JYVÄSKYLÄ STUDIES IN COMPUTER SCIENCE, ECONOMICS AND STATISTICS
47

Juha-Pekka Tolvanen

Incremental Method Engineering with Modeling Tools

Theoretical Principles and Empirical Evidence

University of Jyväskylä
Jyväskylä 1998

Editors
Markku Sakkinen
Department of Computer Science and Information Systems, University of Jyväskylä
Kaarina Nieminen
Publishing Unit, University Library of Jyväskylä

ISBN 951-39-0303-6
ISSN 0357-9921

Jyväskylä University Printing House, Jyväskylä and
ER-Paino Ky, Lievestuore 1998

ABSTRACT

Juha-Pekka Tolvanen
Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence
Jyväskylä: University of Jyväskylä, 1998, 301 p.
(Jyväskylä Studies in Computer Science, Economics and Statistics
ISSN 0357-9921; 47)
ISBN 951-39-0303-6
Finnish summary
Diss.

The main objective of this study is to improve the applicability of information system development (ISD) methods supported by modeling tools. This is carried out by examining and extending method engineering (ME) processes. To draw on an analogy with software engineering, ME develops and improves ISD methods. Instead of introducing a set of standardized methods in an ISD project, we assume that its method requirements vary. ISD projects differ greatly and are more situation-bound than is usually assumed. We suggest that methods should be constructed according to the needs of particular ISD situations and contingencies. To continue the analogy, just as software engineering is guided by ISD methods, ME is guided by (meta)methods. In order to develop ISD methods and improve their flexibility we develop methodical guidelines that are founded on engineering principles. These guidelines specify how knowledge related to methods should be described, analyzed, and maintained for ISD projects, and how it should be adapted into ISD tools. The topic of ME is important, since local method development is common in organizations, and there is a lack of knowledge about the development and use of local methods.

In this thesis we focus on incremental ME. Any organization that builds ISs not only delivers systems, it also learns and creates knowledge about how to carry out ISD, and thus crafts new ISD methods. An incremental approach aims to make this experience systematic, leading to continuous method improvement. Accordingly, methods are a part of organizational knowledge which evolves and needs to be maintained in an organization. This thesis puts forward three principles of incremental ME. First, constructs of method modeling languages for carrying out efficient incremental ME are described. Second, guidelines and mechanisms for collecting and analyzing modeling-related experiences are defined, and their implications for method improvements are explained. Third, the viability of the principles proposed is demonstrated in two cases of incremental ME. The resulting ME principles can be applied in organizations which are developing their own method and need methodical guidelines for this task.

Keywords: Information system development methods, method engineering, metamodeling, computer-aided systems engineering

ACM Computing Review Categories:

D.2.1 Software Engineering: Requirement/Specifications:
Languages, Methodologies, Tools

D.2.2 Software Engineering: Tools and Techniques:
Computer-aided software engineering (CASE)

D.2.10 Software Engineering: Design:
Methodologies, Representation

I.6.5 Simulation and Modelling: Model Development:
Modeling methodologies

Author's Address:
Juha-Pekka Tolvanen
University of Jyväskylä
Department of Computer Science and Information Systems
P.O. Box 35
FIN–40351 Jyväskylä
Finland
Email: jpt@jytko.jyu.fi
Fax: +358 14 603011

ACKNOWLEDGEMENTS

For the realization of this dissertation I am indebted to many people and organizations. The Academy of Finland, the Foundation for Economic Education, and the Technical Research Center of Finland have provided funding for the research. MetaCase Consulting Inc. provided vital connections to the case studies and has greatly increased my research motivation by allowing me to combine both the research and practice of method development.

The advice of my supervisor, Professor Kalle Lyytinen, has been inspiring and significant. It was he who encouraged me into research in the first place, and directed me towards the field of method engineering. I would like to extend my thanks to the external reviewers of the dissertation, professors Juhani Iivari and Richard Welke, and to Göran Goldkuhl, Matthias Jarke, Mauri Leppänen, who have commented on this work as it has progressed.

While carrying out this research I have been fortunate to work in the MetaPHOR project. In fact, the research problems addressed in this work would not have been possible to study to such an extent without the work carried out in the research group. My long term research colleagues, Steven Kelly, Pentti Marttiin, Matti Rossi and Kari Smolander have given valuable comments on this work and implemented the metaCASE tools applied in my research. Steven Kelly also assisted with the correction of the English in this dissertation. The other members of MetaPHOR project, including Janne Kaipala, Pentti Kerola, Janne Luoma, Harri Oinas-Kukkonen, Minna Koskinen, Hui Liu, Risto Pohjonen, Marko Somppi and Veli-Pekka Tahvanainen have all contributed in creating the favorable conditions for research within the project.

I would like to express my thanks to the metamodeling language developers, Arthur ter Hofstede, Michael Heym, Steven Kelly, Kari Smolander and John Venable, who have provided comments about metamodeling functionality and checked the metamodels I made. With respect to the practice of method development, I would like to thank several method developers and adapters, including Ari Jaaksi, Brian Henderson-Sellers, Harri Lindström, Hans Nissen and Pirkko Vesterinen, for their valuable discussions and insights into local method development.

Finally, I would like to thank my parents and my wife Marjut for supporting me throughout the whole research process, and especially Marjut for setting aside her own career to accompany me as I wrote up this thesis as a visiting researcher in RWTH Aachen, Germany.

CONTENTS

1 INTRODUCTION

1.1 Problems in information system development

Though information has become one of the most valuable assets of modern corporations, development of information systems (IS) faces many problems. Among the most important are low productivity, a large number of failures, and an inadequate alignment of ISs with business needs. The first problem, low productivity, has been recognized in the term “software crisis”, as indicated by the development backlog and maintenance problems (cf. Brooks 1975, Boehm and Papaccio 1988, Jeffrey 1987). Simply, demands for building new or improved ISs have increased faster than our ability to develop them. Some reasons are: the increasing cost of software development (especially when compared to the decreasing cost of hardware), the limited supply of personnel and funding, and only moderate productivity improvements.

Second, IS development (ISD) efforts have resulted in a large number of outright failures (cf. Lyytinen and Hirschheim 1987, Charette 1989). These failures are sometimes due to economical mismatches, such as budget and schedule overruns, but surprisingly often due to poor product quality and insufficient user satisfaction. For example, one survey (Gladden 1982) estimates that 75% of IS developments undertaken are never completed, or the resulting system is never used. According to the Standish Group (1995) only 16% of all projects are delivered on time and within their budget. This study, conducted as a survey among 365 information technology managers, also reveals that 31% of ISD projects were canceled prior to completion and the majority, 53%, are completed but over budget and offer less functionality than originally specified. Unfortunately this area has not been studied in enough detail to find general reasons for failures. As a result, we must mostly rely on cases and reports on ISD failures (e.g. Oz 1994).

Third, from the business point of view, there has been growing criticism of the poor alignment of ISs and business needs (cf. Earl 1989). While an increasing part of organizations' resources are spent on recording, searching, refining and analyzing information, the link between ISs and organizational performance and strategies has been shown to be dubious (Smith and McKeen 1993). For example, most managers and users are still facing situations where they cannot get information they need to run their units (Davenport et al. 1992, Rockart and Hofman 1992). Hence, ISD is continually challenged by the dynamic nature of business together with the ways that business activities are organized and supported by ISs.

All the above problems are further aggravated by the increasing complexity and size of software products. Each generation has brought new application areas as well as extended functionality leading to larger systems, which are harder to design, construct and maintain. Moreover, because of a large number of new technical options and innovations available — like client/server architectures, object-oriented approaches, and electronic commerce — novel technical aspects are transforming the practice of ISD. All in all, it seems to be commonly recognized that ISD is not satisfying organizations' needs, whether they are technical, economical, or behavioral. Consequently, companies world-wide are facing challenges in developing new strategies for ISD as well as in finding supporting tools and ways of working (Rockart and Hofman 1992, Benjamin and Blunt 1992).

1.2 Methodical support for information system development

One widely acknowledged approach to solve these problems has been to improve and apply systematic guidelines and procedures for ISD[1]. This type of knowledge is typically incorporated into ISD methods, which we can briefly define here as systematic and predefined guidelines for carrying out at least one complete task of ISD effort[2]. As the considerable amount of effort poured into the method development indicates (cf. Jackson 1976, Gane and Sarson 1979, Lundeberg 1982, Rumbaugh et al. 1991, Booch 1994), the current paradigm within the scientific community advocates methods. Maybe an analogy to the use of methods and techniques in other engineering disciplines (e.g. electronics, civil) or even in less engineering-oriented disciplines, such as recording and composing (Jaaksi 1997), is so close that methods are sought for ISD as well. It is a general opinion among both practitioners and academics that ISD failures are resulting from the application of irrational approaches. ISD methods are viewed as one solution to these problems (Fitzgerald 1996). In fact, the drive towards

[1] Some other organizational and technical innovations include CASE (Computer Aided Systems/Software Engineering), 4GL (fourth generation languages), application package-based ISD, development and use of reusable designs and code, and quality assurance programs.

[2] ISD methods are defined and characterized in more detail in Chapter 2.

better methods and practices is common in all fields of systems development, including business modeling and re-engineering (Bubenko and Wangler 1992, Smith and McKeen 1993), development of IS architectures (Bidgood and Jelley 1991, Stegwee and van Waes 1993), system analysis and design (Olle et al. 1991), and implementation (Jeffrey 1987). In particular, improvements in early phases are believed to lead to higher productivity.

The goal of method development is to build up collective experience of IS development and utilize it to craft systematic development practices. Such experience is obtained through participating in ISD, evaluating methods, and conducting studies on method use. Based on this experience, method developers promote their own concepts, beliefs, modeling languages and procedures. In general, methodical approaches are expected to lead to more acceptable and successful solutions, and to a better-managed development process.

As a result, we currently find hundreds of methods. By taking into account organizations' own "dialects", i.e. methods developed in-house, we can assume that thousands of more or less similar methods are available (Bubenko 1986, Grant et al. 1992). In addition new or improved methods are being introduced continuously. Similarly, there are thousands of tools available for automating and assisting these methods. In fact, nearly all tasks of ISD are supported with software products varying from business modeling tools (Spurr et al. 1994) and CASE tools (Computer-Aided Systems Engineering, Chen et al 1989, Nilsson 1989) to programming environments. Most methods or techniques for ISD are considered impractical or even impossible to use without automated support (Wasserman 1980, Yourdon 1986, McClure 1989, Smolander et al. 1990). For example, there is little point in writing first in some programming language and then making a translation by hand into a machine language, or in checking the correctness and consistency of system designs without tool support. Accordingly, in this thesis our interest is in those method aspects that can be supported with automated tools.

Paradoxically, despite the efforts poured into method development, there seems to be no universal agreement on whether methods are useful in ISD (Lyytinen 1987, Cotterman et al. 1992, Wynekoop and Russo 1993, Wynekoop and Russo 1997). One major reason for this contradiction is the limitation and narrow focus of research: there is surprisingly little empirical knowledge of method use. The vast majority of research has concentrated on developing new methods, or developing frameworks for method analysis (cf. Olle et al. 1982, 1983, 1986, 1988, Blum 1994), comparison (cf. Hackathorn and Karimi 1988, Hong et al. 1993), and selection (cf. Davis 1982, Kotteman and Konsynski 1984)[3]. Furthermore, most empirical studies on method development or method comparison are based on cases, and on limited experiences of method use (Fitzgerald 1991). Because we know so little about how methods are used in practice, there are only shallow generalizations that could explain the success or failure of method use.

[3] Although the literature offers several approaches to classify, understand, compare and select methods there has been no validation of these approaches.

Although we do not know the effects or usefulness of ISD methods, the market has put a great emphasis on tool use and productivity. The market for development tools, such as CASE tools and application generators, has grown steadily during the last decade, and several new approaches have emerged, such as object-orientation and business process re-engineering. As Welke and Konsynski (1980) and Norman and Chen (1992) point out, tools for supporting ISD have evolved together with technical and methodical innovations. Likewise, vendors' investments in building ISD tools have increased. As a result, organizations world-wide have invested in new ISD tools (embodying various methods) such as business modeling tools, CASE, 4GL's (4th generation languages), integrated programming environments etc. (Benjamin and Blunt 1992). Although the rate of diffusion of CASE tools has been slower than expected, it is relatively widely recognized that the rate of diffusion of these tools will continue to increase in the future (Conway et al. 1995, Hobby 1993, Benjamin and Blunt 1992, Friedman and Cornford 1989). For example, a prediction of the CASE world market in 1997 is $1.2 bn (Hobby 1993), and the estimated annual rate of growth is 35% (Conway et al. 1995).

1.3 Local method development

Despite the plethora of ISD methods available, organizations are seldom satisfied with existing methods. Surveys as well as case studies reveal that organizations tend to develop their own local "variants" of methods, or adapt methods available according to their situation-specific needs (Pyburn 1983, Russo et al. 1995, Hardy et al. 1995, Wijers and van Dort 1990, Aalto 1993, Aaen et al. 1992)[4]. This means that methods from outside an organization do not meet the requirements for its ISD efforts, i.e. they are not considered applicable. As a result, the only choices are to abandon the method, try another one, continue the use of the method, or develop methods locally.

By *local method development* we mean organizations' attempts to develop their own method or methods. This means that the local method includes aspects which are not included in any other single method. In our case of modeling related methods these extensions can include whole modeling techniques, new modeling concepts, or new constraints. Local method development is often carried out by combining and modifying existing methods. Surveys investigating the use of methods have shown that 38% (Hardy et al. 1995) or 36% (CASE Research Corporation, cited in Yourdon 1992) of the organizations used methods developed in-house. These, however, can also be adapted to organization, project, or individual needs. Thus, the difference between local method development and adaptation is noteworthy. By *method adaptation* we mean attempts to modify available methods, including local variants, for situational needs. On the adaptation side, a survey by Hardy

[4] The empirical studies on method use and on local method development are discussed in more detail in Section 2.4.

et al. (1995) showed that 88% of the organizations studied had adapted methods in-house. In another study the percentage of method adaptation is similarly 88% (Russo and Klomparens 1993, Russo et al. 1995).

Studies of CASE tool usage (Wijers and van Dort 1990, Aaen et al. 1992) have obtained similar results. They show an obvious need for local method support and indicate that although companies have introduced CASE tools with particular methods, they face difficulties in CASE use due to the limited possibility to adopt and develop situation specific methods (Wijers and van Dort 1990). Different development situations, cultures, skill levels, and types of IS require different ISD approaches. Accordingly, standard-like methods are less common than expected, and less popular than their local variants. This is interesting since tool markets have focused on fixed and method-dependent tools. This may explain the relatively low acceptance of CASE tools which lack method modification and adaptation possibilities (cf. Aaen et al. 1992). Therefore, a need for more flexible and customizable tools has been emphasized (cf. Forte and Norman 1992, Seppänen et al. 1996).

We can find in the literature some case studies of local method development including extensions of current methods (cf. Aalto 1993, Nissen et al. 1996). These describe how methods have been modified and explain reasons for their evolution. There is, however, surprisingly little empirical knowledge available on local method development and method use (Wynekoop and Russo 1993). In fact, most of the reported work has concentrated on developing new standard methods. Though many organizations develop methods in-house or adapt them, we know little as to why and how this is done, or whether local or adapted methods work better. Some studies on method development indicate that although many companies are "rolling their own"; the selection of methods, their development and introduction seems to be done in an ad-hoc manner by choosing tools and methods on a trial-and-error procedure (Smolander et al. 1990). Although local method development is common there is a lack of proven principles. These principles include how to construct and adapt methods for particular needs, how to check the applicability of the method, and how to organize method development efforts.

To understand local method development efforts in more detail we distinguish five steps that every organization faces while developing methods in-house. The identification of these steps is based on analyzing and synthesizing the literature (e.g. Smolander et al. 1990, Tagg 1990, Tolvanen and Lyytinen 1993, Brinkkemper 1996). These steps are illustrated in Figure 1-1 and described in more detail in Section 3.2. It must be noted that the figure is an assertion rather than a generally proven process model of local method development.

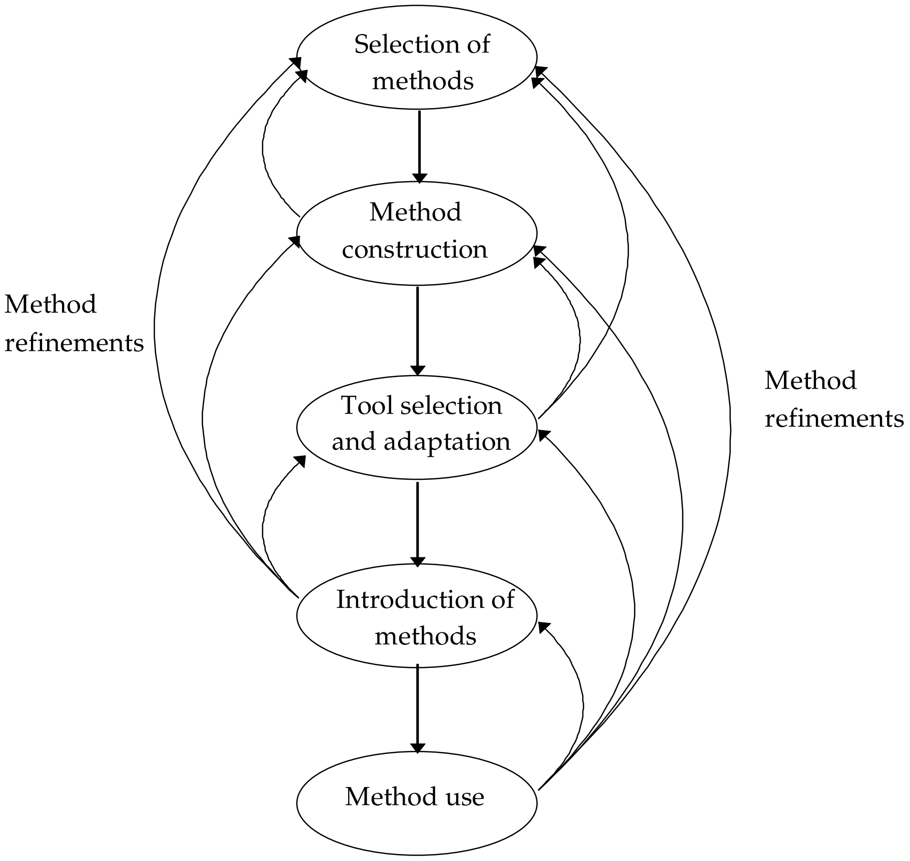


FIGURE 1-1 Steps of local method development.

1) **Selection of methods.** First, every organization or ISD project must make a decision which methods to follow and use. Even an organization that does not use methods at all has made some decision, either explicitly or implicitly. Similarly, cases of local method development (cf. Jaaksi 1997) are often based on a selection of a well-known "text-book" method which is introduced first in its standard form, and later — after gaining experience about its use — modified to better meet local needs.

2) **Method construction** is a task in which selected methods are composed or new ones are created to meet specific objectives of ISD (Kumar and Welke 1992, Heym and Österle 1993). This task includes building, improving and modifying a method by specifying its components and their relationships.

3) **Tool selection and adaptation.** Tool adaptation can be defined as a task in which a given method is represented and implemented for an ISD tool in such a way that the tool can support tasks as prescribed by the method (Tolvanen and Lyytinen 1993). If such a tool modification is not possible, organizations still face the question "which method-specific tools should be selected?" A more detailed discussion about different selection strategies for CASE can be found from Bubenko (1988).

4) **Introduction of methods** deals with various tasks for initiating method use, such as teaching, carrying out possible pilot projects etc.

5) **Method use** refers to an actual ISD effort in which methods are utilized together with supporting tools.

The transitions described show a succession of steps through the selection of methods to their use, but it is also possible to omit some steps. For example, local method development can be carried out without selecting or using any ISD tool, or without proper method introduction. Similarly the steps can be overlapping (Nuseibah et al. 1996). Other transitions are also possible or can even be more common. For example, an organization does not necessarily use, and thus need to select, a computer-aided tool, or an organization can start from the middle step by choosing a CASE tool, and indirectly the accompanying methods.

The specification of transitions suggests a "loop" to method refinements which is of major interest in our study: At each step of method development new experience can lead to method modifications. This part is illustrated by arrows on the left and right sides. An organization or an ISD project not only produces ISs, but also gains and creates knowledge about the ISD. Typically, at least part of this knowledge can be incorporated into ISD methods.

The figure identifies several transitions for method refinements. Such refinements can occur either before, during or after the use of a method. In the former case, method refinements occur during method construction, tool selection or adaptation, or method introduction. For example, the capabilities of the tool for method adaptation can lead to new method modifications. In the latter cases, situations that have taken place during method use are analyzed, generating new insights on how to use methods. This experience-based method refinement can be characterized using Schön's term, reflection-in-action (Schön 1983). In this loop the situation "talks back" and the practitioner reframes the situation. Depending on how experiences are externalized (Nonaka 1994), refinements can take place during or after method use. Refinements which occur during method use — "on-the-fly" — deal mostly with an individual's interpretations, and give new meaning to a constructed method. It must be noticed that this type of refinement often occurs without any documentation, and thus takes place in the dimension of tacit knowledge (Nonaka 1994). If these experiences are made explicit, and thus available for other method users, they can be related to the methods constructed and used to modify earlier methodical understanding. In this thesis we mostly examine method refinements, because we believe that the applicability of methods can only be improved when experience is made explicit for future ME efforts.

According to this *a posteriori* view of ME, an important factor in local method development is the capability of an organization or project to learn of its method use, create knowledge about the applicability of a method, and utilize this knowledge for refinements. This is true not only of method development but also of organizational knowledge creation in general (Nonaka 1994). In this study our interest is not in how efficiently an organization develops ISs, but in how it creates explicit knowledge of the ISD method. Our focus will be on defining, refining, validating and discarding ISD knowledge. Methods are seen as one part of organizational knowledge, which evolves and needs to be collected, analyzed, maintained and purged. We believe that how and to what extent this is done in different situations is of great importance to the success of local method development and to the usefulness of methods.

1.4 Alternative strategies for local method development

Although local method development steps may seem straightforward, there are great differences in how ISD methods are selected and developed locally, and how they can be introduced. To highlight some of these differences we have analyzed the literature on methods, their selection and development (Davis 1982, Sullivan 1985, Olle et al. 1991, Kumar and Welke 1992, Brinkkemper 1996, Odell 1996, Harmsen 1997). Based on this analysis we distinguish three basic strategies for local method development. These are: a text-book approach, a contingency approach and a method engineering approach, either at the organizational level or at the project level.

These strategies can be considered as ideal types in different situations. They differ in the extent of the changes that are made to methods to meet the situation specific needs (cf. Figure 1-2). In a text-book approach a whole method is chosen; in a contingency approach selection is largely made by choosing individual techniques from a large set; in method engineering selection is made by choosing components of techniques (or methods) and by constructing unique components. Hence, the method development strategy applied in an organization can be identified by studying how different the resulting method is when compared to other known methods. It must be noted that these strategies are not mutually exclusive; indeed they are often combined. For example, some modeling techniques may be chosen as text-book techniques because they are considered de facto standards whereas other techniques are be developed from scratch or by extending existing techniques (e.g. in Jaaksi 1997). These differences will be discussed in more detail in Section 2.5.

Each method development strategy extends the scope of modifying methods for local situations. Thus, the text book approach and contingency approach portray a limited adaptation possibility whereas the ME approach suggests that ISD methods should be constructed for the use situation. Hence, organizations which apply text-book methods believe that development situations are generally alike, and thus can be solved with standard solutions. Standardization efforts of methods, like SSADM (CCTA 1995), IDEF (FIPS 1993a) and UML (Booch et al. 1997) are examples of this approach, although they also aspire to other objectives such as communication between different ISD tools. In contrast, organizations which develop their own methods are examples of a different opinion. They believe that development situations in the organization or in projects are very different and furthermore this difference influences the applicability of methods. Different method development strategies also have implications about the maturity of the organization’s ISD process (Humphrey 1989) because ME and detailed modification and use of methods necessitates that an organization is first able to understand its ISD processes, and second measure them to develop better ISD procedures and guidelines. This means that organizations which successfully modify their methods to meet their situational requirements can not be at low maturity levels. An organization must have at least defined their ISD process (Humphrey 1989): successful modification of methods in an organization or in its projects is not possible if their use process is not known. In the following we shall study these three strategies in more detail.

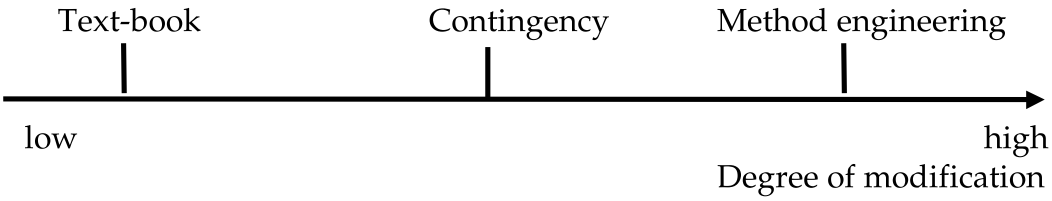


FIGURE 1-2 Strategies for local method development.

1.4.1 Text-book approach

The most common approach to select and introduce methods is probably simple trial and error (Smolander et al. 1990). Organizations choose their methods, either consciously by selecting one of the well known "text-book" methods often backed by consultants, or indirectly by introducing a CASE tool that applies a specific method. A new methodical approach is then introduced without modification. The text-book approach offers a simple strategy for local method development: the method construction and tool adaptation steps do not take place.

The underlying rationale behind this approach is that situations and problems in ISD are similar, or at least similar enough to be analyzed and solved by applying general methods applicable to "almost" all situations. This text-book approach to ISD methods can be characterized as what Schön (1983) calls "technical rationality". According to this approach, situations of practice can be scientifically categorized, problems are firmly bounded, and most importantly they can be solved by using standardized principles. From the technical rationality point of view, we can see ISD methods as universally valid techniques for instrumental problem solving. It must be noted that although the need for flexibility is recognized in some methods (cf. Wood-Harper et al. 1985, CCTA 1995, Booch 1994, Coleman et al. 1994), they do not include mechanisms to modify them according to the various characteristics of ISD.

1.4.2 Contingency approach

An alternative approach for method selection is based on contingency theory. It suggests that there is no universally acceptable method which is applicable in all circumstances. Hence, a contingency approach is based on the observation that situations of practice can be classified, but are more situation bound than the text-book approach expects. Because current methods do not offer general rules for considering situational expectations and deviations (Iivari and Kerola 1983, Vlasblom et al. 1995), contingency frameworks for method selection try to establish this connection by relating methodical needs and available methods. Researchers following a contingency approach (e.g. Davis 1982, Kotteman and Konsynski 1984, Sullivan 1985, Naumann et al. 1980) have tried to identify prominent characteristics (i.e. situation dependencies) which control outcomes of the use of methods and predict their suitability. These characteristics can be technical, such as the type of an IS or the programming language applied; organizational, such as the development culture and maturity; or human, such as the level of experience and learning.

Although the contingency approach in method research is mostly used to analyze situational features of methods, it is also applied for method selection and development (e.g. Vlasblom et al. 1995, Punter and Lemmen 1996, Savolainen 1992). For example, a contingency framework developed in the HECTOR project proposed several situational characteristics, like the type of project activities, ISD environment properties and method/tool properties for tool selection (Savolainen 1992). In contingency frameworks for method selection new methods are not necessarily developed; rather they are selected from those available. Thus, in contrast to the steps of local method development (cf. Figure 1-1), the contingency approach focuses on the selection of an available, appropriate method rather than on more detailed method construction (Kumar and Welke 1992). This bias towards selecting methods from those available leads to "bounded" construction and selection of methods.

1.4.3 Method engineering

Although contingency theory has considerably expanded our understanding of methods suitability, its *a priori* assumptions, once applied as a method selection framework, neglect possibilities for method choices other than those already prescribed. Moreover, contingency-based method selection ignores the impact of organizational learning. Both these problems are addressed in local method development. The first one deals with the insufficient competence to find situation dependencies (cf. Kumar and Welke 1992, Grant et al. 1992). This observation is supported by empirical studies of method use (cf. Hardy et al. 1995, Russo et al. 1995, Wijers 1991, Fitzgerald 1995). They show that situations at an organization, project or individual level often cause changes in methods. Simply, understanding of methods increases while methods are being used. However, this type of knowledge is not usually included in the contingency-based method selection.

The second problem comes from selecting methods (Kumar and Welke 1992). Because varying contingencies will cause changes in methods during their adaptation with new tools, their learning becomes expensive, or even impossible. If research has indicated that ISD professionals do not have enough knowledge and experience of methods (Aaen et al. 1992), how could they be competent to choose between methods? Hence, method selection and development should be considered in relation to both *a priori* contingencies and cumulated organizational experience.

To complement contingency-based method selection researchers have proposed an idea of a detailed method construction in close connection with the use situations (cf. Brinkkemper 1996, Kumar and Welke 1992). Instead of selecting a method purely from an available library according to contingencies, ISD methods should be constructed to meet a particular IS development's needs. This approach is called *method engineering* (ME, Kumar and Welke 1992) as it aims to construct or "engineer" an ISD method according to stakeholders' requirements. Simply, the idea behind ME is the same as behind building any system: just as ISD develops and maintains systems supporting business processes, ME aims to develop and maintain systems for ISD. This is an alternative approach since ME assumes that ISD can often not be carried out solely according to a set of available methods. In fact, according to the ME approach ISD methods should be adapted to local situations even if it requires detailed modification of methods. Here, the fundamental assumptions are uniqueness and difference in ISD situations which can not be solved solely by using general and universally valid methods or general contingency-based selection principles. This approach also necessitates more detailed and systematic steps of method development (cf. Figure 1-1). In particular, the steps of method construction and tool adaptation are emphasized.

ME approaches can be further distinguished by whether they aspire towards an organization-specific or a project-specific method. This division can be also found in practice as described in more detail in Section 2.5. The first one, *organization-based ME*, is based on an assumption that development situations — and thus also supported methods — are alike in an organization and the method can be developed to meet these requirements. In the organization this method is then believed to be appropriate for all projects. Baskerville (1996) calls these methods contingency methods, as they are situation specific for certain types of bounded organizational settings. Numerous examples of these approaches can be found. For example, the Pandata corporation has developed various versions of the SDM method and supporting tool (SDW) to be used in the company (cf. Turner et al. 1988).

Another ME approach is *project-based ME*, which assumes that methods should be "engineered" on a project basis. Because this approach copes with the uniqueness of each ISD setting (Baskerville 1996), it focuses on advancing method knowledge in the context of a single ISD project. Thus, it is believed that development situations differ between various projects. An example of local method development effort in this category is Nokia Telecommunications, whose OMT++ internal method has been developed to be used for designing network management systems for mobile phones (cf. Aalto and Jaaksi 1994, Jaaksi 1997). Although this method might be applicable in other divisions or projects of Nokia it has been developed from one application and a project point of view.

Although there can be an in-house method in the organization, according to project-based ME there is also a possibility to adapt it, or even to develop various project variants. For example, a questionnaire based study (Wynekoop and Russo 1993, Russo et al. 1995) claims that over 2/3 of the companies have developed their methods in-house. It also shows that half of the respondents believed that the organization should use a single method for all projects (i.e. follow an organization-wide method). At the same time, however, 89% of respondents claimed that methods should be adapted on a project level in contrast to using the same method in the whole organization (i.e. follow project based ME). These results clearly show a lack of knowledge of local method development and adaptation. This may be due to the fact that the question of whether an organization has or has not developed an in-house method can be understood differently. Thus, the question should not only be whether or not methods are developed in-house, but also to what extent they are modified, or adapted, and how the modification is done. Unfortunately, the study neither explains the variation to different answers, nor does it reveal whether organizations that have purchased methods from outside are more willing to follow the method than those which have developed a local method. However, the study raises several questions that remain largely unanswered. For example, to what extent do organizations adapt methods? How are these efforts organized? Are ME efforts project-driven or organization-driven? How is knowledge related to methods gathered and organized? How are method refinements carried out, and what is the role of method-related tools in method evolution. In the following section we shall analyze the state of ME research in relation to these questions. This will lead us to formulate our research questions and research approach.

1.5 Research questions and research methods

In this section we shall first describe the research approach adopted and relate it to published research on method engineering. Second, we formulate the research questions, and finally we describe the research method.

1.5.1 Research topic

In this thesis our topic is ME principles for local method development. Reasons for selecting this topic are twofold: First, new situations and challenges of ISD, such as client-server architectures, object-oriented approaches, or business process re-engineering, necessitate the formulation of new methodical approaches. Accordingly, instead of selecting methods from the collection of available ones (e.g. by using contingency frameworks) organizations are facing needs to modify and even to develop local variants of ISD methods (cf. Seppänen et al. 1996). At the same time methods must be analyzed, constructed, adapted into tools and maintained in a different fashion when compared to other method development strategies.

Second, current approaches to method selection and development do not provide adequate support for learning and creation of methodical knowledge. Hence, in this study local method development is viewed as a knowledge creation process which can not be done in a "one-shot" manner. As cases of local method development (e.g. Turner et al. 1988, Aalto and Jaaksi 1994) reveal, in-house methods do not remain fixed over time, rather they have a history with various configurations: parts of the methods are modified, some parts are excluded, and new ones are included. Therefore, methods must be seen as one part of organizational knowledge, which evolves and needs to be collected, maintained and shared. Based on this we argue that an important factor in local method development is the capability of an organization or a project to learn about method use and deploy this knowledge for method refinements. Thus, our research approach is anchored on the one hand in beliefs underpinning method engineering (Brinkkemper 1990, Kumar and Welke 1992) that focus on developing situation-bound methods, and on the other hand in theories of organizational learning and knowledge creation (Schön 1983, Nonaka 1994).

1.5.2 Research domains and related research

Before we formulate our research questions, we will conduct a survey of related research. This allows us to position our research within the context of ME research during problem formulation. In their prominent article Kumar and Welke (1992) describe ME and suggest four domains that have to be addressed in ME:

1) modular method construction,
2) stakeholder value based method composition,
3) need for computer aided support, and
4) organizational support for ME.

In the following each research domain is discussed in more detail and related research is described[5].

1) **Modular method construction**. Several researchers (cf. Kumar and Welke 1992, Harmsen et al. 1994a, Heym 1993) suggest that ME can be carried out by using pre-defined and tested method modules. These modules — often called a component base (Kumar and Welke 1992), or method fragments (Harmsen et al. 1994b) — help specify knowledge about ISD methods in two ways. They either describe a method's static part through its conceptual structure, or the dynamic features of a method, i.e. its procedural part. The first aspect is incorporated in meta-data models (Brinkkemper 1990) which describe the conceptual structure of modeling techniques together with their representations. The latter aspect is defined by meta-activity models (Brinkkemper 1990), or by process models (Marttiin 1994, Jarke et al. 1994). These models contain knowledge about the stages and tasks of a method.

Most research done in this domain has focused on developing metamodeling languages (cf. Welke 1988, Wijers 1991, Smolander 1992, Heym and Österle 1992, Rossi 1998, Marttiin 1994, Harmsen et al. 1994a). Principles for using pre-defined modules and utilizing metamodels for method analysis and refinement have been far less studied. Here research has focused on comparing and combining metamodels (e.g. Hong et al. 1993, Henderson-Sellers and Bulthuis 1996b) and developing metrics for metamodel-based method

[5] A more detailed analysis of related research can be found from Tolvanen et al. (1996).

comparison (Rossi and Brinkkemper 1996). Moreover, advances in metamodeling languages have mostly taken place in meta-data modeling (cf. Welke 1988, Smolander 1992), though some process models (Verhoef et al. 1991, Marttiin 1994, Jarke et al. 1994) as well as integrated meta-data models and process models have been developed (Heym 1993, Marttiin et al. 1995, Harmsen et al. 1994a). Major differences among these approaches can be found in their modeling power and capability, degree of formality, and ways to represent method knowledge. Because ME is a relatively new research field, there is a lack of experience in applying metamodeling and modular method construction principles. A few cases studying ME practices have focused on relatively small methods and mostly on the adaptation of methods to modeling tools (cf. Tagg 1990, Tolvanen and Lyytinen 1993, Cronholm and Goldkuhl 1994). Also some laboratory based experiments on representing method knowledge have been carried out (e.g. Wijers 1991, Verhoef 1993). However, they focus on individual aspects (i.e. how a single developer understands and uses a method) rather than on the use of methods in the large and by many. Hence, most studies reported on method modeling can be found from method comparisons and analysis (cf. Song and Osterweil 1992, Hong et al. 1993). For these reasons, the essential question: "How can we represent, criticize, analyze and refine method knowledge adequately to support local method development in practice?" has largely remained unanswered.

2) **Stakeholder value based method composition**. Because ME can be regarded as a change process, it is relevant that constructed methods meet users' requirements. Hence, ME requires methods and guidelines to identify stakeholders — such as designers, programmers, IS users and managers — and their requirements (Kumar and Welke 1984, 1992). This, in fact, is an essential factor in accepting constructed methods. It can be expected that method users will more easily learn the methods, accept them, and use them if the methods are based on their requirements, in contrast to the situation where introduced methods are purely based on requirements outside the organization. The involvement of method users has been emphasized in recent method development efforts (e.g. UML, Booch et al. 1997) in which method user's requirements and comments are collected more extensively than ever before. Although the participation is important it has not been studied as extensively: identification of stakeholders, dealing with conflicting requirements, and responsibilities in decision making are less studied in the ME literature.

In this research domain few empirical studies have been carried out. Goldkuhl et al. (1992) studied five CASE tool adaptation projects and identified different roles and needs for the tool adaptation. In this study, however, the research focus was on technical issues dealing with customizable tools rather than on local method development. Similarly, other studies of ME (e.g. Tolvanen 1995) have focused on a limited number of stakeholders and a few contingency factors.

3) **Need for computer aided support**. Another research stream in ME has focused on developing tools for capturing method knowledge (cf. Heym 1993) as well as building metamodeling-based tools that can be customized (cf. Teichroew et al. 1980, Chen 1988, Sorenson et al. 1988, Bergsten et al. 1989,

Smolander et al. 1991, Rossi 1995, Kelly et al. 1996). These tools, often called CASE shells (Bubenko 1988), metasystems (Sorenson et al. 1988), or metaCASE tools (Kelly 1994), offer facilities to tailor CASE tools with desired methods. Hence, as ISD methods are supported by CASE tools, similarly metamodeling languages are increasingly supported by metaCASE tools. This symmetry has naturally introduced a more general term CAME (Kumar and Welke 1992, Computer Aided Methodology Engineering) to highlight the role of computer-based tools in ME.

As in CASE research (cf. Wynekoop and Conger 1991) there is a bias in ME research towards building metaCASE and CAME environments rather than evaluating them. There are many articles that describe either principles and requirements for such environments (cf. Marttiin et al. 1995, Harmsen et al. 1994a, Goldkuhl and Cronholm 1993, Heym 1993), or represent how one particular system has been implemented and how it works (cf. Teichroew et al. 1980, Sorenson et al. 1988, Bergsten et al. 1989, Chen 1988, Smolander et al. 1991, Rossi 1995). There is, however, a paucity of research that describes the use of these tools in practice. Only two empirical studies addressing the capabilities of adaptable environments was found[6]: Goldkuhl et al. (1992) studied method adaptations carried out with four different tools and five methods. Marttiin et al. (1993) made laboratory experiments by adapting the same method to three different CASE shells. These studies reveal that CAME tool developers have concentrated so far on techniques that allow tool adaptation rather than on developing techniques and principles for utilizing tool based knowledge about methods for example in method selection, method composition, construction, and reuse. Yet, without proven ME principles, the development of advanced tool support for ME will be slowed down.

4) **Organizational support for ME.** The use of ISD methods always involves a supporting organizational structure and mechanisms that ensure method selection, development, training, use, and maintenance. The key research question here is: "How should ME be organized inside a company together with its ISD efforts?". This research domain is hardly tackled in the ME literature although methods are actually developed, taught and used locally (Wijers and van Dort 1990, Aaen et al. 1992, Aalto 1993): because organizations develop their own versions of methods, these tasks are already being managed somehow. Few discussions available (cf. Bubenko 1988, Tagg 1990, Tolvanen and Lyytinen 1993, Tolvanen 1995, Nissen 1996) study the roles and tasks needed for method engineering. Research in this domain has so far focused mostly on proposing an organizational position of a method engineer. Studies of the other people involved or tasks and organizational structures and mechanisms needed to carry out ME in practice are missing.

[6] Most articles related to use of metaCASE tools (e.g. Tagg 1990) describe only the current adaptation product but do not evaluate the adaptation process.

1.5.3 Problem formulation

The goal of this thesis is to improve the situational applicability of ISD methods that forms a part of a modeling environment. This objective is examined as a problem of method engineering. Our special interest is in incremental aspects of ME. Any organization that builds ISs not only delivers systems as an outcome, but also learns and creates knowledge about ISD methods. In fact, knowledge on ISD and ISD methods is one of the most valuable assets in ISD organizations: methods can be seen as a part of organizational knowledge, which evolves and needs to be collected and shared in an organization. Consequently, creation of new knowledge about ISD methods can be characterized as an incremental learning effort in contrast to selecting methods solely in a "one -shot" manner and using them as readily applicable standards.

According to the incremental approach, an important factor in local method development is the capability of an organization or a project to learn about method use, externalize the experiences into explicit knowledge, and utilize the experiences for method refinements and knowledge creation (cf. Schön 1983, Nonaka 1994). In incremental ME method knowledge is managed by using metamodels combined with method experiences and supported by CAME tools. Our primary interest is not in how efficiently an organization develops ISs, but in how it creates information and knowledge about the ISD and about the ISD methods it applies. Our research objective can also be seen as an aim to develop methodical guidelines for ME. Method engineering is driven by a method, i.e. a metamethod. In fact, Kumar and Welke (1992) define ME itself as a "method for designing and implementing ISD methods".

The motivation for our problem formulation is based on two observations: first, many organizations tend to develop their own methods, and second, there is a lack of principles and guidelines to carry out local method development (Russo et al. 1995). Although there is a plethora of methods available for ISD, hardly any could be found for local method development and for method engineering. To develop principles for method engineering the following research problem is formulated:

How does metamodeling support the local development and adaptation of ISD methods?

This question is divided into two more specific questions:

1) **How completely can meta-data models represent knowledge about ISD methods for modeling tools?** This problem can be defined as a method modeling (i.e. metamodeling) problem. It deals with the modeling power of metamodeling languages and inspects semantic data models as a basis for metamodeling. The problem is examined by seeking metamodeling language constructs to specify detailed method knowledge. Thus, this research question deals with extending support for metamodeling. We use the term meta-data model to denote a description of static method knowledge, in contrast to the dynamics of methods which are captured with process models or meta-activity

models (Tolvanen and Lyytinen 1993), or with other type of metamodeling languages (cf. Section 3.2.2). Strategies for meta-data modeling include modeling of a single technique (i.e. its conceptual structure and representations) and integration of techniques into a method. We concentrate on meta-data modeling because most customizable ISD tools focus on changing the static part of method support, and similarly most reported cases of tool adaptation deal with specifying static aspects of methods (e.g. Tagg 1990, Goldkuhl et al. 1992, Nissen et al. 1996).

This question is important since appropriate metamodeling constructs are needed to describe the methods being developed and adapted (Wijers 1991, Brinkkemper 1996). Research in this area (cf. survey on ME research, Section 1.5.2) has focused so far on modeling single techniques or a relatively small collection of techniques. Moreover, if we want to apply metamodeling as a vehicle for method construction (Kumar and Welke 1992) and tool adaptation (Tolvanen and Lyytinen 1993) this question is of great importance: a detailed metamodel is a pre-requisite for developing tool support for a method. In terms of the steps of local method development (cf. Figure 1-1), this research question deals with method construction and tool adaptation.

2) **How can experience of method use together with metamodels be applied for method refinements?** Because knowledge in general (Nonaka 1994, Schön 1983) and of method use in particular is created by individuals, the ability to build up and capture experience is important for local method development. Our subject here is experience of method stakeholders (such as designers, tool experts, method engineers) which can be used to improve in-house methods. The question deals thus with principles of method refinement. Method refinement is investigated through a process of organizational learning (Schön 1983) in which experience about methods is obtained during method use, and knowledge is created through a continuous dialog with the collected experience and assessment of method use (cf. Nonaka 1994).

Two factors motivate this research question. First, method modeling has not been studied from the viewpoint of incremental method development, i.e. how experiences can be used for method refinement. The traditional approach (cf. Brinkkemper 1996) has been to construct methods once in the beginning of each ISD project rather than to provide mechanisms to gather experiences and relate them to the available method specifications. To extend the ME process we propose mechanisms for evaluating and improving the situational applicability of methods applied in modeling tools. In terms of the steps of local method development (cf. Figure 1-1), this question deals with advancing or refining methods based on experience. Second, empirical studies on method modeling and construction have been laboratory experiments or small cases (cf. Wijers 1991, Verhoef 1993, Tolvanen and Lyytinen 1993). Because of bias in individual developers (e.g. Wijers 1991), we lack knowledge of how organizations or teams develop their own methods. In this thesis we demonstrate the viability of the proposed incremental approach in two cases of method engineering. Thus, method development and method refinements are studied here in a longitudinal rather than snapshot manner, and on a project level. One reason for this can be found from our survey of ME research (cf. Section 1.5.2) which reveals that we lack knowledge of how ME efforts can be organized.

To summarize, this thesis puts forward some principles for incremental ME. These principles aim to systematize local method development. Our special focus is on the evolutionary nature of method knowledge. We argue that an important factor for the success of ISD methods is how an organization or a project creates and maintains method knowledge. In incremental ME metamodels can be used for capturing method knowledge, analyzing methods used, and refining methods based on available experience of method use. By finding answers to these questions, we can analyze available ME approaches and extend the principles and methods of ME. By doing so we can improve the flexibility of ISD methods and overcome the problems faced in the dominant "one-shot" introduction and use of standardized methods. In terms of domains of ME research (see Section 1.5.2), and the problems formulated above the thesis focuses on the first research domain: construction of methods based on meta-data models. The problem addressed, however, is also related to other research domains of ME. In the domain of tool support CAME tools can implement the proposed metamodeling capabilities as well as support experience gathering. In the domains of organizational support and stakeholders' roles the incremental principles suggest how experiences can be collected and analyzed in an organization. Finally, whereas most studies on ME have focused on developing metamodeling languages and tools our study deals with the process of actual method construction and development.

1.5.4 Research methods

Selection of research methods is always dependent on the research setting and problem. At the same time, problem formulation can be done in favor of a particular research method. In this thesis we apply two kinds of research methods. The first research method, used to study the metamodeling related question, "how completely can meta-data models represent knowledge about ISD methods for modeling tools?", is conceptual: we model 17 ISD methods and validate their meta-data models by implementing methods in computer-aided tools. These method specifications are then used to analyze method knowledge as part of modeling tools and to extend languages for method modeling. This type of inductive approach has rarely been applied to such an extent for analyzing and developing metamodeling languages for ME (Tolvanen et al. 1996). Thus, the selected research method complements other research approaches applied (cf. Tolvanen et al. 1996).

The second question, "how can experience of method use together with metamodels be applied for method refinements?", is studied both conceptually and empirically. In the conceptual part we analyze the literature on ME and relate it to the mechanisms of knowledge creation and organizational learning. In the empirical part we follow an action research strategy (Rapoport 1970, Susman and Evered 1978) also applied in IS research (Wood-Harper 1985, Jönsson 1991, Checkland 1991). The need for empirical approach is obvious because ME is a relatively new research area, and thus has received little attention to theoretical and research methodical issues. Especially in (meta)methods and ME efforts we could find neither reported cases nor systematic studies which aim to develop metamethods[7]. This observation implies that ME needs to be studied in its natural setting, i.e. in real life organizations. In other words, we believe that it would be difficult and hard to develop principles for ME in a purely deductive way.

In the study of incremental ME we examine two cases in which methods were developed and adapted to local needs. Both of these cases cover all the steps of local method development (cf. Section 1.3). They allow us to build a rich understanding of method development, and demonstrate the feasibility of the incremental approach. The main benefits of applying an action research strategy is to gain in-depth and first-hand understanding of the processes that take place in an organization in a natural setting. In the studies we gather requirements related to methods and capture this information in meta-data models. The data about an organization's method development effort is collected by interviewing method engineers and users, and by observing the ME process. Also, the modified CASE tools are used to analyze the methods as they are supported with tools. As an outcome of the data collection we obtain different versions of methods (in terms of metamodels and adapted tools) together with reasons for method refinements. On the data analysis side, the explicit relation of method specifications to their changes offers a mechanism to indicate and explain method evolution.

In principle, surveys, field studies and laboratory experiments are all appropriate in studying method development. For example, in studying local method development and use of in-house methods, Russo et al. (1995) used surveys and Smolander et al. (1990) carried out a field study. Moreover, Wijers (1991) performed three laboratory experiments to study method knowledge as understood and used by ISD professionals. However, these approaches focus on obtaining a snap-shot view of practice, or do not offer a possibility to analyze the richness and detail of ME. Most importantly, they do not capture changes in ISD methods as well as an action research method does. In our opinion, these are of great importance when examining incremental ME.

The use of action research does not come without cost: The study does not meet the standards of "positivist" research because the approach offers few possibilities for statistical generalization and the researcher can not exercise control over experimental conditions. However, several researchers have advocated an action research approach in systems development research (cf. Galliers and Land 1987, Galliers 1992, Wood-Harper 1985, Checkland 1991), because the nature of method development and use of methods emphasizes a close interaction between theory and practice.

[7] Although the literature offers some metamethods which include aspects of ME in addition to method construction, they focus on an *a priori* view of ME (cf. Section 3.2) and have not been validated or even demonstrated in real-world ME efforts (Tolvanen et al. 1996).

1.5.5 Limitations of the study

This study has several limitations. One notable limitation is the definition of an ISD method. As the title of the thesis suggests our view of methods is limited in how methods relate to modeling tools, such as CASE. We are interested only in those parts of ISD methods that can be modeled, formalized, and supported in computer-aided environments. Therefore, implicit or hidden parts of methods, such as their value orientation, are excluded from the study.

The second limitation relates to the focus on meta-data models and use of semantic data models. The former means that our interest in ME is only in static aspects of the method, namely the conceptual structure behind modeling techniques. The latter means that metamodels are developed on the basis of semantic data models which by themselves have limitations in IS modeling and presumably also in method modeling. The semantic data models are selected as a basis for metamodeling because they provide support for incremental ME in which maintainability, modularity, ease of use and support for communication among stakeholders are important. Moreover, most large metamodeling efforts (Hong et al. 1993, Heym 1993, Henderson-Sellers and Bulthuis 1996a, 1996b, Hillegersberg 1997) apply semantic data models, and most repositories apply semantic data models in their schema (CASE Outlook 1989).

The third limitation relates to the research method. Despite the benefits of action research studies, such as its closeness to the real world and focus on detail and change, the results can not be statistically generalized. Rather, they allow us to suggest conjectures (Yin 1993) on an incremental basis. The studies can demonstrate that the suggested ME approach can be useful rather than justifying it to be universally beneficial. A thorough examination of incremental, evolution based ME necessitates a longer time scale and larger samples than applied here.

1.6 Outline of the thesis

The thesis is divided into seven chapters (cf. Figure 1-3). After this introduction and problem formulation, Chapter 2 surveys major lines of research on ISD methods, defines ISD, and characterizes the role of a method and tools in its enactment. We shall also clarify mutual relationships between ISD methods and computer-aided environments such as CASE tools. In Chapter 3 we survey the literature on metamodeling and method engineering. Our goal here is first to introduce some principle elements of ME, and second to examine what kind of ME tasks and metamodeling languages have been proposed to address method development. The limitations of current ME approaches, especially related to representing method knowledge and improving methods in use, form a motivation for the development of an incremental ME approach.

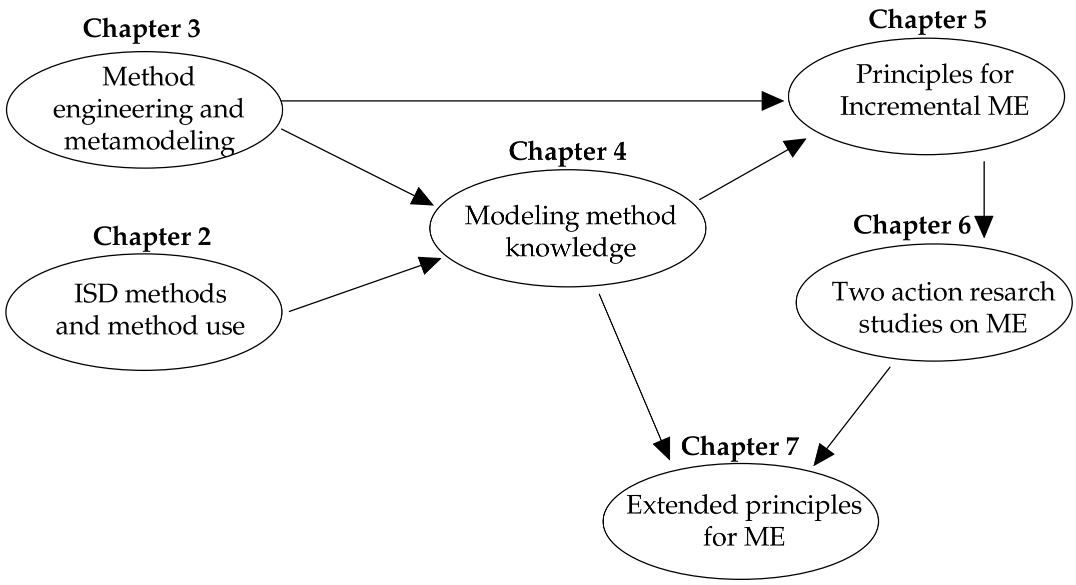

Chapter 3
Method engineering and metamodeling
Chapter 5
Principles for Incremental ME
Chapter 4
Modeling method knowledge
Chapter 2
ISD methods and method use
Chapter 6
Two action resarch studies on ME
Chapter 7
Extended principles for ME

FIGURE 1-3 Structure of the thesis.

In Chapter 4 we study the issue of representing method knowledge, modeling and implementing a large portion of ISD methods into a modeling tool. Thus, our focus here is on studying the first research question, "how completely can meta-data models represent knowledge about ISD methods for modeling tools?". We study what construct s are needed for metamodeling languages by using content analysis to obtain method knowledge from 17 ISD methods. This leads us to propose some extensions to ME and especially to the languages it applies.

Chapters 5 and 6 concentrate on the second research question: "how can experience of method use together with metamodels be applied for method refinements?". Chapter 5 puts forward principles of incremental ME, and studies method development through experience-based method refinement. We propose some ideas for understanding method evolution and the dynamic nature of ME: how the applicability of methods can be evaluated and maintained in changing ISD environments. These principles are applied in two cases (Chapter 6) in which we study local method development in practice. We apply the proposed ME principles together with existing tools to investigate the incremental ME approach. In these studies our primary interest is not on how efficiently an organization develops ISs, but on how they learn about methods during their use, and how this knowledge can be incorporated back into methods. Finally, in Chapter 7 we shall recapitulate major findings of the thesis, and propose issues inviting future research.

2 INFORMATION SYSTEM DEVELOPMENT: METHODS AND TOOLS

Two types of knowledge are essential in method engineering: knowledge of IS development and knowledge of method development. In this chapter we focus on the first of these, information system development, and especially on development methods and tools.

In the following we shall first define ISD and characterize the role of methods and tools in its enactment. Second, in Section 2.2 we shall survey methods based on their characteristics and alternative structures of method knowledge. Third, in Section 2.3 we shall clarify the mutual relationship between modeling tools and ISD methods, referred to here as method-tool companionship. Fourth, in Section 2.4 we shall discuss the paradoxes of method use by looking at the acceptance of methods in general and commercial "t ext-book" methods in particular. This leads us to propose an alternative view of method development, which helps explain the reasons for local method development.

2.1 Information system development methods

We define ISD as "a change process taken with respect to object systems in a set of environments by a development group using *tools* and an organized collection of *techniques* collectively referred to as a *method* to achieve or maintain some objectives" (Welke 1981, Lyytinen 1987). ISD is understood to include development of both manual and computerized parts of an object system. An IS can therefore include both manual and computer-supported parts. Although the definition emphasizes essential components of ISD, such as its social nature and varying objectives, in this thesis we shall mainly focus on the italicized parts of the definition, i.e. on the role of methods and techniques, and their supporting tools.

By a *technique* we mean a set of steps and a set of rules which define how a representation of an IS is derived and handled using some conceptual structure and related notation (Smolander et al. 1990). Olle et al. (1991) and Wijers (1991) call this knowledge a way of modeling. This definition is illustrated in Figure 2-1. By using a technique, system developers perceive, define and communicate on certain aspects of the current or desired object system. These aspects are defined by the conceptual structure of the technique and represented by the notation. By a *tool* we generally mean a computer-based application which supports the use of a modeling technique. Tool-supported modeling functionality includes abstraction of the object system into models, checking that models are consistent, converting results from one form of model and representation to another, and providing specifications for review (Olle et al. 1991).

Examples of modeling techniques are data flow diagrams and activity models. Other techniques can be found from Table 4-1. As a technique, a data flow diagram identifies and names the objects (e.g. process, store) and relationships (e.g. data flow, control flow) which it considers important in developing an IS. Other techniques include other sets of objects and relationships. Modeling techniques also have a notation and a representation form. In a data flow diagram the notation for a process is a circle, and for a data flow a solid line with an arrow-head. The representation form of a data flow diagram is a graphical diagram. Furthermore, a technique defines some principles on how the models should be derived (e.g. decomposition of processes while modeling with data flow diagrams). In other words, a modeling technique specifies which kind of aspects of an object system need to be perceived, in what notation each aspect is represented, and how such representations should be produced.

A *method* can be considered as a predefined and organized collection of techniques and a set of rules which state by whom, in what order, and in what way the techniques are used (Smolander et al. 1990)[8] to achieve or maintain some objectives. In short, we call this method knowledge. Thus, our definition of method includes both the product and process aspects, although dictionaries define the term "method" as meaning "the procedure of obtaining an object" (Baskerville 1996) and therefore emphasize the process rather than the representation (i.e. product of the method use). In contrast, Wijers (1991) notes that most ISD method text-books focus on feasible specifications rather than on the process of how to develop such specifications. In addition, a method also includes knowledge about method users, development objectives and values. We will analyze the types of method knowledge in more detail in the next section.

[8] We aim to avoid the use of methodology altogether; the use of this term has become confused as it originally means the study of methods, but is also used as a synonym for a method.

Examples of methods include Structured Analysis and Design (SA/SD, Yourdon 1989a), and the object-oriented methods of Booch (1991) and Rumbaugh et al. (1991). A short example of method knowledge is in order. The method knowledge of SA/SD can be discussed in terms of the techniques (e.g. data flow diagram, entity-relationship diagram) and their interrelations. In SA/SD the overall view of the object system is perceived through a hierarchical structure of the processes that the system includes. This overall topology is completed by data transformations; how data is used and produced by different processes, how it is transformed between processes, and where it is stored. Moreover, the data used in the system needs to be defined in a data-dictionary and interrelations between data need to be specified with entity-relationship diagrams. Thus, methods describe not only how models are developed but also how they are organized and structured. Furthermore, since ISD methods aim to carry out the change process from a current to a desired state they should also include knowledge for creating alternative design solutions and provide guidelines to select among them (Tolvanen and Lyytinen 1994).

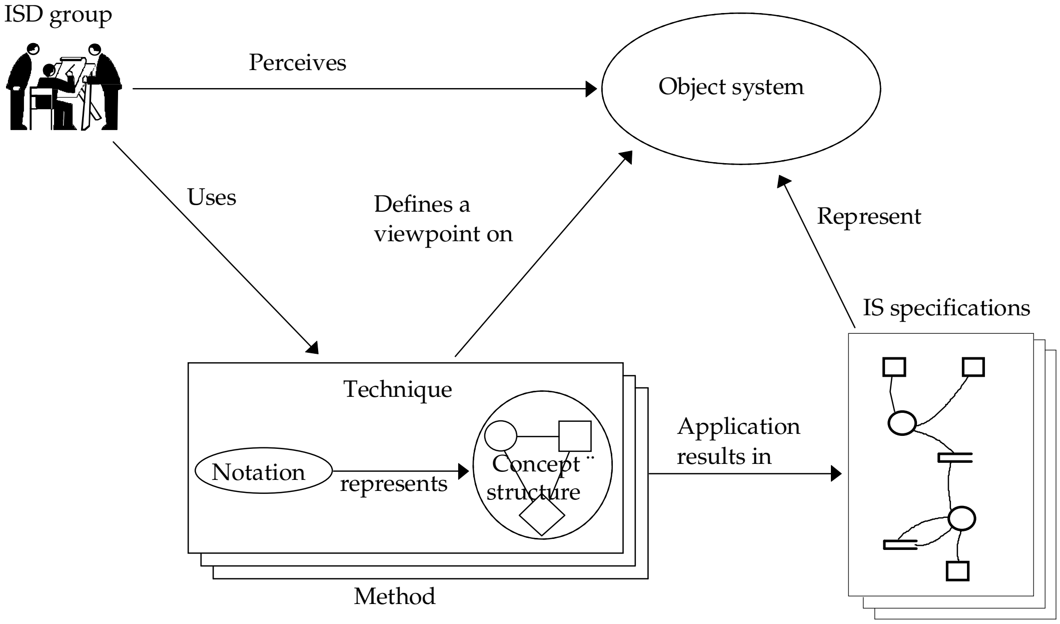


FIGURE 2-1 The role of methods in ISD (based on Lyytinen et al. 1989).

SA/SD and other methods put forward a defined and a limited number of techniques including their conceptual structures and notations. In the same way as there is variety in techniques, there is also diversity among methods (Welke and Konsynski 1980). Different methods include different types and sets of techniques. Interrelations between techniques can be defined differently even between methods which use the same techniques, and the procedures for building and analyzing models can be different. Although there is diversity among ISD methods they include similarities, e.g. they apply the same concepts and notations. To understand these differences and similarities we shall analyze several methods in more detail by describing types of method knowledge.

2.2 Types of method knowledge

The literature suggests many approaches to analyzing and characterizing different facets of methods including their structure, content and use (e.g. Olle et al. 1982, 1983, 1986, 1988, Lyytinen 1986, Hackathorn and Karimi 1988, Wijers 1991, Blum 1994, Krogstie and Sølvberg 1996). These different categorizations are almost as numerous as the methods available. For the purposes of ME, we combine some of them which have been applied in ME research (Wijers 1991, Kronlöf 1993, Jarke et al. 1998) to analyze what type of knowledge ISD methods contain.

The categorization applied here is illustrated in Figure 2-2, whose shape leads us to call it a shell model. According to the model, methods are based on a number of concepts and their interrelations. These concepts are applied in modeling techniques to represent models of ISs according to a notation. Processes must be based on the concepts and they describe how models are created, manipulated, and used with the notation. The concepts and their representations are derived, analyzed, corrected etc. by various stakeholders. In addition, methods include specific development objectives about a ‘good’ IS, and have some underlying values, “weltanschauung” and other philosophical assumptions (Olle et al. 1991, Wijers 1991, Krogstie and Sølvberg 1996).

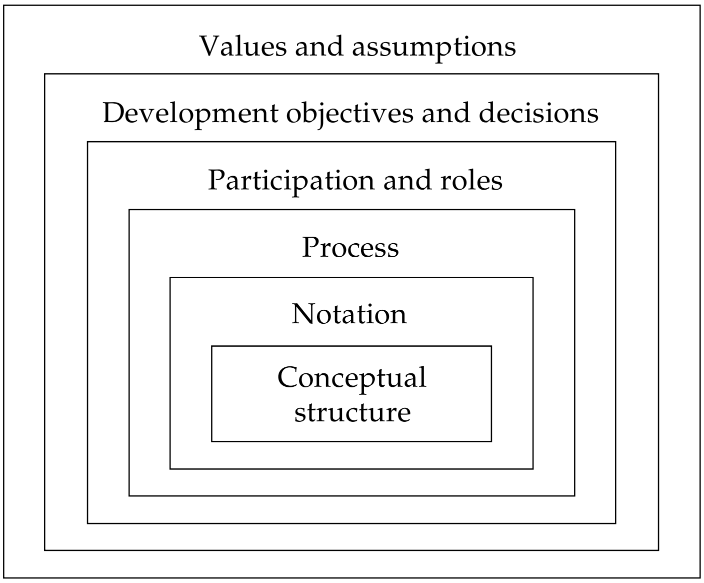


FIGURE 2-2 Types of method knowledge.

The shape of a shell emphasizes that different types of method knowledge are neither exclusive, nor orthogonal. Each type of knowledge complements the others and all are required to yield a "complete" method, although many methods focus only on the concepts and notations included in modeling techniques. To illustrate relationships between different types of method knowledge we can use the concept of functional decomposition as an example (cf. Table 2-1). In the procedural guidelines of Structured Analysis (DeMarco 1979) this concept is described as a top-down refinement of the system starting from the high level diagram. In the modeling technique this concept is implemented as the possibility for every process to have a sub-diagram, and in the balancing of the data flows between the decomposed process and its sub-diagram. The concept of decomposition also affects other method knowledge in several ways: the method should explain who identifies, specifies, and reviews decompositions; the partitioning of the system into a hierarchical structure dominates the design decisions and reveals the underlying assumptions of the method, i.e. that an IS can be effectively designed by partitioning the system based on its processes.

TABLE 2-1 Examples of method knowledge.

Type of method knowledge	Examples of method knowledge
Conceptual structure	Each process may have sub-processes
Notation	Representing sub-diagrams for processes, balancing the data flows between decomposed process and its sub-diagram
Process	Top-down modeling of processes
Participation and roles	Division of labor based on sub-processes
Development objectives and decisions	Design choices are made by partitioning the system into sub-processes
Assumptions and values	An IS can be effectively designed by partitioning its processes

Because of these dependencies it is often impossible to focus only one type of method knowledge. For this thesis, this means that metamodeling the conceptual structures behind modeling techniques is not meaningful if other parts of the method knowledge are not considered. Similarly, it is not meaningful to use a modeling technique just to represent the designs if the underlying values or design objectives are not known. Therefore, when specifying functional decompositions we need to also consider aspects related to the process, or how various design alternatives can be sought using data flow diagrams.

Accordingly, in the following we shall analyze examples of method knowledge using the shell model. The analysis of method knowledge is based on the methods studied in Chapter 4 and on a method survey carried out by Tolvanen and Lyytinen (1994). The analysis allows us to understand methods in more detail as subjects of ME: the applicability of a method is determined partly by how well its specific composition of method knowledge is suitable for the ISD task at hand. By highlighting various aspects of methods, the shell model also provides a clear delimitation of the types of method knowledge that interest us (conceptual structures of modeling techniques). Consequently, the shell model is also applied in Chapter 3, where we inspect what types of method knowledge are addressed with available ME approaches.

2.2.1 Conceptual structure

During ISD it is impossible to analyze and represent the system to be built in its full richness. It is therefore necessary to restrict attention to a smaller number of concepts which are meaningful and sufficient to conceptualize and interpret the relevant parts of the object system. Such a conceptualization consists of a set of concepts, relationships between them and constraints applying to them, forming a conceptual structure.

The conceptual structure forms the basis for other types of method knowledge and therefore all ISD methods are based on a conceptual structure. Some of the concepts are applied directly in notations, e.g. a class with a rectangular symbol as in Rumbaugh et al. (1991), whereas some are related more to the process, e.g. a top-down modeling approach via decomposition; or to design objectives, e.g. clear responsibilities on data usage. Because of the importance of the conceptual structure most arguments supporting a specific ISD method deal with promoting specific concepts. Similarly, most research on method analysis and comparison (e.g. CRIS-conferences, Olle et al. 1982, 1983, 1986, 1988) focus mainly on the conceptual structures of methods. It must be noticed that while defining what aspects can and must be considered during ISD, the conceptual structure of a given method also excludes some other aspects as irrelevant. In this way methods can be seen as enforcing a particular world view via their conceptual structure (Welke and Konsynski 1980). The conceptual structures behind different methods differ for various reasons, but most importantly they vary because of differences in the domain, levels of rigor, and other types of method knowledge considered. These differences are described in the following.

First, because of different ISD contingencies, and differences in the systems being built (from business administration systems to automated robots), a variety of fundamental concepts exist. This is also one of the main reasons why so many methods have been developed. For example, methods which aim to develop single systems (e.g. Yourdon 1989a, Jackson 1976, Ross and Schoman 1977) include concepts such as local functions, data structures, data flows, control flows, and functional decomposition. Methods for managing system architectures (e.g. IBM 1984) focus on internal business processes, data sharing and access rights. Methods for business modeling (e.g. Vepsäläinen 1988, Ciborra 1987, Macdonald 1991) include concepts such as organizational structures and responsibilities, value adding, production and transaction costs (cf. Tolvanen and Lyytinen 1994).

Second, conceptual structures can differ based on their rigor and degree of formality. These explain how well and strictly the relationships between the concepts, constraints and verification rules are defined mathematically. Some less formal and general concepts of a method, such as flexibility in the face of business changes, e.g. in BSP (IBM 1984), are loosely related to other concepts of a method, whereas some other concepts can be defined in more detail. For example, SA/SD (Yourdon 1989a, p 278) defines that each process must be specified with a decomposition or process specification, but not with both. Typically, methods which focus on earlier phases of the ISD life-cycle, such as business modeling or requirements engineering, include less rigorous conceptual structures compared with methods targeted for later phases, such as software design. For example, in BON (Walden and Nerson 1995) the concept of a class is related closely to the equivalent concept applied in a specific object-oriented programming language called Eiffel (Meyer 1992). A more rigorously specified method leads to more uniform descriptions and process, but it will also limit the system developers' freedom in method use situations by reducing the opportunities for contextual modification of the method.

Third, conceptual structures differ among methods in how they cover other types of method knowledge. For example, most methods, like UML (Booch et al. 1997), focus only on the concepts behind modeling techniques. In contrast, other methods like BSP (IBM 1984) also specify procedural guidelines, different user roles, and even state which kinds of deliverables are considered good. Thus, the conceptual structure of BSP also includes concepts which are related to other types of method knowledge, such as processes, participants and design objectives.

2.2.2 Notation

Concepts defined as part of the conceptual structure can be discussed and represented only by using some kind of a notation. In a modeling technique, a notation is always associated with a conceptual structure (cf. Figure 2-1). The association between notation and the conceptual structure defines the semantics of the notation. Notations can be characterized according to the degree of their underlying formal semantics into formal (logic, rules), semi-formal (structured and object-oriented methods), and free form (e.g. rich pictures in (Checkland, 1981)). The degree of formality reflects the underlying conceptual structure, and methods apply modeling techniques with different degrees of formality in different phases of the ISD life-cycle, typically moving from informal to formal (Pohl 1996).

To understand the method knowledge behind notations, its relation to the underlying conceptual structure must be clarified. This relationship is also called the conceptual-representational dimension (Smolander et al. 1990). Viewed from the notation point of view, each notational construct in a modeling technique must be related to some part of the conceptual structure. Ideally, each concept of the modeling technique has only one notational representation, e.g. a symbol. This principle minimizes the overload of notational constructs, and guarantees that all concepts can be represented in the technique. Accordingly, the completeness of representations (Batani et al. 1992, Venable 1993) or representational fidelity (Weber and Zhang 1996), i.e. availability of only one notational construct for each concept, is a well-known criterion for dealing with interpretations between modeling concepts and notations. For example, to describe classes, a modeling technique must have a related construct (i.e. apply the concept defined in the conceptual structure of the method) and define how it is represented (e.g. the "cloud" symbol in Booch (1991)).

From the conceptual structure point of view, each concept does not necessarily have a single notational construct, and may not be supported in the notation at all. The former can be characterized as construct redundancy and the latter as construct deficiency (Weber and Zhang 1996). Although these situations can be considered undesirable they are typical: notations are not necessarily designed to cover the whole conceptual structure, and object system characteristics can be represented by using several notational constructs and modeling techniques. Examples of the former are modeling techniques which apply only a subset of the concepts defined in the conceptual structure. For example, the graphical modeling technique EXPRESS-G is a subset of the EXPRESS language (ISO 1991). An example of construct deficiency is the concept of 'object life-cycle' which does not have any single modeling construct or notational symbol, but can be perceived from a state model of a class (i.e. through instances). Similarly, in BSP (IBM 1984) one key concept is to establish clear data responsibilities. During modeling this is achieved by allowing only one (or as few as possible) process to create a single data class. Therefore, a concept of 'clear data responsibility' can be represented only by perceiving the whole system architecture derived within BSP, as no single construct is available in the modeling technique to represent that notion. In fact, one can claim that 'clear data responsibility' is related to design objectives or to underlying values of the method but not to the modeling technique. Although this claim is true it must be noticed that the modeling technique and notation should also support modeling of data responsibilities. Otherwise, such a design objective can not be represented and alternative choices to achieve it can not be analyzed (Tolvanen and Lyytinen 1994).

Furthermore, all aspects of an IS can not be represented with one modeling technique, and so methods apply multiple techniques, sometimes even to the extent of using several techniques to describe the same aspect. Such different views can serve important goals including communication, analysis, understanding and prediction. As a result, a concept can be perceived in different ways via the different notations applied by different modeling techniques. Hence, construct redundancy is typical in a whole method because it allows the user to interrelate models.

First, different notations can be used to represent models based on the same conceptual structure or the same concepts. An example of the former can be found from UML (Booch et al. 1997) which applies two modeling techniques with different notations yet based on the same underlying semantics, namely an event trace diagram and a collaboration diagram. An example of the latter is the concept of a class which can be represented as a graphical rectangle notation in a diagram or as a string in a cluster chart table (e.g. Walden and Nerson 1995). Second, a notational element may be related to more than one construct of a technique. In this case, the interpretation of the notation depends on the context in which the notation is used. For example, a rectangle can represent an entity in the data modeling context, whereas it represents a terminator in the data flow view. Therefore, the relationship between conceptual constructs of modeling techniques and notations can be many-to-many. On the other hand, all modeling related concepts do not necessarily have notational constructs at all. In particular, concepts related to connections between models (and modeling techniques) are often defined weakly, if at all (Tolvanen and Lyytinen 1994), and thus often have no notation. For example, in most of the object-oriented methods it is difficult to notice from a state model which states belong to different objects (i.e. instances of a class). This limitation, of course, comes from the limitations of representation-related completeness (Venable 1993) and shows overload of notational constructs (Weber and Zhang 1996). In other words, the notation does not distinguish all parts of the conceptual structure.

Independently of the notation, modeling techniques can be classified according to their representation form, the type of format in which the model is represented. The most frequently used representation forms include graphical diagrams, which dominate most methods; matrixes, often used in methods for IS planning (IBM 1984, Andersen 1991); tables, mostly used in methods for early phases of ISD (e.g. Critical Success Factors (Rockart 1979) or Root Definition (Checkland 1981)), indented lists (Goldkuhl 1993); text related to other representations or as separate textual specifications (e.g. Class Description (Coleman et al. 1994), or mini-specs (Yourdon 1989a)) and hyper-text (Isakowitz et al. 1995). Independence of the notation means that the same model can be described in different representation forms but still have the same notational constructs. For example, a graphical data flow diagram can also be represented in a matrix, and the notation elements, e.g. symbols for processes, can be the same (Kelly 1997). The representation form also implies some mappings to a technique and its underlying conceptual structure: For example, a graphical representation with nodes and links implies that a conceptual structure distinguishes between objects and relationships. The modeling techniques analyzed in this thesis are mostly graphical, but include also matrix and tabular representation forms.

2.2.3 Processes

Method knowledge also covers procedural guidelines which describe how an ISD effort should be carried out. The process aspect of the method can be distinguished based on several criteria, but most often it includes modeling related processes (way of working) and management related processes (way of controlling, Olle et al. 1991). The former describes how the ISD method produces its results, the outcomes of the method use, and the latter how the project is planned, organized, and managed. For this thesis, the former is of greater importance, since it is related more closely to the modeling techniques studied here.

Based on our definition of a modeling technique, processes define in what order, and in what way the techniques are used to produce the desired models. To be useful, processes must be based on the conceptual structure of the method. For example, in SA/SD (Yourdon 1989a) a concept of decomposition is reflected in a modeling process as a top-down refinement of the system starting from the context diagram. The possibility to add one sub-diagram for every process must also be supported by the conceptual structure. In contrast, a conceptual structure can also restrict some selections to be made in process. For example, it can include some process-related rules, e.g. that every data flow diagram, except the context diagram, must have a higher level process defined.

The process aspect of the method, however, can not be found explicitly from every method. For example, methods may claim to cover the whole life-cycle of ISD, but actually they offer support for only a few tasks, and are based on limited views of ISD (Kronlöf 1993). The processes can be further divided into those which manipulate elements of the conceptual structure and those which manipulate notations (Lyytinen et al. 1998). Thus, the latter actions deal mostly with the "cosmetics" of the models, such as placing external entities on the border of data flow diagram, or placing super-classes above sub-classes.

2.2.4 Participation and roles

ISD is a group activity in which multiple people participate in different roles, e.g. managers, programmers, designers, and end-users (Olle et al. 1991). Some methods also aim to describe these group aspects, such as organizational structures, responsibilities, and roles that the participating people have. For example, BSP (IBM 1984) defines the stakeholders and different roles needed to define system architectures.

It must be emphasized that most methods do not describe organizational structures related to method use or roles. In fact, most of the methods analyzed in Chapter 4 implicitly assume that they are used only by IS professionals, and mainly by analysts and designers. Partly the participation is implicitly defined according to the intended domain of use: methods which aim to develop a single system naturally have a more restricted set of stakeholders than methods which aim to manage multiple systems or re-design business processes (Tolvanen and Lyytinen 1994). Those which identify roles and other participation-related issues are usually tied to specific ISD tasks in which the participation of end-users or domain experts is important.

2.2.5 Development objectives and decisions

Methods are not only used to describe the current system, they also help to improve the current situation by carrying out the change process. To this end methods also describe how feasible specifications can be sought or alternative solutions generated (Tolvanen and Lyytinen 1994). This is based on the method's implicit or explicit rationale on how a "good"I S should be developed.

Development objectives are general statements about types of solutions considered desirable, whereas development decisions are more explicit and closely related to method use. Examples of the former are the formulation of a system architecture so that it is flexible (e.g. IBM 1984), or re-designing business processes so that hierarchical structures are flattened (Hammer and Champy 1993). The latter are more concrete and describe how the objectives can be obtained. In IS integration methods (e.g. Kerner 1979, IBM 1984, Katz 1990) the main development decision is made based on the degree of (de-)centralization in the organization, and this choice then provides a basis for determining application boundaries. Some IS design methods recognize technical issues, such as hardware capacity, available database management system, and operating mode (Tolvanen and Lyytinen 1994), which should be considered while seeking design solutions.

Development objectives and decisions are related to other types of method knowledge. For example, it is hard to achieve an objective if it can not be perceived, represented and assessed within the method. Therefore, development objectives and decisions should be closely related to the process, notation and the conceptual structure. Sometimes the biggest differences between methods are found in the development objectives: the conceptual models can be partially or even totally alike, but the underlying development objectives can be different. For example, both architecture planning methods (Kerner 1979, IBM 1984, Katz 1990) and BPR methods (Harrington 1991) apply the same concept of a ‘business process’, yet architecture planning methods consider business process structures largely as immutable, while BPR methods aim to change them.

Unfortunately, the link between the objectives and notations and processes often remains unclear. It is rare that all important development decisions are described explicitly, and if described they relate to specific tasks considered problematic by the method developer. For example, Rumbaugh et al. (1991) describe four different approaches which could be chosen to create a schema for a relational database from class diagrams, mainly based on how an inheritance relationship should be transformed into a relational model.

2.2.6 Values and assumptions

Methods are always based on some underlying philosophical assumptions or "Weltanschauung". These can also be called the "invisible" or "hidden" assumptions behind methods (Wijers 1991), or the way of thinking (Olle et al. 1991). For example, Krogstie and Sølvberg (1996) differentiate methods based on three views, constructivistic, objectivistic and mentalistic, based on how reality (in ISD the system to be developed) is observed and what kind of relationship it has with the models.

The distinction between development objectives and underlying values is important since many methods claim to have specific values, but they remain hidden in the method. Another situation is that two methods can aim for the same development objective but with different types of decisions and concepts.

In fact, most of the methods do not explicitly define or even recognize the underlying assumptions.

2.2.7 Summary of method knowledge

In this section we described method knowledge using the shell model. The distinction of different types of method knowledge is relevant for our study to restrict our view of modeling techniques while seeing their role in context. This means that we can focus on those parts of the conceptual structure which are applied in IS modeling.

The shell model also emphasizes the dependencies between different types of method knowledge. Because of these dependencies it is impossible to focus on only one type of method knowledge. In this thesis this means that modeling of conceptual structures behind modeling techniques is not meaningful if other parts of the method knowledge are not considered. Most noteworthy is the conceptual-representational dimension: the dependency between a conceptual structure and a notation. While the notation itself is not one of our interests, the modeling of notational constructs is needed because both the development and use of modeling techniques is difficult without notations and representational forms.

2.3 Information system development tools

The shell model allows us to illustrate the tool support addressed in this thesis: ISD tools include at least a part of method knowledge. Typically tools contain parts of the conceptual structure as their schema definition, support modeling with certain notations, or support the process definition and management (Odell 1996). Tool support is important for our research questions because tools can ensure that method knowledge is also applied and does not remain only as method descriptions (i.e. described method vs. method in use).

While the shell model concentrates mainly on the "deep -structure" of the method knowledge behind ISD tools, the tools also provide support for the surface and physical structures of methods (Wand 1996)[9]. Deep structure denotes those aspects of method knowledge which reflect the domain under development, whereas surface structure and physical structure deal with properties of modeling tools. Surface structure describes user-interface characteristics of an ISD tool, such as how method knowledge behind a modeling technique is visible in dialogs, menu commands and reports. This resembles the notational part of method knowledge. Physical structure denotes the technical means applied in the implementation of the ISD tool.

In this section our focus is on tools which support the use of methods, i.e. way of supporting (Wijers 1991). This formed the third italicized part in our

[9] Originally Wand (1996) used the taxonomy of deep, surface and physical structures to identify aspects of IS, but because ISD tools are also ISs, we use it here to define method knowledge in ISD tools.

definition of ISD (cf. Section 2.1). First, we briefly characterize ISD tools in terms of how they support different phases and tasks of ISD. Second, we describe relationships between methods and tools in more detail through the concept of method-tool companionship. This allows us to explain how tools can support modeling techniques. This is relevant for our research questions, since we seek to apply metamodels in specifying modeling techniques enacted by ISD tools. Thus, it is possible to describe the underlying elements of methods (i.e. a metamodel) on which these tools are based (Teichroew et al. 1980). This focus also means that we believe that the use of metamodeling in local method development is most beneficial when related to customization of tools. Naturally, metamodeling can be applied for reasons other than local method development (cf. Brinkkemper 1990), but local method development aiming only to specify and compare methods takes us only half-way, because the usefulness of a method is realized only when it is applied. Using metamodels without considering their support in ISD tools would be the same as designing an IS without implementing it.

2.3.1 Tool support for information system development

Since the 1970's numerous attempts have been made to support methods via computer tools (i.e. software applications) (Bubenko et al. 1971, Waters 1974, Teichroew and Hershey 1977). Technological developments have lead to a large number of tools that cover nearly all tasks of ISD. At the same time the term CASE (Computer-Aided System Engineering) has been extended to denote all types of computer tools from business modeling and requirements capture to implementation tools.

The concept of CASE is broad and it includes compilers, project management tools, and even editors[10]. In this thesis we examine CASE tools (and methods) in the context of modeling. These modeling tools are usually used to support early phases of ISD. As already mentioned, the term method is restricted in this thesis to mean that part of the method knowledge that it is possible to capture into a formalized part of a tool. Types of methods and tools deployed during different phases of ISD are described in Table 2-2.

As shown in the table, support for business process re-engineering and development include both methods and tools (cf. Spurr et al. 1994). On the method side, process maps, workflow models, task structures and action diagrams are applied (Harrington 1991, Goldkuhl, 1992, Lundeberg, 1992). On the tool side, computing power is applied for example to benchmark, compare, and simulate business processes through models. GDSS (Group Decision Support Systems), CSCW (Computer Supported Cooperative Work) and requirements engineering tools can be used in gathering information and

[10] The need to identify characteristics of different CASE products has lead to several classifications (cf. Chen et al. 1989, Nilsson 1989, McClure 1989) where boundaries are quite fuzzy, like upper- (front-end), lower- (back-end) and mid-CASE as well as toolkits, workbenches and integrated CASE environments. It is also possible to classify tools based on the level of integration: drawing tools without a repository support, project repository-based tools, and organization-wide repository-based tools.

organizing it into a structured format so that it can be used in later phases of ISD. The methodical aspects of these tools rely on brain-storming, interviews and cooperation. In the system analysis and design phases the upper-CASE tools support methods such as conceptual data modeling (ER models and derivatives) and structured analysis and design (e.g. data flow diagrams, decomposition diagrams and state transition diagrams). Most CASE products nowadays focus on supporting object-oriented methods, and recently tool support has been extended towards business modeling (Wangler et al. 1993). In this thesis we also concentrate on business modeling methods which, to a large extent, lack computer support (Stegwee and Van Waes 1993).

TABLE 2-2 Examples of methods and tools in the phases and tasks of ISD.

Phase	Type of methods	Type of tools
Business process re-engineering and development	business modeling, process modeling, work flow modeling, task structures	work flow modeling tools, simulators, business modeling tools
Requirements engineering	brain-storming, interviews, requirements definition and design techniques	GDSS, CSCW, requirements engineering tools
System analysis	data modeling, structured analysis, object-oriented analysis	upper-CASE, interface design tools
System design	data modeling, structured design, object-oriented design	upper-CASE, interface design tools
Construction	mapping from high-level language to machine language, version control	editors and compilers, debuggers, 4GLs, code generators, verifiers, performance analyzers
Operation and maintenance	version control, reverse engineering, configuration management	documentation and reporting tools, reverse engineering tools

The relationship between methods and tools is most obvious in the construction phase: program code written in a high-level language is compiled into machine code. The availability of compilers renders programming methods and languages practicable, because there is little point in writing first in some programming language and then making a translation by hand. During construction and maintenance, computer aided tools can support version control, configuration management, and reverse engineering.

2.3.2 Method-tool companionship

Though the technical realization of the companionship between tools and methods can vary, the need to integrate tools and methods is obvious (Forte and

Norman 1992). On the one hand, tools mechanize operations prescribed by methods by storing system representations, transforming representations from one type of model to another, and displaying representations in varying forms. On the other hand, tools empower users by enhancing correctness checking and analytical power, by freeing them from tedious documentation tasks, and by providing multi-user coordination (access and version control). None of these features could be easily available in manual method use. The companionship between tools and methods has also evolved in response to technical innovations (Norman and Chen 1992). These require extensions to existing methods or entirely new types of methods to support their development (e.g. to cope with distributed systems (Olle 1994)), or then allow new types of methods because technical innovations can be applied (e.g. simulation of state models).

CASE tools do not provide the same level of support for all types of method knowledge. For example, there are more tools that support model building, representation and checking than there are tools that guide processes or provide group support (Tolvanen et al. 1993). Naturally, some aspects of methods lend themselves more easily to automation than others (Olle et al. 1991). Nevertheless some method knowledge need to be present in an ISD tool. The presence of methods can also be viewed using CASE tool support functionality, i.e. each type of functionality necessitates different method knowledge. In the following these are discussed based on support for four different design steps (Olle et al. 1991): abstraction, checking, form conversion and review. Olle et al. (1991) also include a step for decision making, but since it can only be supported through other steps and can not be automated (cf. Olle et al. 1991) we exclude it from the analysis of method-tool companionship.

1) **Abstraction** deals with CASE tool support for capturing and representing aspects of object systems. The majority of steps in design deal with abstractions, and thus it is also the most supported step (Olle et al. 1991). On the level of method-tool companionship this requires that a tool includes all the modeling related parts of the conceptual structure and employs notational representations for them in modeling editors.

2) **Checking** of system descriptions is needed to ensure that models are syntactically consistent with method knowledge. Hence, this design step can be carried out only after some aspects of the object system have been abstracted into models. Checking operates mostly on the conceptual structure and deals with constraints and rules of the method (also called verification rules (Wijers 1991)). Although some checking activities can be carried out by using alternative representation forms, such as matrixes for cross-checking, checking operates mostly on the non-notational concepts. Therefore, to achieve companionship this requires that the conceptual structure of the method includes not only concepts related directly to representation (i.e. abstraction) but also include information to carry out checking. For example, in most object-oriented methods, the link between a state model and a class in a class model is vaguely defined (one good exception is Embley et al. 1992): A state model can include states from several classes and therefore a tool can not analyze whether all attributes of the class have values during its life-cycle. To carry out this type of checking, the method specifications should include a reference from each

state to a corresponding class, or have state models that are used for instances (i.e. objects) of a single class only (as in Embley et al. 1992).

These type of rules concerning the conceptual structures of methods are largely absent, because most methods are developed from a "pen -and-paper" mindset. As a result, we do not have many methods which are developed specially for CASE environments and take full advantage of automation. Furthermore, in methods which apply multiple modeling techniques, the need for checking is stressed. Also, if multiple tools are used, method integration is a prerequisite of successful tool integration.

3) **Form conversion** deals with transforming results from one phase or task to another, e.g. analysis models to design models. During a form conversion an underlying conceptual structure, a notation, or a representation form changes. Examples of such conversions, found in many CASE tools, are model analysis, reporting functions, and code generation. To support these, the conceptual structure should include types and constraints which are not necessarily required for the abstraction or checking steps. For example, to generate program code (e.g. C++ or Java) from a class model each operation representing a method in generated code should include return types as well as access levels (i.e. public, private, protected). These constructs are often missing from conceptual structures of text-book methods. As a result, CASE vendors need to extend methods in order to provide additional tool functionality. It should be noted that not all conversions can be fully automated, but rather often require human interaction.

4) **Review** deals with semantic validity of system descriptions, whereas checking focuses on syntactic properties of the model. Because the review step is often carried out together with the users or experts in the object system domain, the notation part of method knowledge is emphasized here. To ensure that models describe all relevant parts of the system, the notation should be sufficient to represent them. Naturally, the adequate support of the notation reflects the underlying conceptual structure.

Since the effectiveness of the tool is always dependent on the method it is important how a method or its parts are implemented in a tool. In other words, which aspects and which level of detail of method knowledge are supported. In our research, this means that the applicability of methods is partly dictated by how well the tool supports their techniques. Hence, method-tool companionship is based mainly on supporting the conceptual structure and its related notation, and secondly the modeling process and design objectives. The modeling process is relevant because the user interface (i.e. interface structure (Wand 1996)) dictates how the tool can be used and thus affects processes related to modeling: how models are created, how they are accessed, etc. The design objectives are relevant to method-tool companionship because tools should also support generation of design alternatives or produce solutions automatically.

2.3.3 Remarks on modeling tool support

In the majority of current CASE tools method integration has been implemented only partially. Tool developers have concentrated more on producing technical solutions such as repositories and intelligent knowledge-based support in their products, while the methodical part has been given a lower priority. Hardly any CASE tool developers have introduced methods which have been developed especially for CASE environments (Tolvanen and Lyytinen 1993). Furthermore, methods which have been coded as a part of a tool, what we call method-dependent CASE, do not allow the further development or extension of methods according to the situation specific needs. We believe that this technically-driven development of CASE has partly led to the rigidity and weak support of users' native methods.

In our opinion the promise of CASE tools does not lie in the long run in the automated support of old "pen and paper" methods, but in innovative and new uses of computer based methods. Against this backdrop the surprisingly slow diffusion of CASE tools is also more understandable. Research into introducing CASE in an organization reveals that the main problems in the introduction are not the technical changes, but the methodical and cultural changes which the use of the new tool will inevitably cause (Aaen et al. 1992, Aaen 1992, Loh and Nelson 1989, Smolander et al. 1990). These observations are obvious, because the effective use of CASE tools is not possible without an adequate experience and knowledge of method use (Humphrey 1989). Introducing method-dependent CASE tools causes changes in the way of working and in the use of methods. Limited possibilities to adapt the tool into an organization's own standards has often led to growing dissatisfaction among users (Wijers and van Dort 1990).

In contrast to the tool-driven approach, one should select tools so that they fit into the local domain and ISD situations. Several studies of CASE tools (see e.g. Marttiin et al. 1995, Smith et al. 1990) speculate that tool development will lead to method-independent CASE tools, instead of tool-driven development. In the same vein, Bubenko (1988) examines several alternative strategies for selecting CASE tools and introduces seven possible ways to exploit CASE. Four of these, building your own CASE tool, ordering your own CASE tool, integrating several tools and experimentations with research prototypes, (others are wait and see, limited experimentation and buying a method specific CASE tool) allow the adaptation of organizations' methods with the tools. Whereas these researchers have pointed out the demand for flexible CASE support, the technological point of view has still been dominant. Therefore, the opportunities for flexibility in CASE-supported ISD is still at most modest. This problem is discussed from the viewpoint of tool adaptation in Chapter 3.

2.4 Paradoxes of ISD methods

Despite the efforts poured into method development and research, there seems to be no universal agreement whether methods are useful in ISD at all (Lyytinen

1987, Cotterman and Senn 1992, Wynekoop and Russo 1993). For example, Wynekoop and Russo (1993) summarize several fundamental questions on ISD methods which are largely unanswered. Of these questions two are especially important to our study: "are methods actually used in practice?" and "why are local methods developed?" The importance of these questions is further emphasized because of the contradiction between the great efforts made to promote text-book methods and their surprisingly low use in practice. In short, there are thousands of methods available (Bubenko 1986) and new ones are continually developed, but at the same time empirical research reveals that many companies do not use them, and if they do then they have developed their own variants (Hardy et al. 1995, Russo et al. 1995, Fitzgerald 1995, Flynn and Goleniewska 1993).

As a result, it seems that method development is relatively easy since so many of them exist, but methods developed by others do not meet method users' requirements. We can find reports and studies about organizations which have found their local methods applicable or even reported success stories of method use (Jaaksi 1997, Nissen et al. 1996). These observations lead us to analyze two paradoxes of methods in more detail, namely the low acceptance of methods and the popularity of local methods. These paradoxes are important to our research objective of supporting the development of methods through incremental ME.

2.4.1 Low acceptance and use of methods

Although the capability of methods to improve the productivity and quality of ISD has commonly been acknowledged, systematic use of methods is still surprisingly low (Chikofsky 1988, Danzinger and Haynes 1989, Necco et al. 1987, Smolander et al. 1990, Aaen et al. 1992, Fitzgerald 1995). Thus, there is a paradox here between the claimed advantages of methods, which should indicate high use, and the empirical observations revealing low acceptance of methods. This paradox is further emphasized when we consider the amount of work both industry and academics put into the development and study of methods.

The low acceptance of methods is reported by many professionals, confirmed by empirical research and recognized in many studies focusing on the use of tools. For example, Yourdon has estimated (reported in Chikofsky 1988) that only 10% of software professionals have actively used structured methods in their daily practice, and 50% of organizations have tried them at some time. Nevertheless, 90% of developers are familiar with structured methods, emphasizing the low acceptance of methods.

In addition, several empirical studies on the use of methods or tools confirm the estimations on the low use of methods. A study by Fitzgerald (1995) into 162 organizations observe that only 40% of them apply methods. Another study by Necco et al. (1987) into 97 organizations shows that 62% of companies used a structured approach. A study by Hardy et al. (1995) indicates that method use can be as high as 82%. As can be seen, these studies have different or even conflicting results. One reason for the variety lies in the selection of the sample and in the definition of ‘method use’. First, samples are not homogeneous. For example, Fitzgerald (1995) included small companies which did not have large ISD projects, companies which applied packaged software, and companies which had outsourced ISD. These companies were also found to be less favorable to the use of methods, which explains the lower use rate found. On the other hand, studies concentrating on method use normally show higher rates of method use, e.g. 82% in Hardy et al. (1995). Nevertheless, a study by Russo et al. (1995) which focused on organizations using methods still found that 7% of organizations which had claimed in an earlier survey to have a method did not use it. Hence, even if the sample organizations would be the same, respondents can have a different understanding of what methods and method use mean.

Second, distinctions between levels of method use is important, especially the borders between systematic, ad-hoc, and no use of methods. What does it actually mean when ISD professionals say that they follow some method? For example, how fully should method use be defined and documented, how completely should they be followed, and how widely spread and obligatory method use should be in an organization before we can make a judgment that methods are actually used. For example, although in the survey by Hardy et al. (1995) 82% of organizations claim to use methods, it does not mean that they always follow them. In a partial solution to this problem, Fitzgerald (1995) suggests a distinction between formalized and non-formalized methods: a formalized method denotes a commercial or a documented method, and a non-formalized a non-commercial or an undefined method. An organization’s own methods could fall into both categories. By considering only the use of formalized methods the rate of method use drops considerably: from 40% to 26% (Fitzgerald 1995). A field study by Smolander et al. (1990) partly confirms these findings by showing that the methods applied were mostly a collection of loosely coupled informal techniques. Moreover, Russo et al. (1996) characterizes method use based on frequency — used always, seldom or occasionally — to find out the adherence to methods. This categorization shows that most organizations having a method actually apply them (66%).

Thus, the diversity of the meaning of method use and the lack of knowledge regarding how methods are actually used explains differences in survey results. It seems that the use of surveys to study method use and commitment to methods and their actual usage is difficult. As a result, researchers (Wynekoop and Russo 1993, Galliers and Land 1987) have advocated diversity of research approaches. In the case of method use this would generally indicate field studies, case studies, and action research.

Empirical studies, however, reveal the major benefits and drawbacks of method use. Major benefits include enhanced documentation, systematized ISD process, meeting requirements better, and increased user involvement (Smolander et al. 1990, Hardy et al. 1995). Organizations which do not use methods consider the improvements caused by methods to be modest: methods are considered labor-intensive, difficult to use and learn, and as having poorly defined and ambiguous concepts (McClure 1989, Brinkkemper 1990). Methods are also seen as limiting and slowing down development, generating more bureaucracy and being unsuitable (Smolander et al. 1990). Hence, introduction of a method changes the prevailing practices of ISD to such an extent that the method is abandoned or at least its use is made voluntary.

To summarize, method developers have partly failed in introducing methods which would be acceptable by the ISD community at large. There is some empirical evidence which explains which aspects of methods and their use situations influence their success (or failure) (Wynekoop and Russo 1993). The research focus seems to be more on the internal properties and characteristics of methods than on their use situations (Tolvanen et al. 1996). Of course, one may state that the idea of methods is not to apply them as given. In reality, most methods are proposed as universal, i.e. to design inventory systems, automatic teller machines, or mobile phones without considering situational characteristics (Fitzgerald 1996).

2.4.2 Popularity of local method development

A second paradox is related to the use of local methods in contrast to applying third-party methods (i.e. commercial or text-book methods). Surveys investigating method use in organizations (Pyburn 1983, Smolander et al. 1990, Flynn and Goleniewska 1993, Hardy et al. 1995, Fitzgerald 1995, Russo et al 1995) as well as case studies and descriptions of organization specific methods (Kronlöf 1993, Aalto 1993, Jaaksi 19 97, Vlasblom et al. 1995, Nissen et al. 1996, Kurki 1996, Tollow 1996) reveal that organizations tend to develop their own local "variants" of methods, or adapt them (Nandhakumar and Avison 1996) to their specific needs. Hence, there is a paradox here between method developers proposing situation-independent methods and method users who have developed situation-bound methods.

Surveys indicate that local methods are more popular than their commercial counterparts (Fitzgerald 1995, Russo et al. 1995). This partly explains the low acceptance of CASE tools which normally necessitate the use of a fixed method (Wijers and van Dort 1990, Aaen et al. 1992). Among the surveys, both Russo et al. (1995) and Fitzgerald (1995) show that 65% of the organizations which use methods have developed them in-house: their own method is preferred over a third-party one. Other studies obtain similar figures: 62,5% (Flynn and Goleniewska 1993), 42% (Russo et al. 1996), 36% (CASE Research Corporation cited in Yourdon 1992), and 38% (Hardy et al. 1995) of organizations have developed their own methods. Hardy's study, furthermore, claims that 88% of the organizations adapted the methods in-house; the same percentage was found in the study by Russo et al. (1995). Thus, although organizations develop their own methods, methods need to be adapted to different use situations in the same way as with third-party methods. This means that organizations' own methods do not completely fit with the use situations in their projects. Some studies (Hardy et al. 1995), however, have found that organizations which have developed their own methods are more satisfied with them than users of third-party methods. This is quite obvious, since otherwise the local method would hardly have been developed and maintained. On the other hand, few would announce that they have developed a bad method. Thus, it seems natural that methods developed locally are considered better than third-party methods.

Unlike surveys of method use, surveys of local method development get surprisingly similar results, although it would be expected that the distinction between local and external methods as well as between levels of adaptation would be more difficult to make. However, since surveys do not go into details, they do not provide answers about what local method development actually means, or what aspects of method knowledge are modified.

To examine local method development more closely, other research methods such as case studies and field studies are required (Tolvanen et al. 1996). Although local methods are typical in organizations that actually use methods, their selection, development, and applicability is less studied (Wynekoop and Russo 1993). With alternative research methods the modifications of ISD methods could be inspected in detail, e.g. what the development of local method or method adaptation means, as well as how in-house methods differ from third-party ones and how extensive the modifications are. These questions are only partly answered in case studies and reports on local method development (cf. Aalto 1993, Jaaksi 1997, Vlasblom et al. 1995, Nissen et al. 1996, Kurki 1996, Tollow 1996) as they mostly focus on outcomes rather than on differences between local and text-book methods, or how the local method is developed. However, these results are important as a motivation for our aim to develop means for carrying out local method development efforts.

To sum up, many of the organizations or projects which apply methods do not use the methods proposed by others. Commercial methods are modified for example by simplifying or by combining them with other methods (e.g. Jaaksi 1997), or then organizations develop their own methods. This is noteworthy since commercial methods claim to have a well-thought out conceptual structure together with process models and guidance which have worked successfully in other ISD efforts. These methods are furthermore backed by manuals, training programs, tutorials, and tools, necessary when introducing methods. The reason for local method development can not be simply a negative attitude towards something developed outside the organization (i.e. ‘not invented here’ attitude). Development of a local method requires significant expenditure of resources which would not be needed if commercial methods were applied. The relatively high costs, need for resources and recognized ad-hoc method development practices (Smolander at al. 1990) would also discourage local method development efforts. Thus, it seems that the need for more applicable methods is so great that it leads organizations to develop their own methods, either organization specific or project specific.

2.5 Re-evaluation of method use

The two paradoxes above raise several questions about the acceptance and applicability of methods in general, and commercial text-book methods in particular. For example, why develop commercial methods or yet another modeling approach (known as the YAMA syndrome, Oei et al. 1992) if hardly anyone is going to use it? Based on the paradoxes we take a different starting point and re-evaluate the prevailing view of method use. Instead of viewing methods as universal, fixed, and readily applicable mechanisms for instrumental problem solving we view methods more as being situation-bound and describing only part of the knowledge necessary for ISD. Methods are related to an organization's current level of expertise, and they are under constant evolution in organizations which apply them. Thus, the re-evaluation of method use describes a new understanding of methods and seeks to explain the popularity of local methods.

The re-evaluation does not mean that methods should not be standardized or situation-independent, or that commercial text-book methods should not be developed. At least 14% of organizations are still using text-book or commercial methods as specified and without adaptation (Fitzgerald 1995). These methods also provide a starting point for development of local methods. In this study we are, however, concerned with the rest of the organizations: those which develop their own methods, those which adapt available methods, and those organizations which could benefit from methodical support once methods have been defined and constructed to meet their contingencies. Accordingly, in the two following sections we shall define and discuss methods from a different angle suggesting a complementary view of methods — especially of their development and use.

2.5.1 Situation-bound methods

Instead of viewing methods as universally applicable, we advocate that method knowledge is situational. Deriving partially from the popularity of local method development, this is by no means a new claim: several researchers (Wood-Harper 1985, Checkland 1981, Parkinson 1996) also emphasize the importance of situational awareness. For example, Wood-Harper (1985) claims that since method use takes place in real-life situations "a method can not be separated from the problem situation and the analyst's intention and beliefs". As a result, the applicability of method knowledge is always determined in the use situation.

Similarly several method developers (e.g. Yourdon 1992, Walden and Nerson 1995, Booch et al. 1996) argue for situation-dependency and modifiability of methods. Yourdon (1992) supports user-driven method selection by proposing that each developer should use the method that best supports the given situation. Walden and Nerson (1995, p 122) make remarks on extending the use of object-oriented methods to enterprise modeling: "enterprise modeling needs more than the basic object-oriented concepts to be expressive enough. This may very well be true for complicated cases, but the additional needs are probably quite different for different types of organizations". Similarly, although UML (Booch et al. 1996) seeks to standardize object-oriented modeling techniques, its developers have recognized the need to modify the techniques, in particular to better serve different target programming languages. This is especially relevant to UML since it seeks to provide a design-oriented language that provides one level of abstraction over programming languages. Also, methods which are maybe best known for their fixed and standardized approach, namely IDEF (FIPS 1993a, 1993b) and SSADM (CCTA 1995), have abandoned the idea of applying them strictly as specified, and even recommend modifications (Fitzgerald 1996).

Similarly, organizations which have introduced methods have found situational adaptation a necessity. It must be noted that situations affecting the applicability of a method can occur at different levels of an ISD organization: organization, project, or even individual level. For example, in the context of IS planning, Pyburn (1983) states that IS planning must be adapted to the specific organizational context. In the context of software development and use of object-oriented methods Jaaksi (1997, p 71) claims that "every method needs adaptation when taken into use". This means that ISD projects should not be considered as all being the same, as in practice each is to some degree unique (Parkinson 1996). Finally, at the individual level, Wijers (1991) conducted laboratory studies on method use and showed that individual developers tend to change the method while using it.

Although the situations in which methods are used can be different and even opposite between the levels of an ISD organization, the more general levels set conditions on the situational adaptability at the lower levels. For example, an organization-wide method can influence the adaptations made at the project or at the individual level. Unfortunately, there is not much knowledge on how situations at different levels influence local method development. We acknowledge situations from different levels, but like in most ME literature, we emphasize situations which are project specific. This does not mean that we exclude other type of situations; rather the project focus is stressed for relating the developed ME principles to other ME approaches. As discussed in Section 3.2, most ME approaches start by defining methods for the ISD project.

A main problem addressed in this thesis is how to make method development happen according to situational requirements. Methods as described in the literature offer few "built-in" possibilities for modification, and do not provide mechanisms for carrying out required modifications. For example, the methods analyzed in Chapter 4 do not define how customization can be carried out, which are the situational dependencies having an bearing on method modifications, and which parts of method knowledge (e.g. technique, process, etc.) should be a target for modifications. For example, one major difference in the newest version of SSADM (CCTA 1995) compared to its predecessors is that it allows and even recommends method adaptation (earlier, adaptation was not allowed). However, little if any guidance is given on how different parts of the method knowledge should be modified. Typically, guided adaptation includes selecting a full or a limited version of a method (e.g. Booch 1994). This approach offers, however, very limited adaptability in terms of method knowledge. This is not a criticism of "standard" methods, but shows how difficult it is to adapt a standard. One can also claim that build-in adaptation guidelines would not solve the problem because they would make methods even more complex, difficult to learn, introduce and use, and thus decrease their acceptance even further.

To summarize, situation-independent and universal methods are not possible because ISD situations are so different. Applicability of a method in one situation does not mean that it provides successful results in other situations. Similarly, contingency theories (Davis 1982, Kotteman and Konsynski 1984, Sullivan 1985) suggest that the creation of a method which can give the best support in all situations is impossible. This also partly explains why text-book methods are not widely used and why organizations use their own locally developed methods. As a result, the YAMA syndrome, mostly used in a negative meaning, is a natural consequence of the need for situation-dependent methods. If organization or project-specific methods work better and their users are more satisfied with them (cf. Hardy et al. 1995) then why apply third-party methods? In fact, one could even state that we should have more methods and variety in method knowledge to cover various situational characteristics of ISD. It must be noticed that different situations do not necessarily explain all local method development efforts and the YAMA syndrome, since organizations can develop their own methods for marketing purposes, or then because they do not have time to learn from outside. Similarly, an organization's own methods can be promoted to sell consulting or tools (e.g. Frost 1994).

2.5.2 Tacit method knowledge

The underlying paradigm behind many ISD methods is scientific reductionism (Baskerville et al. 1992). This rests on the assumption that the solution can be achieved through a sequence of steps, decisions, and deliverables pre-defined in the method knowledge (Fitzgerald 1996). The expectation of a complete and explicit set of methodical knowledge is, however, too narrow.

The dominant approach underpinning many methods can be characterized as what Schön (1983) calls "technical rationality": situations in practice can be scientifically categorized, problems are firmly bounded, and they can be solved by using standardized principles (Tolvanen 1995, Fitzgerald 1996). This view of development and use of methods is by no means wrong or "bad": it has produced a great deal of knowledge about ISD and led to the development of useful routine procedures which are generally known and used (Fitzgerald 1995). In fact, the main principle of method development can be said to be to provide knowledge about ISD which is explicit and applicable for future ISD efforts. However, not all tasks of ISD fit the view of scientific reductionism. In other words, it is not possible to have full knowledge about the problem (and thus the applicable method) beforehand, nor can pre-defined method knowledge cover all possible situations. Moreover, part of the knowledge related to ISD in general and to methodical knowledge in particular is tacit and thus can not be expressed. Therefore, we claim that the technical rationality is too narrow to address and explain the use of methods as it takes place in practice. As a result, it is our belief that system development can not be completely carried out by following pre-defined methods.

A liberating perspective to support method development is what Schön (1983) calls "reflection-in-action". Here, the fundamental assumptions are uniqueness of situations and tacit, intuitive knowledge (Nonaka 1994). Part of our knowledge of ISD is based on our reflection on the situations in which we find ourselves, rather than being found solely by using predefined methods. Thus, methods need to be maintained based on reflections from practice, transforming tacit knowledge into explicit knowledge. The importance of tacit knowledge partly explains the low acceptance and use of methods, and why successful ISD efforts can be carried out a-methodically (Baskerville et al. 1992) without the use of any "explicit" method. Hence, method is not everything. On the other hand, all ISD efforts can not be carried out based on pure intuition and tacit knowledge (Jaaksi 1997). Therefore, we see the views of reflection-in-action and technical-rationality as complementary views of method development and use: both explicit and tacit knowledge are necessary and useful for successful ISD. Accordingly, a good method should take both aspects into account, on the one hand, providing knowledge which can be rigidly followed as routines, and on the other hand allowing human creativity and spontaneous.

2.5.3 Method use is a learning process

The other assumption behind scientific reductionism (Fitzgerald 1996) is that the developer can obtain detailed knowledge about the problem situation and about applicable methods. This view expects that all necessary knowledge about the method, whether it is tacit or explicit, is available beforehand. In addition to this expectation of complete and explicit methodical knowledge the introduction of methods as readily available "routines" is seen as being easy, and the use of a method assumed to lead to solutions which are repeatable. For example, one of the goals of JSD (Cameron 1989) is to eliminate personal differences and even creativity from the development process. According to this view the key problem for IS developers would be to select the right method rather than to use it.

We question this by emphasizing that method use is a learning process in which the current level of expertise is crucial to successful ISD (Curtis 1992, Hughes and Reviron 1996). The learning process occurs at two levels; in the domain of IS, and in the domain of ISD. The former means learning about successful (or unsuccessful) ISs. The latter means that any organization that builds ISs, not only delivers systems — they also learn how to carry out ISD, and use methods. This learning about methods means that they gain experience about the applicability of methods. This experience can complement the method knowledge they already possess.

The importance of learning about ISD and methods over time was already recognized by Vitalari and Dickson (1983) and Davis and Olsen (1985). According to Argyris and Schön (1978, p 2-3) this forms a double loop of learning in which "error is detected and corrected in ways that involve the modification of an organization's underlying norms, policies and objectives". Single-loop learning is related to immediate tasks, in which error detection "permits the organization to carry on its present policies". In the context of ISD the double-loop learning means modification of the ISD methods. Because ME aims to improve ISD methods, it can be viewed as a learning process in which an individual (Schön 1983), or even an organization (Nonaka 1994), creates new knowledge about methods and how to apply them. Similarly, Curtis et al. (1988) have suggested that both the developer and user learn through the dialectic approach, and Floyd (1987) has advocated a second-order learning process in which past experiences are guidelines for using a method.

The emphasis on learning is important in our discussion because it allows us to explain the low acceptance and use of methods. Although experience is known to be crucial to ISD it is not easy to build up and maintain. In fact, we claim that knowledge about methods can be mostly achieved only by using them. This means that a long time is needed for introducing methods into organizations (Bubenko 1986, Lundeberg et al. 1981), which partly explains why organizations do not use methods: the introduction of methods is a long standing investment which bears fruit only after a relatively long time. For example, Lundeberg et al. (1981) estimated that at least one year is required to introduce a method into an organization. In fact, the first projects where methodical principles are used can often show a decrease in productivity (Aaen et al. 1992).

Another factor explaining the low use of methods is organizations' surprisingly shallow knowledge and experience of methods (see Aaen et al. 1992), and their poor capability to manage ISD (see Humprey 1988). For example, a survey by Aaen et al. (1992) observed that more than half of the organizations considered their knowledge and experience of methods small. Similar results have been found in other surveys (cf. Smolander et al. 1990). Research on software process maturity (Humprey 1988) has shown that understanding of one's own work must precede any further steps in method definition and improvement.

2.5.4 Evolution of methods explained

Instead of viewing methods as finished articles, a view which few method promoters take, methods must be viewed from an evolutionary perspective. Shifts in method knowledge are known (Joosten and Schipper 1996) and an examination of current developments in the field of object-oriented methods, workflow methods or business process re-engineering methods gives no reason to expect that this would change in the near future. An indication of method evolution is that organizations must deal with different method versions, as for example with SSADM (CCTA 1995), introduce new method types, such as object-oriented methods, and abandon old methods which have been found inapplicable for new technologies and applications (Bubenko and Wangler 1992).

Basically, two different types of evolution exist: those reflecting general requirements of technical evolution and business needs, and those relevant to the ISD situation at hand. The former deals with the general historical perspective and the latter with how these general requirements are adapted into local situations and how they affect the method evolution.

2.5.4.1 Historical perspective

The method literature includes several reviews of the development and use of ISD methods (e.g. Welke and Konsynski 1980, Bubenko 1986, Norman and Chen 1992, Moynihan and Taylor 1996). Most of these explain method evolution though an interaction with available or emerging technologies which are used either in the developed systems or in the ISD tools.

Bubenko (1986) analyzed methods from a historical perspective: the need for methods has grown while the complexity and size of ISs has increased. The earliest methods were developed in the 1960's when the first large scale batch and transaction-processing systems were developed. Furthermore, the emergence of databases in the 1970's lead to the introduction of data modeling techniques. At the same time structured design and analysis methods derived their origins from structured approaches and from the evolution in programming languages. Similarly, Welke and Konsynski (1980) characterize advances in technologies, such as database management systems, which were reflected in ISD methods. Likewise, today these surveys could be extended to object-oriented technologies, mobile phones, business process changes, and multimedia. As a result, Welke and Konsynski emphasize that method developers should be aware of technological developments, as they form one key factor in improving and maintaining methods.

Likewise, Norman and Chen (1992) explain method evolution in terms of an evolution of applications developed. They also relate method evolution to CASE tools. Although they primarily discuss the evolution of CASE, a close connection to parallel advances in methods are recognized. For them new applications drive the creation of methods and later lead to the development of CASE tools. Thus, method developers should follow advances in technologies which could support new forms of ISD methods. For example, the emergence of graphical user interfaces and CASE tools supported the introduction and use of methods (Chikofsky and Rubenstein 1988).

Another indication of a method's historical evolution can be found by studying different versions of commercial methods such as SDM (Turner et al. 1988), and SSADM (CCTA 1995). These were developed over long periods of time. For example, SDM (System Development Method), has been developed and updated since 1974 because of the changes in software tools, organizational impact of ISs, and the need to support system maintenance (Turner et al. 1988). Even the newer object-oriented methods have a history of different versions, such as OOD/UML by Booch (1991, 1994, Booch et al. 1997) or MOSES (Henderson-Sellers 1992, Henderson-Sellers and Edwards 1994). Accordingly, some efforts have been made to identify evolution paths between different type of methods, or even to construct a family tree of methods (Smolander et al. 1989). Similarly, there are plenty of studies available which extend methods to support some useful or required design or analysis task, such as distribution (Olle 1994), client-server architecture (Frost 1994), or information systems planning (Stegwee and van Waes 1993).

2.5.4.2 Method evolution in organizations

Another viewpoint on method evolution can be taken by analyzing how organizations develop their methods. This viewpoint is also relevant to our research question about incremental ME. Although organizations' local methods are relatively common we do not know why and how organizations develop their methods, or how frequently methods are refined or updated (Wynekoop and Russo 1993). Since ME is not studied empirically enough (Tolvanen et al. 1996) we must rely on reported cases (cf. Aalto 1993, Jaaksi 1997, Kronlöf 1993, Vlasblom et al. 1995, Russo et al. 1995, Nissen et al. 1996, Tollow, 1996, Kurki 1996, Cronholm and Goldkuhl 1994, Bennetts and Wood-Harper 1996).

In the following the evolution of local methods is inspected by analyzing the "end-products" of ME efforts. This analysis is carried out by focusing on two dimensions of method evolution: the first dimension analyzes how much the locally developed method has changed, and the second dimension how often the method modifications have taken place. These dimensions are illustrated in Figure 2-3 and their measures are discussed below. These dimensions along with the analyzed ME cases allow us to partly explain what method development or adaptation involves.

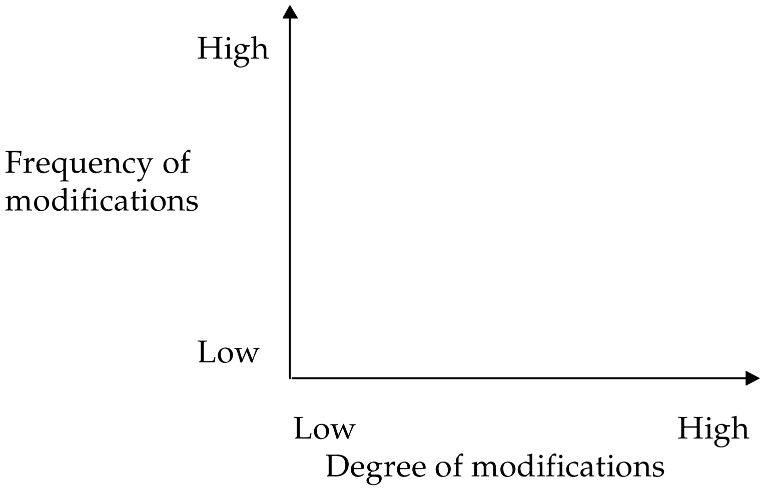


FIGURE 2-3 Characterizing local method development: the degree and frequency of modifications.

2.5.4.2.1 Degree of modifications

The degree of modifications defines how large the changes are that are made to the local method to improve its applicability. These modifications can be (cf. Harmsen et al. 1994):

1) tied to the selection paths provided by a method,
2) based on combining methods, or
3) based on the development of an organization's own method.

This classification allows us to distinguish how much a method used in an organization differs from other methods. The degree of modifications could also compare two changes at different times in the same local method by analyzing the number of method components changed at each time. This alternative dimension is excluded here because ME cases are not reported in such detail that categories could be formed. Hence, in the following each degree of method modifications is discussed by analyzing the current method in use (instead of the current changes).

1) **Selection paths within a method** describe one extreme of ME. Here the only possible modification alternatives are those provided by the method (i.e. built-in flexibility), and thus are limited to a few contingency factors. Examples of these contingencies include development of small versus large systems, the use of prototyping, and the use of application packages (e.g. in SDM, Turner et al. 1988). It is, however, unrealistic to expect that methods should include a much larger set of contingencies and condense them into modification guidelines (Hardy et al. 1995). One clear reason for this is the vast amount of possible contingencies, and even if these could be identified, the growing size of methods.

2) **A combination of methods** for internal use occurs when a chosen method, and its possible selection paths, do not meet the situational contingencies. In a combination (or integration as defined in Krönlof 1993) the local method is based on the constructs offered by several commercial methods, and partly based on modified or totally new constructs. A study by Russo et al. (1996) shows that 37% of the methods used in organizations are combinations of commercial and in-house methods. Accordingly, the adaptation can be carried out either by combining available methods (or method parts, sometimes called fragments, e.g. Harmsen 1997), or by modifying a single method for internal use (e.g. Bennetts and Wood-Harper 1996, Nuseibah et al. 1996). An example of the former is Object-TT (Kurki 1996), which is a company specific method developed by combining available techniques from a larger set of text-book methods. As Object-TT focuses on modeling, it is heavily dictated by the available notations and their underlying concepts. An example of the latter is the modification of the Information Engineering (Martin and Finkelstein 1981) method reported in Russo et al. (1995).

3) **An organization or a project which develops its own methods** faces situations which are outside the set of situations to which known methods are suited. Minor modifications into known methods are no longer sufficient, and thus the developed method does not have any close "relative" among other methods. Ryan et al. (1996) characterizes this category as an effort to develop new conceptual structures (models in their terminology) and related notations. An example of a company which has developed its own methods is USU, a consulting company (reported in Nissen et al. 1996). The method developed, called PFR, focused on rapid requirements capture in team workshops and individual interviews.

Locally developed methods are often considered propriety and information about them is difficult to obtain. Many of the methods which can today be characterized as commercial have a background in an organization's internal needs. For example, Business Systems Planning (IBM 1984) was originally developed to solve the problems which IBM noticed in the management of its own ISs. Similar histories are shared by Objectory (Jacobson 1992) and Octopus (Awad et al. 1996).

2.5.4.2.2 Frequency of method modifications

The second dimension, the frequency of method modifications, explains how often a method is changed (Hardy et al. 1995). More specifically, it measures how often changes in ISD situations are reflected in methods. From the available cases four basic categories can be found:

1) advances and changes in external methods,
2) changes in an organization's ISD situations,
3) a project-by-project basis (once ISD project starts), or
4) continuous refinements within a project.

In the following each category is discussed in more detail.

1) **Method modifications based on advances in external method knowledge** are typical in organizations where methods follow a national or industry standard (e.g. SSADM (CCTA 1995), IDEF (FIPS 1993a), OMG-UML (OMG 1997)), or a method-dependent CASE tool. Thus, new versions are the result of externally decided modifications. Because of the slow standardization process such modifications are carried out infrequently, and do not necessarily relate to organization specific situations. Similarly, if the method is supported by a method-dependent CASE tool, the vendor can dictate the frequency of new versions. Method changes in this category do not normally occur more often than once a year.

2) **Method modifications based on changes in an organization's ISD situations** deal with local method development in which contingencies related to the whole organization change and are reflected in methods. Examples of such changes are outsourcing ISD, introducing new technologies (e.g. Bennetts and Wood-Harper 1996), or starting to develop new type of IS. Hence, the relevant contingencies here are the same for the whole organization. Examples of organization-wide ME initiatives are reported in Cronholm and Goldkuhl (1994) and Kurki (1996). This type of organization-wide method change can occur many times a year. The possibility for in-house method modifications may also be restricted by the CASE tool, as most tools demand a one-shot adaptation (Cronholm and Goldkuhl 1994). Partly for this reason larger organizations have also implemented their own tools (e.g. SDW in Pandata (Turner et al. 1988)) or even applied metamodels to achieve flexibility in changes (e.g. the TDE environment in Nokia (Taivalsaari and Vaaraniemi 1997)).

3) **Method modifications on a project-by-project basis** are considered in ME research to be the most typical. Each project is characterized by individual features which need to be mapped to methods. Modifications are not made during the method use but only at the beginning of every project[11]. Because

[11] It must be noted that organizational units other than a whole company or an ISD project can be identified, such as a department, teams related to developing and maintaining a certain IS, and an individual. Because of the lack of empirical studies on local method development already mentioned, we can not focus here on method modifications

each project is dealt with individually this approach is relevant to project-based ME (cf. Section 1.4.3). For example, in a case reported by Bennetts and Wood-Harper (1996) the successful use of a local method has encouraged an organization to adapt methods for individual projects. Hence, the changes in methods are always based on the schedules of the projects (i.e. a timeframe of months in general).

4) **Continuous method refinement** happens when ISD contingencies are uncertain or change rapidly, e.g. when a new method or methods are used in a new area. Although methods are typically introduced as a whole, the ME efforts analyzed show that method adaptations occur frequently during an ISD project. These modifications do not occur only at the individual level, but also in ISD projects, and in the longer run in the whole organization.

Studies on individual developers' method use (e.g. Wijers 1991) show that methods are gradually changed during their use: e.g. new concepts and new rules are added to the modeling techniques. These personal modifications are, however, often tacit and not shared with other developers. Method modifications are also performed in team-based method use. In this case method modifications are documented and available for others. For example, in Nissen et al. (1996) method modifications related to a supporting tool caused modifications to the method, to the supporting tool, or to both: after the initial method was developed, modifications were made based on feedback from method introduction during internal workshops, during and after the pilot project, and finally after running a few application projects. Third, method modifications also occur in organizations' methods, although not as frequently as in project-dependent methods. For example, clear method modification phases can be found from the ME practices related to the development of one method in Nokia (Aalto 1993, Aalto and Jaaksi 1994, Jaaksi 1997): OMT as a text-book version in 1991, modifications resulting in OMT+ in 1993, and further modifications to create OMT++ in 1994. Moreover, the OMT variant had several smaller and more frequent modifications which were made during its development (Jaaksi 1997).

2.5.4.2.3 Examples of method development efforts

Table 2-3 summarizes the analysis of ME efforts based on the two dimensions discussed above: degree and frequency of method modifications. The table includes ME cases which have been reported adequately enough to be classified. It would be of great interest to also analyze the degree and comprehensiveness of each individual method modification step, rather than looking at the end-product. Unfortunately this is not possible because most of the cases do not describe the method development processes. Furthermore, they usually describe only one or two types of method knowledge which have been modified, like the modeling technique or the ISD process. This naturally makes the classification of ME cases in the Table 2-3 difficult. For example, the method engineers can describe the method developed as a combination of available

occurring in organizational units other than a whole organization or an individual ISD project.

methods (e.g. Jaaksi 1997), but a more detailed analysis of method knowledge can reveal that the method includes many aspects which are not covered by other methods. A simple combination of methods would not lead to such a large modification.

TABLE 2-3 Examples of local method development efforts.

continuous			Aalto 1993/Jaaksi 1997, Nissen et al. 1996
project-by-project basis		Bennetts and Wood-Harper 1996	Tollow 1996
Frequency of modifications organization contingency based		Kronlöf 1993, Kurki 1996, Cronholm and Goldkuhl 1994	
external method based	FIPS 1993a, CCTA 1995		
	within a method	combine methods	own method
		Degree of modifications	

The analysis of the cases reveals that different approaches for local method development are applied. It must be noted that the sample of ME cases is small and thus no firm conclusions can be made, but the analysis does provide some hints about local method development. In some cases it seems to be applicable to follow a standard method and limited adaptation, whereas in other cases larger and more frequent method changes are required. Because of the paucity of empirical research on local method development, the reasons behind these choices are largely unknown. The analysis of method development practices, however, reveals which approaches are not used at all, and can be considered unlikely in ME. None of the organizations has developed its own and radically different method in a short period of time. All the reported cases indicate a more gradual method development process. This is also a reason for developing principles for incremental ME.

2.6 Summary and discussion

In this chapter we have defined ISD, and described methods and tools. First, for the purposes of metamodeling, methods were seen to consist of different types of method knowledge. This analysis focused on method knowledge related to modeling techniques, i.e. on the conceptual structure and notations. Thus, we excluded other aspects of methods and their development. Second, we have described the relationship between modeling tools and methods: the method-tool companionship. This allowed us to show what type of computer support is needed to develop tool support (i.e. abstraction, checking, form conversion and review).

Third, we discussed method use through the notion of method paradoxes. The analysis of method use revealed that the applicability of existing methods is not at all clear, because many ISD organizations do not use the available standard-like methods at all, and have developed their own partially or completely new methods. As a result, the IS research community must admit that we do not know well enough how methods are actually used in development situations, and how important the role of methods is in the success (or failure) of ISD efforts. These paradoxes led us to refine the currently dominating view of methods: we defined methods to be situation-bound instead of universal and standard. We acknowledged that a method is not the sum total of ISD knowledge, as much knowledge about ISD is tacit and can not be provided as readily applicable routines. We emphasized expertise and learning, and viewed methods as evolutionary.

Based on the IS research literature, there appear to be at least three possible ways to research method use. The first is to continue the widely followed research approach to develop new situation-independent and universal methods, compare them conceptually (e.g. frameworks), and use them in cases. However, this approach, despite its use in multiple studies, has proven to be inadequate for resolving problems related to the wider acceptance of methods. The second option is to pursue comprehensive empirical studies on methods in realistic environments (e.g. as proposed by Wynekoop and Russo 1993). Although this proposition is basically correct, it is not a realistic approach for today's organizations. First, they can not stop their ISD efforts and wait for the results. Second, the results of these empirical studies can become obsolete even before they are ready, because of the rapid evolution of the business world and technology. For example, there is not much empirical evidence on the usefulness of object-oriented methods, although this is one of the challenges for ISD in many organizations today. Similarly, there is a paucity of research examining the usefulness of metaCASE tools (Tolvanen et al. 1996).

The third option is method engineering: to focus on mechanisms that support local method development and use. Although many companies are "rolling their own", using local, in-house methods, method development seems to be carried out in an ad-hoc manner by selecting tools and methods on a trial-and-error base. Organizations do not have any principles to guide ME efforts: selecting and constructing methods for particular needs, checking the completeness of methods, or organizing method development efforts. Moreover, organizations face problems in finding and developing tool support and collecting experience of method use. All these reasons motivate the development of systematic principles for ME. In the following chapter, we shall describe approaches or strategies for method selection, construction, and tool adaptation.

3 METHOD ENGINEERING: METHODS AND TOOLS

Two types of knowledge are essential in method engineering: knowledge of information system development and knowledge of method development. In this chapter we focus on the latter, method engineering and especially on the methods, modeling languages and tools of method engineering.

The chapter is organized as follows. In Section 3.1 we define ME and in Section 3.2 we analyze different ME approaches based on their ME process, the types of method knowledge they consider, and the factors or criteria driving ME. These must be described to understand the principles of incremental ME (cf. Chapter 5) necessary to extend the current ME principles. Moreover, tool adaptation as a mechanism to obtain method-tool companionship leads us to explain the role of CAME, metaCASE and CASE tools. In short, we shall focus on creating and maintaining knowledge about modeling techniques in ISD tools. Accordingly, in Section 3.3 we describe metamodeling languages by focusing on how to specify the conceptual structures of modeling techniques. The presentation of metamodeling languages is accompanied with a metamodeling example. This presentation is needed to understand the constructs of metamodeling languages and the evaluation of the metamodeling languages carried out in Chapter 4.

3.1 Defining method engineering

The need for systematic principles to develop situation-specific methods has led to the emergence of method engineering (Bergstra et al. 1985, Kumar and Welke 1992). In a similar vein to ISD, we define method engineering (ME) as a *change process* taken with respect to an ISD object system in a set of *ISD environments* by a *method engineering group* using a *metamethod* and *supporting tools* to achieve or maintain *methods for ISD*. Figure 3-1 illustrates the relationship between method engineering and ISD. In the following we describe this relationship and define ME in more detail by explaining each italicized key concept of the definition.

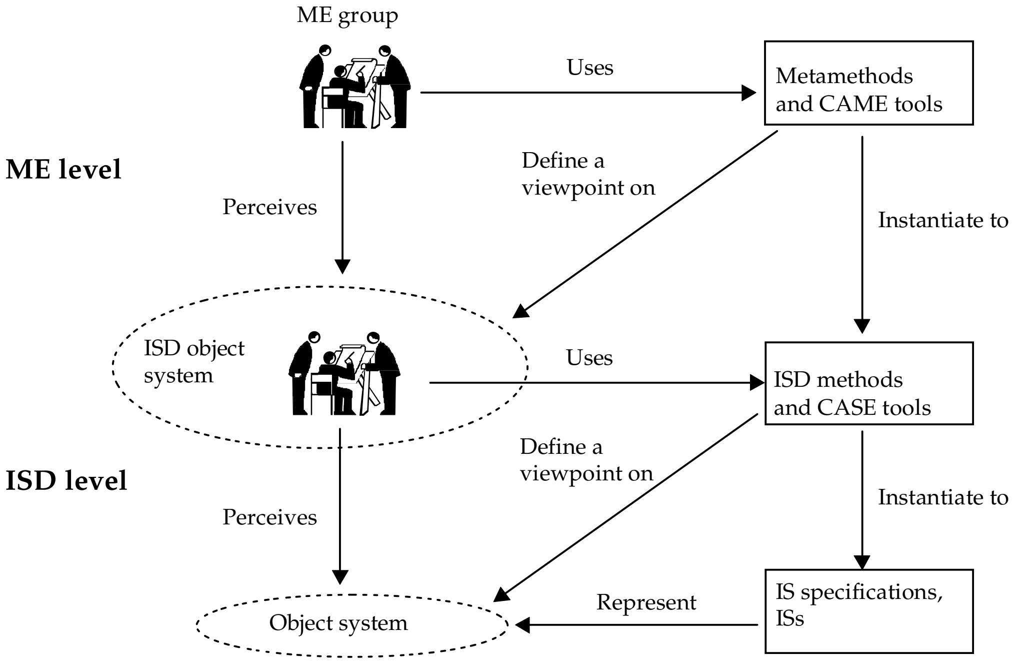


FIGURE 3-1 Method engineering and information systems development.

Both ISD and ME are social processes, in which a number of people act and have an interest. At the level of ME, *method engineers* form a group which perceives the current state of ISD. They are in charge of defining, choosing, modeling and producing method specifications and customizing tools in a similar way to the ISD group creating IS specifications and implementing them. This also distinguishes method engineers from researchers, since the latter are more interested in studying methods (even with metamodels and metamethods) rather than implementing methods for the organization. Method engineers can therefore be considered as developers of ISs for ISD. Often they can be the same group as those carrying out ISD. Furthermore, because the end-users of ISD applications include ISD professionals, they can be expected to be more aware of technical possibilities and thus more demanding than end-users of other type of ISs. This partly explains the importance of stakeholder value based ME (Kumar and Welke 1984, 1992) which emphasizes the role of method users in ME efforts. By stakeholders we mean people who have an interest in method development and method use. These include method experts, tool experts, managers of ISD, IS developers, and IS users. Studies in ME, however, have so far concentrated on developing concepts and principles for ME (Brinkkemper 1990, Heym and Österle 1992), whereas only a few discussions (see e.g. Bubenko 1988, Tagg 1990, Nissen 1996) study the role of method engineers and other stakeholders.

Both ISD and ME aim to deliver an IS, often a computerized one. Method engineers carry out a *change process* resulting in methods and tools which support some tasks of ISD. An example of such a system is a CASE tool customized to support a specific method. During the ME process, a method, or its part, is created, modified, and removed to achieve or improve situation-specific methods. Thus, the goal of ME is to improve ISD by providing better methods and supporting tools. In the ME literature situation-specific needs are understood as a closer relationship between the method and the characteristics of ISD situations (Vlasblom et al. 1995), required problem solving capabilities (Punter and Lemmen 1996), or stakeholders' values (Kumar and Welke 1984, 1992). The "better" in turn implies that the constructed method can be compared in detail with other alternative situation-bound methods or their parts. Each of these approaches to achieve the objectives related to methods is discussed in more detail in the next section.

Both ISD and ME can be supported by *methods*. To differentiate methods between these two levels we use the prefix meta to denote methods and tools at the metalevel, e.g. metamethods, metamodeling and metaCASE. This distinction is also important for this thesis since we focus on studying ME rather than ISD. Like ISD methods, metamethods can be viewed through the taxonomy of method knowledge (cf. Section 2.2). First, a conceptual structure of a metamethod includes concepts specific for engineering ISD methods. Second, the specifications of an ISD method are communicated with a metamodeling notation. Together, the metamodeling concepts and notation form a metamodeling language. As in the term metamethod, the prefix "meta" means that the metamodeling language represents parts of the ISD method in terms of a model of a method, i.e. a metamodel (Brinkkemper 1990, van Gigch 1991). Third, a metamethod includes procedures for metamodeling and constructing methods, and a set of criteria to meet the situational requirements of methods. However, other types of knowledge necessary to carry out ME supported by a metamethod, like the participation and different roles, have been studied far less in ME literature (cf. Tolvanen et al. 1996).

Like ISD, ME too can be supported by tools. This symmetry has introduced the term CAME, Computer Aided Methodology Engineering (Kumar and Welke 1992) to highlight the role of tools in ME. In this thesis we regard the *supporting tools* of method engineers as metaCASE tools (Kelly 1997), also called metasystems (Sorenson et al. 1988), or CASE shells (Bubenko 1988). These tools offer facilities to tailor CASE tools to specific methods.

Finally, an ME process is not performed just once because the *ISD environment* changes. This is emphasized in the definition by the inclusion of the maintenance of methods into ME. The environment also includes stakeholders, who have different, changing, or even conflicting objectives. For example, developers can require methods which minimize errors in a developed IS, managers want the method to improve productivity, and IS users want understandable design documents. The changes and experiences of the method's use raise new requirements for methods and their tool support. As a result, a method constructed at one point of time is not necessarily applicable in the next similar project, or even later in the same project. Therefore, methods have to be maintained and revised. This observation leads to an evolution-based approach where methods are developed incrementally for local and changing needs.

3.2 Method engineering approaches

In working towards more complete principles for ME it is necessary to place this work in the context of similar work reported in the literature. Accordingly, in the following subsections we describe the currently prevailing view of "ideal" ME in term s of its process, criteria, and deliverables. This allows us to analyze alternative ME approaches and describe their underlying assumptions, as well as their weaknesses and strengths. Moreover, and most importantly, the view of current ME principles allows us to describe what are their differences in relation to our focus on ME, namely to engineer modeling techniques for tools.

3.2.1 Method engineering process

The general structure of a ME process (cf. Smolander et al. 1990, Tolvanen and Lyytinen 1993, Brinkkemper 1996, Cronholm and Goldkuhl 1994, Grundy and Venable 1996, Harmsen 1997) is illustrated in Figure 3-2. The model follows the notation of data flow diagrams (Yourdon 1989a) in which processes are circles, external entities are rectangles, and data stores are rounded rectangles. The arrows describe data flows between processes, externals and stores.

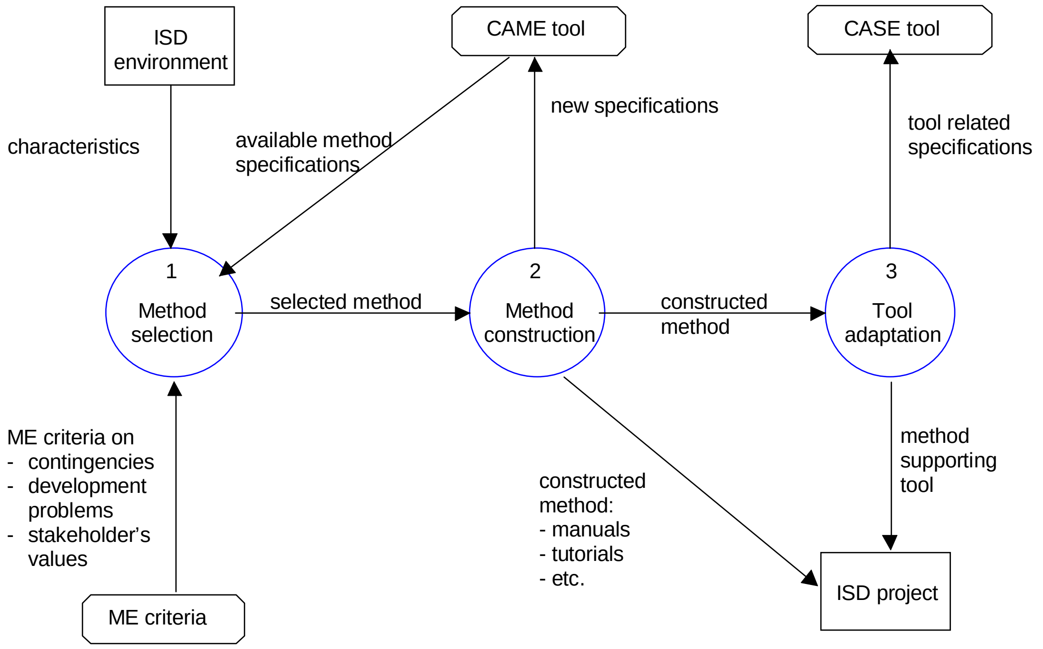


FIGURE 3-2 A data flow diagram specifying ME process.

In the following we outline the ME process by describing each step, namely method selection, method construction and tool adaptation. These are described as processes in the figure. It must be noticed that the figure does not include all steps of local method development (cf. Figure 1-1), such as method introduction, use, or collection of experiences, since the ME literature does not provide any systematic principles for these, although the tasks are usually acknowledged.

In the method selection process the ISD environment is analyzed according to ME criteria. The criteria for methods can be divided into situation-independent and situation-dependent parts. The former criteria are considered desirable for most methods regardless of the situation for which they are developed. Examples of these universal criteria are: easiness to learn, simplicity of use, good support for communication between stakeholders and good support for transitions between different tasks or phases of ISD. These criteria cover more than one type of method knowledge, but can also be specific only to certain types of method knowledge. In our case of constructing modeling techniques examples of general criteria include readability and easy to use.

The latter type of criteria are relevant when we want to increase the applicability of a method for a given situation. Jarke et al. (1998) call these method adaptation criteria, and they are of primary interest for incremental ME. They include classifications of relevant aspects of methods which should be considered to satisfy the objectives for the method. For example, in carrying out IS planning, the degree of centralization of the target organization is suggested as one criteria (Sullivan 1985). If the organization is centralized, IS planning can be performed better with BSP (IBM 1984), whereas de-centralized organizations can be analyzed better with CSF (Rockart 1979). Among the ME criteria we can distinguish between criteria which relate to contingencies, development problems, and stakeholders' values. These criteria are reviewed in more detail in Section 3.2.3.

The selected method (or methods) which meet the ME criteria are constructed, possibly with new method components. This means combining method knowledge from different methods as well as from different types of method knowledge. By focusing on method components (or fragments, Harmsen 1997), and therefore introducing smaller changes, methods can be maintained or even integrated (e.g. Kronlöf 1993) if required. For example, if the programming language changes from C++ to Smalltalk, the method knowledge is modified slightly: it is no longer permissible to use multiple inheritance. Since all criteria are not necessarily met with existing methods, new method knowledge needs to be defined, and some of the method components may need to be removed. The new method configuration is stored into a repository for future selections.

Finally, the method constructed needs to be adapted into a CASE tool. Generally speaking tool adaptation deals with customizing or building a tool for the method, or choosing a set of tools which cover all the method knowledge (for selection strategies see Bubenko 1988). If this adaptation is not carried out then the contribution of the method construction is limited, because a tool could ensure that the method is used as intended. ME without tool adaptation would be the same as developing IS specifications without implementing associated computer-based support. Method introduction also involves non-computer supported parts, such as production of manuals, tutorials, etc., which are not included into the adapted tool. Basically, tool adaptation includes building method-tool companionship, namely the support for abstractions, model checking, form conversion and review (cf. Section 2.3.2). This adaptation requires that the constructed method is modeled with a modeling language offered by the customizable CASE tool. If the CAME tool provides translation of the metamodels into a CASE tool (e.g. Rossi et al. 1992), or method use can be tested in the same tool where the tool adaptation is performed (e.g. Kelly 1997), the tool adaptation becomes easier and faster. Hence, an outcome of a successful ME process is a fully functional computer-based IS for ISD. This defines what developers can store into the repository of an ISD tool, how system descriptions can be represented, retrieved, checked, transformed, and how descriptions are managed.

3.2.2 Types of method knowledge considered

Ideally speaking all sorts of method knowledge and their relationships can be subject to ME: from the underlying conceptual structure to the assumptions and value-orientations of a method. Practically speaking, methods usually only address a few types of method knowledge (Jarke et al. 1998), and methods can be modified by changing only one or a few types of method knowledge (e.g. Kronlöf 1993).

Although the types of method knowledge are related, each type can be viewed independently and represented using a number of alternative representation schemes and mechanisms resulting in different kind of metamodels. Clearly, no method construction is possible without some sort of (explicit or implicit) metamodeling. Thus behind all approaches there are some metamodeling formalisms (cf. Section 3.3). In the following a set of ME approaches are analyzed based on their focus on different types of method knowledge. These ME approaches are taken from a survey of ME research (Tolvanen et al. 1996) and the approaches are summarized in Table 3-1. It must be noticed that a single approach may belong to more than one category, as they typically cover at least modeling of conceptual structures, even if only as the foundation for the modeling notation, procedural guidelines, participation or values.

Because all methods are based on some concepts, ME approaches address the conceptual structure, at least those concepts related to the other type of method knowledge addressed. There are, however, ME approaches which focus mainly on conceptual structure, such as Mercurio et al. (1990), Essink (1988), Olle et al. (1991), and Heym and Österle (1992). In general, these frameworks present conceptual structures or suggest reference models of methods. Hence, here the objective of ME is to identify and establish relevant concepts of ISD and include them in the conceptual structure of the method. A short example of a conceptual structure is now in order. For example, the conceptual structure of most object-oriented methods includes the concept of inheritance, its relation to other concepts, and possibly constraints, such as single inheritance or recursivity. Although this conceptual structure also includes relationships to notations, or to processes, they are often too general to define methods in such a detailed and formal manner that computer support can be built based solely on such a conceptual structure.

TABLE 3-1 Method engineering approaches and types of method knowledge

Type of method knowledge	Method engineering approaches
Conceptual structure	Essink 1988, Merçurio et al. 1990, Olle et al. 1991, Heym and Österle 1992
Notation	Teichroew et al. 1980, Sorenson et al. 1988, Welke 1988, Bergsten et al. 1989, Smolander 1991, Wijers 1991, Bommel et al. 1991, Venable 1993, Hofstede 1993, Tolvanen et al. 1993, Saeki and Wenyin 1994, Oei and Falkenberg 1994, Bronts et al. 1995, Kelly et al. 1996, Grundy and Venable 1996, Harmsen 1997
Process	Wijers 1991, Hofstede and Nieuwland 1993, Tolvanen et al. 1993, Marttiin 1994, Jarke et al. 1994, Rolland et al. 1995, Rolland and Prakash 1996, Harmsen 1997
Participation and roles	Tolvanen et al. 1993, Harmsen 1997
Development objectives and decisions	Jarke et al. 1994, Rolland et al. 1995, Oinas-Kukkonen 1996
Assumptions and values	Kumar and Welke 1984, 1992

The most studied ME approaches are those operating at the notational level, but related to conceptual structures (Tolvanen et al. 1996). These approaches define modeling techniques (e.g. Teichroew et al. 1980, Sorenson et al. 1988, Welke 1988, Smolander 1991, Saeki and Wenyin 1994, Venable 1993). Some of them (e.g. Sorenson et al. 1988, Welke 1988, Smolander 1991, Kelly et al. 1996) are also used as a conceptual schema for modeling tools. Notation-based ME is more concise than those based on conceptual structures: it focuses only on the concepts related to modeling. For example, it defines how the concept of inheritance is represented, related to the representations of other concepts, and how the constraints of the inheritance are supported in a modeling technique. Hence, while focusing on modeling techniques they address the conceptual-representational dimension (Smolander et al. 1990) by defining how concepts are represented. This is typically achieved by relating conceptual structures to their representation definitions (e.g. Smolander et al. 1991, Wijers 1991). For example, Smolander et al. (1991) link the conceptual content of a method into a graphical, diagram-oriented representation. In Kelly (1994) this approach is extended into a matrix representation. As a result, these method definitions can be applied at the ISD level as a modeling technique. The metamodels representing modeling techniques are often characterized as meta-data models (Brinkkemper 1990). This thesis investigates notation-based ME, and therefore we describe in Section 3.3 only those metamodels and metamodeling languages which support a notation-based approach.

ME approaches that concentrate on the processes of method use are less developed than those that concentrate on the conceptual structure and notations of methods[12] (Tolvanen et al. 1996). They can be divided into those that specify the relationships among the modeling tasks (e.g. Wijers 1991, Hofstede and Nieuwland 1993, Rolland and Prakash 1996, Jarke et al. 1994) and those that emphasize the specification of modeling products and the tasks needed to make them change (e.g. Marttiin 1994). The former can be used to describe, for example, when and how inheritance structures are identified, and the latter also in which technique it is represented. These processes are represented in process or meta-activity models (Brinkkemper 1990). A process model is always related to the conceptual structure of a method, but it can also be related to the notation based metamodels, as in the latter example. The manipulation of models is always dictated by tasks and decisions, whether or not these are defined in an explicit process model. In this thesis, we do not consider the process-based approaches to modeling techniques. The strategies of integrating a meta-data model and a process model are discussed in Wijers (1991), Marttiin et al. (1995), Tolvanen et al. (1993), and Kinnunen and Leppänen (1994). Classifications of process models can be found from Dowson (1987) and surveys from Curtis et al. (1992) and Finkelstein et al. (1994).

Other types of method knowledge are more poorly addressed in the ME literature. One reason is the absence of such descriptions in the method books. At the level of ISD participation, ME approaches define the stakeholders involved and the organizational structures related to method use. The metamodel of Tolvanen et al. (1993) includes agent models which specify the activities performed and the agents involved. The Method Engineering Language (MEL) of Harmsen (1997) allows the entry of different roles, such as responsibility, for method components. Therefore, a participation model of ISD defines, for example, who is responsible for finding and creating inheritance hierarchies with class diagrams.

At the level of modeling decisions a variety of design rationale approaches are proposed. These aim to record the design decisions made based on a predefined schema (Ramesh and Edwards 1993). Originally the decision-oriented models focused on decisions behind designs not behind methods, e.g. why an inheritance between two classes is defined as virtual. Design rationale can be also modeled in two other ways which are beneficial to ME: decisions related to method use and decisions related to method construction. An example of the former could be an IS developer's justification why a concept of virtuality is used in inheritance structures. Approaches of this type focus on decisions related to the ISD process. The proposal of Rolland et al. (1995) focuses on the specification of successive transformations of the modeling product, from the viewpoint of the consequences of decisions. Jarke et al. (1994)

[12] Although numerous process models have been proposed for software engineering (cf. Armense et al. 1993), some of which can be used in principle to specify method-related processes, we restrict our attention here to those explicitly developed for ME.

propose a traceability model for tracing processes defined in a guidance model. An example of the latter would be a method engineer's justification of why a metamodel includes a concept of inheritance. Oinas-Kukkonen (1996) proposes metamodel-related rationale, but does not explain its use in more detail. For example, to which types of method knowledge should the decisions on method use be related? Should it be used for recording why a certain concept was used, why it was represented in a specific diagram, or why it was specified in a certain phase of ISD, etc.

Finally, to our knowledge only Kumar and Welke (1984, 1992) have directly focused in the ME literature on the development objectives and values underlying methods. They proposed an ISD-Personal Value Questionnaire (ISD-PVQ), consisting of 86 value concepts, with which a method stakeholder's values can be collected and their relevance can be assessed within three different groups, addressing technical values, such as timeliness of information; economical values, such as ISD costs; or socio-political-psychological values, such as system responsiveness to people.

Although all types of method knowledge can be modeled with ME approaches, it does not mean that all types of method knowledge are modeled fully. First, there are many activities like brainstorming sessions, meetings, etc. which are not even considered to be modeled with current ME approaches. Second, part of the knowledge applied in carrying out ISD is tacit and therefore not expressible. As a consequence, aspects of method knowledge which can not be made explicit can not be improved through method engineering principles, nor supported by ISD tools. Finally, it must also be noticed that there are dependencies between different types of method knowledge (cf. Figure 2-2). Therefore, in local method development it is not meaningful to focus solely on one or a few types of method knowledge. For example, the balancing rules (Yourdon 1989a) necessitate that a notation-related metamodel can recognize mappings between data stores and entities, although these mappings do not have a notational counterpart. Hence, it must be noticed that metamodeling is difficult, if not impossible, to perform meaningfully without considering other types of method knowledge.

3.2.3 Criteria for constructing methods

ME approaches must be also characterized according to the driving factors or criteria used to engineer methods: it is not useful to perceive and model method knowledge if you do not know how it should be analyzed, constructed and maintained. Accordingly, most important to the success of ME is not how well a method can be represented, but how the applicability of a method is improved.

In the following, ME approaches are analyzed based on which kind of criteria they apply to achieve the methodical requirements of ISD. In principle, these approaches fall into three categories, namely those describing the method use environments, also known as contingency frameworks; those emphasizing the importance of problems at hand to be solved by the method; and those focusing directly on method users' requirements. These approaches are summarized in Table 3-2.

TABLE 3-2 Method engineering approaches and criteria.

Criteria	Method engineering approaches
Contingency based	Heym 1993, Vlasblom et al. 1995, Hoef and Harmsen 1995, Punter and Lemmen 1996, van Slooten and Hodes 1996, Harmsen 1997, Brinkkemper 1996
Problem based	Jarke et al. 1994, Punter and Lemmen 1996
Stakeholder value based	Kumar and Welke 1984, 1992

It must be noticed that the three categories are not necessarily the same approaches as those addressing types of method knowledge. Many of the approaches discussed earlier focus mostly on metamodeling but do not explicitly describe what should be done with the metamodels. Compared to research on metamodeling languages, the criteria for engineering methods seem to be less studied. It must also be noticed that they can overlap. For example, a contingency framework can also include some criteria related to different type of problem situations or aspects of the stakeholders' values. Moreover, the same criteria can be recognized with more than one type of method engineering approach.

Each approach focuses on different aspects of ME, and they are therefore limited to some extent. Below, we discuss their weaknesses and strengths as principles for carrying out ME.

3.2.3.1 Criteria based on contingencies

The majority of ME approaches apply contingency frameworks (cf. Section 1.4.2) to characterize an ISD environment and to find situational requirements for methods. This characterization is performed through an analysis of the project's context (Slooten and Hodes 1996), its environment (Harmsen 1997, Harmsen et al. 1994b) or the profile of the situation (Vlasblom et al. 1995). The content and objective of these approaches, however, are the same. The applicability of a method is understood as the closest possible relationship between the characteristics of the ISD situation and the characteristics of the method.

The approaches analyzed use either an external contingency framework (e.g. van Slooten and Hodes 1996), or relate some situation characteristics directly to metamodels (Heym 1993, Heym and Österle 1991, Harmsen 1997). An example of the former is the work of Punter and Lemmen (1996), who aim to apply a contingency checklist to define project strategies and method objectives. An example of the latter is Heym and Österle's (1992) work where they collect characteristics of a method into a metamodel based on fixed classification schema. This classification includes parts focusing on the method (e.g. project management, risk management, system development), application type (e.g. expert, office or real-time system), and life-cycle of ISD (e.g. analysis, maintenance). The advantage of relating contingency factors to the metamodels is the close relation between factors and method components, whereas the external contingency lists are often not explicitly related to method knowledge.

The advantages of using contingency frameworks are obvious. They provide a high-level view of methods, and by covering several characteristics they help identify characteristics of methods which might not otherwise be noticed at all. Also, research has identified a wide variety of different ISD characteristics. However, the use of the contingency approach in great detail is costly as it requires a large amount of resources and skills to manage different methods and situational characteristics (Avison 1996, Kumar and Welke 1992). Its effective use is difficult, because it is almost impossible to classify, or to identify, relevant contingencies beforehand. Moreover, and general to all method development efforts, a combinations of various methods might be impossible because of different and possible conflicting underlying philosophies (Avison 1996).

To cope with the cost part, Punter and Lemmen (1996) aim to provide a ready-made contingency list. However, such a list operates at the general level of method knowledge, and does not allow other method choices than those already prescribed. Examples of the use of contingency frameworks deal with method knowledge in general, such as how much analytic modeling should be used instead of prototyping, or whether methods should be customized (e.g. Slooten and Hodes 1996). As a result, they can not be applied effectively in the detailed construction of methods. For example, it is unclear how factors like the stability of goals (in Slooten and Hodes 1996) or the amount of resistance can be related to the construction or even selection of modeling techniques. Similarly, most of the contingency-based ME criteria do not identify method knowledge in detail. As an example, the concept of inheritance is not addressed in any of the frameworks, although some of the mappings are relatively straightforward; such as single inheritance in class diagrams when a specific programming language is used. Because method knowledge is not addressed in detail, construction and maintenance of methods is not possible with the contingency frameworks discussed above.

Similarly, the cases related to detailed ME, such as customization of techniques or tasks of the ISD process, discussed in Section 2.5.4 (e.g. (Russo et al. 1996, Jaaksi 1997, Aalto 1993, Nissen et al. 1996, Kurki 1996, Tollow 1996, Cronholm and Goldkuhl 1994) or field studies about ME (Smolander et al. 1990) reveal that contingency frameworks for method selection are not used. Similarly, the reported cases of ME (cf. Section 2.5.4) did not apply any contingency frameworks. This does not mean that contingencies do not describe characteristics of applicable methods. Instead it indicates how difficult they are to use in the detailed engineering of methods. Finally, and maybe most importantly, it should be noticed that none of the contingency criteria have been validated for the task they have been proposed for. Although some of them have been recognized to be relevant in past projects (e.g. Slooten and Hodes 1996), their usability in method construction has not been clarified. As a consequence, we do not know whether they are relevant for method construction, which deals with detailed method knowledge.

3.2.3.2 Criteria based on problem solving

An alternative and more detailed approach is to seek applicable methods or their parts based on their description and problem solving capabilities. This approach is closely related to the method knowledge behind modeling techniques: how can relevant aspects of the object system be adequately described with modeling techniques, and how they allow us to find alternative solutions. This approach is more widely applied in practice (e.g. Jaaksi 1997, Tollow 1996) because it is simpler and does not require as much knowledge and resources to carry out as the contingency approach. As emphasized by Jaaksi (1997, p 76), ME efforts take place under financial pressure to solve practical problems. Accordingly, work done by other method developers is considered as a contribution to the method only when it can be expected to produce a significantly better solution (Jaaksi 1997). As March and Simon (1958) have recognized, organizations tend to find satisfactory rather than optimal solutions. The significance of a method solution is evaluated based on the problems at hand, not only by characterizing contingency factors and their changes.

The use of problem-driven ME criteria in practice does not mean that it is better. It has, however, some other advantages in addition to those mentioned. First, since it is problem-focused and derives method requirements from the current case, it is not so idealistic as the contingency-based approach. Second, it is more open to new concepts of methods, because it is not restricted to applying existing method-related situation characterizations. For example, in Tollow (1996) difficulties in developer and end-user communication led to the development of more readable and understandable notations. The resulting modeling techniques were not available in other methods and thus had new concepts and rules. Third, it is more open to the requirements of detailed method knowledge. For example, in Jaaksi (1997) the interest in the modality of dialogs of the user interface was added to the dialogue diagram. This modeling technique was constructed by using the notation of state diagrams with added new concepts, e.g. to describe which of the windows is the main window, which are modeless, and what kind of tasks the user performs with the windows.

Problem-driven ME also has shortcomings. It focuses only on problems identified at the moment and provides little generalization possibilities through frameworks. As with contingencies, the formulation of a problem-driven method construction framework is difficult because of their generality and loose connections to detailed method knowledge. For example, the framework of Essink (1988), used by Punter and Lemmen (1996) in their MEMA model, characterizes the problem domain according to four levels of abstraction (i.e. object system, conceptual IS, data system and implementation) and eight aspects (e.g. goals, dynamics, process structures). Methods are then allocated to the same MEMA model for selection and construction. This approach would, for example, locate ER diagrams and class diagrams under the same problem solving situation. Thus, no distinction can be made between these modeling techniques, or between different dialects of them. Instead, an inductive approach has been proposed (Jarke et al. 1994): a framework should be developed from expected problems and applied to method refinements. This aspect is analyzed in more detail in Chapter 5.

3.2.3.3 Criteria based on stakeholders' values

Kumar and Welke (1984, 1992) address the importance of stakeholders' values or design ideals as requirements of methods. A major difference and strength of their ISD-PVQ technique is the emphasize on participation in ME: the users' requirements are the most important aspect of ISs and therefore also of ISD environments. Thus, the applicability of a method is considered according to how well it supports the method stakeholders' (developers, end-users, managers, etc.) values and expectations of ISD. Simply, the methods developed are more easily accepted if they satisfy the requirements of the method users.

ISD-PQV has also been applied in practice (Kumar and Welke 1984), revealing the domination of technical and economical aspects of ISD at the expense of other values. The need to change the focus of methods from technical aspects of ISs is also noted by several other researchers (e.g. Lyytinen 1986, Avison 1996). The traditional perspective has been to see an IS as a technical innovation and focus less on behavioral and social consequences (Lyytinen 1986). The implication for ME is twofold: on the one hand, methods should include participative and social components to compensate for the bias towards technical and economic issues. Thus, stakeholder-driven ME approaches could be used to move the focus of the ISD group towards other values and design ideals. On the other hand, the current values of stakeholders are also major reasons for the dominance of "hard" valued methods.

3.2.4 Implementation into ISD tools

The steps of ME can be supported by CAME tools. This means that the deliverables, metamodels and ME criteria, can be stored in and retrieved from the CAME repository, and methods can be compared, versioned, and adapted into a CASE tool. The most studied tool functions have been capturing method knowledge (cf. Heym and Österle 1993 , Harmsen et al. 1994a, Verhoef et al. 1991), and building generic CASE toolkits which can be customized for different methods (cf. Teichroew et al. 1980, Chen et al. 1989, Sorenson et al. 1988, Bergsten et al. 1989, Smolander et al. 1991, Rossi 1995, Grundy and Venable 1996, Kelly et al. 1996). The latter type of tools also interest us since we believe that the results of ME efforts should be applied in ISD as a situational method. Because of this focus, we view both the method and the supporting tool as an end-product of the ME process.

The ISD tools can be further divided into two broad categories based on how they model the object systems (Lyytinen 1987). These categories include data-oriented and process-oriented approaches. In ME a similar division can be also observed: there is a variety of metalanguages and CAME tools that model the methods and support the storage of IS models made according to method definitions (Sorenson et al. 1988, Smolander 1991, etc.). In the process camp — with fewer representatives than the data-oriented camp — research has focused on process representations and tools which support the enactment of defined processes (Hofstede, et al. 1993, Wijers 1991, Hidding et al. 1993, Pohl 1996). Since our focus is on the conceptual structure behind modeling techniques we focus on data-oriented CAME tools.

In practice CAME technology is quite new. This can be observed from CAME research which focuses mostly on tool building rather than investigating their usefulness or usability (Tolvanen et al. 1996). For example, to our knowledge only Marttiin et al. (1993) have analyzed metaCASE tools more systematically in terms of their capabilities for establishing method-tool companionship. The earliest pioneer SEM, (Teichroew et al. 1980) was introduced only at the beginning of the 1980's. Most current CAME tools are outcomes of research projects, including MetaView (Sorenson et al. 1988), MetaEdit (Smolander et al. 1990a, MetaCase 1994), RAMATIC (Bergsten et al. 1989), Quickspec (Meta Systems 1989), MetaPlex (Chen 1988), IPSYS Toolbuilder (Alderson 1991), and MetaEdit+ (Kelly et al. 1996). During the last years commercial CAME tools, such as IPSYS Toolbuilder, MetaEdit+ (MetaCase 1996a, 1996b) and Paradigm+ have also begun to appear in the market. Commercial CAME tools are called metaCASE tools[13] and we apply this term because of its wider use. Marttiin et al. (1996) present a framework for comparing and evaluating CAME functionality. Isazadeh and Lamb (1997) and Kelly (1997) review and make partial comparisons of sets of CAME and metaCASE tools.

MetaCASE tools use a set of primitives, which allow them to describe a given method quickly and provide a set of mechanisms to implement tool support for the modeled method. To establish method-tool companionship (cf. Section 2.3.2) the method must be described using the metamodeling language the tool applies. Ideally the metamodeling language used in method construction is the same as that required by the tool, but often the tool-related metamodeling language limits the adaptation (Cronholm and Goldkuhl 1994), or other less formal metamodeling languages are used during the construction and design phase. The result of a ME process is a customized CASE tool which can assist ISD. The customized CASE tool is expected to produce, through its support for the situation-specific method, positive effects on the resulting IS or on the process of its development. In this sense metaCASE tools provide a new approach to establish symbiosis between methods and tools, and offer more degrees of freedom in method and tool selection (Tolvanen and Lyytinen 1993).

The CASE tools developed should not be viewed only as tools for making abstractions. They should also include other functionality which are affected by the method: checking, form conversion and review (cf. Section 2.3.2) are all design steps which need to be taken into account during adaptation. First, checking of the models is always dictated by the underlying metamodel. Because some rules of the method knowledge can not be guaranteed or even checked at modeling time, but only after models are made, the tool adaptation also includes the implementation of consistency checking reports. Second, form

[13] The terms CASE shell (Bubenko 1988) and metasystem (Sorenson et al. 1988) have also been used. These terms usually refer only to functions which allow the implementation of CASE tool support for a selected method.

conversions between different IS models (Fraser et al. 1991) or to programming languages are driven by the underlying metamodel. In fact, requirements to change metamodels often occur because of the demands to generate certain programming code or analysis reports (cf. Section 5.1.2). Third, review of ISD deliverables with end-users is largely carried out via the IS representations (i.e. notations). This may require the use of less formal notations, or simplified versions of the modeling techniques applied by IS developers (e.g. Tollow 1996).

Finally, an additional advantage of using metamodel-based tools is that we can apply them to collect information on method use. In other words, using metamodels we can examine in a systematic and rigorous fashion how developers perceive the IS, in what notation the system is described, and how the models are checked. For example, in relation to experience gathering the metaCASE tools can be used to find which method knowledge is used or not used during ISD. This aspect is discussed in more detail in Chapter 5.

3.2.5 Summary and discussion

The ME approaches described are proposed for representing various aspects of method knowledge and for constructing this knowledge to meet different kinds of situational requirements. The survey reveals a mechanistic view of ME approaches: ME aims to develop methods by specifying and constructing them like machines, and little attention, at least in the published ME literature, is given to the introduction and use of methods. The approaches analyzed (cf. Tables 3-1 and 3-2) also have a narrow view of the ME process and examine method knowledge only at a coarse granularity. Regarding the types of method knowledge at the metalevel (i.e. knowledge behind the methods of ME), most research has only focused on metamodeling languages and conceptual structures, and little effort has been expended on other domains, such as what is the ME process in greater detail or what decision and criteria are relevant to ME. Other more specific limitations of the ME approaches are discussed below.

First and foremost, almost all approaches assume *a priori* construction of methods. Although some of the metamethods (e.g. Brinkkemper 1996, Punter and Lemmen 1996, Harmsen 1997) acknowledge the importance of experiences, they do not propose principles for identifying, collecting, and analyzing experience-based method knowledge. If learning from method use is ignored, methods can not be maintained or redefined based on experiences. This shows that the approaches assume either explicitly or implicitly that contingencies and problems are known beforehand and they are stable during the use of the method constructed. Any changes after method construction, e.g. during tool adaptation, method introduction, or method use, are not incorporated into the methods. Although some of the ME approaches acknowledge the changes, they do not include any steps or provide any mechanisms for refining methods. These approaches do not fit with our view of method knowledge as evolutionary (cf. Section 2.5.4): methods have evolved and changed in general and in organizations, and there is no reason to expect that they would not evolve in future. To our knowledge, only Jarke et al. (1994) focus on analyzing method use as a part of ME. They propose a traceability model to record process-related experiences. Yet their *a posteriori* ME approach does not consider the refinement of other types of method knowledge based on such feedback.

Second, most of the ME approaches are biased towards the selection of ready-made method components (or fragments). Their construction phases mostly consist of composing existing method knowledge, and method choices other than those already available are not considered. As a result, the ME approaches expect that someone has already proved a method or its component, and it is known to be applicable in specific ISD contingencies or problem solving situations. This is paradoxical, since there has been little research evaluating methods according to criteria used in method construction.

Third, the criteria (i.e. contingencies, problem characteristics, or values) used in the ME approaches are far too general to direct detailed method construction. Because of this, the proposed situation characterizations do not support detailed analysis, construction, and refinement of method knowledge. At best, the characterizations can be used to "prefer" a certain collection of techniques and methods. For example, the approaches do not distinguish techniques of object-oriented methods from techniques of structured methods. Although these general driving factors of ME are important in understanding and structuring method knowledge, the examples of local method development show that methods are developed at a far more detailed level. Similarly, the studies on individual designers' understanding and use of methods indicate that method knowledge is different at the detailed level (Wijers 1991): for example, method knowledge is applied differently even at the level of single concepts of a modeling technique.

To summarize, none of the ME frameworks provide explicit principles for collecting and analyzing methods *a posteriori* and therefore do not explain how method refinements can be carried out. In this sense, they aim to deliver a method in terms of a constructed method, while little attention is paid to analyzing how the method is used and whether it has been successful.

3.3 Metamodels and metamodeling languages

This section discusses metamodels and metamodeling languages as used in this thesis to describe tool-supported modeling techniques. This discussion is important because all ME is based on some formalism and because it deals with our research question on modeling method knowledge. Accordingly, in the next subsections we shall define metamodels and metamodeling. This is followed by a representation of different types of metamodels, and especially metamodels which are based on semantic data models.

3.3.1 Defining metamodeling and metamodels

Models play a crucial role in ME, as in all engineering. Only those aspects of a method can be engineered which can be made explicit through a representation. Modeling of methods is not important only in constructing methods, but has also proven to have advantages in systematizing and formalizing weakly defined methods (Tolvanen and Lyytinen 1993), providing a more "objective" approach to comparing methods (Hong et al. 1993, Rossi and Brinkkemper 1996), supporting standardization efforts (e.g. Booch et al. 1997, OMG 1997), and examining linkages between ISD methods and programming languages (Hillegersberg 1997). Metamodeling is also successfully used in building flexible modeling tools (Kelly 1997, Kelly and Smolander 1996), interfaces between tools (CDIF 1997), and repository definitions (CASE Outlook 1989). Metamodels can differ greatly based on their purpose and the type of method knowledge considered. For example, Brodie (1984) analyzed various semantic data models and showed that there is a need for application-specific data models. Similarly, the ME frameworks and underlying metamodels focus on different types of method knowledge (cf. Section 3.2.2).

In its simplest form we can say that a *metamodel* is a conceptual model of an ISD method (Brinkkemper 1990). Metamodels can be further divided into different types depending on what type of method knowledge is modeled. Hereafter, we use the term metamodel to refer to a meta-data model which describes the static aspects of a method. Consequently, *metamodeling* can be defined as a modeling process which takes place one level of abstraction and logic higher than the standard modeling process (van Gigch 1991). The relationships between modeling and metamodeling are illustrated in Figure 3-3.

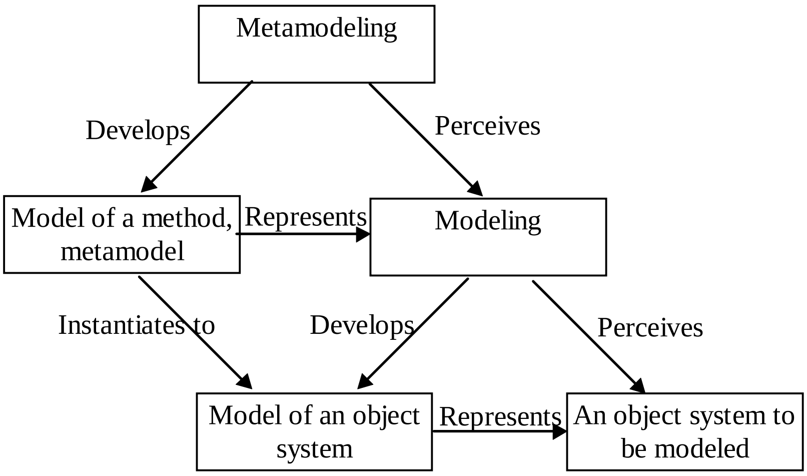


FIGURE 3-3 Metamodeling and modeling (after Brinkkemper 1990).

In metamodeling, the focus is on method knowledge applied in modeling. In the case of meta-data modeling this means the conceptual structure and notation of the method. Accordingly, the resulting metamodel captures information about the concepts, constraints, rules and representation forms used in modeling techniques. IS developers use this knowledge — although often unconsciously — in IS modeling tasks (see Smolander et al. 1990). Clearly, no modeling is possible without some sort of (explicit or implicit) metamodel. The same is also true for metamodeling as it also uses its own methods and tools which, in turn, can be described one level higher in metametamodels (and so ad infinitum).

Kotteman and Konsynski (1984) show that at least four levels of instantiation are necessary to integrate the modeling of the usage and evolution of ISs. A similar observation underlies the architecture of the ISO IRDS (Information Resources Dictionary Standard, (ISO 1990)), and in the universal framework for information activities by Auramäki et al. (1987). The levels and their hierarchy are illustrated in Figure 3-4.

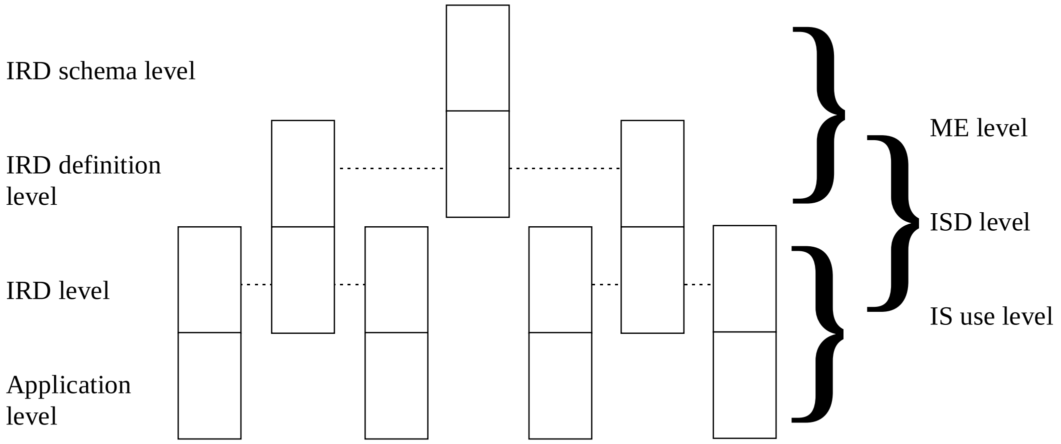


FIGURE 3-4 ISO IRDS repository framework.

The application level includes application data and program execution. An example of the former could be "Juha -Pekka Tolvanen" and an example of the latter the procedure by which this data has been added or removed in the application. This level corresponds to the instances of class-based languages and to instantiations of an IS model.

The IRD level includes database schemata and application programs, plus any intermediate specifications, and also specifications of non-computerized activities (e.g. business processes and work flows). This corresponds to the class level of class-based languages and instantiations of a metamodel (i.e. IS models). An example of the information at this level would be a definition of "customer" information as part of the database schema.

The IRD definition level specifies the languages in which schemata, application programs, and specifications are expressed. It may also contain the specification of possible static and dynamic inter-relationships between these languages, for instance how various design models are linked. This corresponds to the metaclass level of languages such as Smalltalk, and instantiations of a metametamodel (i.e. metamodels). An example of the information at this level would be the specifications of the ER diagram technique and its component types, such as "entity" or "cardinality".

Finally, the IRD Definition Schema Level specifies a metametamodel according to which the IRD Definition level objects can be described and interlinked. An example would be ‘concept’ (Wijers 1991) or OPRR’s ‘Object’ metatype (Smolander 1992).

As the right side of the figure illustrates, these four levels can be grouped into interlocking level pairs. The interlocking is necessary since making sense of instances is not possible without type level information. To illustrate the interlocking pairs in Figure 3-4 the boxes are joined, and to represent that the number of instances is normally greater than the number of types the figure is in the shape of a pyramid. Thus, the hierarchy can be understood as being instantiations in which the higher level forms the type definitions for the lower level instances, in the same way as classes define objects, and metaclasses define classes. A level pair can also be intuitively understood as a database where the upper level is the schema and the lower level the database state.

The lowest pair, IS use, corresponds to application databases, consisting of a schema and of a database state used in daily business. At the database state level the data element ”Juha -Pekka Tolvanen” is useless if its type information is not known (e.g. row in a customer table of a database). The middle pair, ISD, corresponds to data dictionaries or CASE tool repositories used to store models of IS. IS modeling tools also operate at this level. For example, an ER diagram describes a ”customer” as an entity. The topmost pair, ME , corresponds to meta databases, such as metaCASE tools or CAME tools which store models of methods, i.e. metamodels. Here the metamodeling language plays the role of type information and modeling technique metamodels are viewed as instances. For example, an ”entity” is described as an object type, an instance of the metatype ‘Object’, in an ER diagram metamodel. On the ISD level, types included in a metamodel determine what one can observe or describe about the application level while using the method.

Our research questions focus on the topmost pair of the IRDS framework. The first question on metamodeling constructs seeks to find applicable concepts and constraints for metamodeling languages, i.e. to define metametamodels. In other words, metametamodels provide constructs for metamodeling languages. The second research question on incremental ME deals with how the applicable instances of the ME level, i.e. metamodels can be recognized and constructed for ISD. Hence, the research itself can be placed above the IRD schema level, i.e. on a fifth level.

Although we mostly operate with concepts of the ME level we sometimes need to refer also to instances on the ISD level. To distinguish the concepts and the level on which we currently operate we use the following naming conventions. On the IRD schema level, we use the term metatype to denote any of the concepts used in a metamodeling language. On the IRD definition level, one of these metatypes is instantiated to describe a certain method component, resulting in what we call a type (i.e. an instance in a metamodel). Hence, in ME, an entity object type refers to an instance in a metamodel. This then itself plays the role of type on the ISD level, when it is instantiated to an entity used as an element in a model of the object IS.

Since some of the metamodeling constructs refer to instances of models, i.e. to the IRD level we need to refer to them also. For example, to refer to all entities described in an ER model we use the term instances of an entity object type. Because the naming becomes complicated we use apostrophes to refer to things on the IRD definition level. Hence 'entity' means the type described in a metamodel. This means the same as an entity object type although the latter is more precise, since it mentions also the metatype. Because in most cases the metatype is clear, e.g. it is clear that 'class' is an object type not a relationship type, we normally use the shorter naming version.

3.3.2 Types of the meta-data modeling languages examined

Starting from Teichroew et al. (1980), most of the meta-data modeling languages rely on some existing semantic data model (Hull and King 1987). Two types of semantic models, ER-based models and NIAM-based models, have been investigated in particular. Extensions of the ER model (Sorenson et al. 1988, Welke 1988, Smolander 1991, Venable 1993) seek to improve its expressive power by suggesting new integrity constraints (Welke 1988, Smolander 1991, Kelly and Tahvanainen 1994) and verification rules (Wijers 1991), and by representing complex objects (Venable 1993). The NIAM-based conceptual metamodeling formalisms (Bommel et al. 1991, Hofstede et al. 1993, Hofstede and Weide 1993) pursue similar goals, but are often founded on a more formal basis than the ER-based modeling languages.

The reasons for applying semantic data models in ME are the same as in ISD. They are easy to use, support communication, and yet are powerful and formal enough to describe methods and implement them in customizable ISD tools. Moreover, methods in CASE tools are largely based on semantic data models and thus their users are familiar with them. Some models, like ER or NIAM, are even applied at both levels. Because of the symmetry between models in ME and ISD the semantic data model based metamodels should be easy to use and understand for method users. This is especially relevant for incremental ME in which metamodels are constantly used to refine ISD methods. The requirement for ease of use is further highlighted if the users of the modeling tool or their customizers are not familiar with a specific programming language.

Second, support for communication is especially important when multiple method stakeholders need to agree on and participate in ME. Also, metamodels can be applied for teaching (Mathiassen et al. 1996) and method related helps can be generated from metamodels (e.g. MetaCase 1994). Support for easy use and communication are achieved by the use of graphical representation in metamodeling languages. Although some metamodeling languages seek to model methods totally with graphical constructs (e.g. CoCoA (Venable 1993)), in practice they all include some graphically "invisible" metamodeling constructs. In other words, not all constructs of the metamodeling language have notational support. Most of the approaches aim to add extra constraints (Smolander 1992, Ebert et al. 1996), or whole constraint languages (e.g. ter Hofstede 1993) to existing graphical metamodeling languages. Another approach is to apply graphical metamodeling languages to describe only a subset of the metamodeling constraints (e.g. Harmsen 1997).

From the method user point of view, easiness and support for communication could be better achieved with a natural language, rich pictures, or with other similar techniques, but they would not satisfy the third requirement of formality. Moreover, these do not reflect any knowledge specific to ME and metamethods: they do not explicitly describe or implicitly provide guidance on which components, rules, or constraints of methods should be considered during ME. In addition to pure representation of methods, metamodeling languages should provide a formal basis for building tool support. This means that all essential method knowledge must be captured into the formal metamodel.

The modeling power and formality of metamodeling can also be supported by other types of metamodeling languages. For example, for this purpose Saeki and Wenyin (1994) adapted an object-oriented modeling language called Object-Z. Ahituv (1987) introduces a formal metamodel which views an information system as the data flow that moves from one state to another, and by which some existing methods can be modeled. The work of (Oei et al. 1992, Oei and Falkenberg 1994, Oei 1995) introduces a formal language for modeling methods and transforming them into a method hierarchy. Also set-theoretical constructs (Bergsten et al. 1989), and predicate-logic (Brinkkemper 1996, Harmsen 1997) have been applied to metamodeling. However, these fail in other criteria as they neither support communication nor are they as easy to use as semantic data models (although some of them like MEL have close connections with the data modeling side). Semantic data models provide better modularity and maintainability, which are particularly important for incremental ME. Moreover, the development of metamodeling languages mainly to satisfy the modeling power aspect is questionable because this requirement can already be supported with programming languages. Hence, if we concentrate only on achieving the greatest possible modeling power, assembler or C++ is close to the ultimate metamodeling language.

The requirements formulated above are important for incremental ME and direct us to focus on semantic data models. Additional reasons for this focus are the popularity of semantic data models as repository schemas (CASE Outlook 1989), and the dominance of their use in large metamodeling efforts (Hong et al. 1993, Heym 1993, Henderson-Sellers and Bulthuis 1996a, 1996b, Hillegersberg 1997) when compared to other types of metamodeling languages. The former reason allows us to test and validate the metamodels in a tool environment (cf. Section 4.2.3), and the latter reason to compare the metamodels.

3.3.3 Modeling power of meta-data models

Like other modeling languages, metamodeling languages focus on specific aspects of the domain to be modeled, and therefore lead to different types of representations. One major difference between these languages is therefore how well they describe various types of method knowledge. This is related to our first research question, since our aim is to improve the metamodeling power of semantic data models. Ideally, a metamodeling language should capture method knowledge as completely as possible (Griethuysen 1982, Welke 1988, Brinkkemper 1990, Tolvanen et al. 1993). By complete we mean the 100% principle suggested by Griethuysen (1982) in the context of metamodeling, and Welke's (1988) "no loss"criterion in the context of a repository metamodel.

In the following we consider a small example to illustrate the use of different semantic data models in metamodeling. This allows us to describe different approaches to define method knowledge and to introduce them for later evaluation (cf. Section 4.5). We focus mostly on widely-known metamodeling languages. These are summarized in Table 3-3 and illustrated in the following subsections together with an example. The metamodeling languages mostly follow ER-based models, except NIAM which can be considered as an object-relationship model (Kim and March 1995), and MEL which is based on first-order predicate logic, but uses an ER-based graphical notation.

TABLE 3-3 Examples of metamodeling languages.

Acronym	Metamodeling language name	References
ASDM	ASDM	Heym and Österle 1992, Heym 1993
CoCoA	ComplexCoveringAggregation	Venable 1993, Grundy and Venable 1996
ER	Entity-Relationship model	Chen 1976
GOPRR	Graph-Object-Property-Relationship-Role model	Marttiin et al. 1995, Kelly et al. 1996
MEL/MDM	Method Engineering Language	Harmsen 1997
NIAM	Nijssen's Information Analysis Method	Nijssen and Halpin 1989, Hofstede 1993
OPRR	Object-Property-Relationship-Role model	Welke 1988, Smolander 1992

The metamodeling example is based on a small piece of method knowledge as follows:

> In object-oriented design the life-cycle of class instances must be specified with one or more state models. A state model contains states and transitions between two states. A state must be specified by a name and a class may have only one state with a given name. Each transition must be specified with an action which is executed when a transition occurs. An action is specified as an operation of a class.

This example deals mainly with knowledge related to a single modeling technique, but also includes a connection to class diagrams. The example is quite common in object-oriented methods (e.g. Coad and Yourdon 1991a, Rumbaugh et al. 1991) but is made more explicit than is often possible to find from method text-books.

The metamodels made were reviewed by the users or developers of metamodeling languages, except the metamodel made with ASDM and MEL. Users of these metamodeling languages were not available, and their developers did not respond to our inquiries.

3.3.3.1 Entity-Relationship model

The ER model has been commonly applied as a schema for repositories (CASE Outlook 1989), and most of the meta-data modeling languages originate from the ER model. Therefore, several versions and dialects of ER modeling exist. They vary based on whether attributes are allowed only for entities, whether inheritance of entities is allowed, etc. (cf. Batani et al. 1992). Here we apply a version of the ER model which allows the definition of attributes attached to entities, and recognize cardinality constraints between entities (in an ER model this constraint defines how many times instances of an entity can participate in a relationship).

The reasons for extending the ER model for metamodeling are the same as in extending it for IS modeling — its limited modeling power. Figure 3-5 illustrates limitations of the ER model with two versions of our state model example.

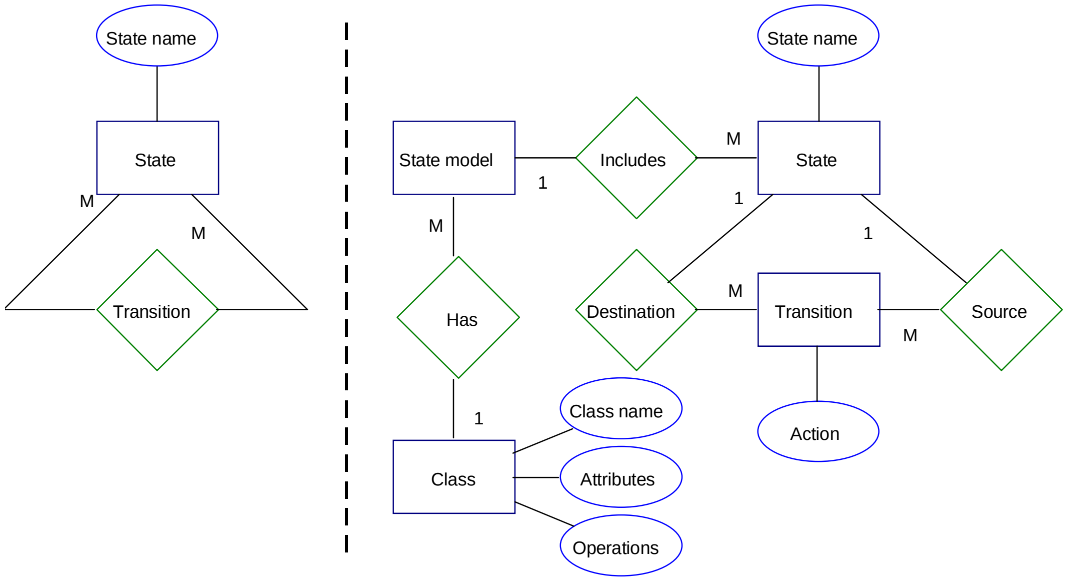


FIGURE 3-5 Two metamodels of state model defined with the ER model.

In the figure, entities are illustrated with a rectangle, relationships with a diamond, and attributes with an ellipse. The cardinality constraints (1 or M in the metamodel) are shown side-by-side with the related entity. The metamodel on the left defines that each state can participate in several transition relationships, and that states have state names. No information is given for example on the state model itself, nor that transitions must be specified with an action.

To specify all rules in the example, the metamodel could be defined differently. For example, to specify that transitions are defined with an action, a transition could be defined as an entity instead of a relationship. The larger metamodel on the right illustrates this possibility. The cardinality constraint value one defines furthermore, that a transition can not be an n-ary relationship. However, when an entity type is used to represent a transition no distinction can be made between design elements that can exist independently (i.e. states) and design elements which exist between independent design elements (i.e. transitions). Thus, in the latter version of the metamodel, there is no explicit constraint (cf. Brodie 1984) to distinguish between transitions and states. Moreover, the metamodel does not specify method knowledge adequately because it allows transitions which are "unconnected" to states. This distinction between states and transitions could be made with the first version of the metamodel since in the ER model relationships can exist only when related to entities. This constraint is thus inherent in the ER model (a basic semantic property of the ER model, Brodie 1984). Moreover, other method knowledge related to state modeling with object-oriented methods is not defined adequately in either of the versions. For example, mandatory action names, unique state names, and the requirement that every class must be specified with state models are not captured with the ER model.

3.3.3.2 ASDM and the reference model of information system development

ASDM is a semantic data model developed at the University of St. Gallen and it has been used to describe the reference model of ISD (Heym and Österle 1992, Heym 1993). ASDM has also been used for metamodeling according to the rules defined in the reference model (Heym 1993). The reference model includes the widest range of method knowledge as it aims to cover other knowledge in addition to that defined in modeling techniques (part of deliverable model of the reference model). These extensions, excluded here, deal with the ISD process, versioning of metamodels, guidelines for integrity, and method related contingencies. Each of these is also represented with different metamodeling languages. Based on the reference model another, existing, metaCASE tool was used to develop a tool called MERET (Heym and Österle 1993), which could represent and compare methods, but not implement them into a CASE tool.

ASDM is used as a notation to metamodel modeling techniques (cf. Heym and Österle 1992). The notation of the language follows the ER model and initially Chen's (1976) version of ER model was used to define methods. The extensions in ASDM deal with inheritance, aggregation and identifying different concepts in the modeling techniques. These concepts are subtyped from an entity type. Because of these extensions, our state model example can be specified more adequately than with the standard ER model (cf. Figure 3-6).

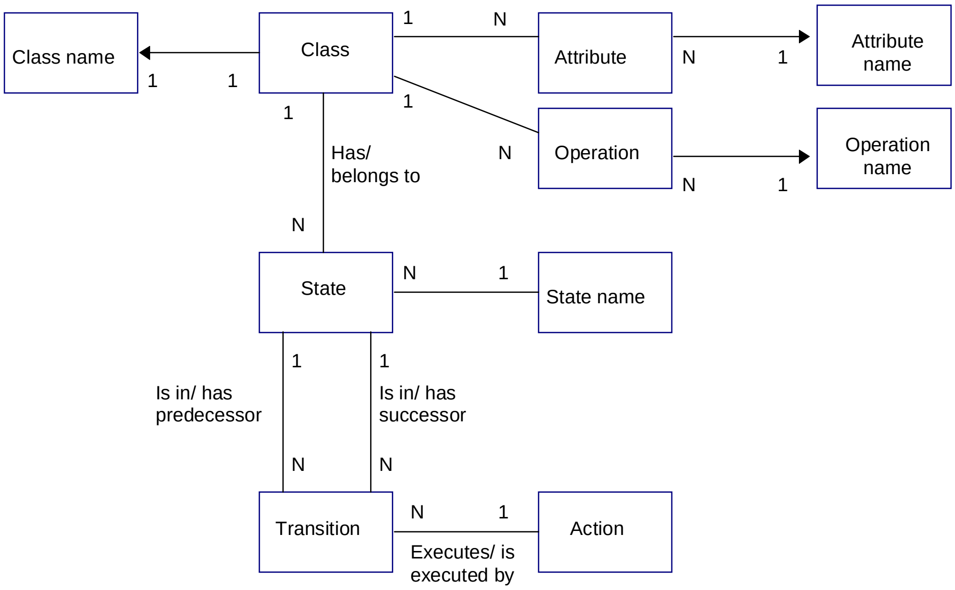


FIGURE 3-6 A metamodel of a state model defined with ASDM (adapted from Heym 1993, p 210).

Although the same entity symbol of ASDM is applied for most of the method concepts, the reference model classifies them into meta-entity types, meta-relationship types and meta-attribute types. Hence, a 'transition' is considered as a relationship type and an action as an attribute type. Furthermore, meta entity types are subtyped into fundamental entity types (i.e. state) and structural entity types (class name) although this can not be easily noticed from the graphical metamodel. Typically, structural entity types are results of an aggregation relationship.

The metamodel based on the reference model shares some of the limitations already discussed for the ER model. Links between modeling techniques are not defined, uniqueness of state names is not defined, and because minimum cardinalities can not be specified we can not define mandatory relationships. Although the reference model distinguishes techniques and their components, the "explains" (Heym and Österle 1992, p 11) relationship does not allow the definition of any rules of the technique-related connections other than the maximum cardinality. Hence, for example, the rule that each class must be specified in one or more state models can not be included in the metamodel. Moreover, the metamodels developed in Heym (1993) do not include these types of connections. The focus of the reference model, as reflected in its name, is on more general method knowledge and therefore it lacks detailed metamodeling capabilities. In fact, Heym and Österle (1993) aim to describe all method related knowledge at a high level of granularity to understand and compare methods.

3.3.3.3 Object-Property-Relationship-Role model

The OPRR model has been developed by Welke (1988) and Smolander (1992). The focus of the OPRR model has been from the beginning to specify single modeling techniques. It extends the largely unspecified 'role' concept of the ER model to clarify the way in which objects participate in a certain relationship. In other words, the role defines what “part” an object plays in a relationship (Smolander 1992). A role can have also properties. OPRR applies the same representation as the ER model and the role type is represented by a circle.

This model forms a specification language for graphical method representations in the MetaEdit tool (Smolander et al. 1991). MetaEdit can be applied as a CASE tool (MetaCase 1994), metaCASE tool (Smolander et al. 1991), or even to customize other metaCASE tools (Rossi et al. 1992). To better address tool implementation and formalization of OPRR Smolander (1992) has added additional constraints into OPRR, namely identifying properties, a duplication policy for object types (whether homonyms are allowed), direction for relationships and modeling technique related data types. A useful feature of MetaEdit is that a graphical OPRR representation can be built up, and compiled in this environment at any time during metamodeling. Therefore the implementation of the CASE tool is easy and straightforward after the graphical representation of the method in OPRR has been achieved (see Tolvanen and Lyytinen 1993). This possibility allows testing the metamodel (cf. Figure 3-7) as a “specification” for IS modeling.

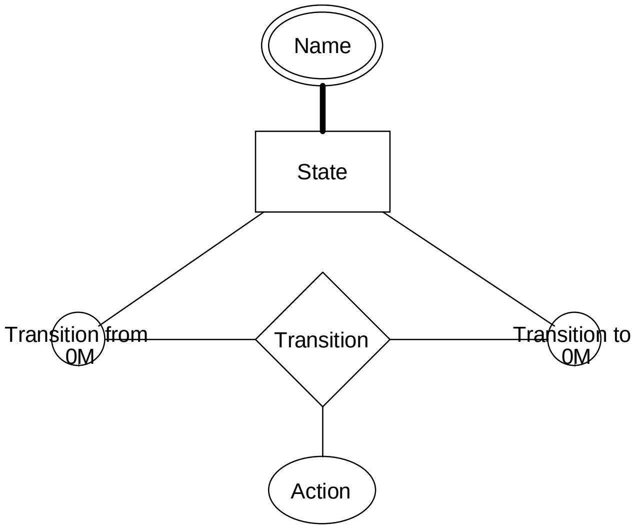


FIGURE 3-7 A metamodel of a state model defined with OPRR.

Regarding extensions to the ER model, the transition now has an action, and states are identified based on their names (double lined ellipse). The duplication policy is also used for states (although not represented in the graphical notation of OPRR): there can not be two different states with the same name. In modeling this means that copies of the same state are allowed, and changing the name of one state is reflected in all copies of that state. The uniqueness of states would make better sense if a state had other properties, like actions executed in the state (as in OMT, Rumbaugh et al. 1991).

Because OPRR focuses on specifying single techniques, a connection between a class and a state model can not be specified. Of course, the relationship could be specified with a normal OPRR relationship (like with the second version of the ER-based metamodel) but no distinction could be made between relationships between techniques and within a technique. Moreover, mandatory actions can not be specified, nor actions referring to operations of a class, nor that state names are dependent on the class they belong to. Thus, according to the OPRR metamodel a state model of another class can not use the same names for states (referring to different states).

3.3.3.4 Method Engineering Language

MEL is a language for describing and manipulating (i.e. selecting and assembling) parts of ISD methods. It is designed specifically to support ME (Brinkkemper 1996, Harmsen 1997). Because of its general focus on supporting all ME tasks, MEL describes both product (i.e. conceptual structure and notation) and process aspects of method knowledge. In this sense it is very similar to ASDM (Heym 1993) although MEL has not been applied so extensively to model ISD methods. Like ASDM MEL too has a supporting tool, called Decamerone, for describing methods and customizing a third-party repository (Maestro II). The selected repository, however, limits the number of possible methods supported. Therefore Decamerone is limited to combining existing methods which can be already stored with Maestro II (e.g. the support for object-oriented concepts has only later been added into Maestro II).

Although MEL is founded on first order predicate logic, its relation to semantic data models can be easily detected. Moreover, MEL also includes a graphical modeling language which is a subset of MEL and very similar to the ER and OPRR models. Figure 3-8 illustrates the metamodel of our example, both with a textual and a graphical part of MEL. Parts of the metamodel which are related to guiding the method selection and modeling process are excluded.

```
PRODUCT StateModel;
IS_A Product;
LAYER Diagram;
( - State;
  - Transition
).

PRODUCT State;
LAYER Concept;
PART OF StateModel;
ASSOCIATED WITH {(StateTransition_1, source),
(StateTransition_2, target)}.

PRODUCT Transition;
LAYER Concept;
PART OF StateModel;
ASSOCIATED WITH {(StateTransition_1,
has_source), (StateTransition_2, has_target)}.

ASSOCIATION StateTransition_1;
ASSOCIATES (State, Transition);
CARDINALITY (0,n; 1,1).

ASSOCIATION StateTransition_2;
ASSOCIATES (State, Transition);
CARDINALITY (0,n; 1,1).
```

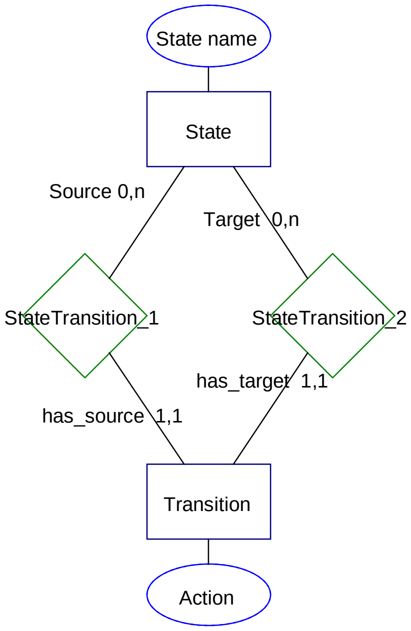


FIGURE 3-8 A metamodel of state model defined with textual and graphical part of MEL.

As the metamodels illustrate the textual and graphical part are largely equivalent. Since the graphical part is only a subset of MEL it is not adequate to define all the method knowledge of our example. The limitations are similar to the limitations of OPRR. In MEL additional constraints can be specified with predicate logic. For example, the following constraint to deny recursion could be added to a transition relationship:

```
Rule1: forall T1 in transition forall A1, A2 in
State [has_source (T1, A1) and has_target (T1, A2)
implies not (A1=A2)];
```

These types of additions are possible for all metamodeling languages, but they provide limited help for method modeling since they do not guide towards modeling relevant aspects of methods. In other words, aspects which are not required in ME at all can be defined as well. This is paradoxical because the aim of methods is to focus attention on relevant aspects of IS, but on the metalevel (i.e. metamethods) this requirement is often ignored. Moreover, as already discussed, these extensions do not support maintainability, ease of use and communication as well as semantic data models.

By inspecting the definition of MEL and example metamodels it is unclear how all method knowledge related to the example can be specified with the predicate logic extensions. These include identity and different scopes: for example that each state must have a unique name among states of the class, and that actions must refer to operations defined for the related class. Unfortunately, the metamodels made to illustrate the use of MEL include only a few examples of detailed metamodels of modeling techniques.

3.3.3.5 ComplexCoveringAggregation

CoCoA (ComplexCoveringAggregation) has been developed to support conceptualization and data modeling of complex problem domains (Venable 1993). As stated in the name of the model, its extensions deal with modeling aggregations which cover entities and named relationships, n-ary relationships, alias naming, and entity categories (through the named roles they participate in). CoCoA has also been applied in metamodeling (Venable 1993) and method integration, and is intended to be used as a metametamodel for a modeling tool (Grundy and Venable 1996). Most of the metamodeling efforts carried out with CoCoA have focused, however, on data modeling techniques, and larger metamodeling efforts including whole methods have not, to our knowledge, been reported.

Since method knowledge can also be considered as a complex object, i.e. as involving shared method elements and multiple levels of granularity, CoCoA performs better than the earlier metamodeling languages (cf. Figure 3-9). The metamodel specifies most of the method knowledge, such as identification of modeling techniques and their components: that more than one state model for a class is possible; and that transition relationships are binary. Furthermore, a covering aggregation is used to describe the components of the modeling technique (large gray box), and that a class can have several attributes and operations. Moreover, an "action" alias is used to denote that class operations are applied in transitions. This means that actions which are not described as operations should not be possible. However, the dependency to operations of a state model related class (or its superclasses) can not be specified.

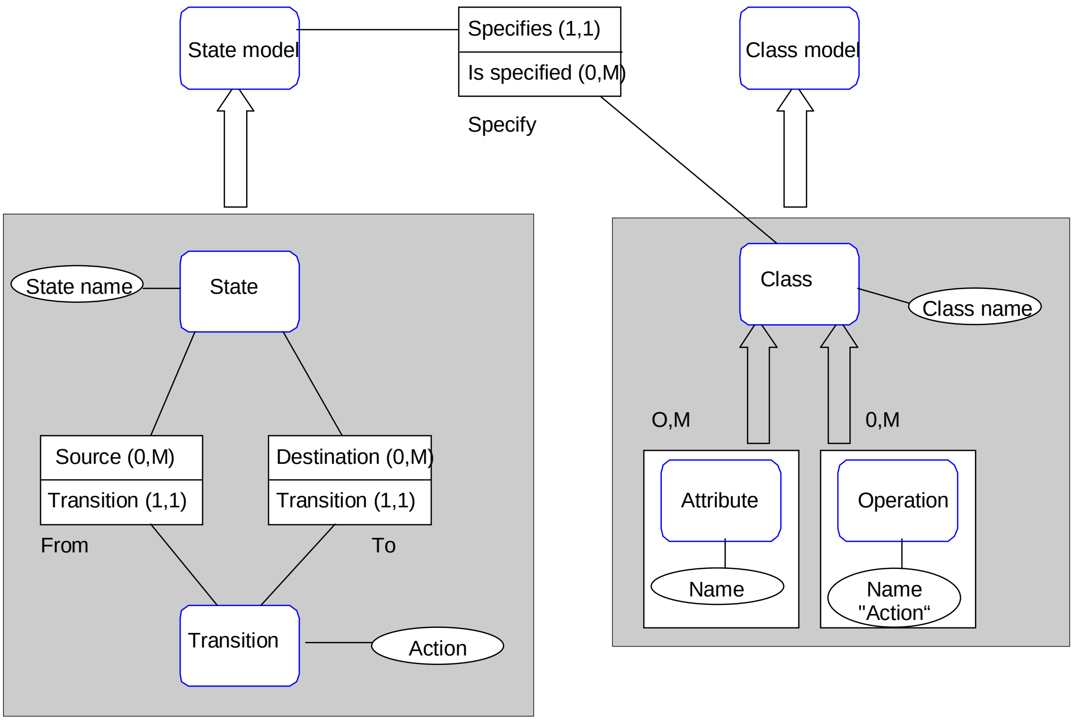


FIGURE 3-9 A metamodel of a state model defined with CoCoA.

All method knowledge related to our example, however, is not specified even with CoCoA. First, no possibility exists to define which attribute values are mandatory: it should not be possible to define states without state names, or transitions without actions, but the CoCoA metamodel does not distinguish between mandatory and optional values. One possibility would be to model mandatory properties with a single aggregate, like an attribute of a class, with multiplicity value of one-to-one (1,1). Second, as in the ER model, the distinction between a transition and a state is not clear since they are not sub-typed as in ASDM (Heym 1993). However, with a minimum cardinality the mandatory participation of each transition in both possible roles is guaranteed. The difficulty to distinguish types which refer to a modeling technique (e.g. state model) and its components (e.g. state) also exists because both are represented as entities. Moreover, if the state model applied n-ary roles, the number of role instances a relationship instance can or must have can not be specified with CoCoA. Third, there is no specification of the requirement that a class can not have several different states with the same name. Implicitly, we can expect that each name must be unique, but CoCoA does not restrict the scope of instances in which the uniqueness should be valid. For example, along all class diagrams, classes with the same name typically denote the same class (e.g. Booch 1991), but several classes can have states named similarly but which still refer to different states.

3.3.3.6 Nijssen's Information Analysis Method

NIAM (Nijssen's Information Analysis Method) has been developed primarily to support information analysis (Verheijen and Van Bekkum 1982, Nijssen and Halpin 1989) but it has also been applied in several metamodeling efforts (e.g. Wijers 1991, Hofstede 1993). It is also a good example of method evolution since several versions of NIAM exist with a wide variety in the terminology. In the metamodeling effort we have applied basic NIAM with the PSM extension (Hofstede 1993). To our knowledge, no modeling tool using NIAM as a metametamodel is available.

The NIAM/PSM based metamodel of the state model example is illustrated in Figure 3-10. A state, a transition, a class, an attribute and an operation are defined as object types and illustrated with circles. As with the ER model, NIAM does not distinguish between relationships and objects. Although transitions could be modeled as a relationship, the constraints could not be specified as it is defined with an object type. Attributes, also called label types (ter Hofstede 1993) or slots (Verheijen and Van Bekkum 1982) are similarly represented with circles but the name of an attribute is described in brackets.

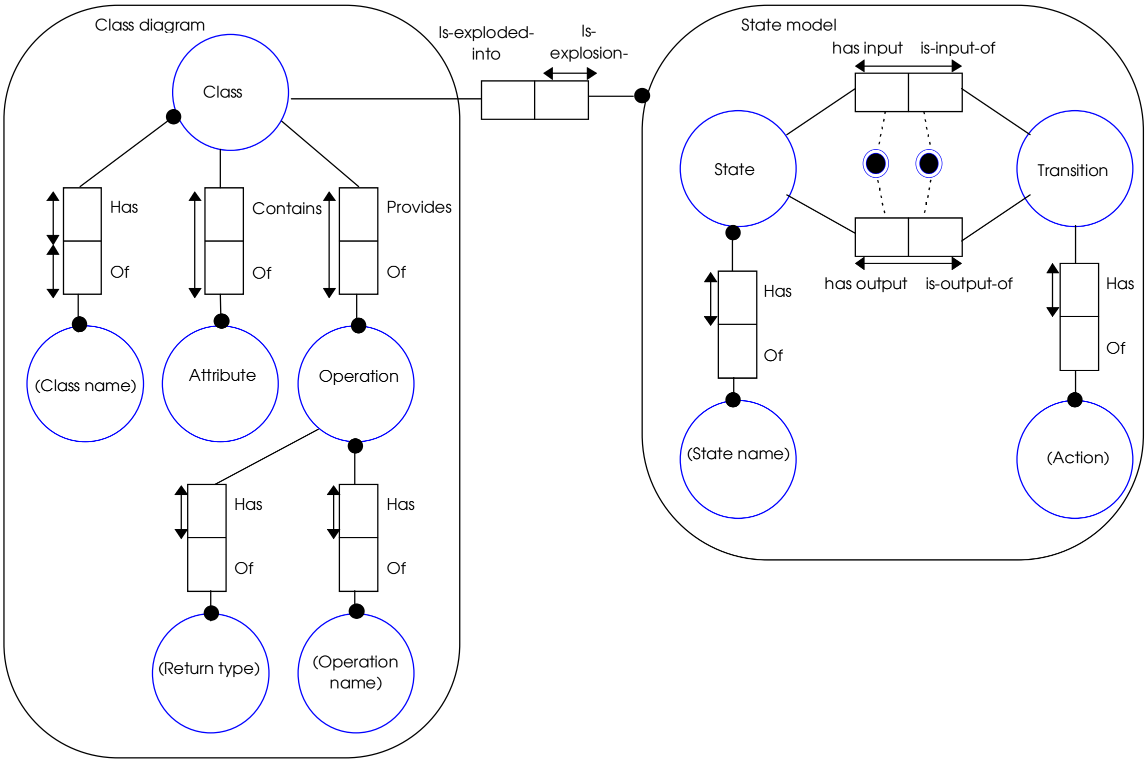


FIGURE 3-10 A metamodel of a state model defined with NIAM.

The linkages between attributes and object types are described with relationships (also called a bridge type or a fact type) although one-to-one relationships (e.g. between a class and a class name) could also be defined by adding the attribute name in brackets below the name of the object type. The use of relationships has the advantage of illustrating different constraints. A total role constraint, illustrated as a black dot on the object or attribute type part of the relationships, specifies a mandatory role. For example, each state must be described with a state name and all state names must belong to at least one state. The uniqueness constraint, illustrated with arrows in role symbols, shows which instances of a role or concatenation of roles must be unique. For example, only one instance of a class name can appear in the relationship between classes and class names. The uniqueness constraint is also used to define that each state can have only one state name, and the same state name can be used as a value for many states. The rule of the example that a class may have only one state with a given name could be added to the metamodel as a relationship with a uniqueness constraint, but then no difference could be made between different kinds of relationships (i.e. those defined between states and transitions of STD and those defined for describing constraints). The total role constraints in input and output roles define that each state has an input or an output of a transition and that each transition has a state as an input or output. This specification can not be used as a metamodel for guiding modeling actively. According to the constraints, creation of the first state would require creation of a transition which could not be possible because other states are not available.

Although NIAM can support most constraints of a state model it does not address multiple interconnected techniques. Therefore, the metamodel includes PSM extensions (ter Hofstede 1993). Modeling techniques are defined as schema types and illustrated as rounded boxes around technique related types. The linkage between state models and classes is described as a NIAM relationship. This relationship can be distinguished from other relationships because it is drawn outside the schema types. The total role constraint and the uniqueness constraint are used to define that each state model must be related to only one class (i.e. other classes can not refer to the same state model). Finally, linkages between the values of actions and operation names can not be defined with the graphical constraints of PSM/NIAM. As with MEL, additional grammars like LISA-D (Hofstede et al. 1993) have been proposed and could be used. For example, a correspondence between an operation name of a class and an action of a transition would then be:

```
Action of Transition PART-OF State model is-explosion-of
Class has THAT Operation name EQUALS Action.
```

3.3.3.7 Graph-Object-Property-Relationship-Role model

GOPRR (Graph-Object-Property-Relationship-Role) has been developed from the OPRR data model (c.f. Smolander 1991, Marttiin et al. 1995, Tolvanen et al. 1993, Kelly et al. 1996, Kelly 1997). It has been developed specifically for metamodeling. The GOPRR model is implemented in a MetaEdit+ metaCASE tool (Kelly 1997, MetaCASE 1996a) which enables GOPRR-based metamodels to be instantiated at any time in the same tool as a model.

We will apply GOPRR in method analysis and metamodeling in Chapter 4. Therefore, the GOPRR model is described in more detail in the appendix. Here we briefly define the main extensions to OPRR. As the extra G in the acronym indicates, the main extension is a graph metatype. It is a collection of all other GOPRR types (including the graph type) chunked together into a modeling technique. The GOPRR model also offers three semantic relationship types between graphs and other non-property types (i.e. object, relationship and role types): inclusion, decomposition, and explosion. Inclusion means that a graph may consist of instances of particular object types, relationship types and role types. Moreover, an object type in a graph type may be either decomposed or exploded into another graph.

In addition to the graph metatype, GOPRR extends OPRR with an abstraction mechanism to generalize and specialize non-properties. It applies single inheritance, where an ancestor can contain a number of descendants, but each descendant can have only one ancestor. Cartesian aggregation is defined so that non-property types can contain any number of property types; in the other direction, a property type can be shared by many non-property types. GOPRR also applies cardinality constraints in a different way than in OPRR. In GOPRR, cardinality defines a minimum and a maximum number of instances of a role type a relationship type instance may have. In other words, this defines whether relationships are binary, specific n-ary, or whether some roles are optional. Other additions of GOPRR include local names for properties and a collection data type[14]. These are described in the example and in the appendix.

Because of these extensions GOPRR can capture almost all the method knowledge related to our example. GOPRR does not have a standardized graphical notation (Kelly 1997). Rather, metamodels are specified through forms represented as windows in MetaEdit+ tool (e.g. Rossi and Tolvanen 1995). One main reason for this choice is the relatively large number of constraints which are not best represented with a graphical notation. If they were added to increase the picturability (cf. Venable 1993) it would make GOPRR-based metamodel representations inherently complex. Graphical notations have, however, been implemented for metamodel representations. Following Kelly and Tahvanainen (1994) and Hillegersberg (1997) we have defined a similar graphical notation to make metamodels readable and more comparable with notations of other metametamodels. This notation is used to illustrate the example metamodel in Figure 3-11.

In the GOPRR representation used, a technique is defined as a graph type represented with a window symbol. Inclusion is described by drawing components (i.e. types) inside the window symbol of a technique (i.e. graph type). The relationship between a property and a non-property includes the local name of the property in that non-property, whether it is unique there, and whether it is the identifying property. For example, a class name is defined as an identifier and is unique among all classes. These constraints are not added to the property type because GOPRR supports reusability of types and these constraints may be different in other places where the property type is used (Kelly 1997). The data type of operations of a class is 'collection', and this is illustrated with a double-lined ellipse. Hence, the metamodel specifies that each class can have a collection of operations and each operation can be defined

[14] Recently GOPRR has been extended with multiplicity constraints, and with checking of property values through a constraint specification language (Kelly 1997).

through its name and return type. More generally, a property type can also be linked to GOPRR types other than an object type.

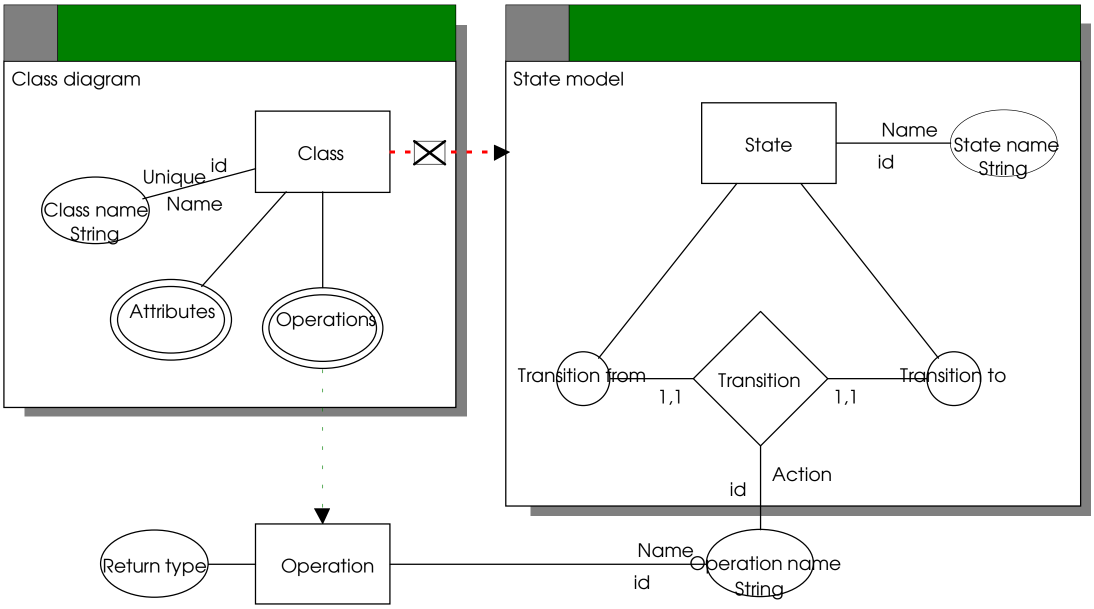


FIGURE 3-11 A metamodel of a state model defined with GOPRR.

Property sharing is used in the metamodel to define that both transitions and operations refer to the same operation name. For the purpose of state modeling, an operation is called an action by defining the local name. Modeling the life-cycle of classes with state models is specified with an explosion link represented as a dotted line with a cross in a box and an arrowhead from the class to the state model.

The GOPRR metamodel, however, does not define all method knowledge. Mandatory actions and state names can not be defined because the metamodel does not differentiate between optional and mandatory values. A recent addition to GOPRR for checking property values, however, can support the definition of mandatory properties. The uniqueness of state names depending on the class can not be specified because the uniqueness constraint is relevant among all instances (as with class names). Mandatory explosions (i.e. each class has at least one state model) and that only one class can refer to a single state model can not be specified. Moreover, although operation name and action refer to the same property type, similarly to the CoCoA alias, no restriction can be defined on the population of property type values. Hence, an action in a state model can refer to any operation defined for any class. The correct definition should be that an action can refer to any of the operations of the related class or its superclasses.

3.3.4 Constructs of metamodeling languages

In this section we defined metamodeling in the context of ME and described a set of metamodeling languages. Our focus is on semantic data models because of their support for communication, ease of use, and a formal enough basis to enable metamodel-based tool adaptation.

The set of metamodeling languages is illustrated by modeling a small example of method knowledge. With respect to our first research question of representing method knowledge "completely", the metamodeling exercises were used to analyze the modeling power of the metamodeling languages. This revealed both similarities and differences among the constructs provided for metamodeling, although the example was so small that not all limitations or strengths could be described. This would require modeling a larger sample of ISD methods, as performed in Chapter 4. The metamodeling exercises, however, clearly showed the limitations of the basic ER model and showed how additional constraints are used to model methods.

In this thesis our main interest is on finding essential metamodeling constructs which could be used as predefined and generalized (meta)knowledge about modeling techniques. In other words, constraints like cardinality in the ER model and explosions in GOPRR already guide engineers to identify, capture and construct certain aspects of method knowledge (i.e. for each relationship at least the maximum cardinality must be examined). The currently used constraints, however, are not adequate as the metamodeling example clearly demonstrates. Hence, instead of applying programming languages or additional grammars (e.g. Harmsen 1997, (ter Hofstede et al. 1993, Saeki and Wenyin 1994) we seek constructs which are relevant in metamodeling with semantic data models. The limited number of metamodeling constructs will help method engineers to focus on perceiving known rules about method knowledge, speed up the metamodeling process, and support communication among the participants in ME efforts. Hence, our aim is to find constructs specific to our domain of metamodeling, in the same way as developers of ISD methods try to find constructs specific to their domains of application.

3.4 Summary of method engineering approaches

In this chapter we defined method engineering and placed our research in the context of ME research. We analyzed the current understanding of ME in terms of its process, the type of method knowledge "engineered", and the criteria used in method construction. First, related to the ME process, the analysis shows the dominating *a priori* approach. Most principles are targeted toward method selection and construction, and little attention is paid to analyzing whether the constructed method is applicable in the task for which it has been promoted, or could the method be improved. Hence, information about methods and ISD contingencies is expected to be known completely in advance. Moreover, learning from method use and the evolution of methods are ignored.

Second, we surveyed the ME criteria that have been proposed to construct methods. These explain how the situational applicability of methods can be improved. Of these approaches, we discussed those based on contingencies, problems, and stakeholders' values. Each of these approaches is limited in its ability to guide method construction in detail. They are too general to provide guidelines for constructing techniques or their parts for situational needs; they rely on existing problems and contingencies; and they do not support the creation of new knowledge originating from an organization's own experiences. Finally and maybe most importantly, none of these approaches seems to be used systematically in an organization's local method development efforts. This is especially important since empirical studies of local method development reveal that few organizations apply systematic customization processes (Smolander et al. 1990, Cronholm and Goldkuhl 1994), but rather follow ad-hoc practices (Hardy et al. 1996, Hughes and Reviron 1996).

Third, based on the shell model (cf. Section 2.2) we analyzed which types of method knowledge are identified and subject to ME. We focused on metamodeling languages targeted to representing the conceptual structures behind modeling techniques. These are also most widely studied in the ME literature. Since conceptual models describing methods should be based on some knowledge representation scheme, we furthermore described a set of metamodeling languages. We illustrated, through a small example, various grammatical and notational constructs applied in metamodeling languages. This example showed some differences and limitations in the metamodeling constructs, and it serves as the background for a more detailed analysis of their metamodeling support in the next chapter.

4 MODELING METHOD KNOWLEDGE FOR MODELING TOOLS

4.1 Introduction

One part of engineering, and therefore also of method engineering, is concerned with model building, analysis, and implementation tasks. Accordingly, it is of great importance to understand how knowledge about model building, analysis, and implementation can be captured, represented, and analyzed (Welke 1988, Wijers 1991, Brinkkemper 1996). Therefore, in this chapter we shall focus on our first research question (cf. Section 1.5.3) dealing with constructs of metamodeling languages:

> "How completely can meta -data models represent knowledge about ISD methods for modeling tools?"

Metamodeling constructs are needed to model method knowledge as completely as possible. By completely we mean the 100% principle suggested by Griethuysen (1982), which has also been applied to the metalevel by Brinkkemper (1990). The principle states that a metamodel should describe all relevant aspects of a method. According to this ideal, a metamodeling language should be capable of modeling all aspects of the method knowledge. It must be noticed, however, that our focus is on meta-data models (i.e. on the static conceptual structure behind modeling techniques, cf. metamodels in Section 3.3.3), and therefore we exclude the modeling of other types of method knowledge (e.g. such as processes in Brinkkemper (1990), or participation in

Tolvanen et al. (1993)). Instead of focusing on general data modeling constructs like classification, generalization, and aggregation provided by semantic data models (Brodie 1984), we shall focus on specific constructs in metamodeling languages that are essential in modeling methods. These constructs extend available data modeling languages in several ways and provide a basis for modeling method knowledge more completely. The proposed constructs are not necessarily essential in other modeling domains, as some deal with specifying knowledge about single techniques (i.e. their conceptual structures, rules and constraints), and some with integrating multiple techniques into a method.

In seeking essential constructs for meta-data modeling we will follow an inductive approach. We shall analyze in detail what kind of knowledge ISD methods in computer-aided modeling tools include, as well as how and to what extent this knowledge can be included into a metamodel which is based on a semantic data model. The analysis of 17 ISD methods carried out as a metamodeling task also includes tool adaptation for each method. This allows us to obtain a detailed understanding of the structure and content of method knowledge. The resultant patterns, categories and rules of a method, defined here as part of method knowledge, are used to derive requirements for metamodeling (Patton 1990). To our knowledge (cf. Tolvanen et al. 1996) this kind of approach has not been applied to such an extent for analyzing and developing languages for method engineering[15]. Because of the inductive research approach, the proposed requirements for metamodeling can not be claimed to be complete, but rather they represent an essential set of constructs needed to model the modeling techniques of the selected "sample". Additional requirements for metamodeling languages may be found if more ISD methods were included in the study.

We will assess the metamodeling capabilities of available metamodeling languages based on the derived requirements: how do they satisfy the essential need to capture and specify method knowledge and serve as a metametamodel? Hence, this assessment extends the analysis of metamodeling languages described in Section 3.3.3. It must be noted that our aim here is not to develop a new metamodeling language, but rather determine a set of constructs applicable for metamodeling, and which are necessary for "good" metamodeling languages. The results of the assessment can be applied by developers of metamodeling languages to improve their languages, and also by metamodelers to specify methods more completely. These extensions will also be applied to specify metamodels in the action research studies reported in Chapter 6.

The chapter will proceed as follows. In the next section we describe the research method in more detail and discuss the metamodeling process. In Section 4.3 we describe some of the metamodels which were developed and CASE tools which were modified. Section 4.4 defines the essential metamodeling constructs needed to model a single technique as well as a

[15] A typical research approach is based on the selection of one (or a few) ISD methods as examples, and on studying how they can be represented with the proposed metamodeling language (e.g. Smolander 1992, Welke 1988, Hofstede 1993, Saeki and Wenyin 1994).

complete method. In Section 4.5 a set of metamodeling languages is assessed in terms of how they support metamodeling. Based on this assessment we identify some limitations in the modeling power of semantic data models and highlight aspects of method knowledge which can not be captured with their data modeling constructs. This provides motivation for further research to extend semantic data models with additional constructs. Section 4.6 summarizes the chapter.

4.2 Method selection and method modeling process

Before analyzing and describing requirements for metamodeling languages we shall first clarify our research method. We discuss how ISD methods were selected for the study, how they were modeled, and how they were adapted into a modeling tool. Each of these steps is described in the following subsections in more detail.

4.2.1 Selecting methods for the study

In the selection of ISD methods we used several criteria. First, we chose methods which are well-known or widely-used. This criterion ensures that the metamodeling constructs are needed for modeling most of the methods used today. Second, because we focus on representing method knowledge in modeling tools, only those methods that could be supported through computer-aided modeling tools were selected[16]. In fact, the selected methods are already supported by another computer-aided environment, either in a method-dependent CASE tool or in a metaCASE environment. Availability of tool support also shows that the selected methods are known, and that they have users. Otherwise there would hardly be such tools available.

Third, the primary criterion for method selection was to find a set of ISD methods which represent diverse approaches and exploit different kinds of conceptual structures. This selection criterion ensures that the metamodeling constructs identified are valid for a wide variety of methods, not just, for example, for modeling object-oriented methods. Therefore the chosen methods include data modeling techniques, IS planning techniques, structured design and analysis methods, object-oriented methods, and business modeling methods. The relatively large number of object-oriented methods included can be explained by the fact that they include more techniques and have richer conceptual structures than other methods (Rossi and Brinkkemper 1996). As a consequence, it is expected that the modeling of object-oriented methods will necessitate the use of more powerful metamodeling languages. Table 4-1 provides a summary of the selected methods together with their individual modeling techniques (i.e. each method consisting of one or more techniques).

[16] The excluded methods typically focus on early phases of ISD, such as blackboard and brainstorming based methods. Similarly, methods related to project management, configuration management etc. are excluded from our study.

TABLE 4-1 Methods selected for metamodeling.

Methods analyzed	Individual techniques
Activity Model (Goldkuhl 1992, 1989)	Activity model Goal list Problem list
Demeter (Lieberherr et al. 1994)	Demeter
BON, Business Object Notation (Walden and Nerson, 1995)	System chart Cluster chart Event chart Scenario chart Creation chart Static model Dynamic model
BSP, Business Systems Planning (IBM 1984)	Problem table Process/entity matrix Process/organization matrix Process/system matrix System/entity matrix System/organization matrix
EXPRESS (ISO 1991)	EXPRESS-G
Fusion (Coleman et al. 1994)	Object model Operation model Object-interaction graph Visibility graph Inheritance graph
IDEF, Integration Definition (Ross and Schoman 1977, FIPS 1993a, 1993b)	IDEF0 IDEF1 IDEF3
ISAC, Information Systems Work and Analysis of Changes (Lundeberg et al. 1981, Lundeberg 1982)	A-graph I-graph Problem table Table of problem groups Table of needs for changes Table of interest groups C-graph D-graph
JSD, Jackson's System Development (Jackson 1976, Cameron 1989)	Data structure diagram Program structure diagram

(continues)

TABLE 4-1 (continues).

OMT, Object Modeling Technique (Rumbaugh et al. 1991)	Class diagram Data flow diagram State transition diagram Use case model
OOA/OOD, Object-Oriented Analysis and Design (Coad and Yourdon 1991a, 1991b)	Object diagram State transition diagram Service chart
Moses (Henderson-Sellers and Edwards 1994)	O/C model Event model Inheritance model
OODA, Object Oriented Design (Booch 1994)	Class diagram State Transition diagram Object diagram Module diagram Process diagram
OODLE, Object-Oriented Design Language (Shlaer and Mellor 1992)	Information model State model Action data flow diagram Object access model Process table
OSA, Object-Oriented Systems Analysis (Embley et al. 1992)	Object-relationship model Object-behavior model Object-interaction diagram
SA/SD, Structured analysis and design (Gane and Sarson 1979, Yourdon 1989a, Ward and Mellor 1985)	Data flow diagram RT data flow diagram Entity relationship diagram Structure chart State transition diagram
UML, Unified Modeling Language (Booch and Rumbaugh 1995, Booch et al. 1996, Booch et al. 1997)	Class diagram Use case diagram Operation table Collaboration diagram State diagram Composite diagram Component diagram Deployment diagram

4.2.2 Metamodeling process

The structure of the metamodeling process is summarized in Figure 4-1. The universe of discourse is that of the selected methods, in contrast to modeling the "real world". Each method was examined and modeled using a metamodeling language in a metaCASE tool. The outcome of the metamodeling effort, i.e. the metamodel produced, was adapted into a tool environment and validated by trying out the method in system modeling.

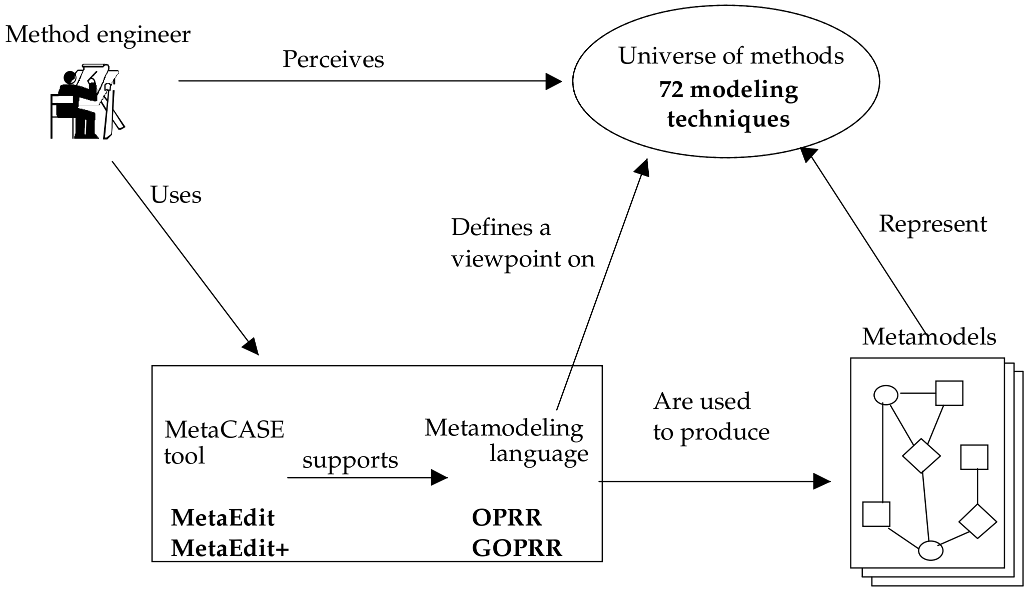


FIGURE 4-1 Structure of method modeling.

Like any modeling task, metamodeling is driven by a number of objectives. In our case metamodeling was based on a content analysis of the published method literature. We tried to follow as closely as possible the method descriptions given in the reference books (cf. Table 4-1) rather than deviating slightly to better suit an envisaged use situation. Content analysis can be defined as a process of identifying, coding and categorizing primary patterns in data (Patton 1990). In our metamodeling study, relevant data about methods was first collected and identified through the method literature. The method literature basically describes the concepts, languages, notations and possible requirements in building tool support for a method. Second, the data was classified into distinct types, allowing us to simplify and systematize the conceptual structure of methods. In our case the classification was based on the metamodeling languages. Naturally, a metamodeling language with a given classification schema restricts our view of methods. Third, the methods were documented as completely as possible via the metamodels.

Another objective for the metamodeling task was the method-tool companionship: the metamodeling task was conducted by examining how selected methods could be supported by a tool. Therefore, the metamodels were "executable" and implemented into a modeling tool (see Section 4.2.3).

The actual method modeling was conducted in two phases. In the first phase, winter 1992-1993, we analyzed a set of methods and developed a set of tentative metamodels (reported in Tolvanen and Rossi 1996). The second phase took place in 1995-1996, when we analyzed and modeled the same methods in more detail. We thus modeled the methods twice. During the first round, we limited our focus to modeling individual techniques and their conceptual structures, whereas in the second phase we focused on method integration, i.e. how different techniques could be combined to form a "whole" method. Because of our interest in tool-supported methods we applied two metamodeling languages that are supported by metaCASE tools, OPRR (Welke 1988, Smolander 1992) and GOPRR (Marttiin et al. 1995, Kelly et al. 1996) respectively[17]. These metamodeling languages were applied because of the tool support available for metamodeling and testing the metamodels (cf. Section 4.2.3), because they succeeded relatively well in the metamodeling exercise (cf. Section 3.3), and because our own metamodeling experience was mostly with these languages.

During method modeling we distinguished the following set of tasks that OPRR (Tolvanen and Lyytinen 1993) and GOPRR related metamodeling must follow. These tasks, applied in several successful metamodeling efforts (cf. Tolvanen and Lyytinen 1993, Rossi and Brinkkemper 1996, Hillegersberg et al. 1998), provided a more detailed view of the classification process of content analysis as adapted to metamodeling. These tasks are:

1) **Identification of the techniques in the method.** Because each method can consist of one or more techniques we first need to identify them (as listed in Table 4-1). Most often an ISD method proposes a number of separate techniques with different concepts and supporting notations, but a technique can also include concepts that are shared with other techniques. For example, in Embley et al. (1992) an object-behavior model (used for describing a life-cycle of a single object through state transitions) can include interaction relationships which are also applied in object-interaction models. Some techniques can also be subsets of other richer and more complex techniques in the same method. For example, in Fusion (Coleman et al. 1994) and in MOSES (Henderson-Sellers and Edwards 1994) an inheritance graph includes only a subset of the concepts used within an object model.
2) **Identification of object types.** The modeling of an individual technique starts with resolving what kind of object (or entity) types a technique recognizes. Object types can be defined as basic elements of a technique that can exist independently of other types in a technique. Examples of object types in a data flow diagram are 'process', 'store', and 'external'.
3) **Determination of properties for each object type.** Each object type has zero to many properties that characterize object type instances. Since object types typically account for the majority of properties of a technique and properties can be shared with other types (Rossi and Brinkkemper 1996), this task is distinguished as a separate task. Identification of properties that belong to types other than object types are discussed in task 6. Examples of property types are 'identifier' and 'name' for an object type 'process'.

   It must be noted that in GOPRR a property type can have an internal structure, an identity constraint, and a local name. For example, the property type 'operations' is of collection data type, and refers to the 'operation' object type which it contains. This object type in turn consists of other property types such as an 'operation name' and a 'return type' (cf. Figure 3-11). In addition to defining a set of property types, one of them can be defined as an identifying property type. In GOPRR, the identifying property defines which

[17] These languages are discussed in Section 3.3.3, in the appendix, and in Tolvanen and Rossi (1996).

property type is used as the non-property type's name. For example, a name of a class comes from its 'class name' property type, rather than from e.g. its attributes (i.e. the 'attributes' property type). When attached to non-properties, a property type can be re-labeled in the context of the attached non-property. This allows to define property sharing between non-property types: instances of two or more non-property types can refer to the same property values (cf. Kelly 1997).

4) **Determination of relationships.** Object types are connected to each other through a number of relationship types. This task deals with the identification of those relationship types that connect object types. Examples of relationship types are 'data flow' in a data flow diagram and 'inheritance' in a class diagram. It must be noted that these can not be defined as an object type in GOPRR because that would lead to incorrect method definitions: in the context of data flow diagrams this would allow data flows which are unrelated to processes.
5) **Determination of roles.** After object and relationship types have been identified, connections between these types need to be established. In the metamodeling languages we applied, these connections are specified by using role types. Examples of the role types are 'sender' and 'receiver' connected to the 'data flow' relationship type, and 'subclass' and 'superclass' connected to the 'inheritance' relationship type.
6) **Allocation of properties to relationship types and role types.** As with object types, relationship and role types can also have properties. These can typically be allocated to relationship or role types after all types have been identified.
7) **Determination of metamodels for individual techniques** deals with making all possible connections between the object, relationship and role types in a single technique. The connections can be further specified according to the cardinality: a minimum and a maximum number of instances of a role type a relationship type instance can have.
8) **Determination of linkages between separate techniques** is needed to form a whole method. Thus, the previous steps are carried out for each technique individually. In general, these linkages define interactions between techniques in two directions: horizontal and vertical (Lyytinen et al. 1991). In the horizontal direction connections or constraints between types or instances in different techniques are specified. For example, data stores in data-flow diagrams are redefined for cross-checking with ER models. The vertical direction refers to linking semantically equivalent descriptions at two consecutive levels of abstraction, such as connecting an ER model with its representation in a relational schema, or transforming a data flow diagram into a structure chart (Yourdon 1989a).
9) **Determination of the representational part of the method.** The use of methods in modeling tools necessitates the specification of notations such as graphical symbols. The representations are needed because the descriptions derived using the method are created, compared, and communicated by humans (Harel 1988). Accordingly, we must define symbols and location information for the elements of the method. Examples of symbols are bubbles

for processes (Yourdon 1989a), and a cloud for classes (Booch 1991). Examples of location include placing elements in the horizontal axis of a matrix and drawing superclasses above subclasses in a diagram. Typically representations are expected to correspond one-to-one to the types specified in a metamodel (Venable 1993). The representation aspect in a modeling tool also includes dialogs, menus, and toolbars: to be used in a tool these must be defined for the method as well.

10)**Analysis and evaluation of the metamodel.** Because method specifications in the literature are often inconsistent, ambiguous and informal there are several alternative ways to model a method. At this step different modeling alternatives, or even versions of metamodels, are analyzed and assessed based on the available method descriptions and modeling tools developed, to ensure that all knowledge of a method is captured in the metamodel. At the same time, we also discuss limitations of the metamodel and point out constructs for modeling ISD methods more completely with a metamodeling language.

Metamodeling, however, is not as straightforward a process as described above. Rather it is an iterative process in which alternative ways to model method knowledge are tried out, analyzed and compared on the basis of their results (Tolvanen and Lyytinen 1993). For example, when several options for understanding and modeling methods were available (e.g. because of poor or imprecise descriptions of methods) we often tried several alternatives. This naturally led to iteration between metamodeling tasks, and developing and testing many versions of metamodels (at least two versions of each technique were developed). Some of these metamodeling alternatives are discussed in Section 4.3, in which the metamodels of three methods are represented. Also some pieces of the methods already modeled by others (such as in Olle et al. 1991, Welke 1988, Brinkkemper 1990, Smolander 1992, ter Hofstede 1993, Venable 1993, Hong et al 1993, Kinnunen and Leppänen 1994, Marttiin et al. 1993, Ebert and Süttenbach 1997, Süttenbach and Ebert 1997, Booch and Rumbaugh 1995) were used to suggest alternative metamodeling decisions and help validate that all parts of the method knowledge were captured.

Although the inductive research approach followed allows us to generalize requirements for metamodeling languages, it also raises some problems. The first one deals with the expressive power of the chosen metamodeling languages (i.e. OPRR and GOPRR). Their predefined classification schemata will influence our view of methods. Second, the metamodeling languages applied could not describe all method knowledge. However, those parts of the method knowledge which we were not able to classify according to the metamodeling language, and thus not described in the metamodels, were recorded into a diary. These additional descriptions were attached into the metamodels as free-form descriptions and are also partly discussed in Section 4.3 when we evaluate the metamodels. In fact, most aspects of methods which could not be modeled are generalized as requirements for metamodeling languages in Section 4.4. Because the tools were driven by the metamodels developed, these unclassified aspects of methods were not taken into consideration during the tool implementation.

4.2.3 Tool implementation

As mentioned above, method modeling included an examination of how the selected methods could be modeled and supported by a modeling tool. Consequently, methods were adapted into two metaCASE tools, called MetaEdit (Smolander et al. 1990, MetaCase 1994) and MetaEdit+ (Kelly et al. 1996, MetaCase 1996a, 1996b). By adaptation we mean a representation of a given method in a tool in such a way that the CASE tool can support modeling tasks as prescribed by the method (Tolvanen and Lyytinen 1993). In the selected metaCASE tools the adaptation is relatively easy and straightforward since metamodels are almost directly applicable as method specifications in the tool.

More important than the tool support, however, was the possibility to validate the metamodels. In every modeling task, the lack of correspondence between the real-world and the model raises a question of validity. Correspondence between ISD methods and metamodels is no exception. In our case, tool-related metamodeling offered mechanisms to ensure an equivalence between the metamodels at the type level (i.e. IRD definition level) and system models at the instance level (i.e. IRD level) by modeling with the method: each element in a model must have a corresponding element in the metamodel. In this sense, the metamodels include only those concepts that are essential, and can be supported by a modeling tool. This also confirms that the metamodels are as complete as possible. Accordingly, the method examples shown in the following sections include both the type level definitions (i.e. metamodels) as well as some instance level descriptions (i.e. models). Similarly, Wijers (1991) claims that complete metamodels are so complex that a full verification of them without tool support is unmanageable. Tool support allowed us to check that metamodels were complete in terms of the metamodeling language used, and to make queries on the metamodels (e.g. what kind of relationships are possible between selected objects, what properties are shared between elements of a technique, etc.). The method specifications described in the following section were produced by querying the metamodels that were stored in the repository.

4.3 Metamodels for method knowledge

In this section we shall analyze what kind of knowledge ISD methods include and how this knowledge can be represented. This provides a basis for identifying essential metamodeling constructs. Method modeling is illustrated by representing metamodels of three methods, namely Business Systems Planning (IBM 1984), Structured Analysis and Design (Yourdon 1989a) and Unified Modeling Language (Booch and Rumbaugh 1995, Booch et al. 1996, 1997). These methods were selected as examples because of their different nature and area of use, and because they demonstrate various kinds of knowledge incorporated into methods. Metamodels of other methods are available through the CASE tools adapted (cf. appendix), and can be found in Tolvanen and Rossi (1996).

In the following, method knowledge is inspected on two levels. First, on the metalevel, we apply the ten phases of metamodeling by specifying individual techniques and by showing how techniques of a method are interconnected. Second, we briefly demonstrate tool support by showing instance level models as they are represented in a modeling tool.

4.3.1 Business Systems Planning

The modeled BSP was based on IBM's Guide on Information Systems Planning (IBM 1984).

4.3.1.1 Metamodel of BSP

In the following the metamodel is discussed based on the metamodeling steps and the use of the GOPRR metamodeling language.

1) **Identification of techniques.** BSP includes six techniques: five matrix-based techniques that focus on relationships between business processes, data classes, systems, and organizational structures of a company, and a problem sheet to analyze the business problems faced during the development of IS architectures. Although the method also includes other project management oriented techniques such as GANTT diagrams, study work plans, and product lifecycle models, we focus here on design-related techniques. The techniques of the BSP are defined as graph types with the GOPRR language, and can be defined as follows (cf. appendix):

```
Graph types = {   Process/Organization Matrix, Process/System
                  Matrix, Process/Entity Matrix, System/Entity
                  Matrix, System/Organization Matrix, Problem
                  Table}
```

2) **Identification of object types.** In BSP, we distinguish five object types, namely: 'business process', 'data class' (or entity), 'organizational unit', 'system', and 'problem'[18]. A business process is defined as "a group of logically related decisions and activities required to manage resources of the business". A data class denotes "a logical grouping of data related to things that are relevant to the organization" (IBM 1984, p 29). An organizational unit denotes departments involved in the study, and a system either existing or planned information system. The object types can be defined as a set:

```
Object types = {  Business process, Organization, Entity, System,
                  Problem}
```

3) **Determination of properties for each object type.** Each object type is described with its naming property, that conveys the meaning of an instance. Because the name identifies all instances of an object type it is supposed to be unique in order to avoid homonyms. As a result, it is not possible to have for

[18] Hereafter we use apostrophes in the text to denote the types of a method.

example entities or systems with the same name (i.e. values). The 'problem' is further characterized with six properties: a 'problem cause' to specify the reason for the problem, a 'problem result' to specify what is the outcome of the problem, e.g. how an organization must currently perform because of the problem, a 'value' to relate estimated costs to the problem, a 'causing process' to attach a business process that is related to the problem, a 'causing entity' to attach an entity related to the problem, and a 'suggested solution' to describe a proposed strategy for solving the problem.

4) **Identification of relationships.** Each of the matrix-based techniques focuses on specifying some relationships between two different object type instances. Accordingly, a relationship called 'data usage' defines the information needs of processes. A 'system support' relationship defines the systems that each process uses, and it is also applied in another technique to define units of an organization that use systems. Thus, this relationship type describes similar kinds of connections of systems in two different techniques, reducing the number of concepts in the metamodel. A 'responsibility' is used to relate stakeholders to processes: which units of an organization are involved with the process. Finally, a 'usage' relationship specifies the entities managed by the systems.

```
Relationship types = {Responsibility, System support, Data  usage,
                      Usage}
```

5) **Determination of roles.** In BSP all relationships are binary relationships with two different object types. In GOPRR these relationships are defined together with the role types and their cardinality rules. The role types identified are 'uses' and 'used' for the 'data usage' relationship; 'uses' and 'supports' for the 'system support' relationship; 'performs' and 'is part of' for the 'responsibility' relationship; and 'system part' and 'data part' for the 'usage relationship'. Although each object type could have their own role types, and thus we could minimize the number of role types in the metamodel, we choose separate role types for each relationship type. According to the GOPRR data model, each role has a cardinality when bound to object and relationship types (cf. Kelly 1995), defining the minimum and maximum number of role type instances a relationship can have. In BSP all role cardinalities are one to one (1,1): a relationship must be connected to only one instance of each role type. The role types can be defined as a set:

```
Role types = {    Uses, Used, Supports, Performs, Is part of,
                  System part, Data part }
```

6) **Allocation of properties to relationship types and role types.** In BSP all information related to connections between object types is related to relationship types. The 'data usage' relationship type has a property type called 'type of use' since data use can be based on creation in which a process creates an instance of an entity, or on using the information contents of an existing entity (more specific usage types, such as update or read, can also be used).

A 'responsibility' relationship type specifies responsible organizational units for each process. This involvement can be as a major responsible decision maker in a process, having a major involvement in a process, or having some involvement in a process. The 'support' relationship type has a property called 'status' describing whether the system support for the organizational unit or process is current, planned, or hybrid (i.e. current and planned). Finally, the 'usage' relationship type does not carry additional properties. The allocations of property types to other types can be defined as follows:

```
Properties = {  Name, Organization name, Entity name, System      name,
Relationship name, Type, Owner, Costs,          Value adding, Value,
Problem result, Suggested           solution, Problem cause, Causing
process,                Causing entity, Responsibility type, Support
               status, Data usage}

Properties of types = {
      <Organization, {Organization name, Owner}>,
      <Business process, {Name, Type, Value adding, Costs}>,
      <System, {System name}>,
      <Entity, {Entity name}>,
      <Problem, {Problem cause, Problem result, Value, Causing
            process, Causing entity, Suggested solution}>,
      <Responsibility, {Responsibility type}>,
      <System support, {Support status}>,
      <Data usage, {Data usage}>,
      <Usage, {Relationship name}>}
```

7) **Determination of metamodels for individual techniques** builds up the individual techniques by defining the bindings (Kelly 1995, Kelly 1997) from the relationship types as follows:

```
Process/Entity Matrix = {<Data usage, {<Used, {Entity}>,
                                       <Uses, {Business process}>}>}

Process/Organization Matrix = {
            <Responsibility,{<Performs,{Organization}>,
                             <Is part of, {Business process}>}>}

Process/System Matrix = {<System support, {<Uses, {Business process}>,
                                           <Supports, {System}>}>}

System/Entity Matrix =  {<Usage, {  <Data part, {Entity}>,
                                    <System part, {System}>}>}

System/Organization Matrix = {
                  <System support, {<Uses, {Organization}>,
                                    <Supports, {System}>}>}
```

8) **Determination of linkages between separate techniques.** Since the data gathered with BSP is interrelated, each object type except a 'problem' is a part of several other techniques. Because the instances of object types are the same in the matrices, i.e. an instance of the 'organization' object type can exist in two modeling techniques, no actual linkages are defined in the meta-data model. In fact, the use of the same instances between the techniques is more dependent on the process in which the matrices are built, and thus should be represented with a process model.

9) **Determination of the representational part of the method.** BSP presumes a matrix and a table for representing models. Each object is represented through its name shown on the axis of a matrix. The relationships are located in the cells and only two relationship types have symbols according

to the value of their properties: the 'responsibility' property can be denoted with different types of cross (i.e. X, X with a dot, and /), a 'usage' relationship between the 'data class' and the 'system' object types is shown with an X, and the rest of the relationships show the value of the relationship's property.

10) **Analysis and evaluation of the metamodel.** The described metamodel of BSP has some limitations because of the restrictions made to BSP, and because of the metamodeling language applied. Naturally, BSP could be modeled differently. To gather information on the distribution of processes and data classes[19], a 'process' could be further characterized with additional properties, namely 'security', 'auditability', 'volume', and 'responsiveness'. For distribution analysis each 'data class' could be specified with its 'use', 'audit', 'security', 'occurrence' and 'currency' properties. These properties could be attached to the 'process' and 'data class' types in the process/data class matrix, or a separate matrix (i.e. a new technique) could be created. The definition of data classes could also be supported by attaching a new property for categorization: a data class denotes a person, a place, a thing, a concept, or an event. Furthermore, to allow a description of each business process and data class a 'description' property could be attached to the corresponding object types. It must be noticed that the search for new method alternatives can lead to modifications of a method or even to the creation of a wholly new technique.

The metamodeling language also caused limitations to the specifications of BSP. Only those concepts of the method that could be described by the GOPRR model (and can be supported by the CASE tool) were specified[20]. First, we could not describe that each instance of an object type in a matrix must participate at least in one relationship and can participate in many relationships. This rule requires a constraint type for object type multiplicity in the metamodeling language, as discussed in more detail in Section 4.4.1.6. Later extensions of GOPRR include, however, such a multiplicity constraint (Kelly 1997). Second, we did not describe that a data class can not be created by more than one process although it could be possible to divide the data usage relationship into two different types, i.e. 'create data class' and 'use data class' relationship types, and a 'create data class' could have only one connected process.

Third, the metamodel does not specify the grouping of a set of processes, and the composition of a set of data classes, processes, and their connections into a module of an architecture plan. This would require a constraint for describing a composite of objects and relationships as discussed in Section 4.4.2.2. The main reason for excluding these rules of the method was the limited

[19] The analysis of distributed information systems is an extension of a basic BSP and thus excluded from our method analysis. For the same reason, an optional technique for analysis of critical success factors was also excluded. Similarly, techniques for ranking development priorities and project management were not included.

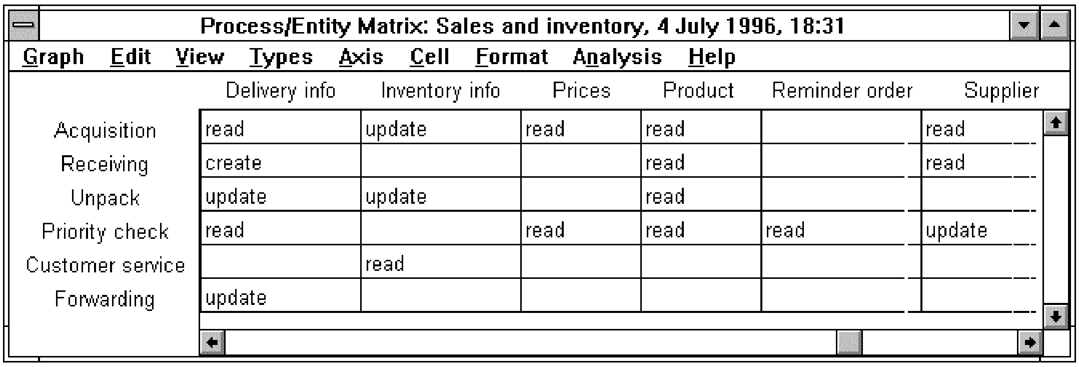
[20] It must be noted that not all method knowledge described with the GOPRR model is included into the set-based definitions. These include cardinality constraints of roles, data types of properties, identifiers of types, and uniqueness of property types. These are, however, implemented in the tool-based metamodels and contained into the discussion of the essential constructs of metamodeling languages in Section 4.4.

tool support: the tool used did not provide support for representing these constructs of a model in a matrix form as described by the method.

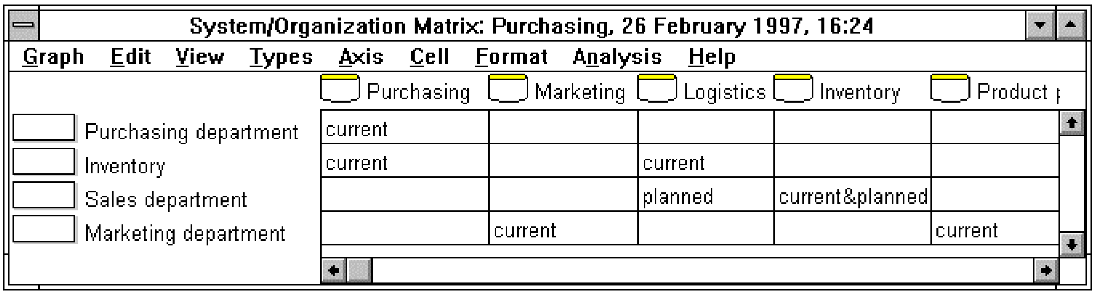
Fourth, we could not describe mandatory properties, so it was not possible to define for example that each business process must have a non-empty name. The specification of mandatory properties would require an additional construct in the metamodeling language (cf. Section 4.4.1.3). Finally, we could not describe the heuristic rules of the method. An example of these rules is the recommended population of 30-60 data classes that could be found in an average organization and specified within the matrix models (i.e. modeling of data classes should not be more detailed in system architecture plans). Similarly, the capturing of multiplicity of types would require an additional metamodeling constraint as discussed in Section 4.4.1.9.

4.3.1.2 Instance models of BSP

Because the methods modeled were also adapted in a modeling tool we can demonstrate the tool support by showing instance level models made with the metamodel developed. The examples of BSP include two matrixes, namely process/entity and system/ organization. Both contain instances of all types of the techniques. In the 'process/entity' matrix the horizontal axis contains entities, and the vertical axis contains processes. The cells of this matrix include data usage relationships: how each process uses each data class. In the 'system/organization' matrix units of an organization and systems are shown. The former on the vertical axis and the latter on the horizontal axis. The cells describe the current status of the system support for each organization.


Process/Entity Matrix: Sales and inventory, 4 July 1996, 18:31
Graph
Edit
View
Types
Axis
Cell
Format
Analysis
Help

	Delivery info	Inventory info	Prices	Product	Reminder order	Supplier
Acquisition	read	update	read	read		read
Receiving	create			read		read
Unpack	update	update		read		
Priority check	read		read	read	read	update
Customer service		read				
Forwarding	update					


System/Organization Matrix: Purchasing, 26 February 1997, 16:24
Graph
Edit
View
Types
Axis
Cell
Format
Analysis
Help

	Purchasing	Marketing	Logistics	Inventory	Product
Purchasing department	current				
Inventory	current		current		
Sales department			planned	current&planned	
Marketing department		current			current

FIGURE 4-2 Instance models described with BSP.

These examples show that the metamodel is complete, because it can represent matrixes as described in the method book, and also follow the constraints of the method. For example, it is not possible to relate two business processes directly through a 'system support' relationship type. Both of the matrixes show instances of all types found from these techniques, expect role instances which do not have any explicit representation in BSP. All in all, the correspondence between the elements of the metamodel and the instance level is clear.

4.3.2 Structured Analysis and Design

Although there are several structured methods available (e.g. DeMarco 1978, Gane and Sarson 1978, Yourdon 1989a) we selected Yourdon's (1989a) version with the original ER model (Chen 1976) for closer analysis. It includes more individual techniques than other dialects with a structured approach and describes the linkages between different techniques in more detail.

4.3.2.1 Metamodel of SA/SD

As with BSP, the metamodel of SA/SD is discussed following the metamodeling steps and documented with the GOPRR metamodeling language.

1) **Identification of techniques.** The main techniques for analysis and design include a data flow diagram (DFD) for describing a network of functional processes, an entity relationship diagram (ERD) for specifying the stored data layout of a system, a structure chart (SC) for describing data interfaces between components, and a state transition diagram (STD) for specifying the time-dependent behavior of a system. According to the GOPRR language these techniques can be defined as the following graph types:

```
Graph types = {   Data Flow Diagram, Entity Relationship Diagram,
                  Structure Chart, State Transition Diagram}
```

2) **Identification of object types.** Unlike BSP, in SA/SD each individual technique has separate object types. DFDs have three object types, 'process', 'store' and 'external' (sometimes also called a terminator). ERDs contain three object types, 'entity', 'attribute' and 'relationship', and both SCs and STDs include only one object type, 'module' and 'state' respectively.

```
Object types = {  Process, Store, External, Attribute, Entity,
                  Relationship, Module, State}
```

3) **Determination of properties for each object type.** Because of the pen-and-paper mentality of SA/SD each object type is described with only a few properties, shown also in the diagrams: only a naming property type is used for most of the object types. The name must be unique among all components to avoid homonyms in the data dictionary. In GOPRR a property type can be renamed with a local name, hence the same property type (i.e. the 'name') can be labeled with a 'store name' and a 'process name' but they still refer to the same property type. Because Yourdon and some other method developers have proposed the use of numbering for processes a 'process ID' is also defined. Other modeling information on object type instances is typically added into an

additional data dictionary including, for example, a definition and examples of each instance. Therefore, in the metamodel a 'documentation' is added for each object type. The 'attribute' object type is furthermore characterized with two additional property types: 'type of data' (e.g. integer, Boolean) and 'constraints' (e.g. primary key, not null).

4) **Identification of relationships.** Since each technique includes different object types the relationship types are separate between techniques as well. In DFDs, SCs and STDs only one relationship type exists. A 'data flow' describes the movement of chunks or packets of information between the object types of DFDs; a 'call' specifies a synchronous hierarchy of modules in SCs; a 'transition' describes possible changes between states. In ERDs two different types of relationships exist, one between an 'entity' and a 'relationship', called 'in relationship', and one between an 'attribute' and the other object types called 'attribute of'. In other words, both entities and relationships can have attributes.

```
Relationship types = {  Data flow, In relationship, Attribute of,
                        Call, Transition}
```

5) **Determination of roles.** In SA/SD all relationships except the 'data flow' are binary with two instances of role types. According to our GOPRR definition of roles, a 'data flow' has two roles called 'sends' and 'receives'. Since data flows can diverge, a cardinality constraint for a receiving role is one-to-many (1,M). In GOPRR, a cardinality constraint defines how many instances of a given role type can exist for a relationship type instance (Kelly et al. 1996). The other role types identified are listed in set form below. The 'sends' and 'receives' roles are also used for the 'transition'; an 'owner part' and an 'attribute part' are used for an 'attribute of' relationship type; an 'entity part' and a 'relationship role' is used for the 'in relationship'; and a 'call from' and a 'call to' for the 'call' relationship type. The minimum and maximum cardinalities for the other roles are one-to-one (1,1): an instance of a role type must be connected to only one instance of a relationship type. The set of role types is:

```
Role types = {Receives, Sends, Entity part, Owner part, Attribute
              part, Relationship role, Call from, Call to}
```

6) **Allocation of properties to relationship types and role types.** In ERD relationship types do not have any explicit properties: they just relate object types together. A 'data flow' can be characterized with its name and 'documentation' property type, a 'call' has a 'call name' and a 'parameters' property type to specify the parameters sent in the subroutine calls. A 'transition' relationship type is characterized with three property types: a 'condition' that must be met to change the state, an 'action' that the system takes when the state is changed, and 'documentation' for adding a textual description of the transition into the data dictionary.

Role types have properties only in the ERD technique. The 'entity part' has a 'cardinality' property for specifying how many times an instance of the 'entity' can relate to another instance of the 'entity' through a relationship. The properties and their allocation to object, relationship, and role types are specified in GOPRR as follows:

```
Properties = {Process name, Process ID, Documentation, Name, Flow
              name, Cardinality, Constraints, Type of data, Attribute
              name, Relationship name, Entity name, Module name,
              Parameters, Call name, Action, Store name, Condition,
              State name}

Properties for types = {
            <Process, {Process ID, Process name, Documentation}>,
            <Store, {Store name, Documentation}>,
            <External, {Name, Documentation}>,
            <Data flow, {Flow name, Documentation}>,
            <Attribute, {Attribute name, Type of data, Constraints,
                        Documentation}>,
            <Entity, {Entity name, Documentation}>,
            <Relationship, {Relationship name, Documentation}>,
            <Entity part, {Cardinality}>,
            <Module, {Module name, Documentation}>,
            <Call, {Call name, Parameters}>,
            <State, {State name, Documentation}>,
            <Transition, {Condition, Action, Documentation}>}
```

7) **Determination of metamodels for individual techniques.** Types in each individual technique are defined by mappings from the relationship types as follows:

```
Data Flow Diagram ={<Data flow,{<Sends, {Process}>,
                                <Receives,{Process, Store, External}>}>,
                    <Data flow,{<Sends, {Store, External}>,
                                <Receives, {Process}>}>}
```

This definition follows the rules of DFDs preventing data flows between stores and externals. Real-time extensions to the DFD could be included by defining the related types and bindings as follows:

```
Real-time Data Flow Diagram = {
      <Signal flow, {<Event from, {Control}>,
                     <Signal to, {Process, External, Control,
                                  Buffer}>}>,
      <Signal flow, {<Event from, {Process, External, Buffer}>,
                     <Signal to, {Control}>}>,
      <Discrete data flow, {<Sends, {External, Store, Buffer}>,
                            <Receives, {Process}>}>,
      <Discrete data flow, {<Sends, {Process}>,
                            <Receives, {Process, Store, External,
                             Buffer}>}>,
      <Deactivation flow, {<Event from, {Control}>,
                           <Deactivated, {Process}>}>,
      <Continuous data flow, {<Sends, {Process}>,
                              <Continuous, {Process}>}>,
      <Activation flow, {<Event from, {Control}>,
                         <Activated, {Process}>}>}

Entity Relationship Diagram = {
            <Attribute of, {<Owner part, {Entity, Relationship}>,
                            <Attribute part, {Attribute}>}>,
            <In relationship, {<Entity part, {Entity}>,
                               <Relationship role, {Relationship}>}>}

Structure Chart = {<Call, {<Call from, {Module}>,
                           <Call to, {Module}>}>}
```

```
State Transition Diagram = {<Transition, {<Sends, {State}>,
                                          <Receives, {State}>}>}
```

8) **Determination of linkages between separate techniques.** Unlike in BSP, techniques of SA/SD model a system using techniques that do not share types, i.e. use the same type in several techniques. Thus, each technique focuses on separate types in describing the system. Instance information can, however, be shared as discussed below. In the following the top-down approach and balancing rules of SA/SD are modeled with GOPRR using explosion and decomposition based linkages. First, because the DFD forms the dominant view of the system, each process can be decomposed into a submodel of a DFD (i.e. functional decomposition). Processes can also be exploded into a SC to specify subroutines and modules of a process, and into a STD to specify which transitions change the state of the controlling process and send or receive data flows from other processes. Second, as data stores of DFDs and entities of ERDs must be balanced, each data store in a DFD can be exploded into an ERD to specify the schemas of the database. Matching store names to entity names on the instance level is achieved in GOPRR by using the same property type for both types. Only their local name may differ (i.e. the metamodel definition shows local names for store and entity, but they refer to the same property type). Third, the STD allows decomposition to partition states. The same partitioning could be applied for the SC as well. It is not defined here, because a single SC should not include modules of several processes. Instead, according to SA/SD the process should be decomposed into subprocesses that have a simpler structure in terms of a number of modules and their subroutines.

```
Data Flow Diagram:
Explosions = {<Store, {Entity Relationship Diagram}>,
              <Process, {Structure Chart, State Transition Diagram}>}
Decompositions =  {<Process, {Data Flow Diagram}>}

State Transition Diagram:
Decompositions =  {<State, {State Transition Diagram}>}
```

9) **Determination of the representational part of the method.** In SA/SD all techniques are graphical notations except the process specification. During method implementation all graphical representations were defined as illustrated in Figure 4-3. Notational aspects that were not possible to define during the adaptation are the representation of relationships in a functional decomposition (i.e. relationships mapping to parent diagram and represented as "dangling" links with one connected object type only), and the tree structure of the SC diagram (i.e. each calling module should be located above the modules called). Other techniques do include similar location recommendations though. For example, in DFD externals are most often placed on the sides of a diagram.

10) **Analysis and evaluation of the metamodel.** Because of the several informal definitions of SA/SD, it is possible to model techniques in many ways. For example, a 'decision' in the SC could be included into the metamodel either by specifying a new role type, such as a 'decision', by adding a relationship type, such as a 'call based on decision', or by adding a property type for the 'call' relationship (see also Welke 1988). Moreover, supertype/subtype

relationships and indicators for associative object type in ERD are not modeled[21]. Also, the data dictionary and process specification are not included, although one can form a dictionary by using information entered into the documentation properties attached to types (i.e. using form conversion mechanisms, cf. Section 2.3.2).

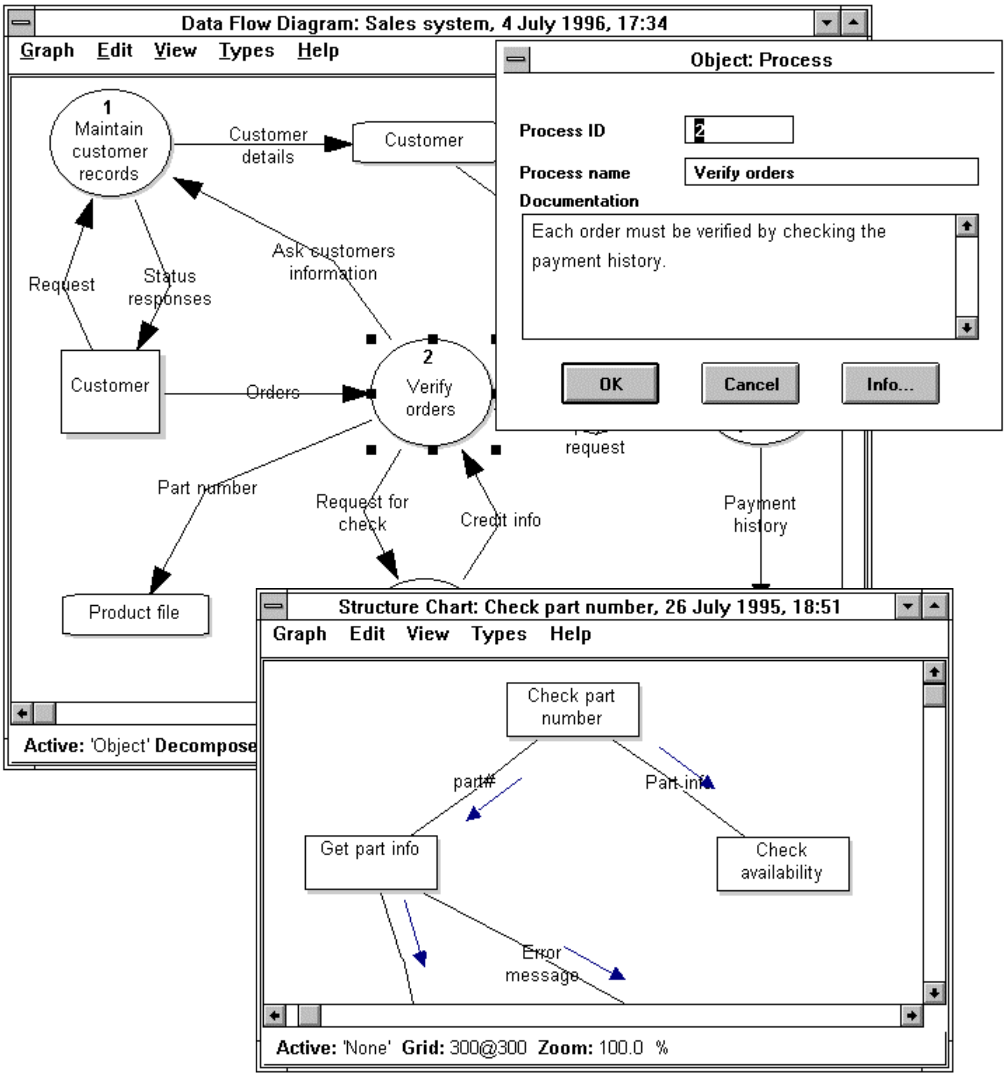
A more important aspect of evaluation is how completely the SA/SD method is modeled and therefore implemented into a CASE tool. Because of limitations in GOPRR and OPRR, some aspects of SA/SD were not captured and supported by tools. First, in a data flow diagram a process should be described in more detail either with an additional subdiagram or with a structure chart, but not by using both techniques. Such a restriction between explosion and decomposition can not be specified with GOPRR. These require a more detailed definition (cf. Section 4.4.2.3). Second, iteration calls between the modules could not be restricted with GOPRR, requiring an additional metamodeling constraint (cf. Section 4.4.1.8). Third, transformations that could be automated, such as transformations from a DFD into a SC at the same level, are not supported. Fourth, and similarly to the BSP metamodel, identifiers and uniqueness of property values were not defined adequately. It was not possible to restrict these to the scope of a single diagram. For example, according to the metamodel each name of a state is unique among all diagrams, not just inside one diagram as required. To model these rules a metamodeling language should allow the specification of different scopes for the method rules (cf. Section 4.4).

Fifth, the metamodel does not support all the balancing rules of SA/SD, such as a correspondence of data stores to entities (based on their names) or correspondence of conditions in state transition diagrams to data flows. Furthermore, dependencies of the property values should be included into the metamodel as well: the possible values for a condition in a state transition diagram can be only those defined in data flows that a related control process receives. Modeling of the balancing rules and dependencies of related instances would require additional metamodeling constructs as discussed in Section 4.4.2.4. Finally, the metamodel does not include multiplicity rules defining the number of roles an object can participate in (cf. Section 4.4.1.5).

[21] Modeling of these concepts, however, is taken into account and specified in the metamodel of UML (cf. Section 4.3.3).

4.3.2.2 Instance models of SA/SD

Figure 4-3 illustrates an example of the use of two techniques in the modeling tool. The upper window shows some processes and data flows of a sales system. To show instances of property definitions and a decomposition, a process called verify orders is viewed with its properties, and the status bar of the window shows that the process is decomposed into a subdiagram. The lower window shows a structure chart.



Data Flow Diagram: Sales system, 4 July 1996, 17:34

Object: Process

Process ID 2

Process name Verify orders

Documentation
Each order must be verified by checking the payment history.

Active: 'Object' Decompose

Structure Chart: Check part number, 26 July 1995, 18:51

Active: 'None' Grid: 300@300 Zoom: 100.0 %

FIGURE 4-3 Two instance models of SA/SD: a data flow diagram and a structure chart.

4.3.3 Unified Modeling Language

After modeling two relatively simple methods we shall next model one of the most complex methods found, the Unified Modeling Language (UML). Because the standardization of UML was under development at the time of metamodeling, the metamodels discussed here are based on several publications about the method and its versions (cf. Booch and Rumbaugh 1995, Booch et al. 1996, 1997).

4.3.3.1 Metamodel of UML

As with the earlier metamodeling cases, specification of UML follows the metamodeling process described in Section 4.2.2.

1) **Identification of techniques.** UML includes as its main techniques a class diagram, a collaboration diagram and a use case diagram. Other diagram types[22], mentioned below as graph types, have a simpler structure and they are not applied as often. We also make some simplifications in techniques by joining the category diagram and class diagram as they include the same types. The only difference is that a category diagram shows instances of a 'category' only, whereas class diagrams can include instances of both a 'class' and a 'category'. The composite diagram has been included in the graph types to support context diagrams for instances of classes (called composites in UML). In a later extension (Booch et al. 1996, 1997) all techniques are expected to use categories, renamed now as package, for organizing large models.

```
Graph types = {Class diagram, Use case diagram, Collaboration diagram,
               State diagram, Component diagram, Deployment diagram,
               Operation table, Composite diagram}
```

2) **Identification of object types.** Because of the unfinished documentation and relatively complex conceptual structure of UML, several alternative interpretations and consequent modeling decisions of object types can be made. First, template classes are not distinguished as a separate type because template parameters denoting generic classes are not mandatory, i.e. a 'class' without defined template parameters is considered to be an ordinary class. Second, a utility class is not identified as its own object type. Rather it is distinguished with a property attached to each class (discussed in the next task).

In addition to the object types of a class diagram, the GOPRR metamodel includes some constraints of UML modeled as object types, such as an alternative association (i.e. 'or-constraint') and 'parallel inheritance'. Parallel inheritance hierarchies, however, could also be described with a property type referring to another hierarchy of the same superclass (i.e. to another inheritance relationship). The fourth major modeling alternative would be to distinguish objects of a class diagram and a collaboration diagram into separate types instead of using a single type in both techniques. Overall, the following set of object types describes one possible outcome of object type identification.

```
Object types = { Class, Instantiated class, Object, Category, Or-
                 constraint, Parallel inheritance, Actor, Use case,
                 Operation specification, Note, Node, Specification,
                 Main program, Body, State, Stop, Start}
```

[22] We also excluded the event trace diagram since it is based on the same underlying semantics as the collaboration diagram (Booch and Rumbaugh 1995).

3) **Determination of properties for each object type.** Compared to other types of methods, object-oriented methods in general, and UML in particular, have more property types. For this reason they are specified here distinct from other metatypes. In contrast to the methods above, in UML some properties are atomic, single valued, whereas others consist of a set of other property types. For example, an attribute of a class has a more detailed internal structure with its own property types (such as an initial value and a data type of an attribute). In the following we first define the properties of object types and later the non-atomic properties.

A class is identified by its name and by the name of a possible category to which it belongs. Although class names must be unique in the enclosing category, the same class name can be used as a type of a parameter. In GOPRR this is modeled by using both global and local names for property types (see also the metamodel of state diagrams in Section 3.3.3). For example, a property type 'class name' can be renamed with a local name for describing a data type of a parameter.

An 'Is utility?' property type is used to define global attributes and operations; 'overridability' (e.g. deferred, leaf, extensible, virtual) is used to define how a class may be overridden by a subclass. 'Template parameters' is used for defining generic classes called templates in UML. An object type 'instantiated class' refers to a template class in two ways: properties of 'instantiated class name' refer to a name of a template class, and 'values' is a set of template parameter values. An 'instantiated class' can not have its own attributes or operations as these are derived from the template class. Categories are identified by a 'category name' and it has an additional 'documentation' property type. These are also added to 'class' and to 'instantiated class'.

A 'class name' is the same property for 'object' and 'class' object types, showing that an object can not belong to other classes than those defined. Hence, objects are dependent on the existence of classes. Other property types for an 'object' are a 'multiplicity' type for specifying the number of instances a related class can have at a time; and 'values' which denotes instances of an attribute of a class. Notes are specified by one property type only, named here 'description'.

Other object types of UML are identified and described just by their name and additional 'documentation' property type. Only a 'node' object type can have a 'multiplicity' property type, and a 'state' object type has 'attribute values' and 'operations' property types. The name spaces of identifying properties are distinguished by using separate property types for identifiers of object types, except in a dependency diagram. Here, the metamodel uses the same property type 'name' for identifying several object types. By defining the instances of this property type as unique we can specify that the same value for a property can not be used to identify other instances of object types in a deployment diagram. Thus, instances of 'specification' and 'body' can not have the same value for this property.

```
Properties for object types = {
      <Class, {Class name, Is utility?, Category, Stereotype,
               Overridability, Attributes, Operations, Documentation,
               Template parameters}>,
      <Category, {Category name, Documentation}>,
      <Instantiated class, {Instantiated class name, Values,
                            Documentation}>,
      <Object, {Object name, Class name, Values, Multiplicity}>,
      <Note, {Description}>,
      <Use case, {Use case name, Documentation}>,
      <Actor, {Actor name, Documentation}>,
      <State, {State name, Attribute values, Operations,
               Documentation}>,
      <Operation specification, {Name, Responsibilities, Inputs,
            Returns, Modified objects, Preconditions, Postconditions}>,
      <Specification, {Name, Documentation}>,
      <Main program, {Name, Documentation}>,
      <Body, {Name, Documentation}>,
      <Node, {Name, Stereotype, Multiplicity, Documentation}>}
```

In UML, some property types have an internal structure. Each attribute has a basic format requiring property types for an 'attribute', namely an 'attribute name', a 'data type', an 'attribute type', and an 'initial value'. Similarly, an 'operation' has an 'operation name', 'parameters' and a 'return value'. The 'parameters' property type is a collection consisting of parameters each having three property types, namely a 'parameter name', a 'parameter type' and a 'default value'. To support specific programming language constructs some additional property types are attached for both attributes and operations. These include 'constraints' and 'visibility' (e.g. public, private, protected in C++). Moreover, 'overridability' (applicable for both classes and operations) and 'method body' are also attached for operations. Finally, parameters for template classes have a name and a type.

```
Derived property types = {
      <Attributes, {<Attribute, {Attribute name, Data type, Attribute
                                 type, Initial value, Constraints,
                                 Visibility}>}>,
      <Operations, {<Operation, {Operation name, Parameters, Return
                                 value, Constraints, Visibility,
                                 Overridability, Method body}>}>,
      <Parameters, {<Parameter, {Parameter name, Parameter type,
                                 Default value}>}>,
      <Template parameters, {<Template parameter, {Parameter name,
                                 Parameter type}>}>}
```

4) **Identification of relationships.** In UML, there is a clear distinction between methods which apply several relationship types, and those using only one or two relationship types. More specifically, class diagrams and use case diagrams apply more relationship types than other techniques. In the following definition the number of necessary types is reduced by applying a 'dependency', a 'note connection' and an 'inheritance' relationship types in several techniques instead of having their own variants in each technique.

```
Relationship types = { Association, Ternary association, Aggregation,
                       Instantiation, Dependency, Note connection,
                       Inheritance, Connection, Message link, Uses,
                       Extends, Participation, Transition}
```

5) **Determination of roles.** As in other methods, binary relationships dominate in UML. Some of the relationships are, however, n-ary at the instance level and some also at the type level. At the instance level, an 'inheritance' and an 'association', like their subtypes 'aggregation' and 'ternary association', necessitate roles with a maximum cardinality greater than one. For example, a 'specialization' role type in an 'inheritance' relationship can have more thanone instance. Hence the maximum cardinality for a 'specialization' role type is many. At the type level, UML has optional role types connected to the relationship types which already have two other role types. Hence the minimum cardinality constraint of GOPRR is defined here as one for mandatory roles and zero for optional role types. Since all optional role types (i.e. 'link attribute', 'or' and 'parallel') can occur only once in a related relationship type instance, the maximum cardinality defined for them is one.

As with relationships the same role types are used in several techniques, such as 'specialization' and 'note part', reducing the size and complexity of the metamodel. The set of role types is:

```
Role types = {Instantiates, Is instantiated, Note part, Object part,
              Has dependents, Is dependent, Associates, Qualified
              association, Link attribute, Specialization,
              Generalization, Parallel, Or, Part, Whole, Sends,
              Receives, Uses, Participates, Is used, Extends, Is
              extended, Receive message, Send message, Connected}
```

6) **Allocation of properties to relationship types and role types.** Compared to the earlier methods, UML relationship and role types have many properties. This is also highlighted by adding some additional programming language specific property types. For example, the 'inheritance' relationship type has property types 'visibility' and 'virtual?' denoting inheritance structures in C++. In addition, associations can be defined as derived, messages have a 'sequence' for numbering, 'arguments' sent and values returned along the message, an 'indicator' for describing exclusive iteration and condition indicators of a message, and a 'link type' for typing message links (e.g. association, argument, global, variable).

Association role types have several property types. These include a description of the role names, their visibility (i.e. public, private, protected), the existence of an explicit order of the set of classes associated with a single object, how the role outside the class is accessed (i.e. read, write, both or none), and how many instances of the class can be associated with one instance in another class with 'multiplicity'. Multiplicity is also needed for describing how many components an aggregate class can have, and how many aggregates a component class can be part of.

In the 'state diagram', the 'transition' is also specified with several property types: an 'event' triggering a state transition, a 'condition' to be met before the state transition can occur, and an 'action' resulting in a change in the state of the object. The action is realized by sending a message to an object or modifying a value of an attribute. Both events and actions have arguments, which refer to a specific value corresponding to a parameter. Finally, a role type 'receive message' has an additional 'adornment' property.

```
Properties for relationship and role types = {
      <Inheritance, {Visibility, Virtual?, Discriminator}>,
      <Association, {Association name, Is derived?}>,
      <Ternary association, {Association name, Is derived?}>,
      <Aggregation, {Name}>,
      <Associates, {Role name, Visibility, Access, Multiplicity,
                    Ordered?}>,
      <Part, {Multiplicity}>,
      <Qualified association, {Role name, Visibility, Access,
                                Multiplicity, Ordered?, Qualifier}>,
      <Whole, {Multiplicity}>,
      <Message link, {Sequence, Name, Arguments, Return type, Link
                       type, Indicator}>,
      <Receive message, {Role name, Adornment}>,
      <Transition, {Event name, Arguments of event, Condition, Action,
                     Arguments of action, Documentation}>,
      <Connection, {Name}>}
```

7) **Determination of metamodels for individual techniques.** In the following bindings for each technique are described. In the class diagram, both associations have four roles, of which a 'link attribute' and 'or' constraint are optional. Inheritance hierarchies between categories and classes are distinguished. Thus, inheritance hierarchies between these object types can not be mixed. Both of these hierarchies have an optional 'parallel' role showing to simultaneous specializations. A use case diagram also allows inheritance among categories, although use cases could also be considered as classes with their own inheritance hierarchy. Moreover, use cases can be related with 'uses' and 'extends' relationships and connected to actors with 'participation' relationships.

Other graph types have simpler bindings: a composite diagram is used for describing the associations and aggregations of objects in the context of a class, and a collaboration diagram focuses on message sending between the objects. A state model is similar to that modeled with SA/SD and an 'operation table' has no bindings. Finally, the component and deployment diagrams describe the physical design and apply a dependency structure among all object types as described below.

```
Class Diagram = {
      <Instantiates, {<Instantiates, {Class}>,
                      <Is instantiated, {Instantiated class, Object}>}>,
      <Dependency, {<Has dependents, {Category, Class}>,
                          <Is dependent, {Class, Category, Object,
                                          Instantiated class }>}>,
      <Note connection, {<Note part, {Note text}>,
                         <Object part, {Object, Class, Category}>}>,
      <Inheritance, {<Specialization, {Class}>,
                     <Generalization, {Class}>,
                     <Parallel, {Parallel inheritance}>}>,
      <Inheritance, {<Specialization, {Category}>,
                     <Generalization, {Category}>,
                     <Parallel, {Parallel inheritance}>}>,
      <Inheritance, {<Specialization, {Instantiated class}>,
                     <Generalization, {Instantiated class}>,
                     <Parallel, {Parallel inheritance}>}>,
      <Aggregation, {<Whole, {Class}>,
                     <Part, {Class}>}>,
      <Association, {<Associates, {Class}>,
                     <Associates, {Class}>,
                     <Or, {Or-constraint}>,
                     <Link attribute, {Class}>}>,
      <Ternary association, {<Associates, {Class}>,
                             <Associates, {Class}>,
                             <Or, {Or-constraint}>,
                             <Link attribute, {Class}>}>}

Use case diagram = {
      <Dependency, {<Has dependents, {Category}>,
                    <Is dependent, {Category}>}>,
      <Uses, {<Uses, {Use case}>,
              <Is used, {Use case}>}>,
      <Note connection, {<Note part, {Note text}>,
                         <Object part, {Actor, Use case, Category}>}>,
      <Extends, {<Extends, {Use case}>,
                 <Is extended, {Use case}>}>,
      <Inheritance, {<Specialization, {Category}>,
                     <Generalization, {Category}>}>,
      <Participation, {<Participates, {Actor}>,
                       <Participates, {Use case}>}>}

Composite diagram = {
      <Aggregation, {<Whole, {Object}>,
                     <Part, {Object}>}>,
      <Association, {<Associates, {Object}>,
                     <Associates, {Object}>}>}

Collaboration Diagram = {
            <Message link, {<Send message, {Object}>,
                            <Receive message, {Object}>}>,
            <Note connection, {<Note part, {Note text}>,
                               <Object part, {Object}>}>}

State Diagram = { <Transition, {<Sends, {Start, State}>,
                                <Receives, {Stop, State}>}>,
                  <Note connection, {<Note part, {Note text}>,
                                     <Object part, {State}>}>}

Component Diagram = {
      <Dependency, {<Has dependents, {Category, Specification, Main
                                      program, Body}>,
                    <Is dependent, {Category, Specification, Main
                                    program, Body}>}>,
      <Note connection, {<Note part, {Note text}>,
                         <Object part, {Category, Specification, Main
                                        program, Body}>}>}

Deployment Diagram = {
      <Connection, {<Connected, {Node}>,
                    <Connected, {Node}>}>,
      <Dependency, {<Has dependents, {Node}>,
                    <Has dependents, {Node}>}>,
      <Note connection, {<Note part, {Note text}>,
                         <Object part, {Node, Category}>}>,
      <Dependency, {<Has dependents, {Category}>,
                    <Is dependent, {Category}>}>}
```

8) **Determination of linkages between techniques.** Because UML suggests several modeling techniques, linkages between them are vital to integrate models. The following explosion and decomposition operators were specified: categories can be attached to class diagrams which can also contain other categories. A class can also be exploded to a collaboration diagram showing the interaction between its objects, to an operation table for describing a functional model (also applicable for actors, use cases, states and for objects of a collaboration diagram), and to a state model for describing the temporal evolution of an object of a given class. A class can also be decomposed into a

composite diagram to describe a specific context for its instances. Similarly classes, actors and use cases can have related state models through an explosion link. State models can be nested through decompositions of states into substates. Finally, a decomposition of categories into instances of the same graph type is added to class, use case, component and deployment diagrams.

```
Class Diagram:
Explosions = {
      <Class, {Collaboration diagram, Operation table, State diagram}>}
Decompositions = {<Class, {Composite diagram}>,
                  <Category, {Class diagram}>}

Use Case diagram:
Explosions = {<Actor, {Operation table, State diagram}>,
              <Use case, {Operation table, State diagram}>}
Decompositions = {<Category, {Use Case diagram}>}

Collaboration Diagram:
Explosions = {<Object, {Operation table}>}
Decompositions = {}

State Diagram:
Explosions = {<State, {Operation table}>}
Decompositions = {<State, {State diagram}>}

Component Diagram:
Explosions = {}
Decompositions = {<Category, {Component diagram}>}

Deployment Diagram:
Explosions = {}
Decompositions = {<Category, {Deployment diagram}>}
```

9) **Determination of the representational part of the method.** On the notational side, the following aspects could not be represented. First, nested forms could not be specified in the same diagram with categories, composites and state diagrams. As the definitions show this was partly solved by using explosion and decomposition structures, even though the relationships between the components of two or more categories can not then be represented. Second, concurrent substates could not be represented by partitioning the state symbol. Third, different symbols for classes could not be defined based on the values of their properties, such as an additional box above the class symbol if parameters are defined (i.e. a symbol for the parameterized class).

10) **Analysis and evaluation of the metamodel.** In addition to the representation dependent aspects, the metamodel of UML could have been made differently. Some aspects of the textual method description were not included, since they were not supported by the parallel metamodel definition given by Booch and Rumbaugh (1995). Metamodels could also include additional programming language specific constructs. In fact, Booch and Rumbaugh, even though seeking for a standard notation for object-oriented methods, recommend situation-bound modifications to align concepts closer to a specific programming language (e.g. Booch and Rumbaugh 1995, p. 4).

A more important aspect of evaluation is how completely UML could be adapted in a CASE tool. This aspect is discussed in the following. The modeling of UML emphasizes the need of scopes for identity and uniqueness of properties. An identifier consisting of two property type instances could not be modeled, nor could the dependency between partial identifiers: the same category should not have more than one class with the same name, and the same class should not have more than one instantiated object with the same name. Modeling of these would require additional constructs in the metamodeling language.

The design orientation of UML and its close relationship to programming languages necessitates support for the naming policy of attributes and operations, such as the naming of classes in Smalltalk with a capitalized first letter (Hopkins and Horan 1995). Some of these syntax definitions would require dynamic changes in other property types. For example, if a parameter is not defined a colon should be omitted from the operation specification. These would require a specific syntax for property values and for checking of property type values (cf. Section 4.4.1.4). The GOPRR model could not describe the multiplicity rules which were applied in the OPRR metamodel. For example, the UML metamodel does not include restrictions on multiple inheritance (i.e. a class can participate several times in a specialization role) or that a class can be part of multiple classes through the aggregation relationship. To model multiplicity rules of methods an additional metamodeling construct would be needed (cf. Section 4.4.1.6).

Because of the wide variety of different graph types, the modeling of UML also highlights requirements to model interconnected methods and complex objects. In interconnections, the metamodel does not allow an operation to be exploded into an operation table. Instead this is carried out by exploding the whole class. Nor can we represent that each state diagram needs to be connected to a class diagram (through an explosion), or to a higher-level state (through decomposition). Modeling of these would necessitate a more detailed specification of interconnections (as discussed in Section 4.4.2.3). In a similar vein, complex objects could not be specified adequately with a decomposition, or an explosion. An example of such a situation is when a component in a complex object (e.g. a substate) can belong to many aggregate objects (e.g. to composite states). Among states, the substates can belong only to one composite state, whereas an object can belong to more than one composite class (Booch and Rumbaugh 1995, p 11, 33). Modeling these complex objects completely would require additional constructs in metamodeling languages (as discussed in Section 4.4.2.2). Finally, modeling UML requires a specification of related properties; i.e. two or more property instances have the same value. An example of this is the requirement specifying that a state model should not have an action that is not defined as an operation in the related class, or that a state should not have attribute values that are not equivalent to those defined in a related class. Similarly, an operation in a class diagram and a message in a collaboration diagram can have several common values, such as name and arguments, which refer to the same property instances. To model this sharing of the same values among different types would require additional constructs in the metamodeling language(cf. Section 4.4.2.4).

4.3.3.2 Example models

Part of the tool adaptation for UML is shown in Figure 4-4. The figure illustrates a class diagram for a banking application in which all classes belong to a stereotype interface that enables code generation for Corba IDL (Iona 1997). The cardinalities of the aggregation relationship are shown between classes named a bank and an account: a bank has multiple accounts, but each account must belong to only one bank. Inheritance relationships based on single and multiple inheritance are shown as lines with an arrowhead. Multiple inheritance is illustrated as the class named premium account inherits both current account and deposit account.

Since UML includes more complex data types than earlier methods, we show dialogs below the class diagram to illustrate the properties of a class and an attribute. The property dialog of an attribute newAccount refines the instance selected from the attribute list of the bank class.

newAccount

Object: Attribute [UML]: Attributes in Class [UML]

FIGURE 4-4 An example class diagram of UML.

4.3.4 Summary

Metamodeling, if properly performed, leads to a detailed understanding of the phenomena under examination. In this section we inspected the conceptual structure of methods in computer-aided modeling tools. Of all 17 methods modeled, three were taken into a closer examination. The structures of the methods were identified, classified and represented with meta-data models. Moreover, CASE tool support was created using the metamodel to validate the method specifications. These efforts form the background for our study of requirements for method modeling languages in the next section.

4.4 Requirements for metamodeling languages

In this section we shall investigate requirements for metamodeling languages using an inductive method. In the inductive analysis (Patton 1990) the underlying patterns, categories and rules of modeling techniques are used to identify and generalize metamodeling requirements. In our case the identification of method knowledge was based on an examination of 17 ISD methods.

Although we could examine general requirements for (meta)modeling, like simplicity and ease of reading (cf. Brinkkemper 1990, Venable 1993), our emphasis is on constructs which increase the modeling power of meta-data models: what constructs are needed to extend available semantic data models to capture and represent method knowledge. Our focus is on providing explicit constraints which deal with a combination of mechanisms provided by the data modeling language (Brodie 1984). Because our focus is on semantic data models, and mostly on ER extensions, the inherent constraints are the basic properties of the semantic data model. For example, there is a distinction between entities and relationships.

Table 4-2 summarizes the proposed metamodeling constructs derived from our inductive analysis. In the following each construct is described in more detail: in section 4.4.1 we describe constructs essential for modeling single techniques, and in section 4.4.2 constructs related to modeling multiple interconnected techniques and complete methods. When describing each metamodeling construct, we show examples of method knowledge that indicate the need for that construct. The examples of method knowledge are based on the methods summarized in Table 4-1.

In addition to the metamodeling constructs, modeling methods requires specifications of the checking mode and recognition of the different scopes for the constraints. These are described in more detail below.

TABLE 4-2 Essential constructs of a metamodeling language.

Metamodeling construct	Checking	Scope of metamodeling constructs		
	Passive or active	method	model	dependent type
Identifying property	a	x	x	x
Unique property	a	x	x	x
Mandatory property	a	x		
Data type of properties	a	x		
Cardinality	a	x		
Multiplicity	a & p	x	x	
Multiplicity over several role types	p	x	x	
Cyclic relationship	a	x		
Multiplicity of type	a & p	x	x	
Inclusion	a		x	
Complex objects	a	x	x	
Explosion	a & p	x	x	
Polymorphism	a & p	x	x	x

By checking mode we mean the strategy to guarantee that the rules of the method defined are followed. The checking is performed on the instance level data either actively or passively. In active checking the rules of a method are mandatory and must be satisfied at all times. In practice actively checked rules are verified each time rule-related instances are created, changed, or removed. An example of active checking is the identifier of a process in a data flow diagram. Because processes must always have identifying numbers, the construct of a metamodeling language describing an identity must be an active constraint. Passive checking, on the other hand, refers to rules of a method which are not mandatory, and are only checked at the modeler's request. Typically, passive constraints are applicable only for completed models. Table 4-2 summarizes which checking types are useful with the proposed metamodeling constructs. In addition to supporting computer-aided checking, a passive constraint type is needed to model methods which allow the modeler to specify incomplete or conflicting models, or when active checking is not possible in practice, e.g. because of the heavy demands on computational resources if the rule was checked. Typical examples of method rules which are passively checked are instructions and recommendations, such as the number of activities in ISAC graphs (Lundeberg et al. 1982), or that each data class in BSP should have only one relationship which creates it. In practice, a data class in BSP may be created by several processes. Hence, to allow the latter situation the 'data usage' relationship type in the metamodel should be specified as being passively checked (IBM 1984). Instead of rigidly enforcing consistency rules, passive constraints can provide some advantages by providing information about possibly conflicting data (Nuseibah et al. 1993).

The scope of method knowledge denotes the instance space in which the rules of the method are relevant. In contrast with what is assumed in most metamodeling languages, not all rules of method knowledge can be specified within a single scope. For example, the uniqueness of a class name and a state name have different scopes (see also the example metamodels in Section 3.3.3): the former is usually unique among all classes defined in all models, even among different techniques (e.g. Henderson-Sellers and Edwards 1994), but the latter need only be unique within a single state model (cf. Embley et al. 1992), or in the context of the dependent class (cf. Booch et al. 1997). The need for different scopes of capturing method knowledge is also recognized by other metamodelers (Hochstettler 1986, Hofstede 1993, Sütenbach and Ebert 1997). Among the methods analyzed we identified three different kind of scopes for metamodeling constructs: a method, a model and a dependent type. Accordingly, a metamodeling language should recognize these scopes. In the following, each scope is described in more detail using uniqueness of properties as an example.

1) **A method** is the largest scope used. It refers to rules that are relevant in all instances of a method used in an ISD project. For example, in many object-oriented methods the name of a class must be unique within all the models made (e.g. Booch 1991): Two classes with the same name can not refer to different classes. Also, if two or more models (e.g. an object model and an inheritance graph (Coleman et al. 1994)) describe a class which has the same name they must denote the same class, even if some of their property types, or the relationships that they participate in are different. As a result, in a metamodel a 'name' of a 'class' must be specified uniquely within the scope of a whole method.

2) **A model** refers to rules that are enforced for all instances within the scope of a single model (based on one technique, or schema types (as in Hofstede 1993). For example, inside a single state transition diagram the names of states must be unique, but other diagrams can have states with the same name which, however, refer to different states. Thus, a uniqueness constraint within the scope of a whole method would be too restrictive and would not describe the method knowledge adequately.

3) **A dependent type** is the smallest scope which focuses on constraints that are relevant for instances that are dependent on the existence of other instances (i.e. masters). An example of a dependent uniqueness rule can be found from an entity relationship diagram in which an entity can not have two different attributes with the same name. However, attributes with the same name denoting different instances are allowed within the scope of a model and a method. For example, another entity can have an attribute with the same

name, but denoting still to a different attribute. Thus, naming of attributes is dependent on the master element (i.e. in our example of an instance of the entity).

The scopes are embedded within each other, and therefore a more general scope includes limited scopes: if a scope is defined for the whole method it includes also scopes for a model and for a dependent type. For example, a constraint for unique class names within the method scope prevents also the use of the same class names inside a model. Any scope, however, does not exclude the possibility of defining other scopes for the same metamodeling construct. Consider a multiplicity construct as an example. At the scope of a model a data store does not need to participate in instances of both 'receive' and 'send' role types of the 'data flow' relationship type, but in the scope of a whole method, each 'data store' must participate in instances of both role types. This means that among IS models an instance of the 'data store' object type must have both updating and reading data flows but in a single diagram at least one data flow must be connected to the data store (i.e. unconnected data stores should not be included). Hence, the multiplicity rule can be defined separately for each scope. The use of method scopes are summarized in Table 4-2, and discussed in more detail in the following subsections.

4.4.1 Modeling single techniques

In this section we shall describe the metamodeling constructs that were needed to model the 72 individual techniques selected (cf. Table 4-1). The first four constructs will focus on specifying characteristics of property types, and the next four on connections between object types. Finally, the last construct address the multiplicity of types.

4.4.1.1 Identifying property constraint

Once types of a method have been introduced, their instances must be identified by using an identifier inside the scope. For example, in a class diagram a 'class' has a 'name' (e.g. Rumbaugh et al. 1991), in a data flow diagram a 'process' has a 'process ID', and in BSP (IBM 1984) an 'entity' has an 'entity name' as an identifying property. The identity of instances is typically based on an identifying property. Relationship type instances can be identified based on the participation with object type instances and/or its properties. In the former case, relationships do not have identifying properties, or in many cases they have no properties at all. An example of the latter case is message passing diagrams (e.g. Coleman et al. 1994) in which messages are distinguished by a number specifying timing and the sequence of message passing because several messages can be exchanged between the same object type instances.

Because some object types, like 'start' and 'end' states (e.g. in Booch 1991, Booch et al. 1996) do not have properties they must be identified based on the context (e.g. a start state of a given state transition model), or have an internal identifier. The former means that the context forms another part of the identifier. The latter one is typically used in CASE tools. Text-book methods, however, do not recognize internal identifiers because of their 'pen and paper' -mentality.

In the methods analyzed all three types of scope were used. First, in most object-oriented methods (e.g. Rumbaugh et al. 1991, Coad and Yourdon 1991a) the 'name' of a 'class' and the 'number' of a 'process' form identifiers inside all models of a project (i.e. method scope). Second, the 'name' of a 'state' identifies states inside a single state transition diagram, but not within a whole method, since two or more state transition diagrams can have states with the same name (referring to different states). Third, an identifier can be dependent on other instances. For example, in UML (Booch and Rumbaugh 1995) classes can have a scope according to the enclosing category: the identity of a 'class' object type is dependent on the 'category' object type it belongs to. Similarly, in ISAC (Lundeberg et al. 1981) the code of an elementary information set recognizes the instances only as a subset of a non-elementary information set. Therefore, the master (i.e. a category in the former and a non-elementary information set in the latter example) also has an identifier, and it forms part of the identifier for instances of dependent type.

The identity constraint can be characterized as an active constraint since in modeling tools they can be analyzed each time an instance of the property type (i.e. value) is created, changed, or deleted. Active checking, however, can lead to time consuming computation and usually CASE tools can not analyze identifiers actively. For example, active checking at the level of the whole method necessitates that all models and their instances are inspected.

4.4.1.2 Unique property constraint

A unique property constraint specifies that an instance of a property type has a unique value inside the enclosing scope. The unique constraint prevents the homonym problems which almost every method warns against: the use of the same value for different instances of a property type. Typically an identifier must be unique, but also other properties may need to have unique values.

Among the methods analyzed a unique constraint is needed in all method scopes. A unique property based on the dependent type can be found from class diagrams (e.g. Rumbaugh et al. 1991) in which a class can have only one attribute with the same name. Similarly, in the ER model (Chen 1976) a name of an attribute must be unique among the attributes connected to an entity. A unique property constraint within a model is relevant for example in data flow and state transition diagrams in which names of processes and states must be unique inside the diagram. Among relationship types a unique property for a message passing sequence (Coad and Yourdon 1991a, Coleman et al. 1994) is relevant inside a single model. In the method scope the identifying number should be unique among all instances of a 'process' object type.

In addition to different scopes, a metamodeling language should be able to specify uniqueness of the same property types for several object types. For example, in Coad and Yourdon (1991a) both abstract classes (i.e. a 'class') and classes with instances (i.e. a 'class-&-object') must share the same property type. Similarly in Booch (1991) a 'metaclass' and an ordinary 'class' can not have the same value for class names (i.e. class name values are unique among both types).

A uniqueness constraint can be considered as being passively checked at least in the scope of a method, since all values of a given property type are not necessarily available and thus can not be checked instantly. In contrast, a model and a dependent type have a limited number of instances and thus can be checked actively.

4.4.1.3 Mandatory property constraint

Some methods include rules which state that properties must have values at all times (i.e. null values are not accepted). Accordingly, a metamodeling language must distinguish mandatory and optional instances for property types. Generally, properties are optional, but identifying properties are mandatory. For example, a 'number' as an identifier and the 'name' of a 'process' object type in a data flow diagram are mandatory, but in UML (Booch and Rumbaugh 1995) the 'name' of a 'state' is optional. To ensure that data dictionaries can be formed parallel to modeling (as proposed in Yourdon 1989a), a documentation property type used in the metamodel of SA/SD for creating a dictionary must be defined as mandatory.

The mandatory constraint is not restricted to any specific scope, such as being dependent on instances of other types, or used in a model. Thus, we expect that the only scope for mandatory instances of property types is the whole method. Furthermore, this constraint can be checked actively in a computer-aided environment each time the property value is changed, or a new instance of a property type is created. In practice a need for passive checking would most likely arise because all properties are not necessarily known while creating models, leading to undefined property values.

4.4.1.4 Data type of properties

Design information captured in properties of other types are specified with various data types. From a metamodeling point of view, data types are needed to restrict the possible values of properties. Recent methods, such as most of the object-oriented ones, tend to have complex data types. One explanation for this is CASE tool support, which on the one hand demands data type definitions to implement the tool support, and on the other hand offers mechanisms to manage larger models and more complex data types.

Among the most typical data types are integer, string, text, and Boolean. A number is commonly used as identifying property or for describing the order among relationships (e.g. Coleman et al. 1994, Coad and Yourdon 1991a). A string is used for short descriptions; a text for a larger body of specifications such as definition in a data dictionary or pseudo code (Yourdon 1989a). Boolean describes single-value "on-off" or "true-false" characteristics such as the persistence of a class (Henderson-Sellers and Edwards 1994). In addition to plain data types, some methods, such as ISAC (Lundeberg et al. 1981) and BON (Walden and Nerson 1995) include more detailed specifications for the internal structure of each data type. For example, in ISAC, only one-digit numbers can be used for identification of activities, and in UML (Booch and Rumbaugh 1995) the possible values for visibility are limited to three (i.e. public, private, protected) and access of attributes into four (i.e. writeread, write, read and none). Some methods have more complicated rules for the textual description: in IDEF (FIPS 1993a, p 11) arrow labels can not consist of reserved words, and in ISAC the numbers of information sets also include the number of the activity creating the set. In BON (Walden and Nerson 1995) the structure of textual properties is the most extreme: there is a whole language for defining instance-related assertions through properties related to other instances. Thus, a metamodeling language should provide, together with the method-related data types, the possibility to specify the syntax of data types, and for checking the syntax. This requirement, however, goes beyond the typical use of data modeling languages as discussed in Section 4.5.3.

In addition to the property types which can be understood of having one value only as above, a metamodeling language must also identify collections. Collections are mostly used in object-oriented methods. For example, a class can have multiple attributes and operations.

Property type definitions can also be extended by defining default values and predefined values. These mean that a metamodel defines some instance values for property types. A default value defines a single instance for a property type to be applied if nothing is added. Thus, it is usually applied with property types defined as mandatory. Predefined values are typical in the cardinality constraints used in data models because they apply different naming policy for cardinality values. Some expect symbols instead of numbers: some describe cardinality with number only (typically a maximum value), whereas others describe cardinality as a pair of values (i.e. minimum and maximum). Checking a data type can be done actively. Because data types do not focus specifically to any scope of the constraint, the method scope as most general seems to be most applicable.

4.4.1.5 Cardinality constraint

A cardinality constraint defines a minimum and a maximum number of instances of a role type a relationship type instance can have. A role construct, used either explicitly or implicitly in all major semantic data models, defines the part played by an object in a relationship, such as in NIAM, (Nijssen and Halpin 1989, ter Hofstede 1993), OPRR (Welke 1988, Smolander 1991), or CoCoA (Venable 1993). The minimum number is typically 1 in the roles of binary relationships, since a relationship can not normally exist independently without connected objects. For example, a ‘message passing’ relationship in an object diagram (Coad and Yourdon 1991a) must have both ‘send’ and ‘receive’ roles. It is also possible, however, to define the minimum constraint as zero to denote relationships that do not need to have other role(s). For example, the object interaction graphs of Fusion (Coleman et al. 1994) allow one to define message passing between objects in which the sender outside the model boundary is not specified. More typical situations of zero minimum cardinality are cases in which a relationship can be extended with an optional role type, e.g.

with an associative object type (Yourdon 1989a) in entity relationship diagram, or with a creation of an object in data flow diagrams (Rumbaugh et al. 1991). Hence, the minimum constraint for the optional role type is normally zero and for a mandatory role at least one. Consequently, the deletion of mandatory roles (minimum cardinality one) removes also the whole relationship and related instances of role types. Moreover, if ternary relationships have their own modeling constructs, as in the class diagram of UML (Booch and Rumbaugh 1995) the minimum role cardinality is 3: Each relationship must have at least three instances of a role type. Otherwise the relationship is a binary one and should be defined with a different relationship type.

Within the methods analyzed the maximum cardinality of a role is either one (1) or many (M). The maximum cardinality is one in binary relationships with two role types, e.g. in the relationships of a structure chart, a module diagram (Yourdon 1989a) and a platform diagram (Booch 1991). Thus, if an instance of a 'call' relationship type exists in a structure chart it can not have more than one instance each of 'send' and 'receive' role types. The maximum cardinality of a role type is many in n-ary (sometimes also called branching) relationships. For example, an 'inheritance' relationship (also called generalization, gen-spec, supertype) in object-oriented methods can have only one (1,1) 'superclass' role but one to many (1,M) 'subclass' roles.

None of the methods modeled include restrictions on the cardinality rule within different scopes. Since they implicitly expect that the same instance of a relationship can exist only among the same role type instances, the most relevant scope is a method. This allows us also to support methods which use the same relationship type instances in several techniques (e.g. an inheritance in Henderson-Sellers and Edwards (1994)). Moreover, checking of both minimum and maximum constraints for the role cardinality are active: They can be checked each time a relationship is created, an existing role is deleted, or a new one added.

4.4.1.6 Multiplicity constraint

A multiplicity constraint is needed to define a minimum and a maximum number of role instances an object instance may have. With the minimum value we can define that an object instance must be connected to at least a specific number of instances of this role type, and with the maximum value that an object type instance can not be connected to more than a specific number of instances of this role type. The need for the minimum constraint can be found from modeling a state diagram (e.g. Booch 1991) in which a 'start state' must be connected to at least one 'send' role of a 'transition' relationship, and from techniques that are based on tree structures, such as JSD (Cameron 1989). An example of the maximum constraint is inheritance found from most class diagrams allowing only single inheritance (e.g. Rumbaugh et al. 1991): a class can only participate once in a subclass role.

Typically, a multiplicity constraint for a role is zero-to-many (0,M): an object type does not need to be connected to an instance of a specific role type, but it can be connected to many instances of that role type. Other common values found for minimum multiplicity are one for mandatory roles, and two for object types which must occur at least twice in a specific relationship (e.g. a 'condition' object must participate in at least two 'connector' relationships (Coad and Yourdon 1991a). Hence, the multiplicity value for a 'condition connector' role type should be two-to-many (2,M).

Methods use multiplicity constraints within the scope of a model or a method. An example of the former is a 'start state' in a state transition diagram: each start state must be connected to one state and thus the minimum multiplicity constraint must be checked for each instance of a start state. A typical example of the latter scope can be found from data flow diagrams in which an instance of a 'data store' must participate in instances of 'send' and 'receive' role types of a 'data flow' relationship type, but not necessarily in one diagram (Yourdon 1989a, p 282). Similarly among all collaboration diagrams (Booch and Rumbaugh 1995) each instance of an 'object' must send and receive at least one message, but not necessarily inside the same model.

A maximum constraint can be checked actively, but the minimum constraint is passive: it can not be satisfied during model building, unless it is zero, because objects can exist while they are not related to other objects (i.e. connected to a role type instance).

4.4.1.7 Multiplicity over several role types

In addition to the multiplicity constraint, modeling of method knowledge requires constraints between different role types. Basically, this constraint is needed to prevent instances of object types that are not participating in any relationships. In other words, this constraint supports a rule stating that an object type instance must participate in at least one of the specified roles. In NIAM (Hofstede et al. 1993) this constraint is called a total role constraint. Examples of method knowledge necessitating the multiplicity constraint over several role types are those of ISAC and SA/SD within the scope of a model. An 'information set' instance must participate either in a 'predecessor' instance, or a 'successor' instance (Lundeberg et al. 1981), and a 'data store' instance must participate at least once in a 'send' or a 'receive' of a 'data flow' (Yourdon 1989a). The multiplicity rules identified among the methods analyzed do not require more complex multiplicity constraints, such as mandatory participation among two or more of the specified roles, or maximum multiplicity over several roles. Together with the cyclic relationship constraint, modeling techniques using tree structures, such as JSP (Cameron 1989), can be specified: one of the modules must be the root of the tree.

All these constraints are originally specified to be applicable inside a single diagram only, i.e. in the scope of a model only. Although the methods modeled do not apply this constraint within other scopes, it could be applied for the scope of the whole method as well (e.g. no more than 10 flows to an external).

Although the metamodeling construct could be checked actively when an object is created or a relationship deleted, passive checking is more suitable. The reason for this is simple: all objects and relationships to be checked are not necessarily available and models would too often encounter this rule leading to heavy model checking.

4.4.1.8 Cyclic relationship

A cyclic relationship involves connections between instances of object types via instances of a single relationship type, thus forming a cycle. Basically, in the methods analyzed two types of cyclic relationships exist: a direct one, in which the same instance of an object type can participate in both ends of the same relationship type instance, and an indirect one, in which the cycle can be formed via one or several additional instances of object types (with associated relationship type instances). It must be noted that the indirect cyclic relationship necessitates two or more instances of a relationship type. Table 4-3 illustrates examples of different cyclic relationships found in the methods modeled. Accordingly, a metamodeling language should distinguish both of these cyclic relationships types and allow method engineer to allow or prohibit them.

TABLE 4-3 Examples of cyclic relationships in the methods modeled.

	Cyclic relationships allowed	Cyclic relationships not allowed
Direct cycle	Transitions in state model (Yourdon 1989a, Rumbaugh et al. 1991) Message passing in object-interaction graph (Coleman et al. 1994)	Message passing in message trace diagram (Booch and Rumbaugh 1995) Call relationships in structure chart (Yourdon 1989a) Inheritance in class diagram (Coad/Yourdon 1991a)
Indirect cycle	Message passing in message trace diagram (Booch and Rumbaugh 1995) Information flow in A-graph (Lundeberg et al. 1981)	Inheritance in class diagram (Coad/Yourdon 1991a) Data structure diagram (Jackson 1976)

An example of a direct cyclic relationship can be found in state transition models (e.g. Rumbaugh et al. 1991, Booch 1994) which allow a transition from a state to itself, and in the object interaction graph of Fusion (Coleman et al. 1994) in which an object can send a message to itself. In other techniques, direct cyclic relationships are prohibited: in a message trace diagram (Booch and Rumbaugh 1995) an object can not send a message to itself, in a structure chart a module can not call itself (Yourdon 1989a), and in all object-oriented methods a class can not inherit itself. Inheritance serves also as an example of a prohibited indirect cyclic relationship. Similarly, indirect cyclic relationships are not possible in tree structures, as in JSD (Jackson 1976). Data flows in data flow diagrams, transitions in state transition models, and message passing in object diagrams (Coad and Yourdon 1991a) can form indirect cyclic structures. None of the methods analyzed, however, restricts the "length" of an indirect cyclic relationship structure, nor presents any specific scope for this type of constraint. Because the dimensions above are not totally orthogonal, only three basic patterns of cyclic relationships were found: those allowing both types (e.g. state models), those allowing indirect relationships only (e.g. A-graph), and those forbidding both types (e.g. tree structures like JSD). Thus, cyclic relationships which allow direct cyclic relationships but not indirect ones were not found from the methods.

Metamodeling constructs for cyclic relationships were not dependent on instances of the same or other relationship types. For example, a data flow from a process to itself can occur even if the same process receives a control flow (Ward and Mellor 1985), i.e. has instances of other role types.

The checking of both cyclic relationships types can be carried out actively although in most CASE tools the checking of the indirect type is passive because of its high computing requirements. Here, all relationships and objects participating in these relationships should be available. The scope of the constraint is all models since none of the methods included any more specific scope.

4.4.1.9 Multiplicity of types

A multiplicity construct for types is needed to define how many times instances of the same type must or can exist inside the enclosing scope. For example, ISAC (Lundeberg et al. 1981) has rules which specify that a maximum of 9 instances of a given type ('activity' or 'information set') should exist inside a single graph. In IDEF (FIPS 1993a) the possible number of functions in a model should not exceed 6, and also BSP (IBM 1984) recommends the number of data classes or business processes in an IS architecture plan. The multiplicity constraint is relevant for both object and relationship types. In most methods, the multiplicity constraint for object types is one-to-many (1,M): They must have one to many instances. However, the multiplicity of the 'start' state in most state transition models is zero-to-one (0,1): start states are optional and only one start state can exist in a model. Whereas object types can have different minimum and maximum values for multiplicity, relationship types are restricted only to a possible mandatory existence (i.e. with a minimum value). For example, at least one instance of a 'data flow' must exist in a data flow diagram and one instance of a 'transition' in a state transition diagram, but an 'inheritance' does not need to have instances in a class diagram. None of the methods analyzed include rules which set a maximum number for the occurrence of instances of relationship types.

Multiplicity of types should not be confused with multiplicity of the same instance: how many times the same instance, e.g. a process named Verify Orders, exists in a model. A metamodeling construct for instance multiplicity seems to be unnecessary since none of the methods includes such restrictions. Typically, to simplify crossing relationship lines in a model an instance can be replicated and all copies have the same properties. For example, in SA/SD (Yourdon 1989a) the same instance of the 'store' object type can be drawn to many places in the data flow diagram. The same relationship instances can also occur, as in OSA (Embley et al. 1992): an interaction relationship can occur both in the object-behavior model and in the object-interaction graph and it has the same properties for both instances (e.g. a trigger and an action). However, it must be noted that the same relationship type with the same instance information can not occur in any method more than once between the same object type instances (i.e. no duplicate relationships are allowed). This constraint is an inherent constraint (cf. Brodie 1984). As a result, it is not necessary to specify this constraint with an additional construct of a metamodeling language.

In the methods analyzed the multiplicity constraint is applicable at the level of a single model (e.g. ISAC), and of a whole method (e.g. BSP). Furthermore, although the multiplicity can be checked actively each time a new instance of a type is added, it should not be restrictive: it should be possible to create models that violate the multiplicity rule during modeling. Checking minimum values actively would also be inappropriate, since new models would violate the constraint.

4.4.2 Modeling interconnected techniques and methods

In this section we describe those metamodeling constructs essential to model interconnected techniques. Although the proposed constructs are mainly a prerequisite for modeling a whole method, some of the required constructs are useful for modeling single techniques.

4.4.2.1 Inclusion of types

The first requirement in specifying a whole method is the allocation of types into techniques. For this purpose, a metamodeling language must include a construct called inclusion (according to Tolvanen et al. 1993). The inclusion can be defined as an aggregation which can exist only between a technique and its types. For example, at the technique level the ER model includes entity, relationship and attribute types. At the type level, the GOPRR definition allows us to describe the graph type 'ER model' and its components. The type level cardinality for inclusion is many to many, since types can belong to many techniques and a technique usually consist of multiple types. For example, in the GOPRR metamodel of BSP the 'business process' belongs to three different techniques (cf. Section 4.3.1).

In addition to object types, relationship types and role types can belong to multiple techniques. For example, an 'interaction' in OSA (Embley et al. 1992) can belong both to interaction models and state models, and an 'inheritance' in MOSES (Henderson-Sellers and Edwards 1994) can be part of class and inheritance diagrams. Because of the similarities in the type level method definitions, these methods also explicitly allow the occurrence of the same instances in different techniques. For example, the same instance of an 'interaction' defined in an object-interaction model describing message passing between a set of objects can also be used to define an external trigger in an object-behavior diagram describing possible states of a single object (cf. Embley et al. 1992).

The checking of inclusion can be regarded as active since this constraint is already specified in the metamodel and does not necessitate an instance based evaluation: each time an instance is created or an existing instance is added to a model it must satisfy the type level definition. Because of the focus on specifying a type of a technique, the inclusion constraint is relevant only within the scope of a model.

4.4.2.2 Complex objects

The majority of the methods modeled, especially the object-oriented ones, apply complex objects. By a complex object we mean an abstraction mechanism which allows us to build aggregate-component structures among the types of the method. The aggregate object suppresses details of the underlying relationship between components (Smith and Smith 1977). Complex objects are also distinguished from aggregation of attributes used to define attributes of entities (Alegic 1988). In line with Iivari (1992) we make a distinction between the concept of a relationship and a complex object. The former is used for example in most dialects of ER-based data modeling languages. In fact, the ER model proposed by Chen (1976) only included relationships. Because our interest is on the type level definitions of methods it must be noted that instance level aggregation structures, such as aggregation (in Rumbaugh et al. 1991) and whole-part (in Coad and Yourdon 1991a), can be described with relationships in the metamodel. Complex objects are used as modeling constructs in specifying functional decomposition (cf. Yourdon 1989a), aggregation (cf. Coleman et al. 1994), concurrency (cf. Booch and Rumbaugh 1995), and clustering (Walden and Nerson 1995). Hence, our focus here is on those structures that are not described with relationship types and necessitate the use of complex objects. Several studies on metamodeling (e.g. Smolander 1991, Venable 1993, Saeki and Wenyin 1994) reveal limited support for modeling complex objects (sometimes also called hierarchical structures) with data model based metamodeling languages.

4.4.2.2.1 Analysis of complex objects in methods

Iivari (1992) reviews complex objects as a conceptual abstraction mechanism and classifies them into five dimensions. These are: 1) dependent/independent, 2) connected/unconnected, 3) mandatory/optional, 4) exclusive/shared and 5) recursive/non-recursive. In the following we describe the categories in more detail and apply them at the metalevel to recognize the different kinds of complex objects used in methods. Based on the analysis we found 11 different kind of complex objects summarized in Table 4-4. The five first rows of the table correspond to the various structures of complex objects proposed by Iivari (1992).

1) **The dependent/independent** dimension defines whether a component object can exist independently of the aggregate object. If a method employs dependent components it leads to a top-down process of model building, since

it is not possible to create components without an available aggregate object. Similarly, in a dependency situation, deleting an aggregate object will delete all of its components in that scope. An example of a dependent complex object is a functional decomposition in a data flow diagram (Yourdon 1989a). Here a process can be divided into a new subdiagram describing its subprocesses. Another example is a composite (Booch and Rumbaugh 1995) in which a class must exist before its component objects (i.e. instances of a class) can be defined. Functional decomposition is also applied in other methods, e.g. ISAC for defining activities with A-graphs (Lundeberg et al. 1981), in IDEF for decomposition of functions (FIPS 1993a), and in other techniques that employ data flow diagrams (e.g. Rumbaugh et al. 1991, Shlaer and Mellor 1992). Here we handle all these as examples of functional decomposition. An example of an independent complex object is a clustering (Walden and Nerson 1995): a cluster symbol (with an attached name) can be drawn around a set of classes to specify that they belong to the same group. Because empty clusters are meaningless, one or more component classes must exist. None of the methods analyzed, however, includes a multiplicity rule which specifies the required number of instance components. The construct for defining dependent components can be checked actively each time a dependent component is created or an aggregate deleted.

2) **Connected/unconnected** defines whether the internal relationships between the components of a complex object can be omitted. This dimension is not valid for metamodeling, since connected components are always possible: None of the methods offers rules which state that the internal relationships can not be specified together with the aggregate object.

3) **Mandatory/optional** describes whether a complex object can or cannot exist without any specified component. 'Mandatory' necessitate the existence of components and a bottom-up modeling approach. Typically, methods which propose their own object type(s) as an aggregate object expect that components exist before the aggregate object is specified. For example, a boundary in an object model (Coleman et al. 1994) should not be specified without the existence of its components (i.e. an empty boundary is not possible). Similarly, empty categories in UML (Booch and Rumbaugh 1995) are meaningless. In contrast, methods applying the same type both as an aggregate and a component often propose a top-down refinement, although a bottom-up approach is also possible. For example, in a data flow diagram, a process can exist even though it is not decomposed into a subdiagram. A mandatory rule can be checked each time an aggregate object is created or its component deleted.

4) The components of an aggregate can either be **exclusive or shared**. Techniques which form hierarchies, like composite (Booch and Rumbaugh 1995), functional decomposition (Yourdon 1989a) or clusters (Walden and Nerson 1995) presuppose that a component can not directly belong to more than one aggregate object. In contrast, a boundary (cf. Coleman et al. 1994) allows that a class (a component) can belong to many functional systems shown through boundaries (an aggregate). Similarly, aggregation structures in object-oriented methods specified with a complex object (e.g. Coleman et al. 1994) instead of a relationship type (as in OMT, Rumbaugh et al. 1991) allow shared

component classes. The notation used here for an aggregation as a complex object, however, easily leads to complex representations once components are shared due to overlapping aggregate representations. Moreover, components that are defined to be exclusive must be checked during modeling: the same instance of a component type can not belong to another complex object.

5) **Recursive/non-recursive complex objects.** This final dimension in Iivari's classification defines whether or not the component objects can be of the same type as the aggregate object. An example of a recursive complex object is a subject (Coad and Yourdon 1991a) which can contain classes or other subjects. None of the methods modeled had rules which required non-recursive structures.

TABLE 4-4 Structures of complex objects in methods.

Dimensions	Yes	No
Dependent component objects?	composite, functional decomposition, nested states	aggregation, boundary, category, cluster, object group, process group, subject, subsystem
Connected component objects?	aggregation, boundary, category, cluster, composite, functional decomposition, nested states, object group, process group, subject, subsystem	-
Mandatory component objects?	boundary, category, cluster, object group, process group, subject, subsystem	aggregation, composite, functional decomposition, nested states
Exclusive component objects?	category, cluster, composite, functional decomposition, nested states, process group	aggregation, boundary, object group, subsystem, subject
Recursive complex objects?	aggregation, boundary, category, cluster, composite, functional decomposition, nested states, object group, process group, subject, subsystem	-
Independent relationship of the aggregate?	aggregation, category, composite, nested states	boundary, cluster, functional decomposition, object group, process group, subject, subsystem
Aggregated relationships?	category, cluster, object group, subsystem	aggregation, composite, functional decomposition, nested states

The five structures applied here reveal some similarities and differences in the use of complex objects as a modeling construct in ISD methods. There are, however, additional differences between the structure and behavior of complex objects that are not yet addressed. For example, the structure of a composite is not similar to a decomposition used in data flow diagrams, nor is a subsystem (cf. Henderson-Sellers and Edwards 1994) similar to a subject (cf. Coad and Yourdon 1991a). To identify these differences two additional dimensions are required, namely independent/dependent relationships of the aggregate object and aggregated/non-aggregated relationships. Both of these dimensions are included in Table 4-4 as the last two rows.

6) **The independent/dependent** relationship of an aggregate object specifies whether an aggregate object in a complex object can participate in relationships which are independent of the relationships of its components. For example, one difference between the structure of a composite and functional decomposition is that in the composite an aggregate (i.e. class) can have relationships, such as inheritance, which are not related to its components. Naturally, the components representing instances of a class have attributes which the class may have inherited. Similarly in nested state models (Yourdon 1989a, 267, Booch and Rumbaugh 1995, p 33) a state which has substates (i.e. a composite state in UML) can participate in transitions which are not defined for any of its substates. The important difference in this dimension is that in functional decomposition a decomposed process can not have relationships other than those included in a subdiagram. This dimension also reveals other differences between relationships of complex objects. Some aggregates (i.e. boundary, subject, and process group) do not participate in any relationships by themselves but only through their components. Here a complex object is concerned with a collection of its components without any specific relationships (cf. Kim et al. 1989).

The case of dependent relationships, however, does not necessarily lead to the use of the same instances of relationship types both for a composite and for an aggregate. We did not include this difference among the dimensions of complex objects, since the difference can be specified in metamodels simply by allowing object types to participate in different relationship types. Independent relationships do not require instance-based checking, since they are already allowed on the type level. In contrast, dependent relationships of an aggregate and its components originating outside the complex object must be checked actively.

7) **Aggregated/non-aggregated** relationships define whether relationships of components connected outside the same complex object are collected into a new instance of the same or a different relationship type. Thus, we expect here that the relationships of components are compressed into new aggregated relationships. This dimension is valid only for those aggregates of complex objects which can participate in relationships. Examples of aggregate relationships can be found from some object models: a subsystem (MOSES, Henderson-Sellers and Edwards 1994) or a cluster (BON, Walden and Nerson 1995) has its own relationships "collecting" the relationships among the components of different subsystems or clusters. In MOSES these aggregate relationships are called a collaboration and in BON a compression of client relationships. The aggregated relationships can be of the same or a different type from the components' relationships. In the former case an object group (cf. Walden and Nerson 1995) can not sent or receive messages that are independent of the messages being send or received by its components. An example of the latter case is a category dependency describing only client-supplier relationships of the categories. These dependencies are, however, based on underlying relationships between classes of different categories. Methods which

do not allow aggregated relationships either apply exactly the same instances of relationships for the components, such as in functional decomposition, or allow relationships for the aggregate which are independent of the relationships of its components (as discussed in the sixth dimension). An aggregation of relationships requires that each time a first relationship is created or the last one deleted the same operation should be executed for the aggregated relationship as well. This demands active checking of aggregation rules.

4.4.2.2.2 Metamodeling constructs for complex objects

If we assume that all these dimensions are orthogonal, we can obtain 128 basic alternatives for complex objects. All possible alternatives identified in Table 4-4 are not necessarily, however, relevant for metamodeling of complex objects. The analysis of complex objects shows that some complex objects follow a similar structure. A subject (Coad and Yourdon 1991a) and a boundary (Coleman et al. 1994), nested states and a composite (Booch and Rumbaugh 1995), and an object group (Walden and Nerson 1995) and a subsystem (Henderson-Sellers and Edwards 1994) belong to the same categories in Table 4-4. These similarities limit the number of different structures found among the methods into 8. As a consequence, the metamodeling language must support the modeling of each conceptual structure of complex objects to capture method knowledge. In the 17 methods selected we found 11 complex object types. It must be noted that other alternative types for complex objects are also possible.

In addition to the seven dimensions of complex objects, the possible scope of this construct divides methods into two categories: those treating complex objects globally within the scope of a method, and those allowing different complex objects of the same aggregate object in different models. Functional decomposition and clustering according to the system view belong to the first category. Each decomposed process or cluster has the same components even though they would be represented in different models. Composite and subjects are examples of the model scope. A class can participate in multiple composite structures, each structure describing its instances (objects) in various contexts. Similarly, the same subject can have different components in different class diagrams.

To summarize, the following aspects of complex objects need to be recognized and represented with a metamodeling language. First, since all methods allow relationships to be described between components the second dimension — connected component objects — seems useless in metamodeling. Thus, we expect that a metamodeling language will not need to distinguish complex objects based on the possibility to have relationships among the components. Iivari (1992) too has doubts about system models that do not specify internal relationships. Second, dependency and mandatory components are alternatives since complex objects can be defined either in a top-down, or in a bottom-up manner. In most situations of IS modeling both of these strategies are possible. Thus, the methods analyzed here provide either one or both of the options. Third, a metamodel should define whether the same object can or can not be a component in many complex objects. None of the methods proposes other restrictions, such as a component having to belong to a specific number of complex objects. Fourth, since non-recursive complex objects were not found, recursive structures do not need to be distinguished in the metamodeling language. Fifth, it should be possible to define complex objects in which an aggregate object can participate in relationships separately from relationships that its components have outside the complex object. Methods apply either the same relationship types for the aggregate object as its components have, like in nested state models, or new relationship types, as in dependencies of categories in UML. Finally, a metamodeling language must have constructs to distinguish relationships of components which must be aggregated to from relationships of the aggregate.

4.4.2.3 Explosion

One of the most common approaches to integrate techniques is linking of a type in one technique to a set of types described in another technique. We call this metamodeling construct an explosion in the GOPRR model (Kelly et al. 1996). According to our analysis of complex objects, explosion structures are typical between different techniques, and they do not carry as much semantic information about the instance level linkages as complex objects. For example, relationships of the exploded type are meaningless as the relationships in the target model are based on another technique. According to most object-oriented methods the behavior of a class from a class diagram (see also example metamodels in Section 3.3.3) or a use case from a use case diagram can be described with state diagrams (cf. Coad and Yourdon 1991a, Booch and Rumbaugh 1995). Similarly, according to the balancing rules of SA/SD (Yourdon 1989a, p 283) each control process must be associated with a state transition diagram. Various explosion structures can be characterized according to 1) the type of the explosion source, 2) the cardinality of the explosion, 3) an exclusive or shared explosion target, and 4) active/passive checking of explosion cardinality constraints. Each of these characteristics is described below.

1) **Type of the explosion source.** Among the methods analyzed, three different kind of explosions could be found depending on the metatype that forms the source of the explosion. First, an object type, like a ‘data store’ or a ‘class’, can be a source for the explosion. A second possible source type is a relationship, such as a ‘transition’ which in a state model explodes into a data flow diagram (Rumbaugh et al. 1991). Third, a property type of an object or a relationship type can also be refined by explosions. For example, in Coad and Yourdon (1991a) each ‘service’ of a ‘class’ can be described in service charts.

2) **Cardinality of explosion.** Among the methods analyzed, several limitations to the number of explosion links are defined. These constraints can be represented by attaching cardinality constraints to explosions. Both a source type and a target technique must have a cardinality constraint and both a minimum and a maximum cardinality are needed for a complete definition. At the source part, a cardinality defines how many explosion links an instance of the source type can or must have. Typically, the minimum cardinality represents whether an explosion is mandatory, and a maximum cardinality

specifies if more than one explosion link is allowed. An example of the minimum cardinality can be found from SA/SD (Yourdon 1989a) in which each data store must be described in more detail with an ER diagram, and in a data dictionary. Most object-oriented methods generate a need for a maximum cardinality: in most methods, the states of a class can be described in several state models. Therefore, the total cardinality at the source type is zero-to-many (0,M). Moreover, because in a data flow diagram a process can be specified only in one process specification the maximum cardinality is one.

At the target part, the minimum cardinality specifies whether a target model needs to be linked to one or more instances of the source type. For example, in Yourdon (1989a) no floating process specifications are allowed, and in Coad and Yourdon (1991a) service charts unconnected to a service of a class are not allowed. Especially in cases of multiple possible explosion links to the target technique, the minimum cardinality of the explosion target is zero. For example, in OMT (Rumbaugh et al. 1991) each data flow diagram does not need to be connected to an object type 'state' as it can be a target for an explosion of an relationship type 'transition' as well. Thus, the minimum cardinality for an explosion link between a 'state' and a 'data flow diagram' is zero. The maximum cardinality of the target type on the other hand specifies whether more than one instance of a source type can explode into the target model. Although in most situations only one explosion link is allowed for the same target, some methods, like FUSION (Coleman et al. 1994) or OMT (Rumbaugh et al. 1991), allow many instances of a source type to explode into the same instance model. Hence, the maximum cardinality is many. In Fusion, a model describing interaction between several objects can be a target for several explosion links from different instances of an 'object' specified in object models. Similarly, an UML collaboration diagram specifies messages sent between several instances of a 'class' described in a class diagram.

3) **Exclusive explosion links** restrict whether instances of two or more different source types can explode to the same target model (i.e. instance model). In SA/SD a process specification can be a target of explosion links from two types, as both a 'process' and a 'control' can have operation specifications. These specifications, however, must be defined as exclusive because a process specification can belong to only one instance. An example of a shared target model is explosion of instances of both a 'class' and a 'class utility' into the same object diagram (e.g. Booch 1991).

4) **Active/passive checking.** Because the cardinality as such does not describe a precedence between a source and a target this procedural aspect can be described with a checking rule: if the checking of the minimum cardinality on the target side of the explosion link is defined as active it can be used to specify top-down structures of explosion links. In the earlier example of exploding a process into process specifications, the mandatory and active checking of minimum cardinality assures that process specifications can not be specified without a related instance of a 'process'. If a source element of a top-down explosion link is deleted, the target models should be removed as well. Active checking of the minimum cardinality on the source part, on the other hand, can be used to define a bottom-up strategy for modeling explosion

structures. None of the methods analyzed, however, applied bottom-up structures. In the explosion links analyzed the maximum cardinality for both a target and a source can be checked actively.

Finally, the explosion constraint must be related to either of the two possible scopes; a method or a model. The method scope defines that the explosion constraint is relevant for all instances of the source type. Alternatively, explosions can be relevant for each instance in a model only. In the former case, explosions are defined for all instances of the type, and in the latter case, the same instance of a source type can have different explosion links in different models. An example of the method scope is an instance of a class which explodes always to the same state model regardless of the class diagram in which it is represented (i.e. a class always has the same lifecycle). An example of a model scope is when a transition in a state model explodes to a collaboration diagram specifying a scenario in which the transition occurs as a message passing (e.g. Booch et al. 1997).

4.4.2.4 Polymorphism

Methods consisting of multiple techniques inspect systems from different views: each technique focuses on a specific view and these different views are integrated in the whole method. In addition to using the same types as a part of different techniques on the metalevel (defined with the inclusion construct) methods apply polymorphism of types to indicate instance level connections (Venable 1993). By polymorphism we denote connections between two or more instances of different types based on sharing the same values as their properties. Types can also be of a different metatype (e.g. an object which is represented as a relationship in another model). In other words, ISD methods use different types to describe the same instances. Polymorphism is applied mostly in methods which use horizontal integration for connecting instances of different models, e.g. names used for data stores in data-flow diagrams being redefined for cross-checking with an ER model.

Different structures of polymorphism can be identified based on 1) coverage over one or more techniques, 2) the number of properties shared, 3) the number of type instances related, and 4) a possible dependency among the types of a polymorphism structure. As a consequence, a metamodeling language should be capable of representing all the different structures discussed below. These different structures are collected into Table 4-5 together with examples to be discussed in more detail below.

1) **Coverage.** Polymorphism can exist between types included in one or several techniques. An example of the former is a qualifier of an association in some class diagrams (e.g. Rumbaugh et al. 1991, Booch et al. 1997). Qualifier is also one of the attributes of a class participating in the association, i.e. the value for the qualifier must also be defined as an attribute in the related class. As a result, it is not adequate to model a qualifier and an attribute only as property types, as they need to be related to indicate sharing of the same property values. Similarly, a discriminator of the inheritance relationship must be an attribute of the superclass. An example of the latter, polymorphism between techniques, can

be found from the balancing rules of SA/SD (Yourdon 1989a, p 283). A data flow in a DFD and a call relationship in a module diagram, as well as a control flow into a control process and a condition of a transition in a state model, describe the same instance in different models. Similarly, almost all object-oriented methods apply polymorphism to describe that an action (or an operation) in a state transition model and a service (or a method) of a class describe the same instance (Coad and Yourdon 1991a, see also the metamodeling example in Section 3.3.3). To specify these structures adequately an additional supporting metamodeling construct is required.

TABLE 4-5 Examples of different kinds of polymorphism in methods.

In one model	In a method
qualifier/attribute	attribute type of a class/attribute value of a state, condition/data flow, name of a category/name of a class, name of a class/name of an object, name of a data element/name of a data set, name of an entity/name of a data store, operation of a class/ action of a transition, service of a class/message, service of a class/operation of a state
One property value	Multiple property values
attribute type of a class/attribute value of a state, name of a category/name of a class, name of a class/ name of an object, condition/data flow, name of a data element/name of a data set, name of an entity/name of a data store, qualifier/attribute	operation of a class/action of a transition, service/message, service/operation of a state
Two type instances	More than two type instances
attribute type of a class/attribute value of a state, condition/data flow, name of a data element/name of a data set, name of an entity/name of a data store, qualifier/attribute	service of a class/message, service of a class/operation of a state, name of a category/name of a class, name of a class/name of an object, operation of a class/action of a transition
Independent	Dependent
name of a category/name of a class, name of an entity/name of a data store	attribute type of a class/attribute value of a state, condition/data flow, name of a class/ name of an object, name of a data element/name of a data set, operation of a class/action of a transition, qualifier/attribute, service of a class/message, service of a class /operation of a state

2) **Number of properties shared**. Polymorphism can be based on sharing more than one property value at a time. This necessitates that more than two property types are involved in the polymorphism. The qualifier example, discussed above, is based on sharing one value only between two property types: the 'qualifier name' and the 'attribute name'. Object-oriented methods like MOSES (Henderson-Sellers and Edwards 1994) apply polymorphism for several instances of property types at the same time: a service of a class in a class

diagram and a message in an event model have several common values, such as name, parameters, and return types. Hence, when messages are described in an event model, properties of a message must refer to a set of the properties of one service. According to the majority of method descriptions, it is not possible to have messages other than those defined as services of classes. As a result, if one of the property type instances is shared, it necessitates also that related instances of other property types are shared as well. For example, the same instance of a message type can not have different return values as properties. Similarly, in UML each object has a class name to indicate the class that the object belongs to, and therefore the attribute values of the object must refer to those defined for the class. Accordingly, a metamodeling language should also represent instance level connections between related properties.

3) **Number of type instances related.** Polymorphism can occur between more than two instances of types. For example, the same instance of an operation of a class may be used as an operation of two or more state transitions, or in some state transition diagrams also as operations of states (Rumbaugh et al. 1991, Booch and Rumbaugh 1995). Here the same value is referred to by property types of multiple non-property types. To distinguish the states and related operations of a single object from the operations needed in communication between several objects, some techniques (e.g. OSA, Embley et al. 1992) include separate relationship types, or even techniques, for this purpose.

4) **Dependence on other instances.** The dependent type can not have other property values than those already defined in other type instances. For example, in UML all objects must be connected to a related class by their name and therefore it should not be possible to create an object which is not instantiated from the defined class. As a result, an object can not refer to classes which are not yet defined. A similar kind of polymorphism exists in MOSES (Henderson-Sellers and Edwards 1994) between a 'message' relationship type used in an event model and a 'service' of a class used in an O/C model. Thus, in a metamodel, a property type of a message called 'message name' must share values already defined as values of the property type 'service name'. In other words, an object should not call for a method of another object which is not available in the called object.

Dependency is optional in many polymorphism structures. For example, the balancing rules in Yourdon (1989a, p 281) between names of entities and data stores do not state which of these must be defined first as long as the names match in the end. In contrast, methods which propose some guidance for applying different types in a specific order require that the dependency is defined. For example, in ISAC (Lundeberg et al. 1981) a 'data element' instance in a data structure diagram is normally defined only after the related 'data set' instance is defined in a D-graph (i.e. the name of a data element refers to the name of a data set). Thus, the procedural part defined in a process model of an ISD method usually requires as a counterpart a specific static structure (Kinnunen and Leppänen 1994, Jarke et al. 1998).

In addition to the identification of various structures of polymorphism, each type of polymorphism must be defined according to its scope and type of checking. The scope of the polymorphism defines the space from which the property type instances can be shared. Based on the polymorphism structures found, all three scopes are possible. The method level includes all instances of the property types. For example, in UML 'class' has a property type named 'category' for specifying to which category a class belongs. The possible values for the 'category' are the names of all categories defined in all category diagrams or class diagrams. Similarly, an object may be characterized by the name of an existing class (e.g. Walden and Nerson 1995, Booch and Rumbaugh 1995). Polymorphism restricted to the scope of a model limits the possible shared instances into those defined in a single model. For example, according to the balancing rules of SA/SD (Yourdon 1989a), actions of state transitions must correspond to the name of the flows defined in a related data flow diagram in which the control process is described. The most complex form of polymorphism is based on a dependency on a specific instance of a given type in contrast to all instances, or instances of a single model. Among the methods modeled their dependency can be found from explosion or from composite objects (Venable 1993). For example, a state diagram of a single object (instance of a class) can have only those actions as a property of transition, which are also used for an object. In other words, the dynamic behavior described in a state transition diagram can have only those actions defined as operations of the related class in the class diagram. Similarly, a state can have as a property only those actions which are defined in the related class (Rumbaugh et al. 1991), and the attribute name of an object must match one in its class (Booch and Rumbaugh 1995, p 5). In our metamodel-based definition of UML, this would require that state variables defined as 'values' in our metamodel would be related to the attribute definitions of a class.

The checking of polymorphism can be either active or passive depending on the dependency of polymorphism, i.e. dependent or independent. Dependent polymorphism implies active checking, as it can be checked at all times that a created or modified type can not have other values than those defined already (i.e. no new values are created). Independent polymorphism can also be checked actively if the modeling tool informs the modeler of the available instances of other property types which a created or modified type could use as instances of its property types. Active checking, however, would necessitate that the polymorphism would be satisfied at all times. For example, according to the balancing rules in Yourdon (1989a) it would not be possible to create an entity if a data store with the same name would exist and vice versa. Thus, independent polymorphism must apply passive checking.

4.4.3 Summary of the metamodeling constructs

Modeling of method knowledge has been recognized as one of the main research problems in the field of method engineering (e.g. Kumar and Welke 1992, Kronlöf 1993, Brinkkemper 1996). In this section we have approached this problem in an inductive manner by analyzing modeling techniques from 17 different ISD methods, modeling them into metamodels and adapting them into CASE environments. This analysis has pointed out various patterns, categories and rules of methods that a metamodeling language should capture to model method knowledge more completely. These were generalized into metamodeling requirements by specifying constructs for metamodeling languages which extend existing semantic data models (i.e. metametamodels in the context of metamodeling).

Although the identification of the essential metamodeling constructs is based on the examination of 17 ISD methods, we see several ways to explore these constructs further. The first and most obvious way is to enlarge the set of ISD methods analyzed. Second, the types of methods included could also be extended from analysis and design methods into other methods of ISD, like project management, programming languages, etc. This is especially important since most methods modeled follow the icon-link structure typical in CASE tool related methods. It would also be relatively easy to propose a method which could not be described with the proposed metamodeling constructs. This means that we can not exclude certain metamodeling construct, but rather only describe those which were needed for our metamodeling effort. Third, the metamodels of the software design oriented methods could be extended towards programming languages, as suggested in some references (c.f. Booch and Rumbaugh 1995). This would raise new requirements for metamodeling, especially related to data types to satisfy the grammatical rules of programming languages, as well as analysis of designs by executing or compiling them.

Finally, other metamodeling constructs could also be identified by analyzing metamodeling carried out in practice. For example, in three metamodeling experiments, 75% of the concepts identified were involved in specialization hierarchies (Wijers 1991, p 174). Although we acknowledge the usefulness of inheritance to simplify metamodels and organize elements of metamodels into more manageable hierarchies (Rossi and Tolvanen 1995) we did not include it among the essential constructs of a metamodeling language for one simple reason: all static knowledge of methods could be described without inheritance. Furthermore, in the metamodeling literature a variety of approaches are proposed for using inheritance: Oei and Falkenberg (1994) propose a metamodel hierarchy for organizing techniques and building transformations between them, Elmasri et al. (1985) apply inheritance for entity types, Kelly et al. (1996) apply inheritance for other types as well as object types, and Venable (1993) and Ebert et al. (1996) extend inheritance to also cover the relationships that the object type participate in. The limitation on describing method knowledge only as it is represented in the literature is recognized in Chapter 6: we apply a metamodeling language together with the proposed extensions to describe method knowledge based on situations and experiences of method use (i.e. also in practice rather than just from the method literature).

4.5 Evaluation of metamodeling languages

In this section we shall apply the results of the method analysis to evaluate the modeling power of several proposed metamodeling languages. Evaluation of modeling power implies evaluation of the constructs languages offer for metamodeling. Similar kinds of assessments or comparisons have been carried out by Welke (1988), Venable (1993), Saeki and Wenyin (1994), and Harmsen and Saeki (1996). In the following sections we shall first review existing comparisons in terms of their focus to clarify how they differ from our evaluation. This is followed by an assessment of a set of metamodeling languages using the metamodeling constructs identified. The section concludes with a discussion of how well semantic data models can serve as a metamodeling language from the perspective of modeling power, and in what way they should be extended to describe method knowledge more completely.

4.5.1 Other studies evaluating metamodeling languages

In his early study Welke (1988) analyzed the modeling power of binary, ER, and OPRR models to compare how adequate they are as metamodels for a repository (i.e. as metaschema). The focus was especially on how completely each metamodel can represent method knowledge, and hence it is close to our approach. The suitability of different metamodels is demonstrated by using structure charts as an example. In conclusion, the limitations of each metamodeling language are discussed and an extended OPRR model, called WOPRR, is briefly proposed for modeling larger methods and more complex techniques. Smolander (1991) has extended the analysis of the OPRR model based on experiences gained building OPRR-based metamodeling tools. Limitations of OPRR are identified in two areas. First, OPRR does not provide possibilities to model n-dimensional structures, such as the complex objects discussed earlier. Second, OPRR does not provide concepts for defining the connections between multiple connected techniques that form a whole method.

Venable (1993) concentrates on modeling complex objects, especially in situations of complex covering aggregation in which an aggregate covers both entities and their relationships. The modeling language proposed, named CoCoA, is compared with a number of other data modeling languages, such as the ER model (Chen 1976), entity-category-relationship model (Elmasri et al. 1985), class model of OMT (Rumbaugh et al. 1991), and NIAM (Wijers et al. 1992). Although the comparison covers languages used for IS modeling, many of them including CoCoA have also been used for method modeling (Grundy and Venable 1996). In the comparison two criteria, the richness and problem domain correspondence, are relevant to us here. Richness refers to there being sufficient semantic concepts to describe relevant aspects of the problem domain, i.e. method knowledge. Problem domain correspondence specifies whether the constructs of the metamodeling language correspond to the aspects of the problem domain (i.e. methods). The analysis reveals a limited support for modeling integrated techniques and demonstrates how complex covering aggregation is relevant in metamodeling. In CoCoA, a complex covering aggregation is mostly used for specifying which object types and relationship types are components of a technique.

Saeki and Wenyin (1994) point out some limitations in ER-based metamodels: how to describe constraints and hierarchical structures (i.e.

complex objects). Based on their evaluation, they suggest Object-Z as a language for method modeling. Object-Z can describe knowledge and rules related to the decomposition of processes in data flow diagrams, and constraints of relationships, such as data flows between stores. Although their study reveals the need for supporting specifications of constraints of method knowledge, no classification of relevant constraints or even other constraint types is mentioned.

Finally, in a study by Harmsen and Saeki (1996) four different metamodeling languages are compared. Some languages included in their study also address process modeling, but all of them include meta-data modeling. The focus of their comparison is on a wider framework of languages for method engineering. Because of the breadth and generality of their framework, the study does not reveal how well meta-data modeling languages can represent method knowledge and how it is related to metamodel-based modeling tools.

Although these studies present comparisons of metamodeling languages, our analysis complements them in the following ways. First, we focus on languages for meta-data modeling in relation to tools. To our knowledge only Welke (1988) has compared metamodeling languages in relation to CASE, and especially to a repository. Second, and perhaps more importantly, these earlier comparisons were carried out on a relatively general level, since most of them do not address detailed requirements for metamodeling. Finally, in line with our inductive approach, our comparison is based on a set of constructs found essential in modeling a large number of methods. Still, the inductive approach limits our analysis to those aspects of metamodeling that are relevant in modeling the chosen set of methods.

4.5.2 Evaluation according to essential metamodeling constructs

As described in Chapter 3, several languages for method engineering (i.e. metametamodels) have been proposed and even implemented into tool environments. In this section our goal is to analyze a set of metametamodels according to the proposed essential constructs. The selected metamodeling languages were already discussed in Section 3.3.3. They were selected because of their focus on meta-data modeling, and intention for use as a metametamodel in an adaptable modeling tool. Thus, we excluded all those metamodeling languages which focus on method representation only and do not enable metamodel-based adaptation of modeling tools such as CASE tools. Furthermore, those parts of the metametamodels which do not focus on modeling techniques were excluded. Hence, from MEL we analyzed only the constructs for specifying product fragments and from ASDM we analyzed only its deliverable model.

Because not all the tools applying the selected metamodeling languages were available to complement our study the assessment is partly biased: it is unclear how the metametamodels proposed can actually serve as a metamodeling language for customizable modeling tools. This observation emphasizes the need for such a tool-related comparison (Tolvanen et al 1996). Tools for MEL (Harmsen 1997), CoCoA (Venable 1993) and NIAM were not available as they exist at the design level only, or include partial implementations for the researcher's purposes only. The most common metametamodels implemented into tools (CASE Outlook 1989), such as ER (Mercurio et al. 1990) and OPRR (Meta Systems 1989, Smolander 1991, Marttiin et al. 1993, MetaCase 1994) are widely described from a tool point of view, and also tool-related implementations have been carried out for GOPRR (Kelly et al. 1996), and ASDM (Heym 1993). The last environment supports only method modeling but not the implementation of modeling tools based on the metamodels developed. We furthermore acknowledge the differences in versions of languages and their evolution, as well as differences in supporting even the same language (e.g. Quickspec and MetaEdit for OPRR) (Marttiin et al. 1993, Smolander et al. 1991). In the analysis we tried to focus on only one metametamodel version which is close to the tool environment. Also, additional grammatical extensions made for the notations of metamodeling languages, such as proposed by ter Hofstede (1993) for NIAM, are excluded, since they could be available for other metamodeling languages as well for directly enforcing certain integrity constraints.

In the following we describe the results of evaluation. The results are also summarized in Table 4-6: the vertical axis includes the metamodeling constructs and the horizontal axis includes the metamodeling languages. A cross means that the metamodeling language meets the requirement and a cross in brackets that the current support is limited. It is obvious that the evaluation of the metamodeling languages is not as clear-cut as Table 4-6 depicts.

TABLE 4-6 Support for metamodeling constructs in different metamodeling languages.

Metamodeling construct	Metamodeling language						
	ASDM	CoCoA	ER	GOPRR	MEL	NIAM	OPRR
Identifying property			(x)	(x)		(x)	(x)
Unique property			(x)	(x)		(x)	(x)
Mandatory property		(x)	x		(x)	x	
Data type of properties		(x)	(x)	(x)	(x)	(x)	(x)
Cardinality				x			
Multiplicity	(x)	x	(x)		(x)	(x)	(x)
Multiplicity over several role types						(x)	
Cyclic relationship	(x)	(x)	(x)	(x)	(x)	x	(x)
Multiplicity of type							
Inclusion	x	x		x	x		
Complex objects	(x)	(x)		(x)			
Explosion		(x)		(x)			
Polymorphism		(x)		(x)			(x)

4.5.2.1 Modeling single techniques

4.5.2.1.1 Identifying property

Identity of types is considered a relevant construct in most metamodeling languages, except in ASDM, CoCoA and MEL. Most languages, however, specify identity of object types only, and they do not distinguish the identity based on the scopes discussed in Section 4.4.1.1. OPRR allows one identifier for object types only, but GOPRR allows identifiers to be defined for other types as well (i.e. graph, relationship, and role types). NIAM normally uses a single identifier specified in parentheses above the entity name but also other keys, distinguished by ‘+’, are possible.

4.5.2.1.2 Unique property constraint

Uniqueness of property type instances is considered in ER, OPRR, NIAM and GOPRR, but inadequately on the scope side. A common extension to ER models for schema design is a uniqueness constraint. In OPRR identifying properties are expected to be unique globally (i.e. inside the scope of a method). In NIAM, the uniqueness of label types can be defined by a reference scheme named in parentheses under the name of the object type. In addition to this identifier description, the uniqueness of label types can be described by a uniqueness constraint attached to a role of the label type connecting an object type to the label type. Thus, the use of the same label for many object types allows for uniqueness inside the scope of a dependent type. Likewise, in GOPRR the property can be unique among the instances of related types.

4.5.2.1.3 Mandatory property constraint

Mandatory property types are identified explicitly in NIAM only. This is achieved by a total role constraint attached to an entity (Nijssen and Halpin 1989) or an object (ter Hofstede et al. 1993) type connected to a label type. Some other metamodeling languages include the possibility to define mandatory properties with other constraints. For example, in MEL (Harmsen 1997) a required clause can be used to define that certain properties must be available in other modeling techniques. In CoCoA the constraint could be supported through a simple aggregation with minimum cardinality one but this would lead to complex structures if all mandatory values had to be specified as aggregations and optional properties as a normal attribute. A common extension in the ER model (Teorey 1990) for defining constraints for attributes and database implementation considerations is the restriction of null values.

4.5.2.1.4 Data type of properties

Since semantic data models underpin the design of database schema, various data types are widely recognized and supported, including basic data types such as integer, Boolean, and string. Basically only two weakly supported modeling constructs are encountered: properties with a specified grammar, and collections. On the grammar side, CoCoA, NIAM, OPRR, and GOPRR support more complex data types based on predefined values or intervals for possible values. On the collection side, GOPRR has an explicit collection data type which can have any of the basic data types as components, or even other types of the method. CoCoA allows the definition of attributes (also called properties) as an aggregate of other attributes, and with the simple covering aggregation entity types may have components of other entity types. For the same purpose NIAM has been extended by power types (ter Hofstede et al. 1993).

4.5.2.1.5 Cardinality constraint

The cardinality constraint is handled only in GOPRR by separate cardinality values for both minimum and maximum cardinality. Moreover, since the cardinality constraints are effective in the scope of a model GOPRR adequately supports modeling of all possible binary and n-ary relationships. Although other metamodeling languages allow n-ary relationships to be specified it is only supported at the type level. For example, in CoCoA a relationship can consist of more than two roles (i.e. type level), but it can not be specified whether instances of a specific role type are necessary (i.e. instance level). In NIAM the modeling of a cardinality constraint requires that the relationship is considered as an object type (as in ter Hofstede 1993, p 45) leading to difficulty in implementing a method support based on the metamodel: there would be no way to distinguish the appearance of objects from relationships if they are of the same type. Subtyping of entities could be applied here, as in the reference model of Heym (1993). In spite of this, the relationships of methods considered as object types can have an occurrence frequency constraint on the related role type to include both minimum and maximum cardinality.

4.5.2.1.6 Multiplicity constraints

A multiplicity constraint is supported by ER, ASDM, OPRR, CoCoA and NIAM. ER and ASDM support only maximum multiplicity (called there cardinality) whereas CoCoA, MEL, and OPRR allow to specify minimum and maximum multiplicity. NIAM supports minimum multiplicity with a value of one (i.e. mandatory participation in a role) by a total role constraint, and maximum multiplicity with a value of one (i.e. instances of a role must be unique) by an uniqueness constraint. Other multiplicity values for maximum multiplicity can be supported by attaching the additional occurrence constraints for roles. This possibility is not allowed for minimum multiplicity (Weber and Zhang 1996) as required to model some methods (such as service charts in which a condition must participate in at least two role instances of the same type). The scope of the multiplicity constraint is left undefined in all metametamodels and assumed implicitly (once not specified) to cover either a scope of a method or a model. CoCoA, however, can capture an aggregation of role types together with the related relationship type and therefore supports both scopes of the multiplicity constraint. If a role type is only part of a single technique the scope for the constraint is a model. If the same role can also be part of multiple techniques (overlapping according to terminology of CoCoA) CoCoA could also restrict to the scope of the method. In principle support for specifying a scope among a limited set of techniques could be supported as well, although this was not recognized essential among the analyzed methods.

Multiplicity over several roles is only supported by NIAM, via a total role constraint attached to two or more roles connected to the same object type. NIAM, however, does not explicitly define a scope for this constraint and it is applied by metamodelers (i.e. ter Hofstede 1993) only for the scope of a model.

4.5.2.1.7 Cyclic relationships

Cyclic relationships can be modeled with all metamodeling languages, but only in NIAM can a direct recursion be forbidden by defining the cyclic relationship (called homogenous binaries by Nijssen and Halpin (1989, p 183)) as irreflective, and an indirect recursion by defining the relationship as asymmetric. In contrast, no method has relationships which are reflective, or symmetric, i.e. none of the object type instances must be related to itself via a cyclic relationship, nor does any connection between two object type instances necessitate another instance of the same relationship type with different role types.

4.5.2.1.8 Multiplicity of type

Modeling the multiplicity of types is not supported in any of the metamodeling languages analyzed. Thus, it is not possible to specify rules for restricting the number of instances of a given type (i.e. how many instances of an object or relationship type must or can occur in the scope of a model or a method).

4.5.2.2 Modeling multiple interconnected techniques

Most metamodeling languages focus on modeling single techniques only. Accordingly, only a few of them distinguish constructs for modeling multiple interconnected methods adequately.

4.5.2.2.1 Inclusion

Inclusion is supported in ASDM, CoCoA, MEL, and GOPRR, although in CoCoA (Venable 1993, p 116) it is applied only in integrating similar types of techniques from different methods. All of them allow the modeling of many-to-many relationships of the inclusion: the same type can be part of multiple techniques, and a technique can consist of multiple types. Because of the deficiency of NIAM in this matter it has been extended with schema types by ter Hofstede et al. (1993).

4.5.2.2.2 Complex objects

One of the most weakly supported constructs in metamodeling languages is the modeling of complex objects. Only ASDM, CoCoA and GOPRR support some structures of complex objects: ASDM provides a structural entity type; GOPRR provides decomposition, instance reuse, and nested data types for modeling complex objects. The last of these, however, does not address complex objects other than by allowing a component to be modeled as a property of an aggregate object (i.e. simple covering aggregation as in CoCoA). CoCoA provides complex covering aggregation to model an aggregate object type and its components as object and relationship types. As mentioned above, NIAM has been extended (cf. ter Hofstede 1993) with a schema type also applicable for modeling complex objects. In the following we assess the modeling support according to the various structures of complex objects (cf. Table 4-4).

In ASDM, entity types resulting from recursive aggregations expect that the component (called the structural entity type) must belong to exactly one aggregate type (called the fundamental entity type). In contrast, in CoCoA and GOPRR component elements are not dependent on the existence of an aggregate object. Thus, they can not adequately specify methods which apply top-down modeling only, such as functional decomposition (Yourdon 1989a) or a composite (Booch and Rumbaugh 1995). However, connected components, which are recognized as essential in all structures of complex objects, can be modeled. The third dimension, mandatory component objects, can be modeled only in CoCoA. In GOPRR components are optional, but in CoCoA both mandatory and optional components can be specified by a cardinality constraint for the covering aggregation, namely one for mandatory and zero for optional components. In relation to the exclusive-shared dimension of complex objects, GOPRR and CoCoA support only the shared dimension by allowing the same instance of a component type to belong to more than one aggregate object. As a result, they can not adequately specify methods which apply complex objects with an exclusivity constraint, such as the clusters of BON (Walden and Nerson 1995), functional decomposition (Yourdon 1989a), composite (Booch and Rumbaugh 1995) or nested state diagrams (Rumbaugh et al. 1991, Booch et al. 1997).

The fifth dimension of recursive complex objects is supported adequately both in CoCoA and GOPRR since only recursive complex objects need to be perceived. In CoCoA recursive complex objects are modeled by using the same names for the aggregate and the component. Hence, the same instance of an entity type is added twice to the CoCoA diagram. In GOPRR, non-recursive complex objects can also be described by having two different graph types (i.e. techniques), one containing the aggregate type and another for the component types. Moreover, the graph type for components should not include the aggregate object type. Because this approach requires two different graph types the choice of the modeling technique beforehand would not be ideal.

The metamodeling languages studied do not make any type level distinctions between relationships of an aggregate or a component. Thus, they all satisfy the sixth type of complex structure, i.e. independent relationships for aggregate object types. The final requirement in modeling complex types is aggregated relationships. This was not supported in any of the metamodeling languages studied.

4.5.2.2.3 Explosion

The third essential construct in modeling interconnected methods is the explosion from a type of one technique into another technique. For this metamodeling requirement GOPRR includes an explicit explosion construct; CoCoA does not provide such a construct explicitly and most of the CoCoA metamodels are based on techniques of a similar type in which the connection is based more on inclusion and use of the same types among several techniques rather than explosion structures between techniques (cf. Venable 1993, Grundy and Venable 1996). Relationships which are outside the domain of complex objects can, however, be distinguished as explosion relationships (see the CoCoA example in Section 3.3.3.5). NIAM does not provide support for explosions because it does not consider multiple techniques at all. Therefore, NIAM has been extended with PSM (ter Hofstede 1993) to support explosions by a relationship from an object type in one schema type to another schema (i.e. technique). These relationships are usually distinguished from other relationships by naming the roles (c.f. ter Hofstede 1993, p 33, 45) since they are represented with the same notation and constraint types. Hence, like CoCoA, NIAM/PSM does not provide any explicit type level construct.

None of the metamodeling languages for describing explosions is adequate. First, no distinction is made about the scope of explosions. The metamodeling languages analyzed thus do not distinguish between explosion of a type instance in a model or among all models. Other deficiencies in the metamodeling languages can be illustrated by various alternatives of explosion structures, namely type of the explosion source, cardinality of the explosion, exclusive or shared explosion target, and dependent/independent explosion target. Each of these characteristics was described in Section 4.4.2.3 and will be applied in the assessment below. Both GOPRR and extended NIAM (ter Hofstede 1993) support explosion structures from object types only, and thus can not adequately model explosion from property types, e.g. from a service to service charts (as in Coad and Yourdon 1991a), or from relationship types, e.g. from a transition to a data flow diagram (as in Rumbaugh et al. 1991). Mandatory explosion is supported in NIAM/PSM only with the extended graphical constructs (ter Hofstede 1993): a total role constraint on the role of the object type participating in an explosion relationship. In principle, other NIAM constraints for roles, such as uniqueness and occurrence frequency constraint, could also be used for describing other necessary cardinality rules. To our knowledge, however, NIAM has not been used to model such an explosion cardinality (cf. ter Hofstede 1993, ter Hofstede et al. 1993). If extended in such a way, exclusion of explosion links could also be described with the exclusion constraint of NIAM added to all role types of explosion relationships. While the cardinality of explosion links is unspecified in GOPRR, mandatory explosion links can not be modeled. Also, many-to-many relationship between instances of object types and graph types are allowed.

4.5.2.2.4 Polymorphism

Polymorphism is not supported at all in MEL, NIAM, ER and ASDM. For example, ASDM allows the same property type for one entity type only. Other languages address it only partially.

CoCoA offers an entity alias which supports polymorphism among entity types only. No entity alias, however, was found from the methods analyzed as none of them includes different types which have exactly the same instance information. The entity alias seems to have be added into CoCoA mainly to support integration of similar kinds of modeling techniques (cf. Grundy and Venable 1996). Moreover, CoCoA does not specify the functionality of such an alias if an aliased entity occurs in a complex covering aggregation. For example, would it have the same relationship instances as well?

OPRR supports dependencies between elements at the level of single valued properties only (Smolander et al. 1991, MetaCase 1994). A reference property type links to instances of the same, or another property type. Since the property type referred to may be named differently on the metalevel and belong to a different object, relationship, or role type, some structures of polymorphism can be supported with OPRR. These include methods which apply polymorphism in a single technique and share single valued (string) property types independently. Sharing of single property values can be supported among multiple type instances by using the reference property type for each type. Since any instance of the referred property can be used as a value for a referring property, the reference scope is the whole method.

GOPRR extends the support of polymorphism found from OPRR to multiple techniques as well as to more complex data types than string or other basic data types. If such more complex data types would be used, necessitating the use of instances of object, relationship or role types, the polymorphism is supported for multiple values. In GOPRR the sharing of a single instance as a property type is called property sharing. GOPRR does not specify any dependency to the property sharing, and it is allowed wherever the same property type is reused.

Support for metamodeling of polymorphism can be analyzed through the support for each kind of polymorphism structure: the coverage over one or more techniques, the number of property type instances shared, the number of types related, and the dependency. The metamodeling languages do not make a difference between polymorphism with instances of one technique or a whole method. OPRR provides support for sharing one property value only; and GOPRR extends this to sharing of multiple connected property values through sharing non-property types. This necessitates that all interconnected properties of a polymorphism structure are collected into a single type. Sharing the same property values can not be limited into two or any other number of instances. As a result, for example balancing rules of SA/SD requiring correspondence between two type instances only can not be modeled. These balancing rules state that an instance of an entity name can belong to only one item in the data dictionary. Finally, none of the metamodeling languages recognizes dependencies among polymorphism structure.

4.5.3 Limitations of metamodeling based on semantic data models

Although metamodels represent a great deal of static method knowledge and customizable CASE tools can automate them they are not complete. First, our metamodeling efforts with GOPRR (and OPRR, Tolvanen and Rossi 1996) show that it has limited support for modeling the rule parts of method knowledge. Second, none of the languages provides adequate support for metamodeling. The obvious reason is the limited metamodeling capabilities of the selected data modeling approach. As a result, we need to discuss the limitations of method modeling using conceptual data models. This topic is especially important because other languages for metamodeling, such as Object-Z (Saeki and Wenyin 1994) or LISA-D (ter Hofstede et al. 1993) have been proposed, which could solve alone, or as extensions, limitations of data model based languages. To guide the development of languages for capturing method knowledge or to find other complementary approaches, we illustrate in the following some key constraint types for metamodeling languages which were not addressed with semantic data models.

The analysis of the limitations of data model based metametamodels is supported by our method modeling studies. First, methods include transformations in which models based on one technique are transformed to another model. This is typical in vertical integration of techniques. According to our metamodeling approach this would necessitate changing the types of instances. For example, in a transform-centered design (Yourdon and Constantine 1989) a network model of processes can be transformed into a synchronous structure chart. Similarly, a data transformation based design approach used in JSD (Cameron 1989) applies generation of initial process structures from the definitions of the data structures. This is especially relevant in CASE tool based methods since they can automate error-prone routines, such as transformations. Transformations, however, are examples of method knowledge that needs information both about meta-data models for retrieving or changing design information (cf. Brinkkemper et al. 1989) and about process models (cf. Marttiin 1998, Marttiin et al. 1996) for guiding and executing the transformation.

Another type of method knowledge that can not be captured with static metamodeling constructs is heuristic rules and recommendations. For example, some object-oriented methods include recommendations on the breadth and depth of inheritance hierarchies, the number of public operations for a class (Booch 1991), or that a single state model should not specify states of more than one class (e.g. Embley et al. 1992).

Third, dynamic relationships among the instances of the same or different type can not be described with data models. Examples of these dynamic aspects in method knowledge are the numbering of an instance based on the number of its creating activity (Lundeberg et al. 1981), or that functions participating in a call relationship can not have child diagrams (FIPS 1993a). Similarly, in Yourdon (1989a, p 283) possible values for a condition in a state transition can not be found among the properties of the related control process, but from the flows it participates in: the possible values for the condition are only those names of flows which are coming into the control process. This type of polymorphism could not be supported with the proposed constructs. Extensions of methods closer to the constructs of programming languages would raise similar requirements. For example, an action defined in a state diagram of a single class should be characterized as a private method, and a message between objects as a public method, in the corresponding method definition of the receiving class. Similarly, modeling of method overloading would not be possible with the proposed metamodeling constructs. The extension of the metametamodels and metamodeling languages in this direction is, however, questionable. Some of the dynamic rules on naming, especially on the identifier side, originate from pen-and-paper oriented methods (e.g. Lundeberg et al. 1981) and are not necessary in computer-aided environments. Thus, there would not be a need to support these in metamodels either.

Fourth, none of the metamodeling techniques support grammar specification for formal textual descriptions, such as process specifications, data dictionaries (Yourdon 1989a), or textual grammars (Walden and Nerson 1995). These are especially important to better integrate modeling tools and models into other tasks of ISD, such as generating prototypes, program code, or visualizing available data and program structures.

Because of these limitations other metamodeling approaches, such as rule-based languages or predicate logic, could be more suitable as extensions and need to be studied in relation to the criteria proposed here. Various other type of languages for metamodeling could also be tested to specify essential constructs of metamodeling.

4.6 Summary and discussion

In this chapter, we have analyzed conceptual structures of methods and proposed essential constructs for metamodeling languages. These constructs were derived from the analysis of ISD methods by modeling the methods and by validating the metamodels by adapting them into a modeling tool. Our focus on metamodeling has captured static aspects of method knowledge for adapting a tool. The constructs are not required only to represent method knowledge, but also to "execute" the methods in a computer -aided modeling tool. Each construct of a metamodeling language supports the implementation of a certain part of the conceptual structure of a method into a modeling tool.

The proposed constructs were divided into two categories, those for modeling a single technique and those for modeling a whole method. In modeling a single technique, four constructs deal with modeling property types, including their identity, uniqueness, mandatory and data type. One construct, type multiplicity, deals with the number of instances of a given type. Four of the constructs deal with connections between objects, namely cyclic relationships, cardinality, multiplicity of single role type, and multiplicity over several role types. Required constructs for modeling interconnected techniques were classified into four aspects: inclusion for specifying the types used in each technique; complex objects for describing types which are treated as being "combined" without explicit relationships; explosion for modeling links between types and different techniques; and polymorphism for specifying the types of a method whose instances share the same values. These constructs are specific for the field of method modeling only, and no suggestions were made of their applicability in other domains.

The analysis of method knowledge together with the proposed metamodeling constructs also serves as a vehicle for assessing existing metamodeling languages. In short, CoCoA and GOPRR seemed to be most comprehensive for specifying method knowledge behind modeling techniques. NIAM also succeeded well but only while modeling single techniques.

In general, the assessment reveals that the current methodical support for method engineering is modest. While in recent years some progress has been made in outlining conceptual and theoretical principles for metamodeling and several metamodel-based tools have been developed (for a survey see Tolvanen et al. 1996) we argue that the available metamodeling languages, mostly based on data models (CASE Outlook 1989), do not provide adequate support for all aspects of method engineering. For example, metamodeling methods offer limited constructs for modeling interconnected techniques. Moreover, we identified conceptual structures of methods which could not be represented with the proposed metamodeling languages adequately or even at all. As a result, we have pointed out some areas for improvement.

It must be noted that we did not discuss method knowledge that could be supported already by all the languages, since this can be found directly from the metamodels. Similarly, during the assessment of the metamodeling languages we did not evaluate which language is more suitable for metamodeling, nor did we try to eliminate the constructs offered by these languages although some of them offered constructs which we did not find essential. This was especially the case with NIAM (see also Weber and Zheng (1996) for the construct overload) since the constraints for equality and exclusion were not found to be essential. These constraints along with some other might, however, be needed in modeling other methods, or if different interpretations of method knowledge and their tool support need to be made. The refinement of available languages for metamodeling, however, is outside the scope of our study.

Finally, various interpretations of method knowledge and its modeling deserve a closer examination. Like all modeling, our metamodeling effort was influenced by alternative interpretations of the method knowledge. Two major reasons for the interpretations and alternative versions of metamodels were tool adaptation and incomplete, or even inconsistent, method descriptions.

First, on the tool adaptation side, method developers have not considered enough possibilities of computer-aided tools but rather maintained the pen-and-paper mentality. Moreover, most remarks on tool use were made on the representational side, rather than on the conceptual side of method knowledge. As a result, in tool adaptation some aspects of method knowledge could be modeled differently. For example, there is no need to introduce additional textual pages separate from diagrams to view and edit more detailed information about the elements of a diagram (e.g. Lundeberg et al. 1981, Goldkuhl 1992) if such information can be added directly to the elements. Similarly, models do not need property types for entering reference information, such as how many representations of this instance exist (Gane and Sarson 1978), or whether the process is decomposed or not. Also the identity of instances does not need additional reference numbers, such as process identifiers, or information codes, widely used in pen-and-paper based methods (e.g. Yourdon 1989a, FIPS 1993a, Lundeberg et al. 1981). The balancing rules applied in many methods (e.g. Yourdon 1989a) could also be implemented differently: instead of referring to names in a polymorphism, we could refer to actual instances. Why refer just to a value of a property type, if the whole instance of an object type or a relationship type is available. These modeling options are typically related to the support provided by a modeling language, or a tool.

Second, a major reason for alternative metamodels was inadequate or even inconsistent method descriptions. As a result, we need to make our own decisions on what specific concepts and rules mean. For example, in most of the object-oriented methods, except OSA (Embley et al. 1992), it is unclear whether a state model can include states of more than one object. Instead of providing methods for systematizing ISD, method developers should apply (meta)methods to systematize ISD methods (Parsons et al. 1997).

5 EXPERIENCE BASED METHOD EVALUATION AND REFINEMENT

In this chapter our aim is to extend the use of metamodels in maintaining method knowledge in evolving ISD situations. Accordingly, we shall develop incremental method engineering principles and thus focus on our second research question (cf. Section 1.5.3):

> How can experience of method use together with metamodels be applied for method refinements?

The question deals with extending the dominant *a priori* ME principles through an *a posteriori* approach. We collect situational experiences of method use for refining methods. Whilst current ME approaches focus solely on the construction phase and expect information about methods and their use situations to be known completely beforehand, we assume that constructed methods are not necessarily applicable in the first place, situations in which they are applied change, and method users learn through their use. *A posteriori* refinement of methods is based on collecting and analyzing differences between intended and actual use of modeling techniques, on studying how techniques have supported modeling, and on understanding how they support problem solving. In contrast to learning about object systems under development (via IS models), our aim is to learn about methods and especially about modeling techniques (via metamodels). The proposed principles complement, but do not substitute, the ME frameworks and (meta)methods.

This chapter is organized as follows. First, in Section 5.1 we describe the motivation for incremental ME in general, and experience-based evaluation and refinement in particular. Second, in analyzing the principles of incremental ME it is useful to place this work in relation to other similar work reported in the literature. Therefore, in Section 5.2 we describe approaches proposed for method evaluation as well as point out some problems and difficulties in such evaluations. This leads us in Section 5.3 to propose mechanisms which, through the use of metamodels, can help to gather, analyze and communicate experiences about the use of modeling techniques. The ME process is explained in more detail from the view of *a posteriori* evaluation and refinement. Finally, Section 5.4 summarizes the chapter.

5.1 Introduction into incremental method engineering

Before extending ME principles further we need to argue for the necessity of an incremental approach and describe why and how the current ME principles are inadequate. This is important because there is a paucity of studies focusing on principles to support method evolution through ME principles (Tolvanen et al. 1996).

In the following subsections we first give basic motivation for the incremental approach and define it by taking an *a posteriori* view of the ME process. Second, different scenarios for refining methods are identified. Identification is based on analyzing the origins of ISD related experiences and distinguishing targets for making method refinements. The section concludes by discussing differences between incremental and more radical ME approaches. This allows us to describe local method development situations where the proposed principles are most suitable.

5.1.1 Motivation and definition

The motivation for incremental ME comes from our re-evaluation of method use (Section 2.5) and from the limitations of current ME principles (Section 3.2). In short, we claim that the situational applicability of methods is difficult to achieve solely through *a priori* ME principles. Major reasons are that 1) the required information for method construction is not sufficiently available, 2) the criteria that can direct method construction are difficult to identify beforehand, and 3) the ISD environment evolves.

Availability of method knowledge

In relation to availability, there are not many detailed metamodels available, nor readily applicable frameworks of ME criteria which are "filled" with known situational characteristics and related to metamodels. Most of the metamodels — which come with metaCASE tools, repositories, or are described in reference books (e.g. Olle et al. 1991) — focus on a limited number of methods and/or on only specific types of methods (e.g. object-oriented methods). Moreover, the metamodels described in books are not usually specified unambiguously and are at a relatively coarse granularity, at least when compared with the detailed metamodels required to model operational techniques. As a result, the pool of methods specified with metamodels for comparison and selection is small. Moreover, the frameworks of ME criteria, like the contingency frameworks applied to ME (i.e. Punter and Lemmen 1996, Harmsen 1997) focus on only a few aspects of methods applicability, and only on general method knowledge (see Section 3.2). The combination of metamodels and ME criteria into a larger baseline is difficult: different metamodeling languages focus on different types of method knowledge at different levels of detail and are not usually related to detailed method knowledge. Even assuming that a large body of metamodels and related ME criteria were available, the maintenance of this knowledge would be a huge or even unrealistic task. This fact also partly explains why contingency frameworks operate with method knowledge at a general level.

As a result, organizations are forced to test methods and try to make them more applicable by "learning" and d eviating from them while they are used. After all, organizations can not stop developing their own variants although theoretically this can be assumed (e.g. Wynekoop and Russo 1993). Similarly, the proposed incremental principles take an *a posteriori* view of ME by seeking to understand the applicability of methods through an organization's experiences. The *a posteriori* view also forms a key distinction to current ME principles reviewed in Section 3.2. It must be emphasized that typically the experience gathering and method refinements are carried out haphazardly and on a trial-and-error basis without any systematic principles (Smolander et al. 1990, Hughes and Reviron 1996).

Availability of method engineering criteria

We claim that it is difficult, if not impossible, to identify beforehand all relevant criteria for method construction. For example, the ME criteria of van Slooten and Hodes (1996) — resistance of end-users, aspects of the system to be analyzed, and management commitment — can hardly be known completely beforehand. Furthermore, as argued in Section 3.2 their relationship to method knowledge is not clear. In fact, van Slooten and Hodes apply the contingency framework in an *a posteriori* manner to analyze whether the criteria proposed in their framework have affected past projects and thus could be relevant to ME. How they can be applied to construct methods is not discussed. In contrast to the current ME view, cases of local method development (Jaaksi 1997, Tollow 1996) show that the characteristics and problems to be solved with methods were not known beforehand because of uncertainty about the problems.

As a result, we claim that the requirement for complete prior knowledge is both idealistic and unrealistic. Consequently, in situations of uncertainty and limited information, the incremental principles focus on improving method applicability through promoting small changes to methods while an organization obtains experiences and learns both about the method and about the IS domain. Although this option is partly dictated by practical needs, it also allows the creation of new knowledge based on experience, regarding both the method and the ME criteria. This is important, because current ME approaches rely on the existing body of information about both methods and ME criteria. Therefore, ME must be viewed as a learning process in which experience of successful (or unsuccessful) ISD efforts needs to be incorporated into future ME efforts: every use situation of methods should evaluate and analyze methods with a view to improving them. In fact, keeping the situational dependency of method use in mind, the most reliable information about method applicability can be obtained from an organization's own experiences. This experience-based learning is generally an incremental process (Miner and Mezias 1996), and a main argument in favor of incremental ME.

Evolving information system development environment

A method use environment is hardly stable because situations can change even during a short ISD effort. These changes also affect the applicability of methods, leaving two options for method engineers: either continue the use of the method in its current state, or modify it to support the new situation. The former option is chosen at the cost of applicability and the latter at the cost of making a new version of the method and transforming models which have already been made. This topic is discussed in Section 5.1.3. Changes in ISD situations are common, and can be seen also in the documented ME cases (e.g. Cronholm and Goldkuhl 1994, Nissen et al. 1996, Jaaksi 1997). These show that once methods have been adapted to tools, requirements for maintaining and modifying the methods for new situations appear immediately. In fact, some of the requirements occur already during tool adaptation, or after a pilot use. As a result of this evolution, methods must be refined continuously.

At the level of the ME criteria, changes in the situation have been identified as changes in contingencies, shifts in problems of ISD, or changes in stakeholders' requirements and values (Kumar and Welke 1992, Joosten and Schipper 1996). On the contingency side, one of the main reasons for introducing an ME approach is the inflexibility of contingency based method selection (Kumar and Welke 1992): in the worst case, a change in one contingency could lead to a selection of a totally different method. Similarly, a typical shift in problems to be solved (Checkland 1981) necessitates changes in methods: only in the case of a tightly defined and enclosed system can a method be presumed to be applicable every time. Finally, neither stakeholders nor their requirements are stable. People participate at different times and new people can raise different requirements and have different values which need to be reflected in methods (Nuseibah et al. 1996). Similarly, stakeholders' assumptions change and they can not know all the relevant criteria (Joosten and Schipper 1996).

To summarize, an incremental approach extends, rather than substitutes, the current principles of ME by focusing on experiences of method use, i.e. on an *a posteriori* instead of an *a priori* view. By evaluating the applicability of methods in a given situation it aims to manage and refine methods. The accumulated experience can lead organizations to extend, modify or purge any part of the method knowledge, such as concepts, constraints, or notations. These refinements are gradual and small in nature, hence the name incremental for the proposed approach. Gradual means that method refinements are applied to the method currently used in a given situation, instead of selecting a radically new method. Small changes are a consequence of the gradual changes: applicability is achieved by modifying selected parts of the existing method knowledge. Before describing the incremental principles in more detail we first take a closer look at method evolution.

5.1.2 Scenarios of method evaluation and refinement

In the following subsections we analyze incremental ME according to two dimensions: the source of experiences and the target of method refinement. Identifying the sources of experiences allows us to find mechanisms to collect experiences and make them available for method engineers. Experiences can lead to method modifications in different phases of the ME process. This is described through the target of method refinements. In the following we shall also address relationships between different types of method knowledge by describing how requirements imposed on a method's conceptual structure affect notations and supporting tools, or vice versa.

5.1.2.1 Sources of experiences

A requirement to modify a method arises when the method does not meet the situational requirements. These requirements can be collected while adapting a method to a tool, while introducing a method into an organization, or while using the method[23]. Each of these alternative sources can lead to iterations in the ME process and to a method modification. In each view the applicability of a method is determined based on different criteria:

1) **Tool related** feed-back occurs when a customizable modeling tool has limitations to support the constructed method (Cronholm and Goldkuhl 1994) or it offers possibilities which have not been considered earlier (Tolvanen and Lyytinen 1993). For example, most of the ISD methods used today still follow a "pen -and-paper"-mentality: they do not take full advantage of computer-based modeling environments. Therefore, the applicability of the method is determined here through a method-tool companionship.

2) **Introduction of methods.** Method refinements can also originate from the introduction or "pilot" use of methods (cf. Nissen et al. 1996), when a larger group of stakeholders can analyze the constructed method and tool support. Here a method is typically assessed in terms of its supporting materials, like tutorials, manuals, example solutions and reference models, as well as features of the method supporting tool. The applicability is determined mostly according to the pedagogical aspects, like how easy it is to learn and introduce the method into an organization. Some ME approaches focus on constructing methods so that they work as a learning device for teaching ISD methods (cf. Mathiassen et al. 1996).

3) **Experience-based** feedback occurs when developers face situations in which they feel that the constructed method is, or is not, applicable. If the method is considered inapplicable, they may rely on their experience more than on the use of the method. Hence, the applicability of the method is viewed in the light of current circumstances. This type of feedback is important because it is founded on actual method use. Several researchers (cf. Wood-Harper 1985, Galliers and Land 1987, Galliers 1992, Checkland 1981, Grant et al. 1992) have

[23] The method modifications can also occur while selecting or constructing the method, but we do not consider them because they are discussed in available ME approaches (cf. Punter and Lemmen 1996, van Slooten and Hodes 1996).

also emphasized the importance of the problem situation in which the method is used as a basis for evaluation.

In this thesis our main interest is on experiences related to method use, although the other sources mentioned are also possible starting points for iterations. Modifications which arise from the functionality of the tool are mostly dealing with technical issues and not related to the applicability of modeling techniques. In other words, our primary focus is not on the evaluation or improvement of modeling tools and their support for method-tool companionship. However, we claim that this type of empirical approach is required for evaluating tools (Tolvanen et al. 1996). Similarly, we do not consider here the effect of teaching approaches, other method-related materials, or the effects of piloting approaches, although we acknowledge their importance.

5.1.2.2 Targets of method refinement

Collected experience can lead to method modifications at different phases of method development. This dimension describes whether the iteration leads to re-select a method or its parts (i.e. start the ME process from the beginning), to construct the method differently, modify the method only to achieve better tool companionship, change the way in which the method is introduced, or to interpret the method differently. Each target is described in the following.

1) **Refine the method while using it.** This possibility means making different interpretations and giving different meanings to the method knowledge. This type of refinement takes place in learning-by-doing when an individual learns by developing an IS. Moreover, it must be noticed that this type of refinement can often occur without any language or documentation. Hence, the refinements can also be tacit (Nonaka 1994, Hughes and Reviron 1996). For method refinements this means that experiences about methods are not shared and thus not explicitly used to modify "intended" method knowledge. As a result, method refinements performed only while individual persons are using the method are difficult to study and systematize, because of the tacit nature of method-related experiences.

An individual person's refinements can be externalized (Nonaka 1994) by making them explicit and thus available to method engineers and other stakeholders. Experiences can also lead to organizational learning if they are collected and shared in some way with other participants. The remaining four phases of method refinement presuppose mechanisms to collect experiences and to make explicit changes to method knowledge.

2) **Changes in the introduction or "piloting"** phase deal with method refinements which change the way in which a method is taught; modify method-related materials, such as example solutions, tutorials, manuals; or demand more easily learned versions of the method. This last approach is commonly used both in text-book methods (e.g. Booch 1991) and in local variants (Jaaksi 1997) by developing "light" versions of the method. These are simpler and easier to learn, yet applicable for small or "first" projects.

3) **Refinement of the method in a tool** aims to make the use of the method easier with the tool. Examples of such modifications are changing the order in which the design information is added to the tool, or changing reports for consistency checking. Issues related to the tool only, such as the layout of dialogs and the order of reports and techniques shown in dialogs or menus fall into this category. Because these modifications change the behavior of the tool, they are explicit and well-structured changes, but deal with the surface structures (Wand 1996) of method knowledge.

4) **Re-constructing a method** represents more profound changes to the method knowledge. These include modifications to the initially constructed method knowledge: existing method components in the metamodel are removed or modified. A re-construction of the method also requires modifications to the underlying rationale applied for selecting and constructing the method in the first place. Ideally, each modification should be evaluated based on earlier knowledge of the method's applicability: why a certain methods was not applicable, and how the modification can improve its applicability.

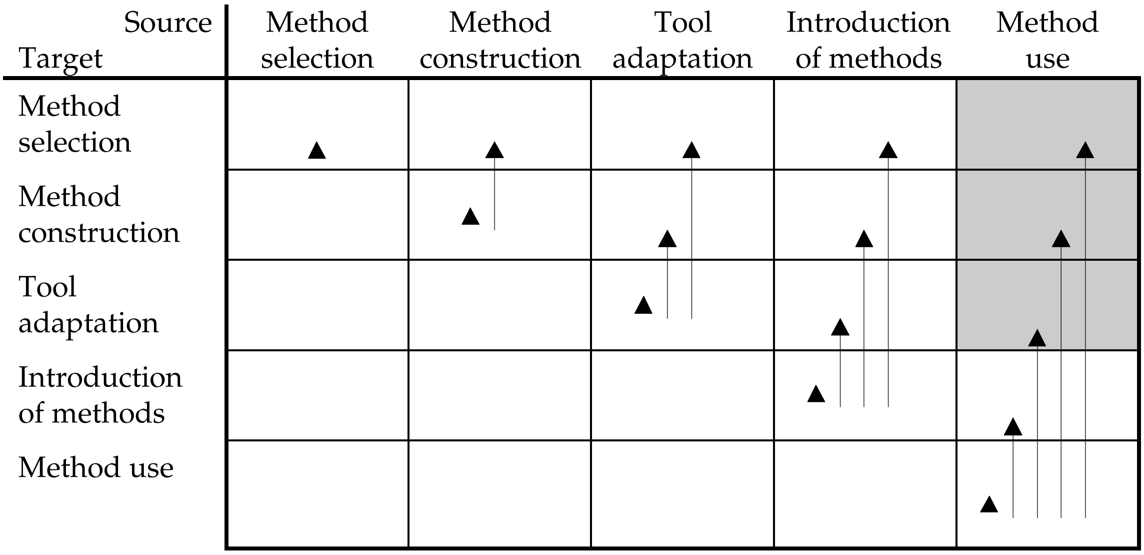
5) **Selecting a new method or its parts** deals with the most profound refinements to methods. Here new method components, like types, constraints, and modeling techniques, or totally new methods are selected. These changes are also typical when the re-construction requires new components or when unforeseen contingency factors arise, or there is a significant change in existing factors (van Slooten and Hodes 1996).

Together the two dimensions, the source and the target, form a space for possible method refinements. These are illustrated in the cells of Table 5-1. The horizontal axis shows the sources of experience and the vertical axis shows the targets of method refinement. The arrows show all possible choices when method refinements can take place: from the starting point of an iteration to the point of making the refinement. These scenarios are important because they allow us to restrict our view to experiences based on method use, and their influence on method refinements.

According to the possible scenarios, experiences about methods can be collected before, during or after the use of the method. Our interest in method refinements is in those related to experiences gathered from method use which can be externalized, i.e. represented, analyzed and refined with metamodels. These situations are grayed in the table. Thus, the first four phases of the ME process, which can also raise requirements for method refinement, are not considered in this thesis. They expect evaluations to be carried out before a method is used and are already partly covered in *a priori* ME approaches. Hence, we believe that the applicability of methods can be known only when the method is used.

The selected scenarios also reflect the depth of method refinements because all refinements made to metamodels expect modifications to be made to later phases of ME, i.e. to tools and their introduction phase. These are, however, also supported in most of the ME frameworks (see Section 3.2).

TABLE 5-1 Scenarios for incremental method refinements.



Source Target	Method selection	Method construction	Tool adaptation	Introduction of methods	Method use
Method selection	▲	▲	▲	▲	▲
Method construction		▲	▲	▲	▲
Tool adaptation			▲	▲	▲
Introduction of methods				▲	▲
Method use					▲

5.1.2.3 Refinements between types of method knowledge

Method refinements can not always be carried out by modifying only one type of method knowledge. Instead, during the construction phase the modifications are interrelated: changes in one part of method knowledge cause changes in other parts of the method.

Based on our focus on modeling techniques, we shall analyze only two fundamental types of method knowledge subject to modifications, namely the conceptual structure and the notation. These types and their relationships are also identified by Kronlöf (1993) and Jarke et al. (1998). Thus, other types of the method knowledge shown within the shell model (Figure 2-2) are excluded. However, to emphasize the companionship between a method and a tool, we also analyze tool-related method refinements. Therefore, method refinements can deal with the following interrelations: 1) conceptual structure and notations, 2) conceptual structure and modeling tool, and 3) notation and modeling tool.

1) **Conceptual structure and notations.** The most drastic modifications in a method occur when a large portion or the whole conceptual structure is changed, as when shifting from structured methods to object-oriented methods. Similarly, domains which are less mature, such as business modeling and requirements engineering, have less stable concepts and thus are more likely to evolve.

Because the underlying conceptual structures are typically the foundation of the method, changes in the conceptual structure affect other types (Jarke et al. 1998): notations which represent these concepts, processes which operate on these conceptual structures, and computer-aided tools which capture, store, analyze and retrieve the models representing those conceptual structures. For example, adding a new type to the conceptual structure requires changing the notations by adding a new notation for that type. Accordingly, the completeness of representations (Batani et al. 1992, Venable 1993), i.e. the availability of a notational construct for each concept, is a well-known criterion for dealing with the relationship between the conceptual structure and the notation. In contrast, changes in notations do not necessarily affect the conceptual structure (cf. Ryan et al. 1996, Kronlöf 1993). With respect to method refinements, we identify here a causal relationship among interrelated modifications: all changes to the conceptual structure should be made before changes in notations. Changes in methods, however, often arise from notations because they are the most visible part of the method.

2) **Conceptual structure and modeling tool.** Method modifications can also occur because a tool can not provide the required modeling functionality, such as an abstraction mechanism, a checking, or a form conversion. Here the method may need to have additional constraints or properties to enable consistency checking, reporting or code generation. For example, most of the object-oriented design methods do not recognize whether an inheritance is virtual although this information is required for generating header files for C++. Although these concepts are added to the tool, they are also defined and maintained in the metamodel.

3) **Notation and modeling tool** deals with the surface structure (Wand 1996) of the method knowledge: method refinements are made by modifying symbols and notations based on the graphical capabilities of the tool.

Although most of the metaCASE tools available provide method adaptation possibilities (Marttiin et al. 1993, 1996) they focus on modifications which are carried out when the tool is introduced. Accordingly, later method modifications are difficult, if not impossible (e.g. Cronholm and Goldkuhl 1994, Nissen et al. 1996). It must be noted that by method modifications we also refer here to situations in which the models made so far are updated along with the modified method (e.g. to support reuse of designs).

5.1.3 Incremental versus "radical" method engineering

Not all method development efforts are necessarily gradual or require small modifications to methods. In general, the literature on the development of business processes and on organizational learning distinguishes between radical and incremental approaches. For example, business process re-engineering (BPR, Hammer and Champy 1993) advocates a radical approach in terms of the rapidity and magnitude of a change, whereas total quality management (TQM, Flood 1993, Oakland 1993) relies on continuous small changes. Similarly, there is a wide-spread consensus on the distinction between incremental and radical models of learning (Miner and Mezias 1996).

Generally speaking, the type of change required and the type of learning are related: carrying out a radical change necessitates that the organization is capable of radical learning (i.e. to implement and introduce a large change quickly). Conversely, continuous small changes to existing processes expect incremental learning. Both approaches can produce benefits for an organization, and both types have advantages and disadvantages. In this sense they provide alternative strategies depending on how often the change is made and how large the change is. These alternatives are also valid in ME. By "radical" ME we mean *a priori* ME approaches in which methods are constructed solely in the beginning of each ISD project. This type of change is also the one most studied in ME research (see Section 3.2). It can be considered radical because each ISD project and each ME case is handled separately. A method is expected to be introduced once and no reflective learning during the method use is incorporated into methods. As described above, incremental ME is based on smaller and more gradual changes. At the extreme end of the scale, method refinement can be continuous and concurrent with the change requests from the method use environment.

Both modes of ME can be useful, depending on the situation, as both modes have their advantages and disadvantages: not all changes to methods can be radical, but on the other hand, small gradual changes may hinder development efforts which require more substantial changes. This also means that not all local method development efforts can be carried out according to incremental ME principles. Therefore, in the following we describe characteristics of ISD organizations or projects where incremental principles are most suitable. These characteristics are summarized in Table 5-2.

TABLE 5-2 Radical versus incremental modes of method engineering

	Incremental approach	Radical approach
Availability of method knowledge	Little	Considerable
Selection criteria	Uncertain	Known
Duration	Long-term	Short-term
Process maturity	Mature	Immature
Degree of methodical change	Small	Major shifts required (e.g. SA to OO)
Variety in target ISs	Few target ISs	Consulting house with multiple customers

As our motivation for the incremental approach showed, ISD environments exist where high levels of uncertainty and unavailability of method knowledge are typical and applicable for incremental principles. Areas of ISD where there are few methods available are, for example, the development of inter-organizational ISs (cf. Tolvanen and Lyytinen 1994), hypermedia systems (Isakowitz et al. 1995), and networked business processes.

Second, the longer an ISD project takes the more an organization will garner experience and the more likely are method modifications. Moreover, longer projects are also often larger and technically more complex, necessitating approaches to combine methods. Similarly, in long-term ISD efforts the technologies used may change and these changes need to be considered.

Third, as emphasized in our re-evaluation of method use (Section 2.5.3), successful method improvements are tied to an organization's own experiences and to the level of maturity. Therefore, ME efforts relate to the maturity of ISD

(Humprey 1988): organizations must have methods in use and an ISD process defined before any systematic experience gathering can be carried out. Also, method refinement efforts expect that methods are specified — otherwise their improvement is difficult (Jarke et al. 1994, Odell 1996). This means that an organization using incremental ME principles must be at least at the defined level according to the SEI maturity levels. In fact, the higher levels of maturity can be partly achieved by using incremental ME principles, in which methods are managed and optimized for the current situation. With respect to maturity, the organization's current method situation reflects the mode of ME. Incremental ME is not necessarily an optimal strategy for initiating radical changes in ISD, e.g. adopting methods to be used for the first time, or moving from structured methods to object-oriented methods.

Fourth, incremental ME principles are more applicable for organizations which can invest in method knowledge. Quite often this is possible only when the method knowledge can be focused on specific areas. These types of situations are typical in ISD organizations which focus on longitudinal projects and on developing a limited number or type of applications. In contrast, consulting houses which provide services to other organizations find their choice of methods largely determined by customers' requirements. As a result, method requirements can change from one customer to another and accumulated knowledge can not be utilized as effectively. Hence, in these cases the method selection and use can usefully be radical.

Finally and perhaps most importantly, one reason for following either of the ME approaches comes from their projected costs and benefits: how to change the method without discarding expensive investments in technology and methods. With respect to the technology investments, metaCASE tools are seen as offering one solution (Seppänen et al. 1996), as they decrease the costs and resources needed to manage method knowledge, and also provide a platform for cost-effectively building new CASE tools for a changed method, or different versions of methods. This also explains our interest in analyzing method use in modeling tools: metaCASE tools which support situation-specific evolution of methods are already available (Kelly 1997), but ME principles are not. With respect to method investments and the process of finding, analyzing and refining methods, it is the task of this thesis to decrease the costs related to ME. In other works, the incremental ME principles provide mechanisms to identify method refinement possibilities, manage method evolution, and automate part of the method refinement process. These decrease the costs of method improvements and thus of the benefits obtainable from engineering methods appropriate to the situation.

5.1.4 Summary

Most of the ME frameworks are based on an *a priori* view of ME in which no method refinements are expected during or after method use (cf. Section 3.3). Therefore, no principles or systematic guidelines for ME during and after method use have been proposed. To overcome this narrow view we analyzed the possibilities for iterations in the ME process. These possibilities were called method refinement scenarios. Based on this analysis we focused on specific scenarios which originate from method use experiences and lead to modifications in method knowledge or method-tool companionship. Hence, our interest is concerned with the deep structure of method knowledge: modifications of the conceptual structure, and its relation to notations.

Experience gathering and refinement should take place in modeling tools. This means that modeling experiences from tool-based method use are collected and analyzed. Although the analysis of "pure" tool support deals mostly with minor changes in the surface structure of a method (Wand 1996), our focus on method use with modeling tools is important for a number of reasons. First, method use in CASE forms a foundation for examining the underlying conceptual structure and notations. It makes ISD more transparent and the products accessible. Second, through the use of metaCASE technology we can allow method engineers to inspect the usage of methods. Third, by necessitating the use of a formal metamodel we expect that method modifications can be made explicit and formal. Finally, the resulting method refinements are also put into use through the tool adaptation. This means that method modifications can be shared quickly and can lead to learning in a whole ISD organization.

An incremental approach can be distinguished from other ME principles by identifying when and how method knowledge is constructed. In the radical mode, ME is viewed largely as an *a priori* method construction process, whereas in the incremental mode experiences of method use are collected and analyzed for the purpose of method refinements. Our focus is on this latter view. We claim that the applicability of the method can not be achieved based on *a priori* construction, but instead need to be investigated while using the method. This allows us to evaluate not only the applicability of *a priori* constructed methods, but also the relevance of the criteria that drive method construction.

We also identified ME situations which are most suitable for the proposed ME principles. These situations are characterized by high uncertainty and unavailability of method knowledge, and by a changing ISD environment. Hence, ISD projects which lack method knowledge or criteria for method construction benefit from incremental ME principles. Similarly, organizations which carry out long-term ISD efforts, often related to specific products, can take advantage of the incremental approach. Finally, incremental ME forms a part of various types of ISD improvements (Humprey 1988, Odell 1996). Therefore, organizations searching for long-term process improvement need incremental ME principles to improve their ISD processes.

5.2 Evaluating the applicability of modeling techniques

In this section we shall analyze some proposed approaches and their weaknesses in evaluating ISD methods. Most research on methods is based on the assumption of applicable methods. ME research is no exception: otherwise ME principles would not be proposed. ME research includes, however, one major difference to most method research, namely the situational dependency.

Whilst method developers often aim to prove the general applicability of methods, ME research is interested in improving the method use in given circumstances. We examine method applicability based on the situation in which it is applied. This is also emphasized in our re-evaluation of method use (cf. Section 2.5.1). Similar definitions are also proposed by Fitzgerald (1991) as an "applicability to the case or circumstances", by Schipper and Joosten (1996) as "serving in the intended purpose", and by Kitchenham et al. (1995) as focusing on specific cases in which a method is used.

Our focus is on studies which address the applicability of modeling techniques. First, we shall analyze which kind of approaches the developers of text-book methods have applied for validating their methods, and what validation approaches are proposed for situational methods. The former covers evaluation of methods in general and the latter focuses on evaluating methods in their use situations. Our interest is in approaches which can be used to collect, analyze and apply experiences to refine methods. This means that the evaluation approach should not only analyze whether and how a method has met the applicability requirements, but also how it could be improved. Therefore, the analysis does not include approaches which are not linked to evaluating method knowledge or which do not offer opportunities for method improvements, such as Kitchenham et al. (1995) and Jayaratna (1994). Second, we acknowledge some key problems related to method evaluation. Some of these problems are partly solved with the systematic ME principles proposed.

5.2.1 Evaluation and validation of text-book methods

The applicability of a method forms a core assumption in all method development. Most method development efforts, however, do not aim to validate the proposed method at all. Only a few are in any way proven or justified for the tasks for which they are promoted (Fitzgerald 1991). As a result, it is difficult to find how claims made in favor of a particular method — such as "more expressive, yet cleaner and more uniform ... than other methods" (Booch et al. 1997, p 15) or "to support a seamless ISD process, or the reversibility of models" (Walden and Nerson 1995) — can be proven.

By analyzing methods modeled in Chapter 4 two kinds of approaches for determining the viability of a method are used: a demonstration of the method in some imaginary or real-world case, and a comparison with other methods. Both of these approaches are also widely used in studies comparing and evaluating methods, e.g. those reported in the CRIS conferences (Olle et al. 1982, 1983). Neither of these approaches, however, can provide strong evidence for method applicability (Fitzgerald 1991). Demonstration (e.g. Yourdon 1989a, Booch 1991) describes how one or more ISs are captured into models. The subject of validation is mostly a modeling technique (i.e. concepts and notations) rather than the process or design objectives which explain the use of such models. Little information is given about the background of the cases, such as contingencies, participants, method users, or alternative solutions. Weaknesses of the method are not considered or mentioned at all. As a result, "validation" means here only that the method can be used in modeling, rather than that it is useful or can lead to better results than other methods.

The latter, the comparison approach, focuses on similarities and differences between the proposed method and other similar methods (e.g. Firesmith et al. 1996, Booch and Rumbaugh 1995). The proposed method is typically used as a yardstick: little wonder that it is often described to be more comprehensive than others. The main emphasis in evaluation is on explaining why the new concepts are required. Here the justification of the method is normally explained at the type level only, because method use is not addressed at all. Comparisons thus focus mainly on modeling techniques and underlying conceptual structures. One reason for focusing on modeling techniques only is that method books do not usually describe other parts of method knowledge systematically. As a result, the main argument for a particular method is based on the endorsement by an authority.

Yet, as is described in method books[24], one can state that the best "proof" for a method is its use: an assessment of a number of ISD efforts following a method shows its viability. Use of a method's popularity as a mechanism to prove its applicability is questionable. First, based on this strategy certain modeling techniques, like ER diagrams (Chen 1976) or data flow diagrams (Yourdon 1989a), should be considered to be applicable. There is, however, a great number of dialects available for these techniques. Also, criticisms against some of the principles they apply have been raised. The different versions of the ER model (e.g. Chen 1976, Batani et al. 1992, Teorey 1990) show that no single variant of the ER model is popular. Similarly, criticisms against the top-down decomposition applied in data flow diagrams has been presented (e.g. Goldkuhl 1990, Booch and Rumbaugh 1995). For example, Goldkuhl (1990) claims that top-down refinement of system leads to costly maintenance of designs and to a loss of information between the different levels. Second, the low acceptance of methods in general (cf. Section 2.4.1) and their adaptation in particular (cf. Section 2.4.2) indicates that most methods as proposed by their developers have failed. Therefore, instead of a general validation of methods, we are more interested in the validation of methods in a given situation. This is discussed in the next section.

5.2.2 Evaluation of methods in the problem context

In our literature review we found only a few studies that aimed to validate the applicability of methods more systematically. These are the validation of action modeling by Fitzgerald (1991), and of trigger modeling by Schipper and Joosten (1996). These can be distinguished from earlier method evaluations or validation approaches by their use of explicit measurements as a basis for the evaluation. In the following these approaches are briefly described by

[24] It may be the case that some validation efforts have been carried out but not described. Similarly, it is most likely that evaluations are performed during method development, but it must be noted that method developers have not described how this has been carried out: i.e. how data is collected, how it is analyzed, and how it has led to improvements in the method.

explaining the context in which they are used, what aspect of the method is evaluated, how data is collected, and what measures are used. Finally, we describe how the evaluation results are interpreted and applied for method refinements. The results of this analysis are summarized in Table 5-3.

TABLE 5-3 Summary of validation approaches

Approach	Fitzgerald (1991)	Schipper and Joosten (1996)
Applied for validating...	technique for action modeling	technique for trigger modeling
Context of evaluation is...	modeling technique as a part of a method and ISD process	one modeling technique
Rationale of method covers...	design criteria and objectives of the technique	method developer's intentions and supporting arguments
Data collection is based on...	examples of object system representations from different modeling domains	intention-related instruments, such as literature study, method metrics, analysis of deliverables, interviews of method users
Target of validation is...	richness of the technique in terms of its modeling power	method developer's intentions with the modeling technique
Validation is based on...	method users' opinion of modeling power	recognized arguments which support intentions, users' priority for intentions
Method refined...	-	if intentions do not change and earlier observations are valid after refinement

Fitzgerald (1991) evaluates the richness of a technique in terms of what is abstracted, i.e. its modeling power. He starts the evaluation by describing the objectives and design criteria of the technique. This forms the basis for the evaluation. He also describes the rationale of the technique (i.e. why the technique has been constructed as it is). In the evaluation the modeling technique is related to the larger context of the whole method, but no clear distinction is made between arguments in favor of the method in general, and those arguing for the specific modeling technique. The evaluation is carried out through modeling and trying to find out how well relevant aspects of the object system could be represented. The main instruments for data collection are the use of examples from different situations (but nothing is explained of the type of situations they were (e.g. domain, contingencies)), and why they were selected as modeling subjects. The results are derived based on the researchers' (acting as method users) opinions of the richness and modeling power of the technique. Thus, the evaluation approach, as noted by the author, is highly subjective and dependent on the selected modeling situations. Furthermore, the approach does not present possibilities to refine methods during or after the evaluation.

Schipper and Joosten's (1996) contribution to method evaluation is their proposal and use of multiple evaluation instruments. They base their approach on reviewing how validation and evaluation of modeling techniques are studied in other modeling-related areas, and what types of validity are recognized in the literature. They propose a model of validation which focuses on observing how the intentions of the method developer, in terms of associated characteristics of a method, are met. The approach focuses on evaluating a modeling technique separately from other parts of the method, and starts by describing the method developer's intentions (e.g. to allow modeling of logistic processes) and characteristics (e.g. easy to learn) for the modeling technique. Next, the rationale for the technique is stated by arguing how the intentions are met and relating them as characteristics of the modeling technique. This method construction rationale is derived similarly in Fitzgerald (1991), but Schipper and Joosten also include characteristics other than those related to modeling power. Therefore, the instruments for observing the characteristics are also different. Schipper and Joosten (1996) propose and emphasize the use of instruments (both qualitative and quantitative) in validation depending on the type of intentions. Various instruments can be used simultaneously to check convergent and discriminant validity. The instruments proposed include a literature study, method metrics, analysis of deliverables, content analysis of interviews and measurement scales for ease of use and usefulness. Of these, method metrics (Rossi and Brinkkemper 1996) and ease of use and usefulness (Davis 1989) are instruments which are not used by Fitzgerald (1991). A major reason for this difference is that Fitzgerald has carried out the validation effort by himself, whereas Schipper and Joosten target their studies to developing an automated method selection procedure in CAME, i.e. to be used also by people other than the method developer.

The study by Schipper and Joosten, however, does not describe how the instruments are applied for data collection and analysis, nor do they provide examples in favor of the evaluated trigger modeling technique. The result of the validation effort should be the list of intentions and arguments derived from the observations based on the instruments. To illustrate the approach some examples are given. A goal to model business processes quickly can be supported if method metrics show that the method is not complex, or users consider it effective and quick to use. Because of the use of multiple instruments the observations made can provide better evidence for how successfully the method fulfills its developer's intentions.

Finally, unlike Fitzgerald, Schipper and Joosten allow method refinements during the validation process to improve the applicability of the modeling technique. The modifications should, however, ensure earlier observations and intentions remain the same. Although these conditions are understandable, since they focus on validating a fixed method, they do not recognize experience based learning, uniqueness of situations and method evolution. Accordingly, in the following we shall analyze method evaluation approaches which accept, or even promote, method evolution.

5.2.3 Evaluation of methods as a part of a continuous ME process

As discussed in Section 3.2, the evaluation of the applicability of methods in ME research is dominated by *a priori* evaluation occurring in the method construction phase. To our knowledge, only the learning based approaches to method development (Checkland 1981, Kaasbol and Smordal 1996, Wood-Harper 1985, Mathiassen et al. 1996) indicate the importance of experiences and learning from method use as key mechanisms both to evaluate and to refine methods. Checkland (1981) advocated the learning based approach to method development and evaluation by introducing a cycle of action research in which experiences on method use provide the main source for method modifications; first by using the method and second by learning method use. This cycle is illustrated in Figure 5-1.

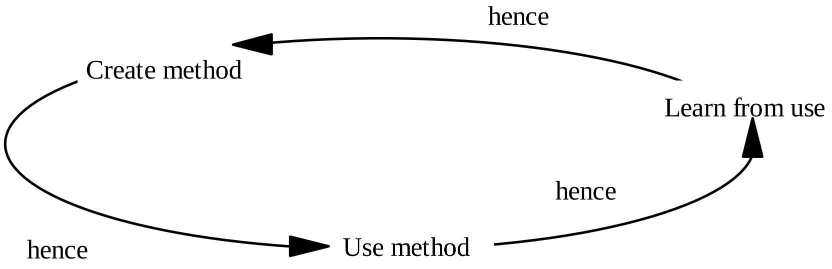


FIGURE 5-1 The evolution of a method through a learning cycle (Checkland 1981, p 254).

According to this cycle, ME can be viewed as a continuous and never-ending process, in which experiences are elicited from working with the method. Checkland has used the action research cycle as a key mechanism to develop Soft Systems Methodology (SSM) by repeating the cycle in many development cases and situations. In fact, the cycle for developing SSM began in 1969. The main reason for the cyclic learning based approach seems to be the difficulty of developing methods into a new field. In the case of SSM this means development based on methods applied successfully in developing "hard" systems into methods applicable for soft, human activity systems. This point also confirms the motivation of the incremental approach to ME made earlier (Section 5.1.3).

Because the cycle of method evolution is carried out as action research it is sensitive to the context in which the method is used and thus situation-bound. Although SSM includes the idea of incremental ME the objective of the learning cycle has been to develop general or universal principles for developing human activity systems. In other words, the iterative cycles have not been used to develop SSM towards situation-specific needs in the same sense as in the ME literature, but rather towards learning about various situations in which the SSM is applied. Moreover, Checkland emphasizes that SSM is not a method in the same sense as defined in this thesis but something between a philosophy or a framework, and a method[25]. However, as Checkland notices, some parts of SSM are very close to our definition of modeling technique (e.g. CATWOE). Therefore, from the ME point of view, the learning cycle has been used to define method knowledge in terms of concepts, processes, assumptions and values. Because of our interest, we shall focus only on the incremental development of SSM's concepts and modeling techniques.

In Checkland's approach the applicability of a method is evaluated based on its strength as a working device in a process of developing human activity systems. As such it applies a general question for the evaluation: "Was the problem solved" (Checkland 1981, p 192). However, he does not provide detailed principles of how experiences are collected (other than case records), analyzed, and applied when starting the next cycle: i.e. creating or modifying the method. Of course it can be claimed that such more concrete and systematic principles exist but they are not reported. In general, the concepts and notations used to develop conceptual models are not defined and thus not evaluated according to any systematic principles. The only reported exceptions are the root definition according to CATWOE and the sequencing between the stages of the method. Especially the former is relevant for our study, since CATWOE is closest to our definition of a modeling technique. The applicability of the concepts behind the root definition (Customer, Actor, Transformation, Weltanschauung, Ownership and Environmental constraints) were studied by seeking a dozen well formulated root definitions from earlier projects to test that the concepts behind the mnemonic CATWOE could be found. As a result of this analysis, they conclude (Smyth and Checkland 1976) that the CATWOE concepts are relevant because they would speed up the process of finding root definitions and enrich debates. The sequence of the method's tasks is another example of experience based evaluation, although it focuses more on the process than on modeling techniques. In SSM the transitions between modeling tasks, and thus also between modeling techniques, are left open because examination of earlier studies has revealed that different starting points and sequences are possible. From a modeling technique point of view this means that "conceptual models" of the system under development can be made before root definitions or vice versa. An obvious reason why other parts of the SSM modeling techniques are not evaluated is the universal nature of the method: different human activity systems require different types of conceptual models — which can also be seen from the case studies documented — and therefore their validation in a universal manner is difficult (cf. Section 5.2.4).

The importance of Checkland's view of method development (1981, 1991) is that it highlights the continuous learning cycle and shows that this cycle occurs at several levels: the IS level, the method level and the ME level. Although Checkland (1981) did not promote the learning cycle as a mechanism to develop methods (in the same sense as defined in this thesis) other researchers have applied it directly to ME (Wood-Harper 1985, Avison and

[25] For the same reason SSM has not been included among the methods modeled in Section 4.

Wood-Harper 1990, Mathiassen et al. 1996, Kaasbol and Smordal 1996). These studies are described in the following.

The developers of Multiview (Wood-Harper et al. 1985) have applied an action research cycle when using and testing the method. Similar to SSM, Multiview was developed as a fixed method and thus it supports narrow situational adaptability (Harmsen 1997) through in-built flexibility. One major distinction of Multiview from other methods and from situational adaptability is that it follows a contingency approach to select among the several techniques it includes. However, concrete suggestions are not given either for ME criteria, or for choosing the components of Multiview (Harmsen 1997).

Mathiassen et al. (1996) have applied the action research cycle for developing a method, called OOA&D (Mathiassen et al. 1995). Here the applicability of the method is evaluated based on how it has supported teaching as a learning device. By eliciting experiences from the students in a class-room setting they have shortened the method refinement cycle. As with other studies, no concrete ME principles for collecting, analyzing and refining methods are given. The authors have, however, distinguished some types of method knowledge which should be specified in ME, namely concepts, guidelines, principles and patterns, but no further details are given about these. The first three are covered by our taxonomy: conceptual structure, process and design objectives. The last one, patterns, deals more with instance level information as it shows partial solutions to IS modeling tasks in specific domains.

These approaches have, however, several limitations in addition to their universal view of methods. First and foremost, they do not include any explicit mechanism to collect, analyze and refine methods. This would be required for more systematic ME. Thus, after method use, no mechanisms are used to study whether the method has been applicable. As such they are general frameworks of method evaluation, rather than applicable principles for evaluating and refining modeling languages.

Second, in all of them an iterative cycle is carried out by the method developers rather than by others. For example, in Mathiassen et al. (1996) the role of students in refining the method is not explained, nor is the frequency of modifications. As a result, the modifications are highly dependent on the method developer’s opinions. No indications are given as to how a larger group of stakeholders can participate in the cycle. In other words, the process and roles involved in ME are not described.

Third, based on what is reported, the learning cycle is applied at a general level rather than related to the method knowledge (an exception is the evaluation of the CATWOE concepts in SSM). Because method knowledge is defined loosely in these approaches, the approaches do not apply any ME languages or tools. If such a more systematic approach to method development had been applied (as proposed by Parsons et al. 1997), it is obvious that method knowledge could also have been specified and evaluated in more detail. One reason why such approaches have not been followed may lie in the aim of situation independent applicability: it is difficult to specify method knowledge in detail and at the same time for general purposes. A good indication of this can been seen in the development of the UML method and its versions (e.g.

Booch and Rumbaugh 1995, Booch et al. 1996, 1997) which have become less specified in terms of details documented in metamodels as the need to satisfy more general situations has increased.

To summarize, there is a surprising and disappointing lack of well-documented method evaluation cases, evaluation mechanisms, and criteria. As a result, it is hard to find out from the ME point of view why methods like OOD&A (Mathiassen et al. 1996), or Multiview (Avison et al. 1990) are constructed as they are. For example, which evidence from the use experiences show that a concept of a cardinality should be used in object models (Mathiassen et al. 1996) or in entity models (Avison et al. 1990)?

5.2.4 Problems of *a posteriori* evaluation

The analysis of the evaluation approaches, and especially their limitations, is not intended to be a criticism of the method development approaches. Rather it indicates the difficulty of method evaluation and why one of the key research questions, "Are methods useful?", has remained unanswered. In this section our aim is to discuss the difficulties in making *a posteriori*, use based, evaluations of methods. This view is important, since it allows us to describe how incremental ME principles could solve these problems, and which problems it can not solve.

First, one major reason why method developers have not evaluated or validated their approaches lies in the difficulty of such a task. By applying 'scientific' research methods to method evaluation and validation we can not satisfy requirements of scientific theory testing, which involves reducing domain complexity, controlling data collection, and meeting replication requirements (see Galliers 1985, Fitzgerald 1991, Grant et al. 1992). The application of a scientific method typically involves construction of an experiment so that only one or a few factors are identified and studied at a time. This involves breaking the research subject into smaller parts for examination with a smaller number of factors. Hence, the experiment is first conducted in a standard way and then a number of times with one factor changed (ceteris paribus). A larger set of factors can not be considered at a time because of their possible interactions. Thus, an understanding of the applicability of a method, i.e. the big picture, would be constructed on the basis of these small factors. This type of research setting is, however, hard to achieve in daily ISD practices.

The replication requirement is also difficult to meet in ME research because ISD and thus method use is considered situational, or even unique. In this sense, the requirement for replication could be met only in situations where the ME criteria are the same. Moreover, if differences in a method's applicability occurred between similar (in terms of ME criteria) ISD efforts, there would probably be factors which had not been identified. These factors could even be considered as candidate criteria for ME.

In terms of ME, the evolution should deal with inspecting the applicability of method knowledge according to the ME criteria used in the construction phase. In other words, *a posteriori* evaluation could focus on studying how *a priori* factors were satisfied. Was the method applicable in the expected circumstances and contingencies? Did the method help solving the development problems? Did the method satisfy its users' requirements? Because of the expected complexity of ME criteria it is difficult to study one or some of these in different cases and expect that other criteria do not interfere with the results.

Second, coming up with hypotheses that show the applicability of methods is problematic, because the hypotheses can not be formally tested. According to the scientific approach, when several independent studies have consistently supported the hypothesis it will become a theory or even a law. This type of proof of method applicability is not available, and as Fitzgerald (1991, p 662) sarcastically notes, this has troubled IS research very little. In the context of ME, confirming a hypothesis means that there is some evidence that a method has been applicable. For example, in the case of validating the root definition method, Checkland (1981, p 227) notices that the existence of CATWOE concepts does not guarantee a good definition, but it provides evidence that in a well-formed definition such concepts are used. Coming up with hypotheses is, however, important because we can reject them by finding aspects of applicability which were not fully supported (Kitchenham et al. 1995). In other words, incremental method refinements occur only when a method has not been fully applicable.

A third difficulty in studying method applicability is to ensure that the method has actually been used (Jarke et al. 1994). In terms of our ME scenarios this means that each source of experience should be based on verifiable experiences. In our subset of method knowledge, this problem is bounded: the study of method use in terms of modeling techniques is easier to analyze than the use of other types of method knowledge, such as process (as in Jarke et al. 1994), or that design objectives and assumptions of the method are actually followed. This is also an obvious reason why most validation approaches focus on conceptual structures and modeling techniques. This does not mean that the study of method use in terms of modeling techniques is without problems. For example, method users can apply other modeling techniques than those proposed by the method engineers, and the study of intermediate models, design sketches, or different working versions of models is labor-intensive and costly to analyze for the purposes of ME (Hofstede and Verhoef 1996).

Fourth, the acquisition of experiences is difficult because experiences are personal and subjective (Nonaka 1994), they deal with situations that occurred at one point in time (Schön 1983), and they are often tacit: not all experiences can be made explicit and thus used for method refinements. Not all method knowledge is explicit: practitioners' method knowledge is partly embedded in their practices and can not be fully described. Furthermore, collecting experiences can be time-consuming and costly. As a result, method evaluations and refinements seem to be highly subjective. For example, Fitzgerald (1991, p 668) believes "that the best that can be achieved is that people may be convinced about a technique's applicability and usefulness only by argument and example, not by any concept of scientific proof". It must be noted, however, that subjective perceptions and opinions are vital for the acceptance of methods.

Finally, it is difficult to find what has been the role of a modeling technique (Checkland 1981). A modeling language can be evaluated based on what it has abstracted from the current situation (Fitzgerald 1991) but whether it has provided alternative solutions or choices among them is more difficult to evaluate. As the analysis of the method evaluation literature showed, evaluation has mostly been based on the researcher's concern that the problem has been "solved" or the problem situation has been improved (Checkland 1981). On the level of a whole method, an evaluation can be carried out more easily (e.g. Kitchenham et al. 1995) because method knowledge can treated in its entirety. Thus, detailed alternative compositions of method knowledge can be neglected. For example, problem solving capabilities can be measured based on the number of errors in the developed program, or whether the IS developed satisfies the user's requirements. Hence, a method is treated as a whole. In addition, there remains a question whether the problem has been really solved with the method, or have they been solved through other means (e.g. the whole problem disappeared because of external changes). Naturally method users can judge the influence of methods, but evaluation research does not discuss enough how the method users' experiences are collected and analyzed for improving methods.

5.2.5 Summary and discussion of method evaluation approaches

In this section we have analyzed approaches for carrying out *a posteriori* evaluation of modeling languages. Our aim was to seek mechanisms for collecting and analyzing methodical experiences, because we believe that the applicability of a method can only be known when the method is used. In short, the analysis shows a lack of instruments for evaluation, and problems in carrying out such evaluations. There seems to be no generally recognized way to determine if a modeling technique has been applicable. The reasons are summarized below.

First, the most important limitation of the approaches is that they do not aim to apply evaluation results to improve the methods. Methods are considered as a whole and evaluation is not targeted to inspect them in more detail. Instead of making small changes to the methods, evaluators often seek to obtain a general proof or disproof. Second, none of the approaches describe the method evaluation process in detail and only Joosten and Schipper (1996) describe some explicit instruments for evaluation. Even in their case, the use of the instruments during the actual evaluation is not explained in detail (Schipper and Joosten 1996). Some of the instruments, like method metrics, do not deal with method use at all. Similarly, most of the instruments applied are used in snap-shot cases. Third, all approaches target the validation to situation-independent methods. Although they recognize various situations of method use, they do not recognize that a method could be situation-dependent. In terms of ME, the evaluation is not targeted only to study whether a method has been applicable in the current case. Some possible reasons for this focus are the search for generality, an aspiration to follow scientific methods, and the method developers' desire to prove their own methods.

To characterize the incremental approach in relation to the others described above, we have to focus on detailed method knowledge. Similarly, our primary aim is not to seek for a universal validation of methods following a "scientific" proof. Instead we focus on situational validation in which better applicability is sought by making gradual changes to a currently used method.

5.3 Principles for incremental method engineering

In this section we shall describe principles for *a posteriori* and continuous method engineering. These principles are described through the steps of incremental method engineering, and the mechanisms applied in each step. The steps deal with collecting experiences, analyzing experiences, and refining a method for a current situation. These steps are lacking from other ME approaches and together with *a priori* ME they form an "iterative loop" of incremental ME. Hence, we claim that both *a priori* and *a posteriori* steps are required. The *a priori* steps were already described in Section 3.2 and the *a posteriori* steps are described in the next section (5.3.1).

Throughout these steps we apply three mechanisms that seek to improve methods. These mechanisms are based on analyzing the differences between an intended and actual use of modeling techniques (Section 5.3.3), on studying the role of techniques in modeling object systems (Section 5.3.4), and on understanding how they support problem solving (Section 5.3.5). As the review of method evaluation approaches showed, these mechanisms are not the only possible ones. They are relevant for improving tool-supported methods and managing methodical changes through metamodels. Together with these method refinement mechanisms, we apply metamodels and method rationale to collect and analyze experiences as well as refine methods (Section 5.3.2). In comparison with other evaluation approaches these make the method improvements more systematic and render refinements visible.

5.3.1 Process of incremental method engineering

To extend the *a priori* view of ME approaches we propose some complementary principles. These extensions are illustrated in Figure 5-2. The data flow diagram shows the steps of ME, with the three steps of incremental ME illustrated by grayed processes. These steps deal with gathering experiences, analyzing method use, and refining a method. Together with the *a priori* steps, they form an iterative cycle in which method improvements can take place gradually using method stakeholders' experience (cf. Checkland 1981). In the following we shall outline each step and their linkages to the steps of *a priori* ME.

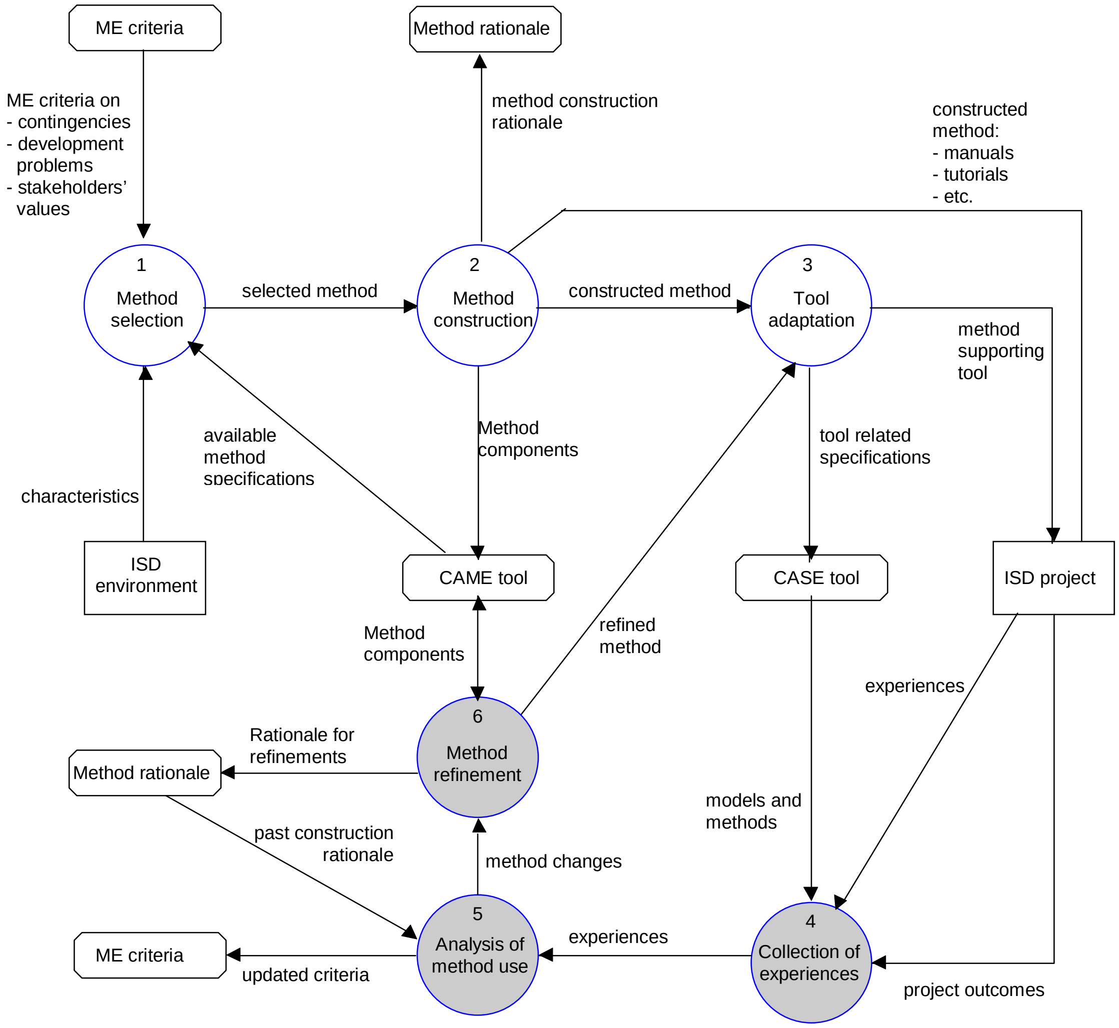


FIGURE 5-2 A data flow diagram specifying the incremental method engineering process.

In carrying out experience-based method evaluation the accuracy and availability of feedback must be enhanced. This improves experience-based learning (Huber 1991). The accuracy of collected experiences is enhanced by relating experiences to metamodels and to the method construction decisions. Their use is discussed in more detail in Section 5.3.2. The availability of method use experiences is enhanced by collecting models and metamodels, by collecting outcomes of an ISD project, and by interviewing stakeholders. The collection of models as deliverables is similar to the ideas proposed by Fitzgerald (1991) and Schipper and Joosten (1996). Models provide data on how modeling techniques were actually used. Because we focus only on meta-data models, the models only describe the end-result of method use rather than the modeling process. In the context of metamodel-driven modeling tools, the collection of models and metamodels can be automated since they are both stored in a repository of a metaCASE tool.

In addition to the model-based deliverables, the outcomes of the project are inspected. These deal with the results of the ISD process changing or improving the problem situation of the object systems. Interviewing method stakeholders obtains situational experiences of method use. Both unstructured and structured interviews can be used for data collection. An unstructured interview closely resembles a normal conversation and allows a method user to apply his or her own concepts and aspirations to specify method refinements. Typically a refinement demand becomes apparent from the modelers' observations of the limitations of a method in use situations (e.g. Tollow 1996, Jaaksi 1997). Structured interviews are based on predefined questions which are known to reveal refinement possibilities. The mechanism of incremental ME described in the remaining sections of this chapter forms the basis for questions for the structured interviews.

The second step deals with analyzing experiences in order to improve a method. This step is carried out by the mechanisms of experience analysis described in the following sections (cf. Sections 5.3.3-5.3.5). In short, the mechanisms deal with:

1) **Type-instance matching:** inspecting differences between an intended (i.e. metamodel) and actual use of a method (i.e. models).
2) **Modeling capabilities:** analyzing the capability of the method to abstract required aspects of the object systems into models and to keep them consistent.
3) **Problem solving:** analyzing the capability of the method to generate alternative solutions and support decision making.

The mechanisms are designed so that they reveal those aspects of a method which can be a target for refinements. In other words, if the analysis phase suggests a method modification it reveals that the *a priori* constructed method was not sufficiently applicable.

Evaluations of method use can lead to modifications of method knowledge and tool support. Modifications related to the conceptual structure or notation take place by adding, subtyping, joining and removing components of the metamodel and by specifying a related notation. Each of the metamodel-based refinements can be operationalized through the same metamodeling constraints as in the method construction (cf. Section 4.4). The re-constructed method is stored into a CAME tool, from which new components can also be selected. Tool re-adaptation is a necessity if a metamodel has changed (cf. method refinement scenarios, Section 5.1.2). Not all refinements, however, necessarily require changes in the method. There are changes that deal with the way the method is supported by the tool. For example, the consistency of model data can be improved by adding checking reports without modifying the metamodel. The modification of a CASE tool must be emphasized, because the advantage of method improvements comes when the refined method is used in a modeling tool. This enables the sharing of refinements and makes possible a new evaluation cycle.

An improved method is not the only outcome of the incremental approach, because the evaluation allows the creation of new knowledge for future ME efforts. Based on current ME approaches this knowledge should be related to the ME criteria in two ways: to confirm or to reject the criteria used in the method construction, or to add totally new criteria. In fact, the only way to use frameworks of ME criteria is to "fill" them with criteria that have worked in past situations. This necessitates that the realization of ME criteria is assessed in terms of new criteria, changed criteria, and whether the *a priori* set of criteria is still relevant. This means that method engineers should analyze the ISD environment continuously, not just for the initial method construction. Paradoxically, ME approaches which aim to apply available frameworks of ME criteria have neither validated them nor considered how information about situational applicability is found.

5.3.2 Use of metamodels and method rationale in incremental method engineering

As with most attempts at organizational improvement, improvements to the current state are difficult to make if the practices currently followed are not known. Changes can be made but no information is available on the effects of the change nor whether they can be considered as improvements. Similarly, incremental ME can not be carried out effectively if information about a method and reasons for its promotion are not known. The former, method knowledge, is described in metamodels, and the latter, method rationale, is described in ME criteria and decisions made during method construction. Both of these are used to collect, structure and analyze experiences. Use of them increases the accuracy of the cause-effect relationships between an engineered and a required method. Their use in ME is described in the following.

5.3.2.1 Metamodels in incremental method engineering

As in method construction, a metamodel makes method knowledge explicit. Incremental ME applies metamodels beyond the method construction step. In the first step of incremental ME, metamodels provide a mechanism to collect and structure experience: method stakeholders' comments, observations, and change requests can be related to the types and constraints of the method. This helps make experiences explicit, and helps focus on those experiences which are related to the method.

For the analysis step, metamodels allow the detection of those parts of the method which are subject to further analysis. The analysis possibilities are available through the same metamodeling constructs that were applied in describing the method. As in method construction, alternative method refinements can be made and compared by using the metamodeling constructs. During an iteration of the incremental approach, metamodels provide a history of method refinements, since all changes to the method can be found by comparing metamodels made at different points of time. Figure 5-3 illustrates the method evolution through "constellations" of metamodels.

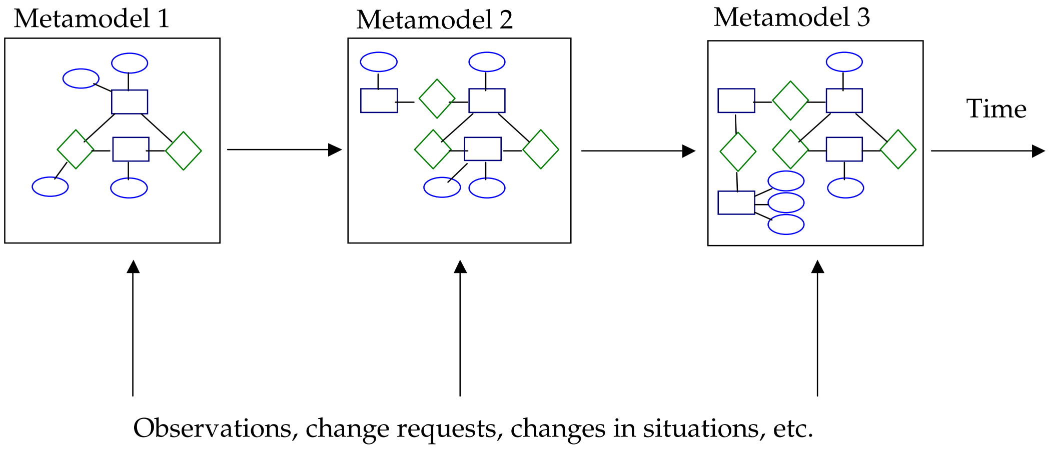


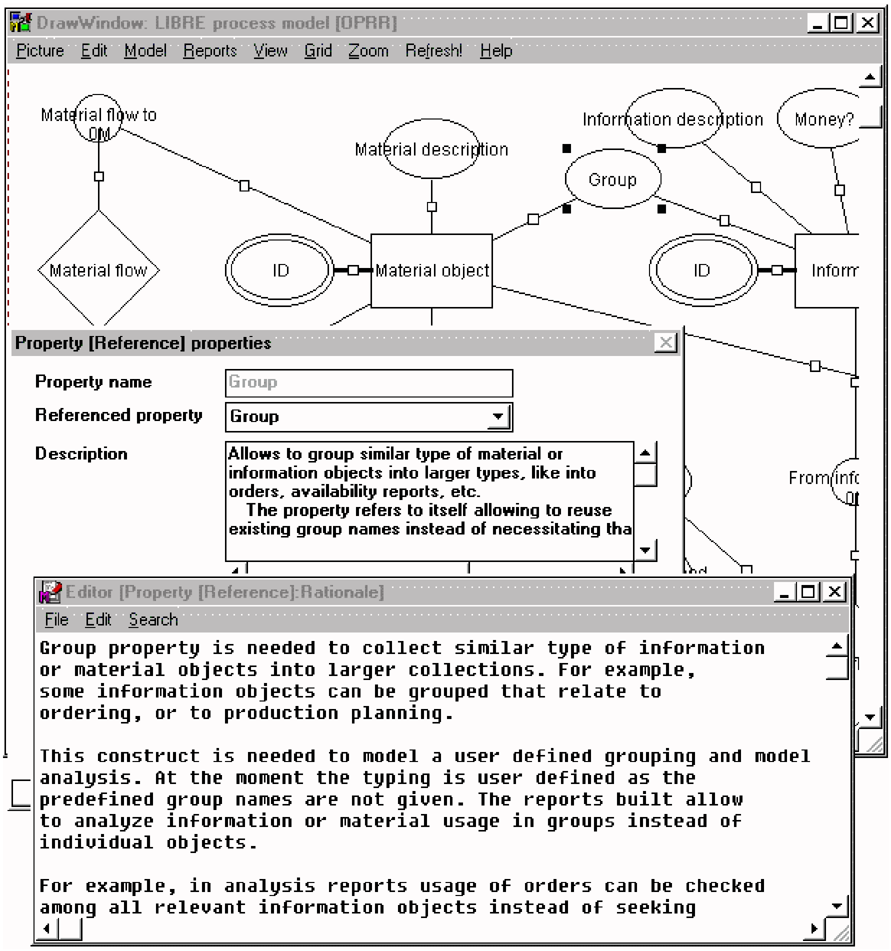
FIGURE 5-3 Method evolution in metamodels.

5.3.2.2 Method rationale in incremental method engineering

Metamodels alone are inadequate to manage method refinements, because they can not explain the evolution of a method. Therefore we need method rationale. Method rationale occurs at two different levels depending on the users (Jarke et al. 1994, Oinas-Kukkonen 1996). For method engineers, method rationale is an explanation why certain types or constraints of the method are included in the constructed method. We call this a method construction rationale. Ideally, each type and constraint in a metamodel should be justified. A sample of method construction rationale from our action research study (cf. Chapter 6) is given in Figure 5-4, in which an explanation for a 'group' property type is given.

The topmost window describes part of the metamodel in which a 'group' property type is defined. The middle window shows specifications relating to the property type. These include the name of the property type, that an instance of the 'group' refers to values of existing groups, and an explanation of the type for method users. The lowest window describes the reason why the 'group' property is needed. In the example, the rationale for using the grouping is the need to collect similar kinds of information or material objects. For example, an analysis can include information about business processes which only use information related to orders, such as sales orders, quick orders, repairing orders, orders sent by someone other than the original customer, etc.

Instead of applying a predefined schema for method rationale we have left it unstructured. Use of predefined schemata could limit the possibilities of information gathering, since there are not many studies on method rationale (Jarke et al. 1994, Oinas-Kukkonen 1996).


DrawWindow: LIBRE process model [OPRR]
Picture Edit Model Reports View Grid Zoom Refresh! Help
Material flow to OM
Material description
Information description
Money?
Group
Material flow
ID
Material object
ID
Inform
From info
Property [Reference] properties
Editor [Property [Reference]:Rationale]
File Edit Search

Property name: Group

Referenced property: Group

Description: Allows to group similar type of material or information objects into larger types, like into orders, availability reports, etc.
The property refers to itself allowing to reuse existing group names instead of necessitating tha

```
Group property is needed to collect similar type of information
or material objects into larger collections. For example,
some information objects can be grouped that relate to
ordering, or to production planning.

This construct is needed to model a user defined grouping and model
analysis. At the moment the typing is user defined as the
predefined group names are not given. The reports built allow
to analyze information or material usage in groups instead of
individual objects.

For example, in analysis reports usage of orders can be checked
among all relevant information objects instead of seeking
```

FIGURE 5-4 An example of method rationale for a ‘group’ property type.

This detailed example also reveals the gap between currently proposed ME criteria and their linkages to detailed metamodels: none of them support relating situational requirements to individual types or constraints of a method. Some of the ME approaches (e.g. Heym 1993, Harmsen 1997), however, support relating information about method use situations and contingencies to metamodels based on predefined schemata. For example, Heym and Öterle (1992) collect experiences in terms of the focus of the method (e.g. project management, risk management, IS development), application type (e.g. expert, office or real-time system), and phase of the ISD life-cycle (e.g. analysis, maintenance). A similar approach is followed in MEL (Harmsen 1997). These approaches, however, do not explain how these more detailed descriptions are obtained, nor are they related to detailed metamodels.

The use of method rationale in incremental method evaluation necessitates that more detailed construction explanations are related to metamodels, instead of referring solely to ME criteria. It helps in understanding the effects of method modifications: what capabilities are lost from the original method if a method element is removed or changed. It also makes possible argumentation about possible new method types.

Method users can understand method rationale differently. For them method rationale explains why certain types or constraints of the method are or are not used in models. We calls this method use rationale. The collection of method use rationale is important because it reduces the subjective flavor of experiences, makes a decision on method use more explicit, and allows users to relate their method experiences directly to method knowledge. This is important, since all experiences are individual, and therefore can be either supporting or contradictory.

The rationale of method use, however, is not normally documented and to our knowledge none of the modeling tools allows the capture of decisions about method use; only decisions about design choices (i.e. design rationale (Ramesh and Edwards 1993)). Therefore, it is the task of method engineers to collect the rationale of method use. A similar data collection approach is followed by Wijers (1991) while eliciting individual developers' modeling knowledge. It must be emphasized that Wijer's studies are not related to *a priori* and restrictive method knowledge. *A priori* means that methods are not improved. Instead, existing practices are documented with metamodels. The restrictive method knowledge means that modeling was not following an "engineered" method in the same sense as in tool-supported modeling. This means that in Wijer's study, the modeler's own method knowledge was allowed, and in fact intentionally sought. In our case, the tool ensures that models are always related to modeling techniques defined and known *a priori*. Because of a greater variety in method use, Wijers applied interviews, analysis of developed models, and think-aloud protocols, and recorded method use with video cameras. This active participation during method use allowed the discovery of detailed modeling knowledge and revealed knowledge about the modeling process. Because active participation is costly and time-consuming it can usually be applied for only one or a few developers' modeling experiences at a time. Thus, in a large scale method development effort where experiences are gathered from several users the approach is not necessarily cost effective. There, active participation can be used for inspecting method use among selected users from different roles (developers, user, managers etc.).

In incremental ME, therefore, the method use rationale is collected through structured interviews based on the evaluation mechanisms. This means that method use rationale is not collected completely; only those aspects which deal with the evaluation mechanisms are covered. In other words, method use rationale is collected only when it seems to differ from method engineers' intentions (i.e. from method construction rationale).

5.3.3 Type-instance matching

The first technique in incremental method engineering, type-instance matching, is an analysis of method use through the models developed. Analysis of models typically takes place at the instance level. For example, metrics are used to analyze system models (e.g. Low and Jeffrey 1990, IFPUG 1994, Rask et al. 1993) and method metrics are used to analyze metamodels (e.g. Rossi and Brinkkemper 1996, McLeod 1997). In ME and especially in an incremental approach, it is important to analyze both levels together: to compare IS models with metamodels to inspect whether the constructed modeling technique has been used. According to the metamodeling approach, the types of the constructed method are described in a metamodel (i.e. IRD definition level, ISO 1990) and instances of these types are described in models (i.e. IRD level). Hence the name for this method evaluation and refinement mechanism.

Analysis of intended and actual use of modeling techniques is similar to seeking differences between prescribed process models and recorded process models, proposed by Jarke et al. (1994). Some key differences must be noted between these approaches. For process modes, the traceability model collecting what has happened is broader than the guidance model defining the process to be followed. While evaluating the differences between these process models, it is also important to ensure that the predefined process is actually followed by developers. In tool-supported modeling, it is not possible to develop IS models which are not based on the metamodel. As a consequence, while analyzing type usage through models we can more reliably expect that the developers have actually used the constructed method (i.e. each instance has a type definition, cf. Section 3.3.1): the tool ensures that active constraints are satisfied and informs users about violations of passively checked constraints.

The close relation between models and metamodels offers also possibilities to automate data collection, since all the necessary information about types and instances is available in the repository. Hence, a metaCASE tool should support queries for both levels simultaneously. This functionality is not available in external CAME tools which are separated from method use (i.e. operate only at the IRD definition level). This automation is especially important while analyzing complex methods, projects which have developed multiple models, and projects which have multiple developers. The last of these is important because it helps highlight differences between people and reveal their modeling preferences.

Type-instance matching can be performed in two phases: first by focusing on the usage of basic types, and second by analyzing related constraints. Both of these are discussed in the following subsections.

5.3.3.1 Usage of types

To investigate the usage of types, we must collect data about whether each type of a method (e.g. object types, relationship types, or property types) is or is not used. The data collection can be fully automated by inspecting instances according to the types. This approach does not automatically lead to a method modification, because the number of instances that a type has does not by itself explain the relevance of a type. Moreover, because the analysis can suggest alternative modifications the results of type use must be clarified by interviewing method users after the preliminary analysis has been made.

Because models are always based on metamodels, three alternative modifications to methods are possible while inspecting the usage of types. These are 1) remove types which are not used, 2) divide, or specialize types which refer to different kind of instances, and 3) combine, or define linkages between types which refer to similar or related instances. These alternative refinement options are illustrated in Figure 5-5 with corresponding numbers.

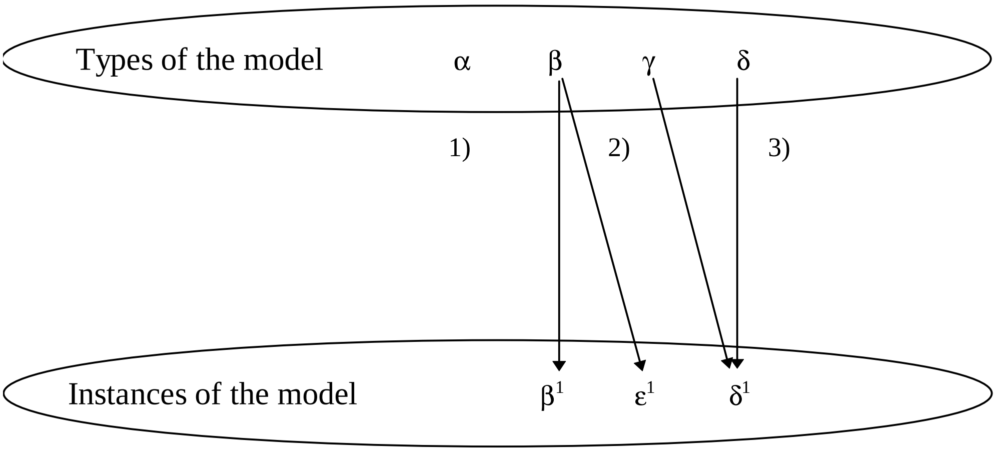


FIGURE 5-5 Alternative method refinements while analyzing usage of types.

The upper ellipse refers to a set of types of a method (i.e. instances in a metamodel), such as α, β, γ, δ. The lower ellipse describes instances of a model, such as β^1,ε^1,δ^1. The mapping between these levels follows the IRDS framework discussed in Section 3.3.1. Reading from the top, models are always created based on type level information. Reading from the bottom, models are always read and interpreted based on the types and their representations.

1) **Remove unused types**. Inspection of unused types is relatively straightforward. Types which are not used at all or have few instances may be irrelevant in the modeled domain and can be removed or combined with other types. This means that a method has a redundancy of modeling constructs, or that not all constructs were relevant in this modeling situation, or that the method users are insufficiently trained to make adequate distinctions. A method can also have unused types if all proposed types or constraints can not be found from the object system, or they are not considered cost-effective to model (e.g. because they are labor-intensive to identify).

Checking for unused types is important in simplifying methods. Similarly, organizations which have adapted external methods often simplify them radically (e.g. Jaaksi 1997). Especially in cases where local versions are made for the first time there is a risk of ambitiously modeling "everything" for incorporation into a metamodel.

2) **Division or subtyping of types** is required if the same type refers to different kinds of instances. This means that modeling constructs are overloaded and new types, constraints, and related representations are needed. For example, specification of classes which are persistent (e.g. MOSES, Henderson-Sellers and Edwards 1994) and at the same time deal with application interfaces (e.g. UML, Booch and Rumbaugh 1995) is not possible

according to any of the object-oriented methods analyzed in Chapter 4. To capture both of these characteristics, additional instance-based information must be specified. Although the analysis is based on semantics, and therefore can not be evaluated solely by analyzing models separately from the real-world, some pointers to this kind of need can be found from models:

- Method users may extend modeling techniques by using different naming policies for instances. This kind of modification is a common form of tacit, on-the-fly modifications (Wijers 1991). An example of such an extension is to name similar instances with a specific suffix indicating the similarity. Naming extensions used can be also found from a data dictionary, or from a documentation property type.
- Instances of the same type which are based on different wording (e.g. nouns versus verbs, or singular versus plural), or use of other distinctions (e.g. capital and lower case letters) may indicate that a single type is inadequate to differentiate instances.
- Instances based on different wording can be further analyzed based on the property types used. An overload of modeling constructs can occur if instances of the same non-property type have instances with different property types. For example, in the case of relationships, a flow which is named with a verb and described with parameters can indicate that the flow represents a function, a procedure, or an operation. In contrast, a flow which is named with a noun may indicate only data passing. These different kinds of flows could also be distinguished at the type level (e.g. relationship types for an operation and a data flow). In the case of object types, we can analyze differences between the relationship types the object type instances participate in. If objects of different wording participate in different relationship types they may denote different object types.

The resulting refinements can be carried out either by introducing a new non-property type, or by using a characterizing property type. A new non-property type is required if instances of a non-property type have different properties or constraints, e.g. a different type multiplicity. If the only type level difference is the need to classify instances then a characterizing property type is sufficient. Depending on the tool support, different representations may require new types. If the representation of a type can be changed based on instance information (e.g. depending on the value of a property) the creation of new types for notational reasons is not required.

3) **Combinations of types, or definitions of linkages between types** are required when there is redundancy among modeling constructs, i.e. a method has several instances of different types which refer to the same real-world or semantic entity. The use of several types that refer to the same thing is not always undesirable because it allows one to inspect an object system from different perspectives, and thereby to integrate techniques. Similarly, the metamodels developed in Chapter 4 show that the use of different types to specify the same instance information is relatively common. For example, in some situations an external entity in a context diagram (like in Yourdon 1989a) can correspond to an entity in an ER diagram (Wijers 1991, p 171). In other

situations only data stores of a data flow diagram can be specified as entities. Redundancy of types in a single modeling technique, however, is not considered desirable, because it makes modeling time-consuming by introducing additional complexity (Weber and Zheng 1996).

Some linkages are already defined in a constructed method, but we are interested in finding linkages which are not defined and could be included into a method. These can be found by analyzing:

- Instances which include the same values as their properties can indicate interrelations. Especially if values are shared among identifying property types, type level linkages could be defined. This refinement supports the maintainability of models and enables consistency checking (cf. Section 5.3.4.2).
- Instances which are nouns, verbs, or adverbs formed from the same root word, and which belong to different types can indicate some kind of relation at the type level. Also, synonyms can indicate that different users apply different modeling constructs to describe the same instances. The wording and possible synonyms can be inspected from the data dictionary related to models and by interviewing different developers about their naming policy.
- Instances of object types can furthermore be analyzed based on the relationships they participate in. If instances of different object types which are named similarly (i.e. same wording, synonyms) participate in similar relationship type instances they probably denote the same instance. A similar approach is often the only way to find out the class which a specific state in a state model describes: if a transition to a state includes actions which the same class includes as its operations, the state describes a part of the life-cycle of that class's objects[26].

The resulting refinements can be carried out either by combining types or by defining constraints which allow instances to be linked. A combination of types is not applicable if the non-property types have different property types, participate in different relationships types, or have different constraints. It is also possible to have different types which share exactly the same property types, constraints, and participate in the same relationship types. For example, in Coad and Yourdon (1991a) the only differences between classes and class-&-objects are their semantics (i.e. class is an abstract class as it does not instances) and representations (i.e. single lined rectangle for a class, double-lined rectangle for classes with instances).

A more detailed analysis of these refinements would require that method use is inspected in relation to other types and to more detailed constraints of the method. Accordingly, most of the modifications deal with refinement of method knowledge at the level of constraints.

[26] The state could also belong to superclasses. The analysis can be further improved by analyzing neighboring instances. For example, if the transition occurs from a state of another class the operation should be defined as public (e.g. Booch et al. 1996).

5.3.3.2 Usage of constraints

Evaluating the usage of constraints is concerned with inspecting how the rules of the modeling techniques were applied. It extends the analysis from types to constraints. As with the type usage, the inspection of constraints can lead to the removal or addition of constraints: Some of the constraints defined may have been too strict, or conversely some might not have been used at all.

Data collection on the use of constraints is performed on the basis of the constraints described in the metamodel. In the following we describe what constraints need to be checked based on the essential constructs of metamodeling (see Chapter 4.4). Basically, most of the constraints used in a single technique are straightforward to analyze, whereas constraints related to integrating techniques are more complex. Below we discuss each constraint and the method refinements that can be suggested on the basis of its usage.

1) **Identifying property.** Properties which have inconsequential or dummy values are not applicable for identifiers. Accordingly, the identity constraint can be removed, and perhaps another identifying property type or types defined in its place. It must be noted that the identity does not deal with identity in a repository, but rather identity among design information. This is only meaningful for humans, since computer-aided modeling tools normally use internal identifiers (see also Section 4.4.1.1). New candidate identifiers can be found from other property type values. For example, instances in a dictionary property type can reveal candidate identifiers.

2) **Unique property.** Values of property types which are slightly changed or are based on different wording (because the tool does not allow the same instance values) may denote that the uniqueness constraint is limiting modeling in the defined scope of the constraint. The scope of the constraint can be refined to include a smaller set of values, for example from all instances of a given property type to instances in a single model, or the whole uniqueness constraint can be removed. In contrast, if different instances of the same type can not be distinguished, compared, or checked adequately a uniqueness constraint needs to be added.

3) **Mandatory property.** As with identifiers, a large number of dummy values added to satisfy the mandatory constraint should lead to its removal. Alternatively, property types which always have values may indicate that the property type should be defined to be mandatory.

4) **Data type of properties.** Although tools normally ensure that data types are followed, the use of complicated data types, default values, and predefined values can be analyzed. Property types which allow free form text can include definitions which should follow a certain structure or syntax (like CATWOE discussed in Section 5.2.3). These can be added as new property types, or alternatively a syntax could be defined for a property type.

A default value and predefined values can be modified to speed up modeling. A default value can be changed if another value is more commonly used. For property types with a mandatory constraint the most used value is declared as the default. Also predefined values which guide selection, such as stereotypes or multiplicity values (e.g. Booch et al. 1996), and which are not used may be removed: they slow down modeling and make use of the method more complicated.

5) **Cardinality** defines whether instances of a relationship type are binary or a specific n-ary. Because all possible alternatives of participating roles and objects do not necessarily appear, nor are all cardinality values used, the refinement possibilities of the cardinality constraint can not be studied fully by analyzing models.

Some aspects, however, can be analyzed from model data. If only binary relationships are allowed the need for an n-ary relationship can be recognized when multiple relationships with the same property values are created for the same object. For example, an inheritance relationship defined as binary will need to be defined as n-ary if a class participates in several relationships in the superclass role and with the same discriminator value. This requires a change to the maximum cardinality constraint. The minimum constraint can be changed to one if all instances of a given relationship type use the specified role type, i.e. changing an optional role to be mandatory.

If n-ary relationships are not used the cardinality constraint can be removed. Another option would be to create a specific relationship type for n-ary cases, as in OMT (Rumbaugh et al. 1991). More detailed refinements, like n-ary relationships only being used for specific instances of an object type or specific cardinality values, must be carried out together with method stakeholders.

6) **Multiplicity constraints** deal with the number of role type instances an object type instance may have in a model. The constraint can be bound either to instances of a single role type, or to instances of different role types. As with the evaluation of cardinality constraint, not all multiplicity alternatives are necessarily applied during modeling and therefore their suitability can not be analyzed solely from model data. The following principles, however, help identify refinement possibilities:

- Existence of role type instances for all object type instances may indicate that the role type should be defined to be mandatory, i.e. minimum multiplicity should be one.
- Existence of only one instance of a role type for each object type instance indicates a one-to-one constraint value (1,1). Alternatively, a passive checking for a maximum value could be used to define that an object should only have one role type instance: in some cases, which the users should be informed about, multiple roles would still be possible. An example of such a case is a recommendation to use single inheritance (i.e. each class only participates once in a subclass role). This option is also relevant for instances of several role types.
- Role types which are defined as mandatory and have "unnecessary" instances may be made optional (i.e. minimum multiplicity of zero). Examples of unnecessary instances are roles and related relationships which are not specified with property values. Changes to the checking mode are not relevant here because both modes are possible only for the maximum multiplicity. Similarly, it must be noted that role types which

are not used at all have already been inspected through the type usage analysis.

7) **Cyclic relationships.** Analysis of cyclic relationships based on model data can only lead to removing cyclic relationships. If cyclic relationships are not allowed, but required, this implies that not all aspects of the object system can be represented. Additional objects to overcome the prohibited cyclic relationships could be analyzed, but this would require semantic analysis.

8) **Multiplicity of types.** If system models are scattered into multiple small models, a minimum constraint can be applied to remind users, but not to ensure (because of the passive checking mode, cf. 4.4.1.9), that instances should be combined into smaller number of models. Alternatively, the creation of large and overly complex models can be prevented (with active checking), or discouraged (with passive checking) by setting the maximum multiplicity constraint for selected object types. The multiplicity of types is related to complexity management which can also be supported with other metamodel-based constraints, e.g. complex objects, explosions, and polymorphism.

9) **Inclusion**. In contrast to analyzing all instances of a type, an analysis of inclusion means that instances are analyzed inside a single modeling technique. For example, a 'library class' (Henderson-Sellers and Edwards 1994) can be useful in a specific modeling technique but not in all techniques, or vice versa. In addition to the use of non-property types, the occurrence of their property type instances needs to be analyzed since it is typical, at least in the methods analyzed (Chapter 4), that not all information on the same non-property type is required in different modeling techniques. For example, in the metamodel of UML (Section 4.4.3) an 'object' can be used both in a class diagram and in an object diagram with a property type 'values'. This property type is used to describe instances of attributes of a class, but it is not necessarily relevant in class diagrams but only in object diagrams. This reveals polymorphism and is analyzed through the polymorphism constraint below.

10) **Complex objects** deal with an abstraction mechanism which allows the modeler to build aggregate-component structures. Based on the usage of complex objects, the most straightforward method is to determine which are not applied and remove them as inapplicable. A more detailed analysis necessitates that different characteristics of the complex object type are examined.

- A component type can be declared dependent if all instances of a component type occur in complex objects.
- A component can be declared mandatory if all aggregate objects have instances of the component type.
- A component type can be declared exclusive if none of its instances belong to other complex objects.
- A component type is shared if the same instances of the component type belong to several complex objects.
- A constraint for aggregated relationships can be defined when all relationships of the components outside the complex object are also defined for the aggregate. Whilst the constraint is undefined there exist

redundant modeling tasks in order to maintain consistency (cf. Section 5.3.4.2).

It must be noted that not all dimensions of complex objects (cf. Table 4-4) are analyzed because they are not relevant, or can not be analyzed from model data. For example, connected components and relationships of an aggregate can be analyzed with a multiplicity constraint. The limited use of complex objects can also be a consequence of the "dictatorship" of a tool. For example, aggregate object types may remain unused because they have mandatory components which are not applicable in the modeled domain. Hence, the constraints of the complex object could be checked passively, although our analysis of complex objects suggested that they can always be checked actively. Passive checking would allow violation of the constraints of complex objects but would still inform the method user about the inconsistencies.

11) The **explosion** constraint deals with organizing and structuring multiple models. The original metamodel can either ensure (with active checking) or encourage (with passive checking) the use of explosions. While analyzing the current usage of the explosion constraint, the following situations indicate a need for modifications:

- Explosion structures which are not used may be irrelevant and removed. Alternatively, the active checking which specified explosion structures as mandatory may be the reason why they are not used: passive checking could be applied instead.
- Explosions should be defined as mandatory if all instances of a specific type (i.e. a source) or a technique (i.e. a target) participate in explosion structures. A mandatory explosion structure is defined with a minimum cardinality value of one, or alternatively if only one explosion exists with a minimum-maximum pair of (1,1).
- Passive checking can be applied if an explosion structure is not used for all instances of a source type or a target technique. The use of active checking can not be analyzed from the resulting models because it deals with the modeling processes, i.e. whether all explosion structures were created in a top-down or in a bottom-up manner. This would be possible in modeling environments which allow queries on instance creation times.
- A constraint of a shared explosion target could be removed if only one instance of a source type refers to a model.
- A constraint of an exclusive target model can be considered too restrictive if it leads to the creation of multiple models which have fewer instances, or which have multiple shared instances. The former can be partly analyzed by inspecting the multiplicity of types in models, and the latter by inspecting the occurrence of the same instance values.
- A model scope constraint is inaccurate if the same instance has the same explosion links in multiple models. For example, a class has the same life-cycle, i.e. an explosion to the same state model, although it is represented in different models. Hence, the method scope should be used.

12) **Polymorphism** means two or more types sharing the same property values. The evaluation deals with analyzing polymorphism structures which are not used and by seeking instances which can indicate polymorphism. Based on the former option, polymorphism structures which are not used may not be suitable for the modeled domain. It must be noted that not all structures of polymorphism are necessarily described by sharing property values. Instead other constraints of a modeling technique can be used for this purpose. For example, an object can have both an 'instantiation' relationship type to a class and an object can be identified by a property type 'class name' (e.g. in Booch and Rumbaugh 1995).

Based on the latter option, new polymorphism structures can be defined to support reuse. These linkages are typical between different techniques where no clear representation for linking is available. First, the analysis is carried out by seeking values among different property types which are based on the same wording, suffix, etc. This results in a set of types which describe the same model data. This approach is similar to analyzing overloading of modeling constructs with type usage (Section 5.3.3.1). For example, as in the metamodeling example in Section 3.3.3, values for actions in a state diagram and values for operations in a class diagram are the same.

Second, a number of types participating in a polymorphism structure must be inspected. For example, a value "add customer" can be used as an instance of an 'operation name', an 'action name' and a 'message name'. This means that the three types share the same value. Third, the size of a polymorphism unit must be analyzed, i.e. how many instances of different property types are shared together. For example, actions and operations typically share only instances of the naming property types since actions do not include operation-related characteristics, like parameters or access levels. Also, parameters of a message in an event diagram (Rumbaugh et al. 1991, Booch et al. 1996) are the same as operations. Hence, the values shared include both the name and parameters. This is illustrated in Figure 5-6. An action, an operation and a message denote the same instance values. In a state model, actions are defined with a name, but operations of a class also include parameters and access levels, e.g. public. A sequence property type is only meaningful for messages in a message diagram.

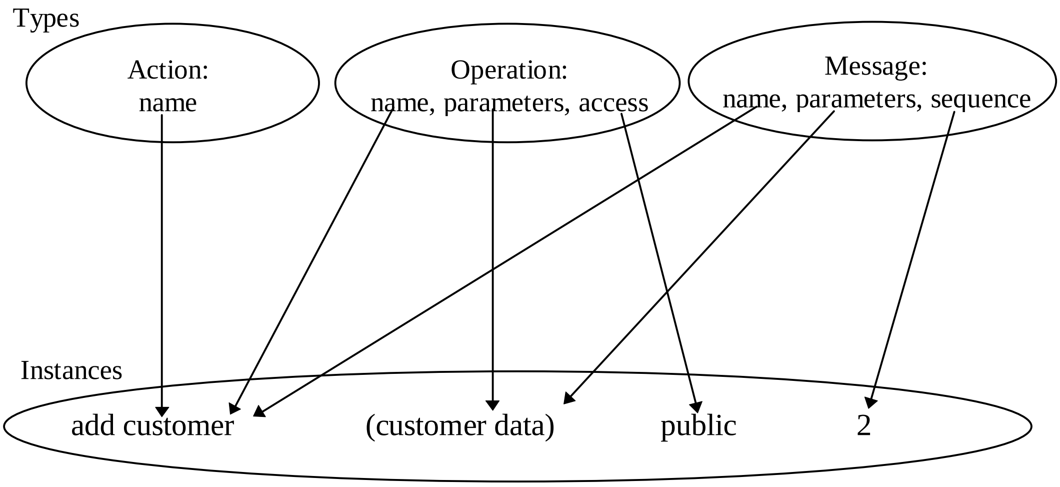


FIGURE 5-6 An example of polymorphism structure.

Fourth, dependency among polymorphism structures requires that the modeling process be analyzed. This would allow us to find the types for which the shared instance values were first defined. For example, candidate operations should be added first into a message diagram and therefore operations of the class should refer primarily to already defined messages. Although analysis of dependencies deals with the modeling process, some of them can be recognized from model data. If a property type always has a value used in another property type, the former type may be dependent. Alternatively the checking mode for the dependency can be changed: active checking of dependency is required if models need to fulfill the rule at all times, and passive checking is used for an optional dependency.

Finally, the scope of a polymorphism can be changed if shared instances belong to a smaller scope than originally intended. Alternatively, if polymorphism is not used, because the instances the user wanted to refer to were outside the permitted scope, a larger scope could be specified.

5.3.4 Modeling the object system

The second approach to incremental ME is analyzing modeling capabilities. Tool-supported modeling capabilities are divided into abstraction and consistency checking (Olle et al. 1991, see also Section 2.3.2). The former means the capability to describe relevant aspects of object systems, and the latter means the capability to maintain consistent models.

The evaluation of modeling capabilities requires information about the object systems modeled and thus extends evaluation from the IRD level into the application level of IRDS (cf. Section 3.3.1). As a result, the evaluation must be conducted in close cooperation with method stakeholders. This means interviewing stakeholders in addition to analyzing model data. Interviewing is used to collect opinions on the method use and requests to change the method. The evaluation questions described below focus on structured interviewing.

5.3.4.1 Abstraction support

A conceptual structure behind modeling techniques suggests an abstraction to describe an object system (cf. shell model, Figure 2-2). An abstraction means perceiving some aspects of the object system while ignoring other aspects. Two issues must be noticed while evaluating tool-supported abstraction capabilities. First, a tool provides a set of concepts which are limited from a syntactic and a semantic point of view. Therefore, non-diagramming concepts[27] (Wijers 1991) or other additional concepts can not be applied. Although a tool can include free-form modeling techniques, they are not adapted *a priori* to the situations and therefore their evaluation is excluded here (for this type of analysis see Wijers 1991). Second, not all aspects of the object system are necessarily represented in a similar notation to that used in paper documents because a CASE tool uses dialogs, linkages between models (e.g. an explosion structure), and a data dictionary to capture specifications.

The evaluation of abstraction support can be analyzed by examining how object systems could be represented, by analyzing difficulties in making representations, and by inspecting differences among method users. These are described below:

1) **Are all relevant aspects of the object system perceived with the method?** The limitation of abstraction support can be recognized when some aspects of the object system can not be perceived and represented with the modeling techniques. This requirement sets out the goal that the method must capture essential "objects" of the design problem and convey relevant information about them. As the review of method evaluation studies showed, this is the most common approach (e.g. Schipper and Joosten 1996, Wijers 1991, Fitzgerald 1991). Based on the evaluation, refinements can be made by:

- Adding new types which illustrate aspects to be modeled. These can include a new non-property type (e.g. an object type which has property types and other constraints) (e.g. Jaaksi 1997), several types, or a whole modeling technique (e.g. Tollow 1996).
- Adding a new property type (or types) that characterizes currently used non-property types (e.g. for subtyping entities, as in Wijers (1991)).
- Adding new relationship (or role) types for describing specialized connections between objects.
- Removing constraints which restrict abstraction. Examples of possibly restrictive constraints are multiplicity constraints (limiting the number of relationships which an object type instance can participate in) and cardinality (limiting the number of roles a relationship can have).

[27] Non-diagramming concepts refer in Wijer's (1991, p. 170) study to additions to the modeling technique which are made when all aspects of object systems could not be explicitly specified with the modeling technique. Examples of such concepts in his study include a 'problem' and an 'external party'.

2) **What types have been difficult to use?** Difficulties in making abstractions can indicate that the method does not "fit" the object system, or has not been introduced and taught well. If the difficulties are related to an inappropriate method, its conceptual structure can be redefined.

3) **What types have been used differently among individual developers?** Differences in method use can be due to individual differences and modeling preferences. For example, some developers can use state models to describe an object's life-cycle as the method engineer intended, whereas others can use them for interface design (e.g. Jaaksi 1997). Similarly, method developers can have different requirements from end-users (Tolvanen and Lyytinen 1993). Although individual differences of method use exist (e.g. Wijers 1991, Hofstede and Verhoef 1996) and can be supported through ME, ISD is a group activity. The modeling results should be based on a common understanding of modeling concepts. This is important for communication, minimizing misunderstanding etc. Hence, method refinement should strive to find linkages between related views of method users (Nuseibah et al. 1996).

5.3.4.2 Checking support

Problems of insufficient computing power are most noticeable in ensuring the consistency of models. Use of checking related constraints in the metamodel will result in well-defined and complete model instances. Consistency checking supports the maintainability of models, decreases the redundancy of modeling tasks, and supports traceability by informing of side-effects of changes. These emphasize both vertical and dynamic integration of conceptual method connections. The checking support is emphasized in ISD efforts where multiple models are developed with different techniques (integration among techniques or methods) and by different people (coordination among method users).

With respect to metamodel data, checking can be carried out either actively or passively. Active checking ensures that models continuously satisfy the constraints of the metamodel, whereas passive checking requires a modeler's attention. In modeling tools passive checking is typically implemented with checking reports which inform method users about violations.

Although consistency is checked with various algorithms it is always dependent on the underlying metamodel data. Checking support can be evaluated with the following questions:

1) **Are the developed models consistent?** This question can be partly analyzed by checking if the models satisfy both the active and passive checking rules defined in the metamodel. Active checking is already ensured by the tool and passive checking can be analyzed by running the consistency reports made during the tool adaptation. If models are not consistent, either consistency rules are not applicable or they are not used. In the former case the checking related constraints can be removed, and in the latter case active checking can be required.

2) **Is manual work required to keep models consistent?** This question deals with finding new constraints to maintain consistent models. Redundancy of modeling occurs when the same instance information must be changed several times in order to maintain consistent models. This error-prone and time-consuming task can be reduced by adding constraints to the metamodel and by providing new checking reports based on the constraints added. Also, difficulties in tracing how changes in one model affect specifications elsewhere can indicate a need for new constraints. Redundant work exists in the following situations:

- The addition of an instance requires the addition of the same information elsewhere in the models. For example, creation of a new action in a state diagram necessitates that the action is specified among operations of a class (cf. metamodeling example in Section 3.3.3). This situation would require that the action names are dependent on the operation names.
- A change to an instance requires that other instances must be updated. For example, a change of an entity name must be reflected in a data dictionary and in data flow diagrams (Yourdon 1989a).
- The deletion of design data requires searching and removing the same instance information. For example, the deletion of a class might require the manual deletion of related state models or removal of individual states. The former could be automated by defining a minimum cardinality of one for the explosion target (i.e. the state model), and the latter by using dependent polymorphism among states and classes (i.e. each state must refer to and be dependent on a class through its name, as in UML (Booch et al. 1996)).

Some refinements can be carried out by changing the checking mode, or the scope of the constraints. Active checking can be used for constraints which are defined but not applied or when the use of passive checking is considered tedious. The scope of the consistency related metamodel constructs can be changed if consistency is not ensured among instances outside the defined scope: the scope is refined from a smaller set of instances (e.g. dependent type) to include a larger number of instances (e.g. model or method).

5.3.5 Supporting problem solving

Methods are not only used to describe a current situation but also to carry out a change process with respect to object systems. This necessitates that a method supports the seeking of candidate solutions and deciding amongst them (Tolvanen and Lyytinen 1994). Both of these can be supported by a CASE tool with form conversion and production of documents (Olle at al. 1991, cf. Section 2.3.2). Form conversion provides mechanisms to seek alternative solutions by manipulating design data according to method knowledge (i.e. according to the conceptual structure or the notation). Deciding among solutions can not be directly automated but can be supported through the provision of documents for review and comparing candidate designs with the current configuration.

The evaluation of problem-solving capabilities is not much addressed as most approaches (cf. Section 5.2) focus mainly on modeling support. One reason for this focus is the evaluation of methods separate from their use situation. The analysis of problem solving capabilities reveals which parts of the method knowledge are required to seek alternative solutions and which are needed only for abstraction and checking. For example, while generating code from object-oriented methods, not all concepts of the method are needed: although message diagrams are important in understanding object interaction, they are not required since the design data related to program code is already represented in class diagrams.

As with the evaluation of modeling support, the evaluation of a method's role in problem solving requires the participation of stakeholders. It involves an inspection of the application level to refine the IRD definition level (i.e. metamodels); in other words, a comparison of development outcomes and the method's role in producing them.

5.3.5.1 Support for form conversion

Form conversion in a CASE tool means an analysis and a comparison of design data, simulation, generation of program code, and building of prototypes. Like consistency checking, form conversions are carried out by algorithms (e.g. checking reports or transformations) but they are only possible if the metamodel specifies and maintains the necessary design data. This means that aspects other than those found directly from or derivable from models can not be converted. In other words, conversions are largely dictated by the abstraction capabilities. Naturally, method knowledge is also included in the conversion algorithm (e.g. the syntax of the generated language), or can be added by developers during the conversion (e.g. a choice among approaches to convert an inheritance into a relational model, see Rumbaugh et al. (1991)).

The evaluation of method support deals with analyzing how well it provides concepts and notations for form conversions. A conversion of conceptual design data takes place, for example, when a schema for a database is generated. Conversion of representational data occurs when the conceptual design data remains the same but the notation changes. For example, BSP (IBM 1984) determines boundaries between ISs by organizing the data classes and business processes into a matrix so that a minimal number of connections occur among ISs. During this conversion only representations of data classes and business processes are clustered according to the use of data. Form conversion capabilities can be evaluated with the following questions:

1) **Can the required analysis be made using the models?** Although analysis of models is dictated by the rationale that suggests modeling concepts, model analysis can reveal the need for new concepts. For example, during workflow modeling, a demand to analyze bottlenecks may arise. This, however, is impossible if the models do not capture information about capacity and throughput times. This suggests additions of property types to the workflow modeling technique. Similarly, inspection of encapsulation requires that attributes and operations of a class can both be specified directly with the

specification of a class (e.g. Rumbaugh et al. 1991), or in a class related models (e.g. Coleman et al. 1994).

2) **Can alternative design solutions be generated from models?** A method should include rules which allow the conversion of models into various design alternatives by using the metamodel data. For example, to generate alternative solutions based on the level of (de-)centralization of the organization, the method should describe organizational structures. Similarly, interaction scenarios between classes can be examined by describing a significance for events (e.g. Awad et al. 1996).

3) **Does the design satisfy requirements of later phases or external tools?** An outcome of modeling is a design solution which can be implemented or further analyzed with other methods or external tools (e.g. with a simulator, a programming environment, a code generator, or a reporting tool). Therefore, vertical integration with other tools and methods (cf. Table 2-2) must be provided. In other words, the requirements of later phases must be satisfied to provide an integrated method. For example, although UML (Booch et al. 1996) supports the generation of CORBA IDL interfaces (Iona 1997) better than other methods analyzed in Chapter 4, its support is not complete. As an example, UML does not consider context clauses for IDL operations. Hence, the UML metamodel can be extended with property types for context expression. Metamodel extensions towards programming languages are further discussed in Hillegersberg (1997).

The analysis of form conversion capabilities typically leads to extending the conceptual structure with new types and constraints. If a conversion suffers from unavailable design data, for example because modeling tasks can not be completed unless instance values are added to the models made earlier, constraints can be added. These include a mandatory constraint for property types, multiplicity of types, and multiplicity of roles. In addition to changes to a metamodel, changes are also required in form conversion algorithms.

5.3.5.2 Support for review

Information system specifications which can be understood and reviewed by stakeholders are of great importance for validation. Tool support for review consists of the provision of information for stakeholders, such as summary reports for managers, less formal descriptions of the selected domain for end-users, and formal specifications for programmers. The documents produced can vary based on the conceptual data and their representations. Since a review is always dictated by what is abstracted, the evaluation of review support deals mostly with representational issues. Tool support for the review step can be analyzed with the following questions:

1) **Can validation of IS models be supported?** The metamodel must help to validate the system descriptions in relation to stakeholders' desires and needs. This requirement is partly overlapping with the consistency criterion. There is, however, a marked difference: validity deals mostly with the semantic adequacy, whereas consistency focuses mainly on the syntactic properties of the models. Therefore, validity can not be assessed by exploring the metamodel

alone, but method users can provide information about which concepts and representations they find useful in validation.

2) **Does the method correspond to users' natural concepts?** Development methods are developed to satisfy developers' cognitive needs related to design tasks. Therefore, it would be an advantage if methods were similar to users' existing concepts and patterns of thought. For example, Olle et al. (1991) suggests different graphic representations for different types of users: experts from different areas of the object system may require different concepts from those employed in the underlying techniques. Similarly, less formal notations and icons can be applied.

The analysis of review support typically leads to extending the method in terms of providing different notational constructs, and simplifying the method for different use situations.

5.3.6 Remarks on the *a posteriori* mechanisms

In this section we have put forward mechanisms for evaluating methods in a given situation. These mechanisms refine a method by adding, changing, and removing parts of the method knowledge. In other words, they evaluate which parts of the modeling techniques need to be simplified or extended. If the mechanism reveals requirements to change the method, it means that the constructed method may not be applicable in a use situation. The mechanisms are summarized in Table 5-4. The steps of incremental ME, i.e. collection of experiences, analysis, and outcome of refinements, form the vertical axis, and the *a posteriori* mechanisms the horizontal axis.

TABLE 5-4 Mechanism for method evaluation and refinement.

Steps of incremental ME	Type-instance matching	Problem solving capabilities	Modeling capabilities	Method rationale
Data collection	Metamodels, completed models	Project outcomes, interviews, models, tool support	Models, interviews, tool support, method change requests	Method use decisions
Analysis	Differences between types and instances	Support for form conversion and decision making	Support for abstraction and consistency checking	Individual differences, method engineer's intentions
Major outcome of method refinement	Removed types and constraints	Added types and constraints, improved method-tool companionship	Add types and constraints, improved method-tool companionship	Updated method rationale

As the proposed mechanisms show, we emphasize modeling and problem-solving capabilities. They are mostly used in cases of local method development (cf. Section 2.4.2) and in the method evaluation literature (cf. Section 5.2), and they can be related to detailed method knowledge. Neither contingencies nor stakeholders' values imply modifications of detailed metamodels, although some changes in contingencies or value-based ME criteria could be accommodated in a metamodel.

It must be noted that the preceding mechanisms are not the only ones possible for evaluating methods. They are relevant for our research question of supporting method improvement through metamodels. The collection and analysis of experiences, as well as method refinements, are carried out through the metamodeling constructs. These organize the experience gathering and make method modifications more explicit and formal. The matching of types and their instances mostly leads to purging of method knowledge, because extensions which enlarge the metamodel are not possible. In other words, analysis of the use of a method in a tool can only show things which the tool has not prohibited. In contrast, the evaluation of modeling support and problem solving capabilities mostly lead to extensions of method knowledge. Extensions are largely a result of method users' requests which arise from the application level. This also means that *a posteriori* ME requires the participation of the method engineer in ISD to obtain application level knowledge. This supports the claims that a method engineer must be one of the stakeholders of ISD, such as a project manager (Odell 1996).

Because the mechanisms are overlapping, they can suggest conflicting modifications. For example, an analysis of explosion structures can show that each instance must be exploded, but the analysis of type multiplicity reveals that resulting models have only a few instances. As a result, the choice of an appropriate refinement must be made together with method users. Moreover, neither the mechanism nor the refinements should be prioritized. Therefore, the preferences of stakeholders can emphasize different mechanisms and resulting refinements. For example, Hofstede and Verhoef (1996) propose to be less ambitious with regard to the level of consistency and promote simple representations (i.e. a small number of graphical symbols).

5.4 Summary

This chapter has focused on complementing existing method engineering principles by introducing an incremental approach. This approach is motivated by the limitations of *a priori* ME approaches. In short, *a priori* ME is not interested in method use, and it assumes that the constructed method is understood and applied as the method engineer intended. In contrast, we believe that method knowledge and method construction criteria can not be known completely beforehand. Moreover, we claim that an ISD environment is not stable, because method use situations change and method users learn about their methods. These method use characteristics were also emphasized in our re-evaluation of method use (Section 2.5). As a result, at some point of time a method becomes less applicable for the tasks for which it was promoted.

Certainly it is possible that a method can be found to be fully applicable during the ISD project. Even in these cases it is of key importance to learn about method use. The learning aspect is also a key difference between *a priori* and *a posteriori* approaches to ME. Instead of expecting that a method engineer is responsible for all improvements to a method, the incremental approach emphasizes the role of method users and their experiences. In other words, method development should be based on stakeholders' experience and situational needs, in contrast to selecting methods solely by using 'universal' ME criteria. The focus on experiences is also relevant because learning through experiences has been identified as a main way of learning about methods (Hughes and Reviron 1996).

The incremental approach is clarified through method engineering scenarios. The scenarios illustrate steps of ME in which modifications can occur. The scenarios are used to explain incremental ME principles: we are interested in experiences which arise from method use, which can be made explicit, and which can contribute to method refinements. Explicit means that method improvements are not tacit, nor individual knowledge, but can be discussed and shared in an organization. This is important because learned methods often become tacit and "invisible" (Wastell 1996) and an in dividual developer's productivity (Davis et al. 1991) can be reduced by methods (Fitzgerald 1996). The aims of method refinements mean that we evaluate methods primarily for improving them in a current use situation.

To relate our incremental approach to other studies, we reviewed the approaches proposed for situational method evaluation and validation. This analysis pointed out that most of the evaluation approaches do not follow any systematic evaluation procedure for data collection or analysis. They are carried out mostly by method developers, and they do not aim to systematize the method improvement process. Moreover, unlike ME they aspire to a general situation-independent proof (or disproof). This proof has been found difficult to obtain (Fitzgerald 1991) as it necessitates that evaluations could be replicated, the variety and complexity of ISD environments reduced, and data collection limited to factors relevant to method use. Our approach is different. We aim to evaluate methods in situations in which they are applied and use an organization's own experiences as a source for method improvement. In this sense method modifications are subjective, but generalizations can be found by iterating in cycles of incremental ME.

The incremental approach is described through the mechanisms for collecting and analyzing experiences for the purpose of method improvements. The mechanisms deal with differences between intended and actual use of a method, the modeling power of modeling techniques, and a method's support for problem solving. In each case the experience is collected and analyzed differently and can lead to modifications of a method or a tool. For each mechanism, principles for collecting and analyzing experiences are described and alternatives for possible method refinement are explained. First, the approach collects experiences and analyzes the applicability of modeling techniques through the use of types and constraints (in a metamodel) for representing an IS (in models). Second, it focuses on mechanisms that emphasize the capability of a modeling technique to abstract relevant aspects of the IS and maintain the consistency of these models. Third, the suggested mechanisms evaluate the support of modeling techniques in problem solving. This evaluation deals with the capability to provide alternative solutions through form conversions and to support review and validation of models.

6 AN EXAMINATION OF INCREMENTAL METHOD ENGINEERING: TWO CASE STUDIES

In this chapter we shall demonstrate the viability of the proposed method engineering principles by analyzing two cases of incremental ME. Our focus on real-world method development efforts means that we will face two major differences in metamodels: hereafter the metamodels developed are situation-bound, and method applicability varies as its use situations change.

So far we have modeled ISD methods as they are described in the method literature: they are "universal", standard and largely fixed. In Chapter 4 each method was specified using a single metamodel and no situational method modifications were made. Recently, method developers have adopted metamodeling for describing meta-data models, e.g. Booch et al. (1997) present metamodels for their Unified Modeling Language, and Henderson-Sellers and Bulthuis (1996b) for their Open Modeling Language. These metamodels, however, neither suggest modifications of methods nor provide different method versions for different situational needs. Although some situational needs are identified (e.g. Booch and Rumbaugh 1995), versions that can meet these situations are not specified. In incremental ME, metamodels are made based on situational needs. At the same time we can demonstrate that the metamodeling constructs are relevant for modeling situation-bound methods, not only applicable for modeling text-book methods, as we used them in Chapter 4.

Based on the re-evaluation of method use (cf. Section 2.5) we shall focus on supporting the evolution of methods. Two cases of local method development are analyzed longitudinally and the methods constructed are evaluated *a posteriori* using the principles of incremental method engineering. These principles seek to externalize experiences of the methods' use and channel them back into method improvements. This allows us to address our second research question on how to refine methods through modeling experiences. Possible method refinements resulting from *a posteriori* analysis demonstrate that the *a priori* method was not as applicable as intended. If some refinements occur, these justify our conjectures that local methods are evolutionary and need to be maintained. Alternatively, if no method refinements are needed, then *a posteriori* analysis can be considered unnecessary, or the evaluation mechanisms were inapplicable to improve methods.

The chapter is structured as follows. First, we describe the action research method followed. Second, two cases of local method development are discussed using the steps of incremental ME. Finally, the cases are analyzed by soliciting lessons about local method development, about method engineering principles, and about the incremental approach.

6.1 Research method for method engineering cases

Empirical studies of local method development are rare (Wynekoop and Russo 1993, Tolvanen et al. 1996). This makes the selection of a research method for our case more difficult: we can not directly build upon other work and confirm (or challenge) its findings. Though several studies have investigated method use in practice (see the studies discussed in Section 2.4 and surveys by Wynekoop and Conger 1991, Tolvanen et al. 1996, Sauer and Lau 1997, Wynekoop and Russo 1997) they operate at a general level of method knowledge. Surveys and field studies do not address detailed method knowledge; rather they show that adaptations occurred (see Section 2.4). Case studies have been carried out on method introduction and use, resistance to change (Wynekoop et al. 1992), social defense (Wastell 1996), and stakeholders' interests (Sauer and Lau 1997), but none of them addresses the situational fit of the method use. As a consequence, most of the ME approaches reviewed in Chapter 3 are unproven for local method development efforts. Those approaches which include demonstrative cases do not go into details (e.g. Punter and Lemmen 1996) and address only *a priori* ME, i.e. mostly the construction phase.

Because ME is a relatively new research field, complementary research efforts and a variety of research methods are needed. To achieve the necessary pluralism we need more empirical studies (Tolvanen et al. 1996). Too often ME approaches, metamodeling languages and metaCASE tools are developed without an empirical grounding. Among empirical research approaches we believe that action research is appropriate to examine ME. Several researchers give support to this research approach (cf. Galliers and Land 1987, Galliers 1992, Wood-Harper 1985, Checkland 1981, Grant et al. 1992) in the context of studying ISD methods. Reasons for applying the action research method in our studies are manifold: it resembles incremental ME, it is iterative, it allows us to go into details, it is situation driven, and it offers possibilities for longitudinal observation.

Before we describe how the action research was carried out in this thesis (Section 6.1.2), and its similarities with incremental ME (Section 6.1.3) we briefly describe the action research method.

6.1.1 Action research method

Action research can be understood as a variant of a case study and a field experiment (Galliers 1992). Analogously to a case study, action research uses evaluations of particular subjects, such as an organization, a group of people, or a system at a point of time. It attempts to capture the "reality" in greater detail and typically no control of the phenomena is exercised. Unlike a case study, in action research a researcher participates and acts in the area of study and simultaneously evaluates the results of this participation. This dual role means that the objectives of the research are twofold: on the one hand, the action researcher aims to improve the situation in the organization. Thus, action research resembles any organizational development or consulting effort. On the other hand, the action researcher aims to contribute to scientific knowledge by creating generalizable concepts and theories of the problem setting and its behavior. The generalization is necessary for future settings, and for researchers to build better theories.

The close interaction between theory and practice in action research means that during the research process, the roles of a research subject and a researcher can be reversed (Galliers 1992). As a result, the process of the action research separates the phases where action is taken, and where its results are evaluated (Checkland 1991, Jönsson 1991, Baskerville and Wood -Harper 1996). The dual role necessitates that action researchers be aware that their presence will affect the situation. Unlike case studies, the action research method permits intervention of the researchers into the events. In fact, the possibility to plan interventions and record them for evaluation purposes forms the essential mechanism of action research. The intervention can vary from direct intervention as an equal coworker, to indirect intervention through a catalyst role. An example of direct intervention would be participation in the method selection, and an example of indirect intervention would be playing an expert role in tool adaptation. However, in both modes the changes to be made must be planned and the effects of the actions recorded. This part of action research resembles a highly unstructured field-experiment. The process of the action research method is described in the next section in which its application in this study is explained.

The possibility to test and refine principles, tools, techniques, and methods, as well as to address real-world problems, makes the action research method very appropriate for organizational development (van Eynde and Bledsoe 1990) and for IS research (Baskerville and Wood-Harper 1996). The advantages of the action research method compared with other approaches come from the possibility to obtain a deeper, first-hand understanding of the situation. The action research method allows collection of information which would be difficult to obtain by outsiders, and permits use of longitudinal research designs (Checkland 1981, 1991, Baskerville and Wood-Harper 1996).

Action research also has limitations. The approach offers few possibilities for statistical generalization, and no possibility to exercise control over experimental conditions. Because action research is largely interpretive, its results can also be interpreted differently by individual researchers. The dual role of the researcher also raises some ethical problems: the goals of practice and research can be conflicting. For example, organizations often expect quick results whereas the researcher may expect slower and more gradual progress. Conflicts can also be faced by the individual researcher having to act in both roles simultaneously. The funding structure behind action research raises a dilemma when the researcher is financed by the organization examined. Although the funding indicates some commitment of the organization to the study and access to data as a "worker" of the organization, the researcher must seek to satisfy the organization's objectives as well. For example, it is not usually possible to study failures by consciously planning them. As in all research, action research must be planned to obtain scientific knowledge, and to overcome or minimize the limitations of the research approach followed. In the following section we describe the action research method followed in this thesis.

6.1.2 Using action research in studying method engineering

Several models for action research can be found (cf. Baskerville and Wood-Harper 1996, Checkland 1991). They all consist of steps like planning actions, taking actions, and evaluating their results. In addition to these, entry and exit points to an organization must be planned (Buchanan et al. 1988). In studying incremental method engineering, the use of action research can be described according to the process model illustrated in Figure 6-1.

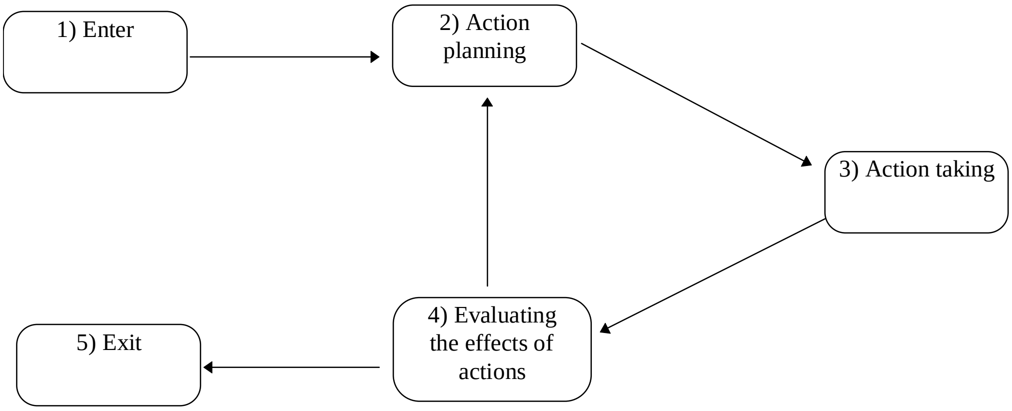


FIGURE 6-1 Action research process model followed.

1) **Entering** means getting access to real-world ME cases and establishing the action research. The criteria for selecting cases can be derived from our problem formulation. The site must be in the middle of a ME problem, deciding how to engineer a method for a particular IS development need?. In our study, the

cases included two organizations which needed methods to carry out specific system development efforts. The first case was related to a business process re-engineering effort involving an inter-organizational IS development in a trade organization. The second case deals with developing sales and outbound logistics in a cardboard mill. Both of the ISD environments were suitable for our study because they lacked methods and detailed method selection frameworks. Both cases had several external stakeholders, were dynamic and consequently had a high uncertainty. Both cases also provided the possibility for longitudinal observation: access to the organizations was possible also after method construction.

Access to cases was obtained within a larger research project in which both companies participated. The funding was not based on the work carried out for the organizations but instead through the organizations' participation in the research project. Participation was arranged as development projects. In these projects, ME efforts were organized as subprojects. In both cases, the ME projects had specific goals and a separate project plan which described its resources and schedule. In the latter case, the ME project also had a separate budget.

2) **Action planning** involves decisions about the objectives and questions of the study, and shows how the study bears on these objectives. The research objective of our studies was to demonstrate the viability of incremental ME. We examined whether situational methods could be specified using the proposed metamodeling constructs, and whether the *a posteriori* ME principles could be used to refine methods. The former was studied by analyzing whether all aspects of modeling techniques could be described with a semantic meta-data model. The latter was inspected by analyzing the outcomes of the ME efforts and the use of methods. If the *a posteriori* evaluation mechanisms neither revealed possible method refinements nor supported learning of method use, they could be considered inadequate. Further studies could then be carried out to analyze which circumstances favor incremental ME and which favor a more "radical" ME approach (cf. Section 5.1.3).

During action planning one also determines data collection mechanisms. To understand the constructed methods and possible method changes in detail, the data collected included metamodels, adapted tools, method manuals and other method descriptions, such as domain models and algorithms for model-based analysis. Method construction rationale was collected from documents describing the requirements for methods, from memos and minutes of the ME projects' meetings, and by participating in the ME process. To understand the actual use of methods, the models developed, the analyses created, and the project documentation were collected. In particular, access to design data stored in tool repositories was important because it allowed to inspect how the method was actually used with the tool. The project reports describing the deliverables of the ISD projects formed anther source of information about method use.

Data collection also covered requirements to change methods and method versions. These were captured in documents relating to ME, and were outcomes of using the mechanisms of *a posteriori* ME. Finally, method engineers were interviewed twice during the study: first while the method improvements were being sought with the mechanisms of *a posteriori* ME, and second at the end of the project. The latter interviews were applied to verify earlier observations and to establish satisfaction (or dissatisfaction) with the method and tool developed. Interviews were not recorded but notes were taken and the resulting descriptions were checked by the method engineers interviewed.

3) **Action taking** carries out incremental ME as described in Sections 3 and 5, i.e. applying both *a priori* and *a posteriori* principles. Both ME phases were carried out by collaborating with stakeholders. They identified ISD problems and provided experience to assess method applicability. The method users were partly the same people who engineered the methods. In the wholesale case, the researcher also participated in the ISD efforts, but in the mill case, participation was limited to tool adaptation and guiding the evaluation mechanisms. Hence, in the latter case the researcher played an expert role. In this sense the latter case was closer to a case study.

4) **Evaluating the effects of actions** analyzes whether the actions have been taken as suggested and how they have affected problem solving. The results of ME actions were described in a baseline documentation which was checked for correctness by method engineers. Using this document, ME actions were analyzed based on metamodels, supporting tools and method manuals. The evaluation included an analysis of whether the metamodeling constructs were applied, and whether their use was considered successful. In the *a posteriori* phase, we analyzed whether changes followed incremental ME principles, and whether the changes improved methods.

5) **Exit** stops the action research cycle. Cycles of action research were simultaneous with the cycles of incremental ME. Each time a method was evaluated the results were recorded and analyzed. This allowed us to carry out a longitudinal study rather than inspect only a snap-shot of the methods (as in Section 2.5 while analyzing the descriptions of ME cases). Because of the time and resource constraints, the actions were limited to one cycle.

Although the cycles were simultaneous they occurred at different levels (Checkland 1991). The ME cycle deals with learning about methods. The action research cycle deals with learning about ME in general and about incremental principles. Both of these cycles should be documented as part of the research (Jönsson 1991). To clarify the different levels of actions we shall briefly compare the action research method and incremental ME.

6.1.3 Comparing action research and incremental method engineering

The description of the action research method shows that incremental ME resembles it in many ways. These similarities are summarized below:

Both are iterative and focus on long-term changes. In principle they form never-ending processes in which learning is used in the consequent cycle.

Both ISD and ME methods are studied in real organizations and in actual use. This focus allows the study and improvement of methods and related technologies. Because incremental ME principles are being promoted only action research and field experiments can be applied to study their viability.

Both are situation-dependent, which can also be considered a weakness if statistical generalization is an objective of the research. On the ME level, the need for situational dependency was already discussed in Section 2.5. On the action research level, situational dependency is reduced through research cycles and by carrying out studies in different organizations (in this thesis two organizations).

Despite these similarities a key difference must be recognized: while the incremental principles operate at the ME level, the action research operates one level higher. These levels are summarized in Table 6-1 based on the domain in which they are applied. In this chapter we operate at the research level since our interest is to study the incremental ME principles. The research subject is method engineering and the mechanisms of incremental approach.

Because of this focus on the research level, in the following sections ME efforts are described from the research point of view. Studying ME principles, however, necessitates that we also describe how methods were constructed (the ME level) and how they were applied (the ISD level). This means that the cases are reported at three levels as follows: Sections 6.2 and 6.3 describe ME cases which followed incremental ME principles (i.e. the action taking part of action research). Although the reporting focuses on how methods were specified and evaluated with the metamodeling constructs, the ISD level must also be recognized. This allows us to explain how the constructed methods were applied and the rationale for their use. The outcomes of action research (i.e. the evaluation part of action research) are described in Section 6.4, in which lessons learned from ME and from the incremental principles are discussed.

TABLE 6-1 Levels of research, method engineering, and information system development.

Level of action	Domain studied	Main actor	Method
Research level	Incremental ME	Researcher	Action research
ME level	ISD method	Method engineer	Incremental ME
ISD level	Application	IS developer	ISD method

6.2 Case A: Wholesale company

This section describes the action taking part of the action research: how incremental ME principles were followed in a wholesale company. The section is organized according to the process of ME (cf. Figure 5-2). First, we describe the background of the company in Section 6.2.1 and characterize the ISD environment in Section 6.2.2. These characterizations are applied in method selection and construction. The results of the *a priori* phases are described in Section 6.2.3 by discussing the metamodel and tool support implemented. Section 6.2.4 briefly describes the method use. The remaining sections focus on an *a posteriori* view: Section 6.2.5 describes the use of evaluation mechanisms, and Section 6.2.6 clarifies refinements and lessons learned from the methods. The research results (i.e. the evaluation part of action research) are described in Section 6.4.

6.2.1 Background of the study

Case A was carried out in a major Finnish wholesale company. Its central line of business is to buy goods and to deliver them to customers through a central warehouse and regional distribution centers. During the study the company was in the middle of a major business reorganization, in which it was decided to remove the regional distribution centers: i.e. a move from a three-level to a two-level organizational structure. The ISD efforts focused on the company's order entry and purchasing processes which had multiple functions, covering both intra- and inter-organizational functions. The main ISD objective was to re-design the ordering and purchasing processes, and develop ISs to support the two-level organization.

The case was chosen because it was thought to be complex enough, and moreover it implemented the idea of business process-driven modeling that covers both hierarchy-based and market-based business processes. In fact, the modeling was carried out in four organizations. In addition to the wholesaler, these were a manufacturer/supplier, a regional distribution center, and a hardware store. Because most of the regional distribution centers and some of the hardware stores were also partially owned by the wholesaler the network can be further characterized as a quasi-market.

The objective of the study was to develop methods which would help identify opportunities to improve order entry and purchase processes. In both of these processes IT plays a significant role. The order entry relates mostly to selling: through quota processing and order receiving to a delivery. The purchasing includes processes that deal with the company's own buying tasks in inbound business operations. In the inter-organizational setting these processes are connected: the stakeholders of the order entry are the company's customers, and in purchasing they are suppliers and manufacturers. In other words, these functions form a net of interrelated processes among companies. Because the business modeling study involved four organizations, the wholesaler's order entry activities had to be seen in connection with the hardware stores' purchasing activities, and so on. Although these functions were common to all four companies, the business development effort was carried out by the wholesaler. Accordingly, the method construction was based on the wholesaler's requirements and problems.

The ME effort was organized as a separate task inside the ISD project. The method was constructed by a person from the wholesaler's IT department and by the participating researcher. In addition, help from external consultants was obtained during problem characterizations. The company had recently hired consultants to carry out a study of the company's logistics. The results of the study were used to characterize the ISD environment and identify problems expected to be addressed with the method.

6.2.2 Characteristics of the ISD environment

The initial requirements for method support were quite general. The method should address inter-organizational processes and it should allow the definition of an architecture for the networked organization. Moreover, because of the importance of the underlying logistics of delivered goods, the method should recognize material flows together with information flows (as proposed by Bititci and Carrie (1990)). These requirements were revised in more detail based on the characteristics of the object system environment.

The initial requirements revealed, however, the necessity of a method engineering approach. First, no contingency framework for method selection was found that could address the basic characteristics of the problem context, such as inter-organizational systems. This was found out by a study reported in Tolvanen and Lyytinen (1994). In fact, the knowledge of developing and modeling inter-organizational ISs is relatively modest; not enough to develop a contingency framework (Stegwee and Van Waes 1993, Vepsäläinen 1988, Clemons and Row 1991, Tolvanen and Lyytinen 1994). Second, we did not find any business modeling method that would satisfy the requirements to model inter-organizational processes, and to specify the network's information architectures (cf. Teng et al. 1992).

6.2.2.1 ISD experiences and method knowledge

In the wholesale company, experiences of methods included data modeling and process modeling. These were part of a method called TKN (Information Processing Advice). The TKN method was mostly used for the requirements engineering and analysis phases. For example, the data modeling part of TKN had been used for conceptual modeling and analysis, but not for schema design. One reason for this was that implementation was outsourced.

The external consultants applied Yourdon's (1989a) structured analysis and a supporting CASE tool (System Architect) in their study. The tool use was considered necessary because of the size of business models, but the method was not considered suitable. Because the method was targeted to develop individual ISs it did not address (de-)centralization, responsibilities among different organizations, or architecture definition. The CASE tool offered method adaptation possibilities by allowing the addition of new attributes (property types in our metamodeling terminology) to existing method types. This support, however, was too limited. No analysis could be made based on the property types added and they only supported the abstraction part of the method-tool companionship.

6.2.2.2 Characteristics of the problem context

Because of the lack of contingency frameworks, the criteria for method construction were sought from the wholesaler's problems. Thus, a characterization of the organization and ISD problems formed the main entry point for method engineering. These characteristics and problems had been identified during the company's own strategy process, and through a recent study that dealt with the company's logistics. The problems are listed below. The numbering of the list allows us to identify their influence on the constructed method. The following problems had been recognized:

1) **Inadequate understanding of other stakeholders' purchasing processes.** Understanding of the external environment was found to be inadequate for the provision of a good external service. Moreover, the shared and fragmented knowledge about business processes (e.g. goals, resources) within the industry made it difficult for the wholesaler to streamline its boundary operations towards a more cooperative environment. For example, in the industry and even in the company's local outlets different rules were applied in purchasing and delivery, including non-uniform product and code standards.

2) **Duplicate tasks and routines.** One of the most obvious problems was the duplication of effort. Each company had its own ordering and purchasing functions and associated supporting systems in which the data was entered. Moreover, the data in the IS is primarily used to serve each organization's own needs. From the network point of view this has led to sub-optimal solutions and to unnecessary complexity in workflow. The wholesale company had already taken some steps towards external systems integration (e.g. data entered once served multiple functions and even multiple organizations), but data integration was still seen a problem. Duplicate tasks in the network increased costs, created errors, and lead to longer turn-arounds.

3. **Customer satisfaction** (i.e. service level) was problematic. Satisfaction had been measured to be quite high from the wholesaler's point of view, but it was considered low on the customer's side. The reason for opposing opinions was not due to different service objectives, but rather due to the way how purchasing and delivery information was shared. Because customers' opinions were not based on statistics, it was expected that better sharing of order and delivery information could improve the service level.

4. **Lack of coordination**. Incompatible systems duplicated data entry efforts and decreased information availability (i.e. data sharing, access rights). The latter was seen to form a major problem in developing shared business processes and supporting ISs. These ISs can share and transmit order and purchasing related information, such as inventory status, orders, quotations, up-to-date price lists, product descriptions, invoices and electronic money transfers. The sharing of information, however, needs to be planned. A concrete example of this was faced in inventory systems where suppliers or buyers had to check another company's product information.

5. **Unsatisfactory turnaround times.** Because of the fragmented logistic functions the turnaround times were not satisfactory. This increased inventory costs. Normally, companies knew their own inventory levels but could not check whether any other store or regional wholesaler "downstream" had a sufficient stock of a given product. Furthermore, this poor availability of delivery information tied with a complex ordering process increased throughput times. Thus, process integration between companies along the value chain was necessary to speed up cycle times and reduce inventory levels.

6. **Lack of demand information**. Because the wholesaler's purchasing system was heavily dependent on marketing information, and on estimated sales, up-to-date market information played a significant role. However, the company did not utilize the marketing information well enough. Moreover, the availability of market information was assumed to be of interest to other participants in the industry (i.e. suppliers, importers, and manufactures).

6.2.3 Business modeling method constructed

Here we shall introduce the modeling techniques using a metamodel and discuss their tool support. We describe how methods were selected and modified to fit the characteristics of the problem context.

6.2.3.1 Metamodels

Two well-known methods formed a starting point for the method construction, namely value chain and value systems (Porter 1985, Macdonald 1991), and Business Systems Planning (IBM 1984).

The method construction was guided by the ISD characteristics and problems. During the construction step we applied metamodeling to specify the methods and their interrelations. Figure 6-2 contains a metamodel of the selected parts of the methods and their interactions. The model is based on the GOPRR metamodeling technique discussed in Section 3.3.3.7 and in the appendix.

The first part of the business modeling effort was to describe interrelated business processes and their relations. This part we call value process modeling, after Macdonald (1991). The value process models describe value adding processes and their dependencies while providing products and services to the "final" consumer. Although the traditional value chain (Porter 1985) concentrates on the value adding capability via different types of processes (i.e. inbound, operation, outbound, etc.) we extended it to include delivery-related properties, such as 'location', 'capacity', 'volume' and 'turnaround time'. These properties we defined as optional whereas 'type of process' and 'process name' were considered mandatory. The mandatory constraint, however, could not be modeled into the metamodel and was therefore not actively checked. The checking of mandatory constraints was enabled by the analysis reports implemented (i.e. passive checking).

Although in a value chain most information and material moves downstream, we also wanted to model the opposite because it allows us to analyze problems related to rework. In other words, duplication of work (cf. problem 2) often occurs as a result of failures or defects in providing services (Harrington 1991), causing a return "upstream" in the chain. This is specified in the metamodel by allowing customers and business processes to send (participate in "flow from" role types) information and material.

Each process was further described by an actor to illustrate process responsibility. In cases where the necessary information was not available a process could be decomposed. According to the metamodeling constructs this structure was defined as a dependent, non-mandatory and exclusive complex object. The metamodeling language, however, did not support these more detailed characteristics of complex objects (see also Section 4.5). It allowed, however, aggregating different levels of value process models and business processes. In the GOPRR metamodel this is described with a decomposition link (a dotted line with an arrow-head).

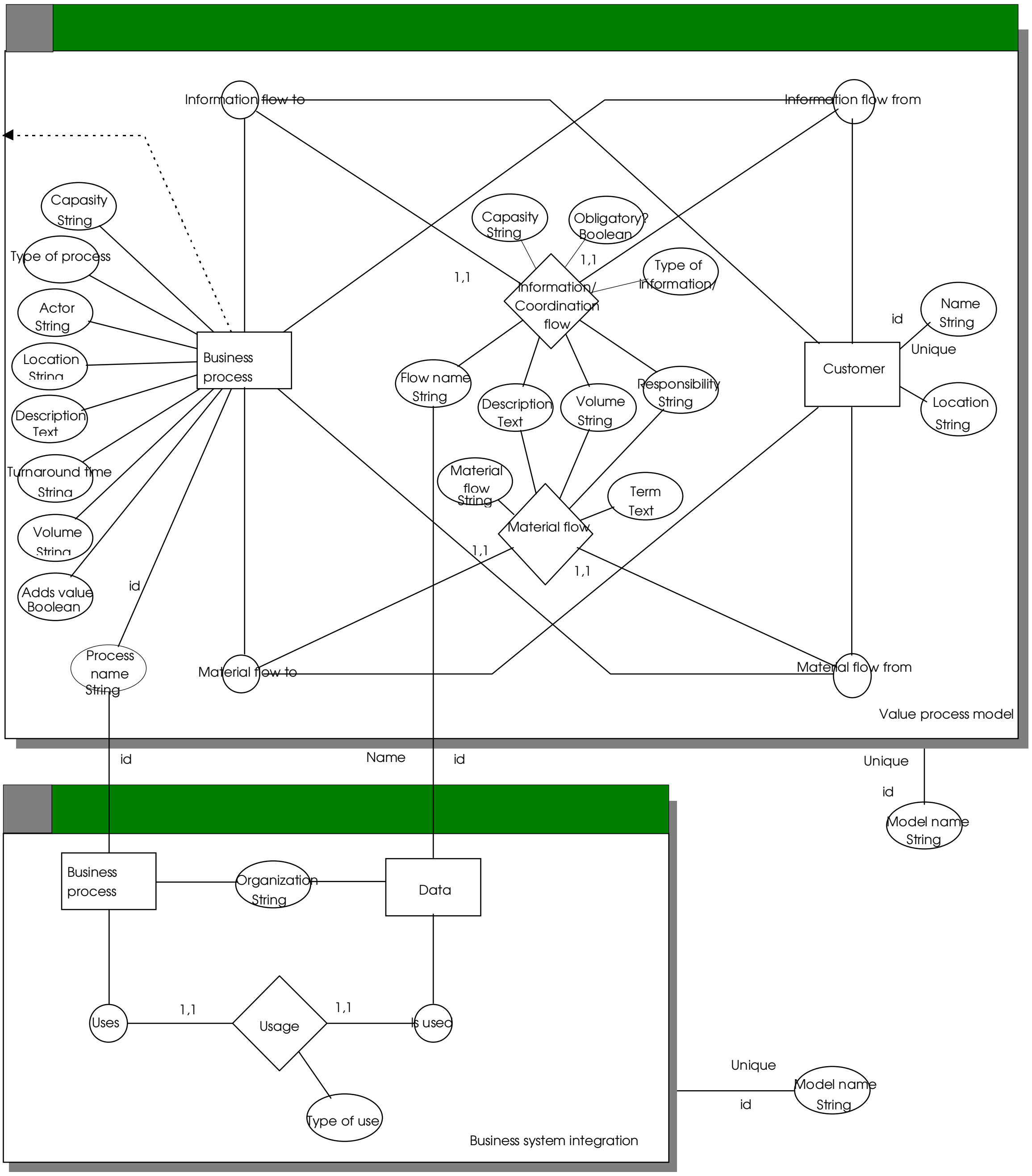


FIGURE 6-2 A metamodel of the *a priori* constructed method.

The process models concentrated on material flows and on process information. In this way, it was possible to identify information requirements for processes that control material handling (cf. Bititci and Carrie 1990). Both flow types were characterized by their name, description, mean volume, and responsibility. Material flows were further defined by possible terms of delivery. Information flows were specified according to their type (i.e. order, payment, report or control), maximum capacity, and status (obligatory, optional). Accordingly, the aim of the value process modeling was to establish a common description of a network of ordering and purchasing processes (problem 1), identify duplicate tasks (problem 2), and help to focus on areas which could considerably improve customer satisfaction and cycle times (problems 3 and 5).

The level of IS integration among the companies was modeled using a business system integration method, which was a modified version of BSP (IBM 1984). The use of the original techniques included in BSP (see Table 4-1) was limited to modeling data use in business processes using CRUD (create, read, use and delete) matrices in architecture planning. The modeling techniques were integrated through polymorphism: the names of business processes should be the same in value process models and integration models. Similarly, data described in architecture models was expected to be specified in value process models. In other words, the system architecture should not have data classes which were not specified as instances of flow types in the value process models.

The method also supported modeling of market based IS integration solutions instead of focusing on integrating processes inside a hierarchical regime. This was achieved by dividing data handling processes among different organizations (a property type 'organization' in the metamodel, see Figure 6-2). Each business process was characterized with the organizational unit it belonged to, and thus organizational dependencies were represented. In BSP this is achieved by inspecting organizational units against business processes. Thus, unlike BSP the integration method described IS architectures where each company had both local and inter-organizationally shared business processes and data. Moreover, it defined the inter-organizational responsibilities, data sharing and data availability (e.g. create, use). The objectives of the integration method were to address and solve problems related to inter-organizational IS architectures, to improve coordination through shared data (problem 4), eliminate duplicate data and processes (problem 2), and to improve availability of market information (problem 6).

6.2.3.2 Tool adaptation

Both modeling techniques were supported by a computer-aided tool. The value process modeling was supported by a metaCASE tool, and the business system integration was supported by a spreadsheet tool.

The metamodel of the value process model was implemented in a metaCASE tool called MetaEdit (MetaCase 1994). The notation of the value process model is represented in Figure 6-3, in which a high level view of the wholesale process is described. With respect to the other parts of the method-tool companionship, checking and documentation reports were implemented. The checking reports operated on those aspects of method knowledge which needed to be checked passively, or were not possible to capture in the metamodel. The checking reports included unconnected object types (i.e. minimum multiplicity one) and undefined properties (i.e. mandatory property types). The multiplicity of types was not inspected because only two object types and relationship types were used. The documentation reports included dictionary reports and flow reports. The dictionary report describes property definitions for all instances of the 'business process' and the 'customer' object types. The flow reports describe use of information or material from the business processes side (i.e. flows in, flows out) and from the flow side (i.e. which business processes use a specific information flow). The reports on information flows were used to build the architecture models into a spreadsheet. The value process model captured most of the design data required for architecture definition, except the type of usage and the organization. The organization information could also be detected from the model hierarchy, although it was not included as separate property type in the metamodel. The flow reports also served as a basis for documentation and to deliver models for validation and further inspection.

Because of the use of a non-metamodel driven tool for business system integration, metamodel based method knowledge could not be applied. The reason for this was the lack of matrix representation support in the metaCASE tools reviewed (cf. Bidgood and Jelley 1991). The matrix representation was considered a necessity because it allowed the analysis of large architecture models among four organization types in a condensed form and the representation of couplings between processes and data. Matrices also provided an abstraction required to develop alternative architectures based on information availability.

6.2.4 Method use

The ISD project took over half a year, and seven persons from all four organizations were involved. Most effort was needed to develop the wholesaler's downstream activities. The participation of a supplier organization was limited because they were only interviewed to obtain their requirements. The value chain of the wholesale process is described in Figure 6-3.

The figure is based on the value process model. The model describes major parties and business processes. Organizations participating in the ISD are illustrated through grayed business processes. The value process model describes only material-based relationships (represented as thick lines with an arrow head). During the ISD project, delivery, ordering, and purchasing related controlling information flows were described. In addition, each participating organization was modeled in more detail by decomposing business processes.

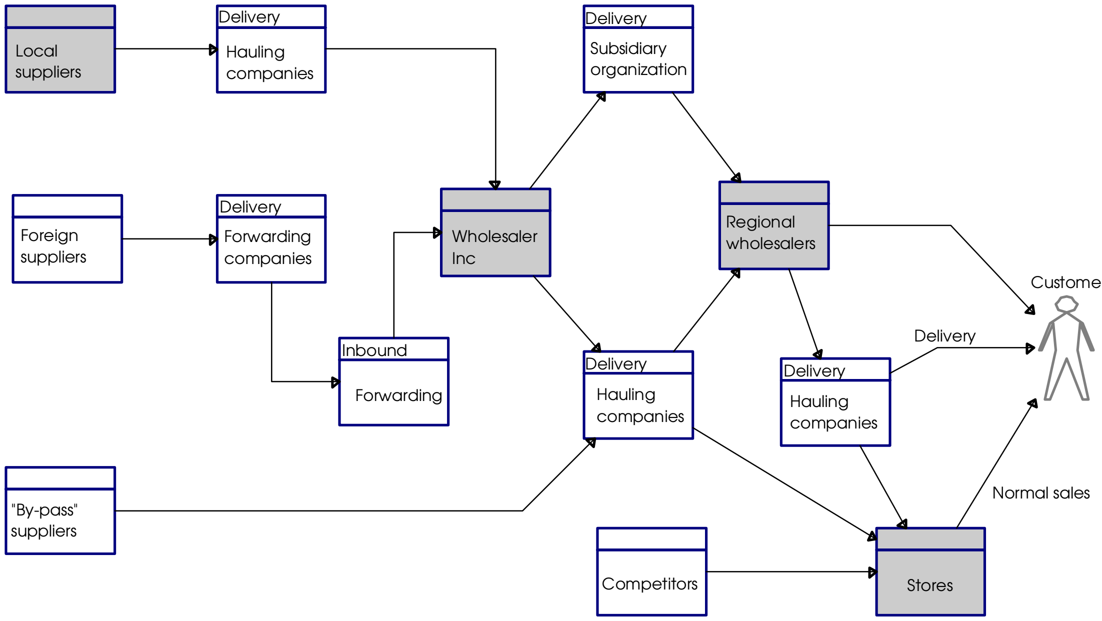


FIGURE 6-3 Value chain of the wholesaling industry (modified and partial).

The order entry and purchasing system was decomposed into around 60 business processes, 140 different information flows, and 30 material flows. The main outcome of the project was three solutions for managing purchasing and ordering related processes. These alternative approaches were differentiated based on the responsibility given to different actors. All these alternatives required a new IS for sharing ordering/purchasing related information. A "pull" solution configures the chain based on the market needs as recognized by the stores: all ordering functions and related purchasing functions of the wholesaler are based on sales. A "push" solution means the opposite. It offers control mechanisms for the wholesaler to monitor sales from the field. This provides better prediction for the wholesaler's purchasing functions, and offers possibilities to balance inventories. A hybrid solution means a combination of these based on the type of goods: for example, sales of low volume products are difficult to predict requiring a market-based strategy (i.e. the pull alternative), whereas seasonal products could be planned by the wholesaler (i.e. the push alternative).

The alternative solutions and their influence on problem solving are described in the next section since they were applied in evaluating modeling power and problem solving capabilities.

6.2.5 The *a posteriori* method engineering

In this section we explain how the method was refined during the case based on the experiences from method use. We first apply type-instance matching. This part of the study was conducted by the researcher/method engineer alone.

Second, we assess the applicability of the method in terms of how well it supported business modeling. Third, we try to identify the role of the method in ISD. These latter two evaluations were conducted by interviewing the stakeholders based on the method evaluation and refinement mechanisms described in Section 5. The stakeholders involved were from the wholesale company and mostly from its IS department. Hence, because the problem characterization and method construction was accomplished by the wholesaler, the method refinement was accordingly conducted from the wholesaler's point of view.

6.2.5.1 Type-instance matching

Type-instance matching deals with inspecting how the constructed method has been applied. The comparison is made between the method's intended use (as seen from the metamodels) and actual use (as seen from the models). In the following we describe only the results of this evaluation, i.e. those differences between models and metamodels which suggested method refinements (cf. Section 5.3.3 for details). Therefore, those questions or evaluation alternatives which did not reveal any differences are excluded. Similarly, it must be noted that not all constraint-related evaluations will be inspected, because the metamodeling language could not capture them.

6.2.5.1.1 Usage of types

1) **Unused types**. All non-property types were used but several property types had few, if any, instances. None of the unused property types were redundant with other property types, but they were not used because design information could not be found, or was not considered cost-effective to find. The 'turnaround time', 'capacity' and 'volume' were defined for only 5% of instances of the 'business process'. The business processes which included these property definitions operated at a detailed level, or at the organizational boundaries. The 'actor' was defined in 20% of the business processes because this was considered redundant while decomposing processes. In other words, actors of lower level business processes were the same, or specific groups of those in the higher level business process.

As a result, the property types could be removed from the value process model. Although some other property types had few instances they were not removed. The 'volume' and the 'responsibility' was defined in only 5% of the material flows, but for almost each information flow. Because no special reason for treating the flow types differently was found (other than the primary focus on information flows during the project) no modifications were made to these property types.

2) **Dividing or subtyping of types** was considered necessary in two cases. First, processes had differences in their naming. Some high-level processes were named according to organizational units (e.g. inventory) whereas other were tasks of employees. For the latter cases, the 'turnaround', 'capacity, and 'volume' property types were applied. This suggested that processes must be divided into higher level business processes and into employee tasks. Second,

because of the inter-organizational setting, several information flows with the same name referring to different flows were modeled. For example, an order had a different meaning and content in different companies. This could be detected from models which had organization-related descriptions related to flows. For example, "confirmations of an order are delivered directly to shopkeepers". Although this was acceptable while modeling information flows of individual companies, it was not desirable for making an information architecture for the whole network. Therefore, the flows/data should be specified in terms of the organization and its content.

3) **Definition of new linkages** between property types was suggested in only one situation. Actors and responsibilities of flows shared the same values. Also, the direction for sharing property values was found, because all actors were also specified in flows. This aspects is analyzed in more detail later.

6.2.5.1.2 Usage of constraints

Analysis of constraints is limited to those defined in the metamodel and supported by the tools. Some of the constraints which could not be captured into the metamodel, however, could be supported by the tool. These constraints include the unique property, the mandatory property, and the multiplicity constraints. For example, a tool could warn about property types which are not defined, although such a mandatory constraint was not defined in the metamodel.

Active checking of the mandatory property constraint was considered important because all classifications of property types were not specified. As a result, separate architecture models could not be created automatically for the current ordering system (i.e. by selecting all order-related information flows from the value process model). The 'type of information flow' property type included also values other than those which were predefined. The most used was delivery related information. It was considered relevant for logistics modeling and had to be added to the predefined values to speed up the modeling work. This addition was also considered important for analyzing management of delivery information.

Several business processes had flows with the same name, suggesting the need for n-ary relationships (a role's minimum cardinality greater than one). Although this indicated duplicate modeling effort in situations where design data is updated, the use of n-ary relationships was considered unnecessary. Moreover, binary relationships could be used for the same purpose. Our metamodeling constructs did not even have a constraint which would necessitate the creation of n-ary relationship if two binary relationships with the same instance information existed.

Multiplicity constraints over two role types could not be supported but the model indicated that this should be the case for all object types in both techniques. In other words, existence of an instance of either of the role types suggested that the role types should be defined as mandatory (i.e. minimum multiplicity one). Moreover, a typical recommendation in architecture design, that only one process should create data (i.e. be responsible for it), was present in the models. Modeling the present state suggested, however, that it should be possible to model more than one data creating process.

The specification of complex objects had to be changed: dependency and non-mandatory rules were applied while decomposing business processes, but an exclusive component constraint was not. One reason for this was the need to combine detailed process models and the development of different versions for representing alternative solutions. Instead of hierarchical leveling with exclusive components (similar to decomposition in data flow diagrams) the value process models were unified at lower levels showing detailed workflows between companies. This required shared business processes in complex objects. Second, the analysis of scope for the constraint suggests a change from the method to the model. Otherwise different versions using the same process could not be made: a tool would necessitate aggregated relationships for all instances of the process regardless of the model where it is defined (i.e. decomposed or combined process models). This would result in a model which included all relationships (and a whole model hierarchy) instead of specifying those necessary only for the current version.

Analysis of values among different types revealed one new candidate for a polymorphism structure between the 'actor' and the 'responsibility'. Here the same value could be used although they are semantically different. The actor means the acting part in the business process whereas the responsibility is used to define the instance responsible for delivering the data.

6.2.5.2 Modeling capabilities

The tool supported modeling with abstraction and checking capabilities. Before evaluating these we describe how the method was used in modeling the object system. These characteristics are the same ones which drove the method construction. First the way of modeling is described and then abstraction and checking capabilities are evaluated.

1) **Inadequate knowledge of stakeholders' processes.** Because of the inter-organizational nature of the object system, the wholesaler's knowledge of partners' processes was modest. In general, only processes that related to costs or interactions at the organization's boundary were documented. In synthesizing this fragmented knowledge the value process model proved to be useful. Its main impact was that it helped to describe all business processes related to order entry and purchasing, which were shared processes in all companies. As is typical in logistics, the specification of material flows between multiple participants and their mappings to controlling information flows were considered useful. In particular, process dependencies and responsibilities were revealed which helped participants see information handling policies.

The main difficulties in abstraction related to characterizing processes with logistic information. These were already recognized as unused types (i.e. unspecified turnaround times, capacity and volumes related to processes). In most cases the business process information was not found, and if such was specified, it was related to processes at organizational boundaries, or to an individual's tasks. Moreover, the value process model operated at too general a level. This demanded modeling of a detailed workflow. Process modeling was found redundant in maintaining process related information between different levels of the process hierarchy. For example, a turnaround time of a business process should not be smaller than the sum of those specified to its subprocesses. The manual maintenance of the property values was one reason why such data was not specified. This required derived data types or checking reports which could calculate business process related characteristics from the properties of its subprocesses.

2) **Duplicate tasks and routines.** In networked processes, effort duplications occurred at the department and especially at the company level. In the study, system integration models were used to describe network-wide processes that use or create similar local data. Examples of such processes were order entry and delivery notification. The value process showed the structure of tasks, but not how the processes are carried out: In particular, the analysis of the current situation required descriptions of more detailed tasks structures and decisions. For example, the value process model did not describe alternative possibilities to make orders depending on the current availability of goods. This suggested a concept of a decision in relation to the task structures.

Because modeling tools were separate, maintaining consistency between models created duplicate work. Each change needed to be updated to other types of models and the information flow report from value process models to integration models was only used once when the whole network was transformed into a spreadsheet.

3) **Customer satisfaction on delivery** did not involve any other modeling concept or constraint than the involvement of customers (i.e. stores). The modeling support therefore dealt with specifying delivery related information flows together with the customers of the wholesaler.

4) **Lack of coordination.** The possibilities for inter-organizational business integration were estimated by deriving IS architecture models for each company and then later integrating them into a network wide model. During modeling, difficulties arose because of homonym and synonym problems in the data, and because the same data class could contain different information. In order to specify IS architectures in more detail — e.g. differences in data classes among companies (e.g. in orders or inventory data) — data modeling was regarded as important: the currently used techniques were considered inadequate to examine these differences.

5) **Unsatisfactory throughput times.** One objective for modeling was to gather data on logistic measures (i.e. capacity, turnaround times, and delivery conditions) to help find efficient solutions. In practice, however, we faced several obstacles in accomplishing this task: the smaller companies did not have the required information on their logistic measures, or it was not in the required format. Although all companies knew in detail their material handling processes which operated at the organization's boundary, information about internal processes and about non-cost items was seldom available. Because logistic measures give a detailed picture of the efficiency of the organization this information was at times kept secret. Moreover, the modeling revealed the need

for different modeling constructs at different modeling granularities (i.e. detailed tasks are specified differently from general business processes).

6) **Lack of marketing data.** The availability of marketing data was modeled like any other information flow. The value process models were used to identify the wholesaler's and stakeholders' information requirements, and the integration model was used to inspect data coordination aspects. As with modeling shared data, the models had to be supported by tools for data modeling (e.g. ERD).

6.2.5.3 Problem solving capabilities

In incremental ME, evaluation is carried out by comparing modeling outcomes and method principles used to achieve these outcomes. We inspected this using form conversion and review mechanisms. Form conversion means the capability of a tool to analyze models and generate candidate designs. Review mechanisms mean production of documents for stakeholder needs and validation.

In the following this evaluation is described. First we describe the project outcome and then the role of the method is discussed.

1) **Knowledge about stakeholder processes** was improved by using the value process models. These helped participants correct or verify their assumptions of process dependencies and find information that originated outside their organization. Thus, the value process models mostly supported the validation and uniform documentation of processes among companies. In the form conversion part, the process and information flows were also converted to tentative design data in the business integration model. As a result, all use-based connections between processes and data could be automatically converted into the CRUD matrix. Other types of usage could not be converted, because no indication could be given in flows as to whether a business process for example had created or only updated the data.

2) **Duplicate tasks and routines**. The business integration method allowed the identification of redundant information handling processes and generation of alternative candidate designs. This is similar to BSP (IBM 1984) with the distinction that data availability in our case is based on different organizations. Hence, solutions were sought by inspecting outcomes of different data integration and sharing possibilities between companies. These alternatives included, for example, that the wholesaler's inventory information is available in real-time for the stores during purchasing, or that manufacturers can have access to the wholesaler's inventory and sales information. As a result, duplicate tasks, both in the order entry and purchasing activities, were removed through improved information sharing between companies. These changes also simplified processes by reducing their complexity, especially in tasks related to handling special kinds of orders, order confirmations, and out of stock reports. The spreadsheet tool did not automate solution generation, although this could have been defined based on the metamodel.

3) **Customer satisfaction.** As a result of the modeling effort, customer satisfaction was improved by offering more accurate information through an

on-line ordering system about products, the customer's order base and delivery status. These changes were obtained by first modeling purchasing processes and then customers' information requirements. The proposed solutions decreased customers' uncertainty, improved the wholesaler's responsiveness, and moved redundant tasks (such as recording follow-up of orders, and re-ordering, and related decision making) from the wholesaler to stores. These changes were also presumed to bind customers more to the wholesaler. Although none of the metamodel constructs were directly used to analyze or improve customer satisfaction, the recognition of delivery information in the instance models allowed the recognition of availability of delivery information.

4) **Lack of coordination.** One of the project outcomes was the overall IS architecture. The method allowed the construction of several candidate designs, including both "hierarchy based" and " market based" data integration. By hierarchy based integration we mean local and company related information modeling, and by market based integration we mean the integration of data across multiple companies. As an example of a candidate design based on a market driven approach, we proposed order entry and purchasing systems which focus on supporting stores and distribution centers by employing the wholesaler's or even the manufacturer's inventory and delivery information (i.e. the pull solution). A totally opposite approach would have offered improved control mechanisms for the wholesaler (i.e. the push solution). For example, by gathering sales and inventory information from the field, the wholesalers could unify processes downstream in the chain, e.g. to control product mixes, or provide information for marketing and inventory control for stores. By these changes the wholesaler could achieve economies of scale and further improve its own purchasing processes. In line with the wholesaler's business strategy, the selected data coordination mechanisms tightened the relatively free mechanisms towards a more uniform and cooperative one. Because of the flexibility of demand, the suggested solutions still allowed a pull solution for selected products and customers. At the same time it also offered a more controlled service to other customers or goods which are easy to handle and predict (such as goods which have a stable demand, a cycle in patterns, or can be delivered quickly). Because of the lack of full CASE functionality, this part was not supported by automatic conversion mechanism provided by matrix based tools (e.g. Kelly 1994). However, conversion reports provided design information to manually build integration models.

5) **Unsatisfactory throughput times.** One objective for ISD was to gather data on logistic measures that help find efficient solutions. The value process models did not offer enough information about task structures or logistic measures. Because of unavailable data, such analyses could not be made with the tool, although the analysis functionality (flow-in/flow-out reports) was implemented. Hence, the method failed to offer immediate solutions that could improve cycle times or decrease inventories.

6) **Lack of marketing data.** Solutions for information gathering included an application for summarizing order and sales data to support the wholesaler's purchasing processes. This data also attracted interest outside the company, especially among the manufacturers. One feasible solution for this problem was

an on-line communication system, which would allow the wholesaler to make queries downstream, e.g. about campaign products sold, or information about marketing progress and delivery schedules. In solving this problem, both methods were applied. The value process models were used to identify the wholesaler's and stakeholders' information requirements and the integration model was used to inspect coordination aspects.

6.2.6 Method experiences and refinements

The outcomes of the method evaluation were two-fold. First, it offered possibilities to refine the used method, and second it supplemented existing knowledge about methods and method contingencies. In our case, method development focused mainly on addressing the networked material flow. Accordingly, we shall concentrate in the following on the contingencies related to the organization's logistic ISs. Experiences from value process modeling confirmed earlier observations (cf. Österle et al. 1990, Macdonald 1991) of its applicability in process integration. Especially in cases of multiple companies (e.g. with customers and suppliers), the method helped clarify both information and material based process dependencies. Moreover, the method was found to be applicable for network-oriented modeling where the knowledge of the business is dispersed. At the same time, the method presumes a strong commitment from participants, especially in cases where the same modeling accuracy and detail is required.

Problems in data gathering revealed, however, that the method is not suitable in cases where the processes are not documented, or where they are constantly changing. Furthermore, the value-oriented approach seemed to be appropriate only in modeling higher level views. Therefore, in situations where a more detailed representation was required, and we lacked general process measures, other methods were needed. The task of business system integration was likewise hindered by the lack of information. This was especially the case in dealing with inter-organizational relationships, where each company had a similar kind of data (such as an order), while its actual content differed greatly. Thus, although most methods for IS architecture definition do not strive to develop detailed data models (Österle et al. 1990), our modeling case clearly demanded the use of such methods. Like most methods for architecture definition (e.g. Business Systems Planning), the business integration method is suitable for organizations which are centralized (Sullivan 1985), and where some architecture and system specifications already exist.

A second outcome of the incremental ME was method refinements based on method use. The suggested method refinements are defined by changing the method specifications. It must be noticed that none of the required changes to the method could be predicted earlier. As the method assessment clarified, the necessary changes to the method related to modeling task structures and data. In the case of value process modeling, specifying detailed task structures required more detailed constructs (as in problem 2 for specifying more detail tasks, or in problem 5 for finding unsatisfactory throughput times): Value process models are not rich enough in dealing with a fine granularity of modeling where we want to describe a team's or an individual's task structures. Some of the necessary data (such as cycle times in problem 5) could be derived only through modeling system dynamics (cf. Jordan and Evans 1992). For these reasons, we examined techniques suitable for modeling business and task dynamics (e.g. Dur 1992). Detailed models of tasks could be utilized in representing dynamic features of logistic processes.

The modeling technique to be used for describing task dynamics and its connections to the value process model is shown in Figure 6-4. In the new metamodel each business process can be further specified either by a new value process model, or by a task structure. In a task structure, a 'task' depicts actors and their jobs, a 'transition' specifies an order between tasks, and a 'decision' possible alternatives and choice logic. The 'task' is further characterized with properties which originally were related to the 'business process'. Hence, task modeling can support information gathering about the capacity, volume and turnaround times which were found difficult to specify at higher levels. The use of task structures could be further specified to enable analysis features. These could include data about actors' workload, delay and priority of tasks, transitions, and other behavior to handle alternative conditions in transitions. These analyses were not made because the aim of the study was not to tune individuals' tasks structures, but rather to design the overall architecture of the ISs.

In carrying out system integration the requirements for a more detailed data analysis could be satisfied by connecting an entity-relationship diagram (ERD) to the business system integration method (see Figure 6-4). This refinement related mostly to making higher level abstractions and improving the analysis of common/shared data, i.e. problem 4. Here data classes identified in the business system integration models were defined in terms of ERDs. This was expected to allow the specification of different views of the same data and inspect differences in local data, e.g. in ordering, where information requirements are often different. Another example can be found in purchasing, where the wholesaler's information requirements are totally different from those of regional wholesalers and stores, and where the terms of delivery and prices are permanent. The conceptual structure of an ER diagram followed the TKN method already used in the wholesaler's IT department, and was similar to the metamodel developed in Section 4.3.2.

In addition to these new modeling techniques the existing ones were modified. The type-instance matching added new predefined values for property types, such as delivery information to the classification of information flows. Similarly, a polymorphism structure was defined between the 'actor' and the 'responsibility'. This modification speeded up modeling and improved consistency: it allowed to reflect changes in one actor value to all other flows or business processes which referred to the same value.

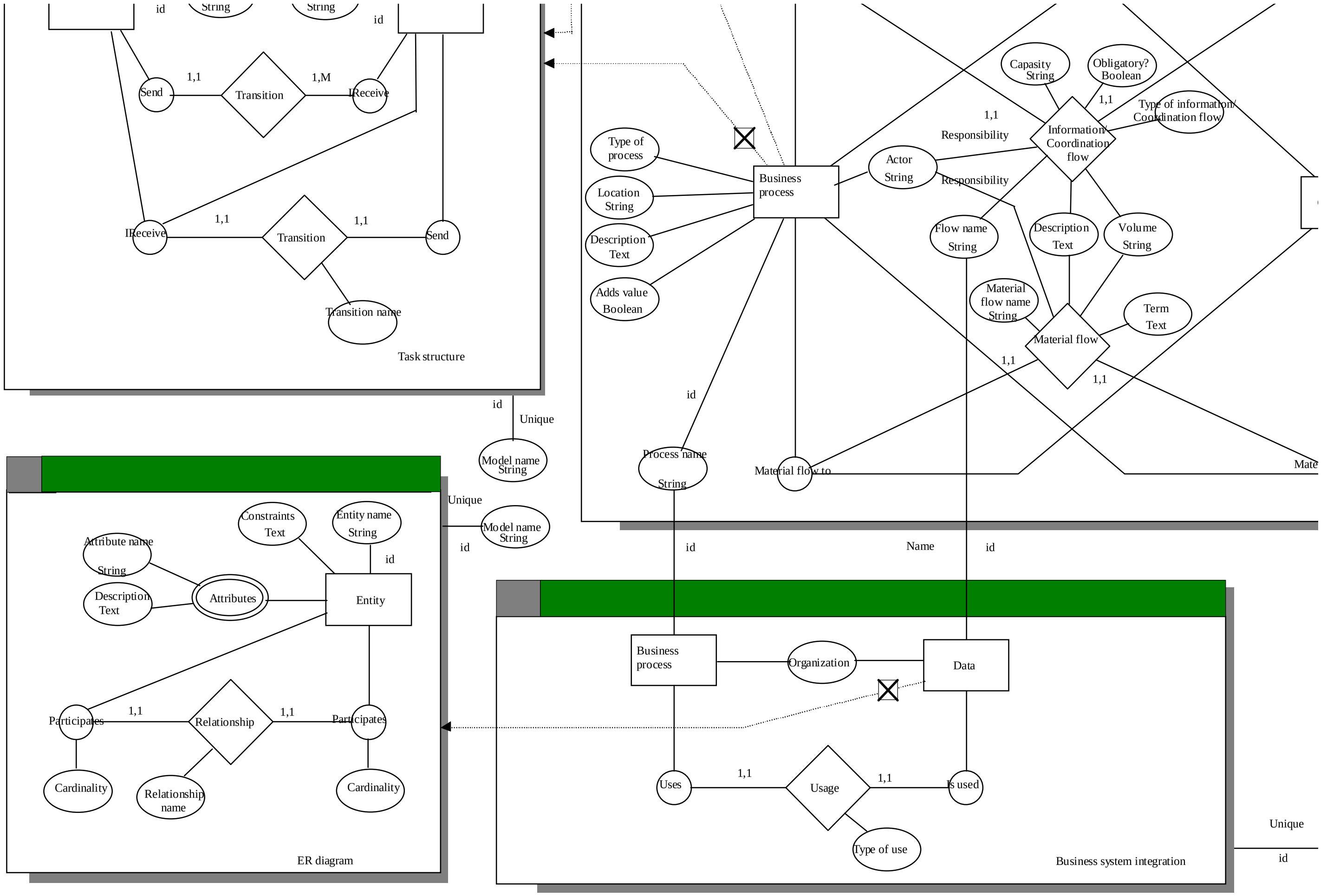


FIGURE 6-4 Method after refinements.

The method evaluation also suggests changes which could not be captured into the metamodel or supported by the modeling tool. Because of the limited metamodeling power of OPRR (see Section 4.5), the metamodel could not adequately specify identifiers, uniqueness and mandatory properties. Other constraints which were needed and not supported related to multiplicity of roles, complex objects, and polymorphism. This means that the tool could not check actively that the method knowledge was followed. These constraints can, however, be supported passively through reports.

In addition to the metamodeling constraints applied for evaluation, the case reveals a need for a derived data type. By a derived data type we mean a property type whose instance value can be calculated from other instances values. For example, turnaround times needed to be calculated from lower level task structures. Similarly, derivation of these values can be performed with reports. For example, if actor names are not given they could be derived from the aggregate business process.

Consistency checking problems suggest the use of a single modeling tool which supports different representation forms. This modification, however, is related more to the required features of the modeling tool than to the method, and therefore is not considered further here.

6.3 Case B: Logistic processes and a cardboard mill

This section describes the ME efforts carried out for developing logistic ISs. Unlike the wholesale case, the aim of ME was not to develop a project specific method, but rather a domain specific method. The method was engineered by a consulting company for redesigning business processes related to logistics. While reporting the case, we focus on method evaluation in using the method for modeling outbound logistics of a cardboard mill.

This two-party setting is reflected in the structure of the section. First, in Section 6.3.1 we describe the background of the method development effort. The *a priori* ME phases are described in Section 6.3.2, along with the metamodels and tools implemented. Section 6.3.3 characterizes the ISD environment in the cardboard mill and Section 6.3.4 briefly describes method use. The remaining sections focus on the *a posteriori* view: Section 6.3.5 describes the use of evaluation mechanisms and Section 6.3.6 the refinements. The outcomes of the action research for both case A and case B are described in Section 6.4.

6.3.1 Background of the study

The case involved two organizations: a large research and consulting company systematizing business process re-design (BPR) practices, and a cardboard mill undergoing BPR. This two-party setting means also two entry points for our action research study: first to the consultant company developing the method, and second to the mill as an application area for the method. In this section we describe the background of the former: the consulting company and its BPR method. The cardboard mill is described in Section 6.3.3.

6.3.1.1 Data model of logistics

The action research study was directed towards ME from the start, because the methods and tools applied by the consulting company were considered inadequate. The decision to develop their own method was supported by a relatively large evaluation of logistics-related modeling tools (Lindström and Raitio 1992) and by piloting and using various methods (including IDEF (FIPS 1993a), communication matrices, state models, and data flow diagrams). The method evaluations were not systematic; rather, they were based on a trial-and-error procedure. ME was expected to support more fine-featured method construction and tool adaptation. In fact, entry to the company was obtained because of the decision made to apply metaCASE technology for building tool support for their own method.

At the time the study was started, part of the method selection process had already been carried out. The result was a metamodel of logistic processes and ISs which can be considered as a reference model for developing logistics (i.e. at the IRD definition level). This model was called a data model of logistics, in contrast with reference models of logistics, which include example solutions (i.e. at the IRD level). The data model was developed based on experiences in developing logistics in different type of companies.

The data model of logistics was specified by following a variant of an ER model and by using examples. Because of the ER model, the logistics data model included only a few modeling-technique-related constraints (i.e. a multiplicity of a single role and an identity) and no representation definitions. In fact, the model focused primarily on defining key concepts and their relationships rather than modeling techniques. Examples of the concepts were a chain, a process, a task, a job, an organization, a resource, and a transfer (split, join, or copy). The data model was complemented by defining the semantics of each concept and by defining major attributes of the concepts. The objective of ME was to construct a method based upon the data model of logistics and other model analysis related requirements. These are discussed in the following section.

During the study, the ME effort was organized into a separate project. The method was engineered mostly by three consultants. In addition, some feedback about the method was obtained during pilot use from the manager responsible for sales and delivery logistics. My role in the *a priori* method construction was limited to the tool adaptation, i.e. modeling the method according to the metametamodel applied in the selected metaCASE tool, implementing the required checking rules and reporting algorithms, and making connections to external tools. With respect to the *a posteriori* ME principles, my role was related to introducing and teaching the evaluation principles, and carrying out the evaluation together with the method users. During the study the *a posteriori* evaluation was carried out after the ISD project.

6.3.1.2 Requirements for the constructed method

As already mentioned, the basis for the ME effort was the data model of logistics. Because the model focused mainly on the conceptual structure it neither defined how logistic processes should be represented, checked, analyzed and documented, nor considered method-tool companionship. It emphasized concepts required for understanding object systems rather than carrying out a change process. Therefore, the main emphasis in the ME effort was in the analysis and model checking part: what should be checked and analyzed about logistic processes for the purpose of re-design, and how this analysis should be supported by a tool. In this sense, ME was driven by the formulation of the logistic related problems to be analyzed.

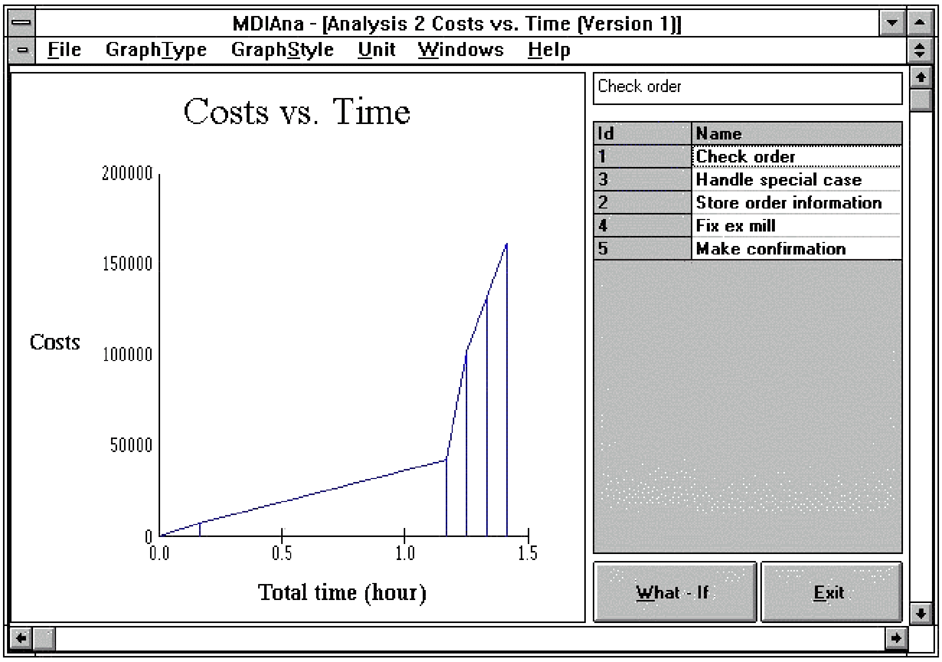
In the following we describe the type of analyses which were intended to be carried out while developing logistic ISs. Each of the analyses raises requirements for the method construction (cf. Section 6.3.2). The suggested analyses were partly a result of analysis needs faced in earlier ISD efforts, and partly adopted from other methods (e.g. Harrington 1991, Dur 1992, Lee and Billington 1992, Johansson et al. 1993). The following types of analyses were considered:

1) **Minimize delays.** In logistic systems it is essential to improve the cycle time because delays increase costs. A cycle time is the total length of time required to complete the entire process (cf. Harrington 1991, Dur 1992). It includes working time, and also waiting and reworking. Delays in the process are defined through tasks with the most idle time in relation to working time. Therefore, the analysis deals with comparing effective processing time to whole cycle time. The timing was considered to be calculated from tasks and from transitions between tasks (cf. Harrington 1991). Moreover, the analysis was planned to be carried out on a subset of the network and also, if required, to the whole network.

2) **Minimize costs.** Processes which have high costs should be selected for further analysis. In logistics, the cumulative cost should be analyzed together with the consumption of time (cf. Figure 6-5). This means for example that higher costs are acceptable if they improve the cycle time, or that small cost tasks which do not improve cycle times may not be acceptable.

3) **Minimize non-value adding tasks** deals with evaluating the process to determine its contribution to meeting customers' requirements (Harrington 1991). In short, real-value-adding tasks are the ones that a customer is willing to pay for. Hence, the objective is here to optimize a process by minimizing or eliminating non-value-added tasks. With respect to logistics, the analysis is related to cycle times and cumulated cost.

4) **Simplification of processes** deals with removing tasks from the process which add complexity and make understanding of the process difficult (Davenport and Short 1990, Harrington 1991). The result would be fewer tasks and task dependencies which make the whole process easier to understand. The simplification is based on analyzing processes which have complex information flows, involve checking, inspection of others work, approvals, creating copies, and receiving unnecessary data.


MDIAna - [Analysis 2 Costs vs. Time (Version 1)]
File GraphType GraphStyle Unit Windows Help
Costs vs. Time
Costs
200000
150000
100000
50000
0
0.0
0.5
1.0
1.5
Total time (hour)
What - If
Exit

Check order

Id	Name
1	Check order
3	Handle special case
2	Store order information
4	Fix ex mill
5	Make confirmation

FIGURE 6-5 Cost-cycle time chart (cf. Harrington 1991).

5) **Organize around processes** deals with re-designing an organizational structure based on a workflow and an overall process structure (Johansson et al. 1993). In other words, instead of following current responsibilities and resource allocations, the organizational structure should be formed around the process. Here, the required analysis covers information or material connections between workers or organizational units. This also means that the BPR effort should not focus on modeling current organizational responsibilities, but rather on building these based on the workflow.

6) **Minimize re-work and duplication of work**. Candidate tasks for removal can be identified from iterations in the process (e.g. returning information), from tasks which are identical and performed at different parts of the process, from tasks which create the same or similar information (often by different organizational units), and from tasks which are exceptions or correct outcomes of other tasks. The analysis of re-work and duplication of work is performed by following the workflow of a certain item (e.g. an order).

The focus on logistics-related analysis had the following consequences: the method had to develop alternative solutions based on the model data, provide concrete measures, and allow the tracking of changes in performance with the same analysis measures. The modeling part of the method had fewer, more general requirements: the method should resemble other used methods, be simple and apply graphical modeling techniques.

6.3.2 Constructed method

To understand the context of method evaluation and refinement subjects we shall introduce here the modeling techniques and tool support. On the method side, we describe the metamodel and how the method requirements were supported by the method specification. On the tool side, we describe what checking and analysis reports were implemented.

6.3.2.1 Metamodels

Method construction began by choosing modeling techniques which are compatible with the data model of logistics. By compatible we mean that they provide the same concepts and relationships as the logistics data model, or allow them to be derived from the conceptual structure of modeling techniques. The selected techniques included an activity model (Goldkuhl 1992) for describing the workflow, and an organization chart (Harrington 1991) for describing organizational structure. These modeling techniques were modified by adding new types and constraints required by the analyses and by the integration of the techniques. This task was supported by metamodeling and by reusing the metamodel of activity modeling already included in the metaCASE tool. Figure 6-6 represents a metamodel of the techniques and their interactions. The figure uses the GOPRR metamodeling technique (cf. appendix). The constructed method and its relation to the analysis requirements are described in the following.

The activity model describes material or information connections between several tasks. For this purpose, the metamodel includes concepts of ‘task’, ‘material object’, and ‘information’. Each of these object types are characterized with property types required for carrying out model based analyses.

The ‘task’ has an identifier as a property type because similarly named tasks could exist. The identifier, however, could be unique inside the method scope. An ‘operation’ property type was applied to specify the contents of the task and possible instructions for carrying it out. As in data flow diagrams, each task could be decomposed into subtasks (i.e. another model). In Goldkuhl (1992) an activity (called a task here) is characterized by its location, doer and trigger. In the constructed version, location information was not used since it was not needed for carrying out the required analyses. A trigger was related to flows related to a ‘task’, i.e. a ‘condition’ property type. A doer was represented by relating tasks to organizational units. This aspect was modeled as a polymorphism, in which the organization names are referred to by tasks and organizational units. The implementation of the metamodel did not allow dependency so that tasks could not refer to organizational units other than those already specified. A similar structure would also be needed to share resource names among instances of a ‘resource’ and the ‘task’. This deficiency also influenced the modeling process: task structures could be specified before organizational units and resources.

The ‘task’ has property types named a ‘processing time’ and a ‘total time’ to analyze cycle times (requirement 1, cf. Section 6.3.1.2). The timing values were further specified with a unit of measurement (e.g. day, hour, minute) enabling calculation of cycle times. Cost analysis (requirement 2) was supported by attaching a ‘cost’ property type for the ‘task’ as well as for an ‘information flow task’ and for a ‘material flow task’ relationship types.

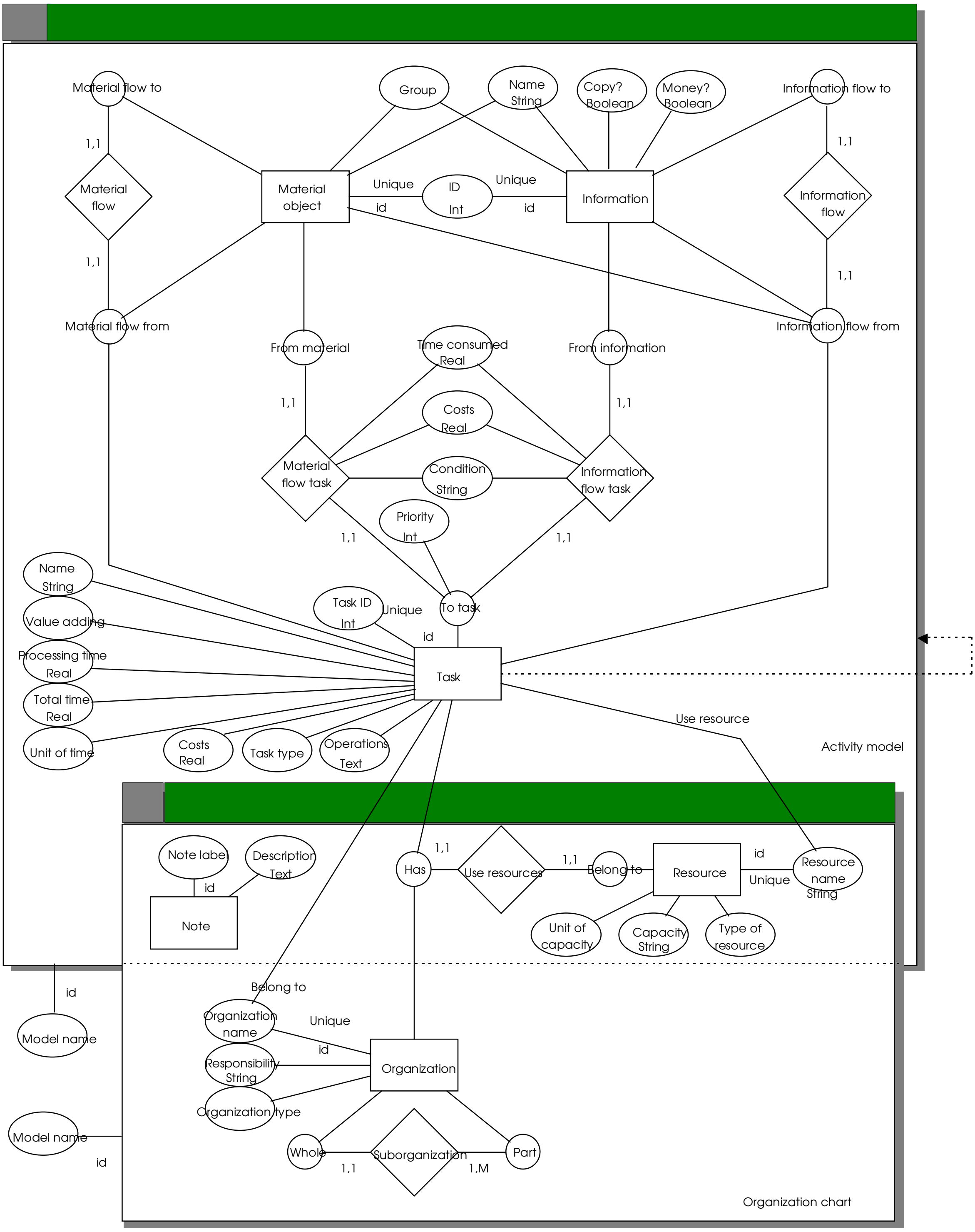


FIGURE 6-6 Metamodel of the *a priori* method.

The ‘task’ object type was further characterized by its type (i.e. approval, check, decision, information update, input, storing, transfer, or mixed). This characterization allowed the simplification of processes (model analysis requirement 4) by highlighting inspection and checking tasks to be removed or combined (e.g. Hammer and Champy 1993, Harrington 1991). Similarly, analysis of value adding (requirement 3) was carried out by characterizing tasks with a 'value adding' property. Value adding included four categories, (business-value-added, real-value-added, no-value-added, mixed) and it was calculated from the estimated value before and after a task (Harrington 1991). This characterization was also used in analyzing cycle times and delays (requirement 1 and 2).

An 'information' and a 'material object' were characterized by a 'group' property type that combined a collection of materials or information. In this way, it was possible to analyze workflows of specific information or material groups and identify complex (requirement 4) or duplicate tasks (requirement 6) (e.g. all tasks related to invoices). Moreover, the 'information' was characterized with property types 'money' and 'copy'. The former specified money and the latter that the specific information was a copy rather than the original information object. These were not required by the analysis reports, but were included into the method to provide compatibility with the logistics data model.

The metamodel included two basic relationship types, material flow and information flow, which were each split into one type for task outputs and another type for task inputs, leading to four relationship types in all.

The 'material flow' and 'information flow' relationship types specified outputs of a task. As in Goldkuhl (1992) a material object can include information, but not vice versa. To model a composite of information or material objects, the 'information' and the 'material object' could participate in both roles of a flow. This allowed us to describe, for example, that a delivery includes a cargo list and shipped goods. Alternatively, an additional modeling technique could be applied to describe composite objects.

The 'information flow task' and 'material flow task' relationship types specified inputs of a task. These flows were characterized with a 'cost' and a 'time consumed' property types to support analysis of costs and delays. A 'priority' property type was added to the 'to task' role type to model urgency handling among several information or material flows. This property was added to the role because the modeling tool did not allow properties of relationships to be represented graphically.

An organization chart specified organizational units and a hierarchy among them. An 'organization' object type was characterized with a 'name', a 'responsibility', and a 'type'. A 'responsibility' was required to identify owners of the tasks and an 'organization type' classified the organizational units into a company, a division, a department, or a working team. Resources were modeled with a 'name', a 'type' (e.g. machine, human, IS), and a 'capacity'. Resources were related by a 'use resource' relationship type to organizations and tasks. Therefore, the 'resource' can have graphical instances in both modeling techniques. In the metamodel this is described by including the type in both graph types (inclusion in GOPRR). Similarly, a 'note' object type is used to add free form comments in both modeling techniques. It must be noted that the 'task' can also refer to the 'resource' by sharing the values of the 'resource name'. This possibility was added because of the desire to simplify activity models (instead of representing all resources and their relation to tasks with a graphical notation).

As a result, the constructed metamodel included information about organizational units and their resources. This was considered to support structuring of the organization according to the process (requirement 5), i.e. connections between tasks could be applied to find organizational units which have cooperation.

It must be emphasized that not all method knowledge could be specified with the metametamodel. Examples of unmodelable method knowledge included mandatory property types (e.g. an identifier of the task), multiplicity over several role types (e.g. unconnected tasks), and different scopes (e.g. resource name unique inside the organizational unit). Moreover, method construction raised the same requirement for a derived data type as in the wholesale case: for example, identifiers of lower level tasks should be derived from identifiers of higher level tasks. The lack of metamodeling power was partly solved with checking reports as discussed in the next section.

6.3.2.2 Tool adaptation

Both modeling techniques were supported by a metaCASE tool, MetaEdit (MetaCase 1994). As a result, models could be developed to carry out abstraction according to the metamodel. The notation of the activity model is represented in Figure 6-8. It illustrates part of a production planning process.

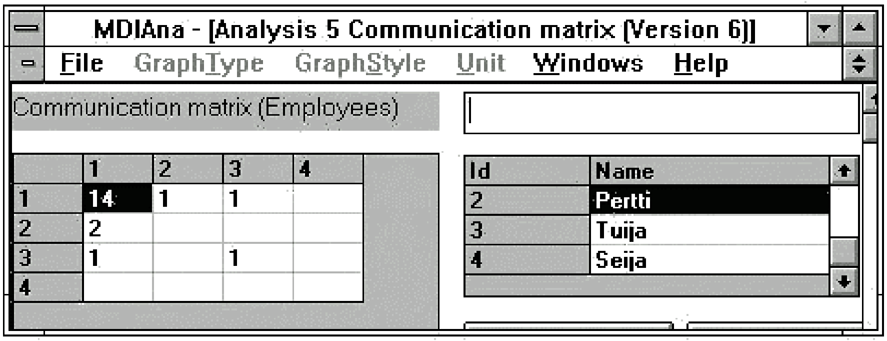
As part of the method-tool companionship, reports for checking, review, and analysis were implemented. These automated reports complemented the manual checking and analysis. The checking reports operated on those aspects of method knowledge which had constraints to be checked passively, or were not possible to capture in the metamodel. The reports covered unconnected object types (i.e. minimum multiplicity one), and undefined properties (i.e. mandatory property types). The documentation and review reports included a dictionary report that listed tasks, items (both information and material), and resources. These reports resembled manual documents followed in activity modeling (cf. Goldkuhl 1989). Moreover, tasks were also reported by their type, possible value adding, and the people carrying them out.

Most emphasis during the tool adaptation was placed on defining reports which carried out the required analyses based on the model data. For the purposes of analysis, the modeling tool included a report which transformed selected model data into the relational database format of an external analysis tool. This tool provided the following model analysis functionality:

- **Elapsed time analysis**, i.e. how much time (effective and waiting time) is used in selected tasks. This analysis addresses delays (requirement 1). Different alternative scenarios could be analyzed using a what-if analysis by changing the property values.
- **Cost versus time analysis,** i.e. an analysis of a chain of tasks based on costs and time consumed in each task. This analysis addresses cost minimization (requirement 2) and is illustrated in Figure 6-5. As with the

elapsed time analysis, property values could be changed to generate alternative scenarios for a workflow.

- **Item workflow analysis**: this report describes time and costs related to a specific item or item group. It allows the identification of errors, re-work, or duplication of effort related to items (i.e. instances of the 'information' or the 'material object'). As with the other analysis reports, cost and time values or tasks could be changed to generate alternative scenarios.
- **Architecture matrix:** this model illustrates the creation or use of items or item groups between organizational units. It allows the analysis of duplicate tasks (analysis requirement 6) which create or update the same data.
- **Communication matrix:** (see Figure 6-7) this illustrates the connections between workers or organizational units. The communication matrix can be derived from the flows of the activity model sending information or material. The communication matrix is generated automatically from the activity model, and it was considered to help in structuring the organization according to the workflow (requirement 5).



Communication matrix (Employees)

	1	2	3	4
1	14	1	1	
2	2			
3	1		1	
4				

Id	Name
2	Pertti
3	Tuija
4	Seija

FIGURE 6-7 Communication matrix.

Each analysis report could be restricted by defining the scope for the models to be included in the analysis. This restriction can be made based on the version of models, selected tasks (i.e. a chain), organizational units, groups of information or material objects, or organizational units/workers.

In addition to these analyses, the tool generated reports which classified tasks according to their value-adding, type, and responsibility. The inspection of value-added properties allowed the analysis of non-value adding processes in relation to costs and cycle times (requirement 3). Hence, it complemented the earlier analysis. Classification of tasks according to their type was considered to support the simplification of processes (requirement 4). It focused on checking, approval, and information updating tasks, which are often candidates for removal. Finally, classification according to the responsible person allowed inspection of the coherence among individual workers' tasks. Each report also included additional process information such as processing time and the description of operations or guidelines.

6.3.3 Characteristics of the cardboard mill

The method was used in developing outbound logistics of a cardboard mill. The mill produces specialized cardboard, mainly for the European packing industry. The study focused on analyzing the current delivery process of the mill. The delivery process was influenced by a cooperation with an export association, and with companies responsible for transportation and harbor operations. In contrast to the wholesale case, the development efforts were limited to one company, i.e. to the mill and its parent company. Because the problem context was logistics centered, the constructed method addressed the characteristics of these problems in the cardboard mill.

6.3.3.1 Characteristics of the problem context

Most of the marketing and sales were made by Finnboard, an export association of Finnish board mills. The export association provided on-line data interchange with their customers and international sales offices. This system provided a virtual and instantaneous means of placing status inquiries and new orders, in contrast with the 12-day norm of the industry (Konsynski 1993). As a result, many mills acting together and leveraging this technology were able to appear to the outside world as one large "virtual" company. The integrated system of Finnpap/Finnboard is described in Konsynski (1993). Because the export association was seen to decrease the competition among mills, its use in the form described has been recently (and after the study was conducted) banned by the European Union. In addition to the sales made by Finnboard, the mill had its own customers among the subsidiaries of the parent company. These sales were made without the assistance of Finnboard, and we call them the mill's "internal sales", in contrast with the sales made by the export association.

The main problems addressed in the ISD process related to variation in the delivery process and poor predictability. The delivery process varied considerably depending on the sales and delivery channel (i.e. internal versus Finnboard). Among internal sales the variety was greater and even more dependent on the customer. These in turn made the process more complex, which required additional resources and increased cost. This problem had already been detected in the mill. Its marketing manager reported that the delivery process had recently been streamlined: all variation and exceptions had been eliminated. However, it was still considered complex and therefore one of the objectives of ISD was to further simplify the delivery process (requirements 4 and 6 used for method construction). This was also of great interest to the consultants, who wanted to apply their method and the developed tools. By modeling the delivery process in detail, which had not been done before, it was expected that the resulting in-depth understanding would further improve the process.

Because of the northern location of the mill and the southern location of its main customers, transportation and logistics placed a central role. The low costs of the cardboard compared to its inventory costs required that the cardboard was always manufactured based on the available transportation capacity. All deliveries were planned on the principle "just-in-time for transportation". Moreover, during the study the demand for cardboard was good and the mill was operating at full capacity. Hence, manufacturing in advance was not possible. This emphasized accurate production planning in the mill. Therefore, ISD focused on improving timely delivery and minimizing logistics costs. Both of these analysis targets were taken into account in the method used(requirements 1 and 2).

It must be noted that not all aspects of the method were considered to be needed. They were, however, included in the method used because these additional analyses had not been used in earlier ISD efforts. In this sense, the experiences of the consultants are counted for the constructed method rather than as *a priori* identified characteristics and problems of the mill to be addressed.

6.3.3.2 ISD experiences and method knowledge

The cardboard mill had limited experiences with ISD methods. In contrast, the consultants responsible for carrying out the effort had relatively high expertise in methods and method selection. This was also indicated from the existence of the data model of logistics and earlier cases from other companies. One of the consultants had studied artificial intelligence systems for contingency-based method selection.

6.3.4 Method use

The ISD project took place in the cardboard mill but also included personnel of the parent company. The project took almost one year, and around twelve people were involved. Most effort was spent on specifying production planning and delivery. During the project these processes were represented by 90 tasks, 140 different information flows, and 30 material flows. An example of a model related to production planning is illustrated in Figure 6-8. The model is based on the activity modeling technique.

Modeling began by defining task structures and validating the activity models. This took most of the time related to method use. Once the task structures had been validated they were refined by adding properties about individual tasks and flows. At the same time the task structures were supplemented with organizational structures and by connecting resources to the tasks. This step was supported by the organizational structure chart.

The models were divided into those dealing with internal sales and those dealing with Finnboard sales. The analysis of the processes was conducted according to the analyses discussed in Section 6.3.2.2. Without going into details, all tool-supported analyses, except those related to cost, were carried out. Cost-related modeling and analyses were not performed because of a lack of time. The project outcomes included three major recommendations to improve production planning and delivery.

First, the delivery process had to be simplified by removing variation in the process. This result came as a surprise. For example, the marketing manager stated: "I thought we had already streamlined our delivery process, but now we have to streamline it some more". The report of the development project summarized that although the variation was not considered remarkable, it doubled the resources needed. The extra complexity was most notable in internal sales. The modes of operation were more homogeneous in Finnboard sales. This could be easily detected by comparing the workflows (e.g. tasks involved and resources needed).

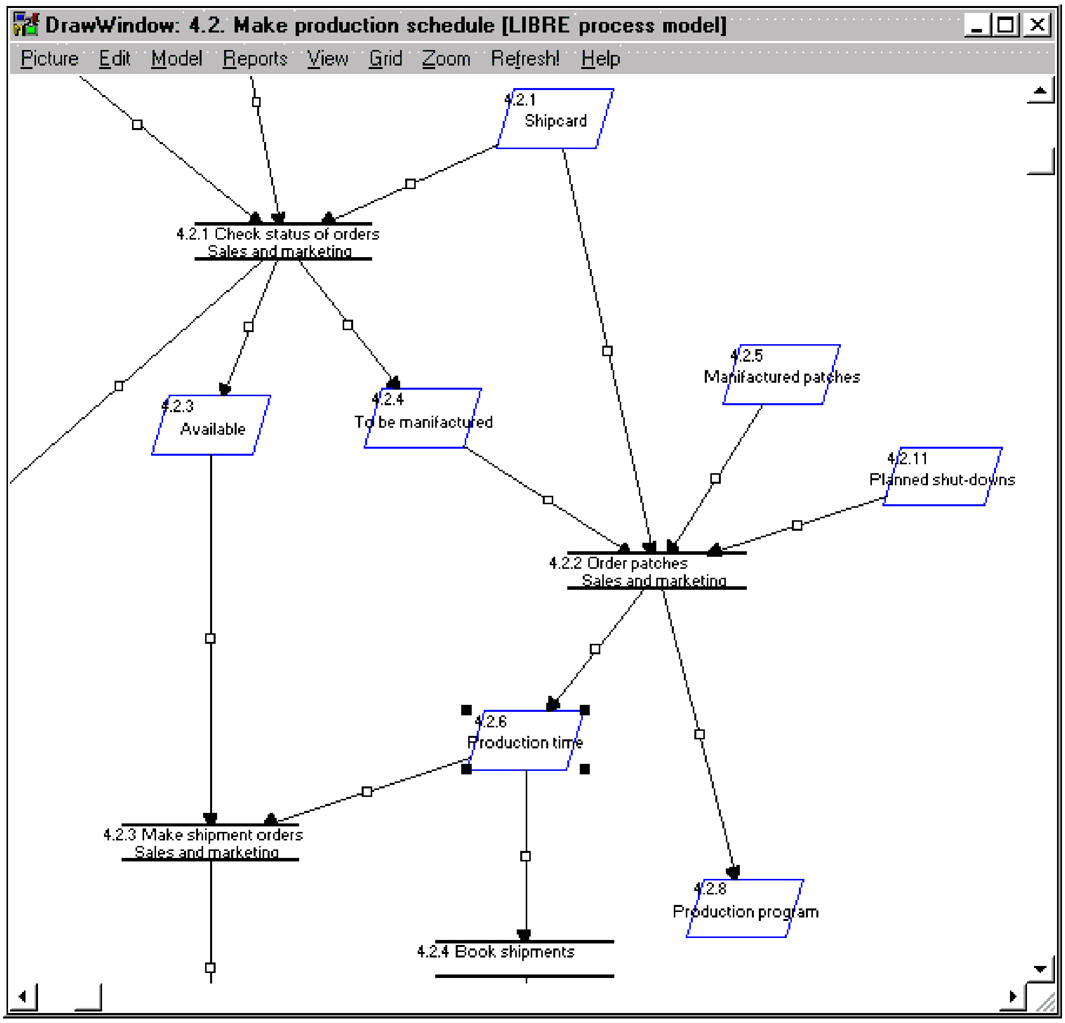

DrawWindow: 4.2. Make production schedule [LIBRE process model]
Picture Edit Model Reports View Grid Zoom Refresh! Help
4.2.1 Shipcard
4.2.1 Check status of orders
Sales and marketing
4.2.3 Available
4.2.4 To be manifactured
4.2.5 Manifactured patches
4.2.11 Planned shut-downs
4.2.2 Order patches
Sales and marketing
4.2.6 Production time
4.2.3 Make shipment orders
Sales and marketing
4.2.8 Production program
4.2.4 Book shipments

FIGURE 6-8 Model of production planning tasks (modified).

Second, better principles for exception management were needed: exceptions took more than half of the total time in delivery management (analyzed through elapsed time, and item workflow). One reason for the relatively high rate was unclear and varying responsibilities. For example, when a change occurred, notification to other parties in the delivery process was haphazard and each party (customer, mill, harbor, transportation company, ship) made and requested several unnecessary confirmations.

Third, internal sales included tasks which duplicated effort. Tasks such as checking order validity and saving order information were not relevant.

Because of the variation, one proposed option was to make the internal sales more similar to that of Finnboard sales. This would necessitate consideration of the current service level in which the mill would take into account the special requirements of each subsidiary company. The resulting better predictability would help production planning.

More detailed analysis of the processes was not possible for two reasons. The variation in the process required that the model-based analyses addressed average situations and excluded frequencies. Furthermore, analysis of cost and value analysis was not conducted.

6.3.5 The *a posteriori* method engineering

In this section we explain how the method was evaluated and refined. We first apply type-instance matching: this part was conducted by the method engineers. Second, we assess the applicability of the method in terms of how well it supported business modeling. Third, we identify the role of the method in problem solving. These latter two evaluations were carried out by the method engineers.

6.3.5.1 Type-instance matching

Type-instance matching inspects how the constructed method has been applied. The comparison is made between the method's intended use (as seen from the metamodels) and actual use (as seen from the models). In the following we describe the results of this evaluation, i.e. the differences between models and metamodels which suggested method refinements (cf. Section 5.3.3 for details).

6.3.5.1.1 Usage of types

1) **Unused types.** Because the analysis reports required detailed data the method was followed strictly. For example, analysis of delays required time related properties to be specified (i.e. have values). Some property types, however, were used infrequently. These included the property types 'money' and 'copy'. Second, property types characterizing flows were not applied. Therefore, analysis of delays did not include time consumption related to flows. Third, costs related to tasks or flows were not modeled. As a result, these property types could be removed from the method.

2) **Division or subtyping** was not required because modeling constructs were not overloaded. The main reasons for this was that the use of the 'group' and 'type' property types allowed for user-defined classifications. The analysis of the free form 'operation' property type, however, indicated new data types. Some tasks included data about error rates and frequencies which could be included as new property types and used in analyses.

3) **Definition of new linkages** between types was suggested in only one situation. 'Responsibility' and 'resource name' had the same values. This suggested polymorphism, to make existing values available between these property types. This would speed up modeling and decrease typing errors. Several task names also included information or material object names. For example, a task called "refine annual budget" delivers as output an "annual

budget" which is an instance of the 'information' object type. This is illustrated in Figure 6-9. However, refinements could not be made here because in some modeling situations the value of an information or a material object was either an input or an output, and the name of a task did not necessarily refer to any information or material object. These naming-based connections, however, could be checked using reports. For example, a report could inform of tasks which did not refer to any of the related information or material objects.

6.3.5.1.2 Usage of constraints

Analysis of constraints was limited to those defined in the metamodel and supported by tools. It must be noted that although the metamodeling language did not support all constraint definitions, the tool checked some of the omitted constraints passively using reports. These reports identified violations of the unique property, mandatory property, and multiplicity constraints. The first two of these in particular were needed to carry out model-based analyses. An identity constraint related to one property type was not enough since there was a need to distinguish versions. This defect was solved by extending all model data with a version number during a conversion of the models. Similarly, checking of unused property types informed about values which were not yet specified but were required by the reports. The model data, however, was often supplemented in the analysis tool because passive checking did not guarantee model completeness. If all property types had been defined as mandatory while making preliminary task structures, entering all task specific data would not have been possible. Alternatively, a weaker constraint technique could be created for modeling preliminary task structures.

A uniqueness constraint was defined only for identifiers. The tool actively ensured the uniqueness of identifiers. The data types defined were found to be adequate, although the predefined values needed some refinement. As storage and transfer were not used while classifying tasks (i.e. the 'task type' property type) they were removed. Value adding was not applied as planned because the classification was too detailed. Instead, a Boolean value (valued-added, no-value-added) was found to be sufficient.

The cardinality constraints in the activity model were not changed. Flows which split or join information or material objects could be created by attaching additional instances to an instance of the 'information' or the 'material' object types.

Constraints on role multiplicity could not be specified adequately in the metamodel. Instead. reports inspected connected and unconnected object types. Model data suggested that in a model scope the 'task' should have a minimum multiplicity constraint (one) for all related role types (i.e. 'material flow from', 'process to', and 'information flow from'). An 'information' and a 'material' should have the same minimum multiplicity, but on the scope of the whole method. Hence, in a single model, an instance of 'material' or 'information' should participate in at least one role, but inside the method in all possible roles, i.e. be both an output and input to a task. This necessitated the use of a multiplicity constraint over several roles.

The metamodeling language did not support checking of cyclic relationships. Therefore, possible cyclic relationships between organizational units (e.g. department consist of itself) could not be checked actively. The tool reports allowed checking only direct cyclic relationships and thus here the method implementation was inadequate. In activity models direct cyclic relationships could be denied because they take part in several object types. For example, the metamodel did not allow direct connections between tasks and thus required information flow or material flow based connections. The method, however, allowed direct cyclic relationships to be created between information and material objects. The initial objective for allowing cyclic relationships was to keep the method simple and use flow relationships to model whole-part structures. Figure 6-9 illustrates the whole-part structure in an activity model in which a budget consists of other information items.

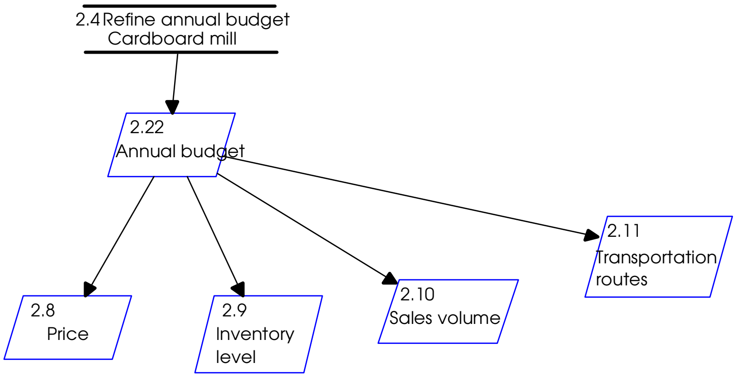


FIGURE 6-9 Modeling whole-part structures in the activity model.

Type multiplicity could not be defined in the metamodel and the tool could only inform about the number of type instances in a model, or in the whole method. Based on the model data, all object types except ‘material object’ and ‘resource’ should have a minimum multiplicity constraint of one in the scope of a model. Because not all activity models included instances of ‘material object’ and ‘resource’ the scope for type multiplicity should be the method. As a consequence, information flows and suborganization relationship types should have instances in all models. The maximum multiplicity constraint was not changed because the models were not considered to be too large (e.g. the largest model had 34 object type instances).

The specification of task hierarchies had several errors because neither the metamodel nor the tool could enforce the complex object constraints. The metamodel only allowed the specification of non-mandatory components, and the reporting capabilities of the tool did not support the checking of complex objects. The required checking included exclusivity of components as well as aggregated relationships. At best, the tool could produce reports which collected constraint-related data for manual checking. This naturally led to error-prone and tedious model checking, decreasing the reliability of analyses.

Polymorphism was applied in two cases in which a task referred to an organizational unit and to the resources it used. Instead of referring to the value of a property type the reference could include the whole object type. In other words, instead of referring to an organization name a task could refer to the whole organizational unit. The advantage was the possibility to inspect specifications (i.e. properties) of organizational units and resources during activity modeling. Hence, the polymorphism unit would be the whole object instead of a single property. Finally, instances of 'responsibility' and 'resource name' had the same values. This suggested a polymorphism structure: sharing the same instance value between these property types.

6.3.5.2 Modeling capabilities

The method was constructed to support logistic analyses. In the following the modeling capabilities are analyzed using the evaluation mechanisms. The suggested refinements are summarized in Section 6.3.6 as changes in the metamodel.

6.3.5.2.1 Abstraction support

The use of the method raised new requirements for describing the logistic processes of the mill. First, there was a suggestion that the life-cycle of important information and material objects would be modeled in separate models. By the life-cycle we mean all the states of an information or material object and transitions between these states. Examples of the states of a material object representing an order are received, checked, accepted, delivered, invoiced, etc. The activity model primarily described sequences and connections between tasks, but the life-cycle of each item was scattered over several models. Only analysis reports illustrated the life-cycle concept through tasks which related to a certain item, or item group. Second, the consultants suggested a new property type which could be defined in modeling (i.e. typed during modeling). In the mill case, tasks in particular were considered to need extra information about error rates or broken items. The addition of a new property type instead of free-form description data in the current 'operations' property type was emphasized because the analysis tool required structured descriptions. Third, it was suggested that information and material objects could include information about volume data and a property for free-form description.

Major difficulties in modeling were related to the variation in the business processes. Two kinds of variation were detected. First, the delivery process differed greatly depending on the type of customer, tasks involved, and task specific properties. This could not be solved by modifying the modeling technique but rather by introducing generalizations (e.g. typical, problematic, etc.). Hence, the developers needed to introduce different versions (e.g. internal sales versus Finnboard sales) and find representative cases of the processes in each version. A second kind of variation related to frequency. The method expected that task characteristics remained stable and volatility could not be modeled. For example, an exception in the process could increase workload temporarily and cause long-term delays. The proposed solution for this deficiency in the method was a 'frequency' property type attach to the 'task'.

Because modeling work was carried out by two people, and others mostly reviewed the models, no major modeling differences between participants were detected. Moreover, the consultant acted both as a method engineer and an IS developer, and could explain and teach the method to other stakeholders.

6.3.5.2.2 Checking support

During model maintenance most efforts focused on the task hierarchy and on the property type 'task'. This needed to be consistent within the hierarchy. Because the metamodel did not adequately specify these constraints (i.e. a complex object) the resulting models had several inconsistencies. For example, it was required that the modelers updated the aggregated relationships in a task hierarchy and that tasks were exclusive (cf. constraints for complex objects in Section 4.4.2.2). The variation in the process emphasized maintainability problems because a change in one task required changes in other models.

The task hierarchy highlighted property-based dependencies between tasks. For example, the processing time of a task should not be less than the processing time of its subtasks, or a task should not be defined as value-adding if none of its subtasks were value-adding. This demanded creation of a new data type which allowed derivation rules to be defined and related to a selected set of property types. Similarly, the numbering of tasks based on a task hierarchy required a lot of manual work: it was the modeler's responsibility to update identifiers when the task hierarchy changed. To speed up the modeling process it was suggested that the tool would use internal identifiers (and output these to the analysis tool). Similarly, to speed up modeling work, timing-related property types needed to include measuring units. The initial metamodel included a pair of property types, i.e. one for the value and one for the related unit. Both these requirements were surprising because they were not found during the initial method analysis (Chapter 4).

6.3.5.3 Problem solving capabilities

The method was constructed to automate analysis tasks. Hence, the form conversion and review capabilities were emphasized during the evaluation of the method. Surprisingly, most benefits were outcomes of modeling rather than of analyses. Although most tool-supported analyses were carried out, their contribution was disappointing. The automated analyses found few improvements and their results were considered dubious because of different interpretations. Instead, most benefits of analyses occurred from the identification of those aspects of processes which required further analysis (e.g. the most time consuming tasks, or slack resources). It must be noted that not all analyses were relevant in the mill case, but all were included since the consultants wanted to test the whole method.

6.3.5.3.1 Form conversion support

Form conversion denotes a tool's capability to analyze models and generate candidate designs. In the CASE environment the conversion functionality was provided through analysis reports. Accordingly, we evaluate the tool's contributions to analysis of the model data and identification of design solutions.

1) **Delays** were analyzed by inspecting the elapsed time in tasks. The delay analysis revealed that exception management is time-consuming, and that internal sales are over 20% more time-consuming than Finnboard sales. Although the analysis allowed the comparison of effective time and waiting time, candidate designs to optimize processing time were not sought. In other words, no what-if analysis was carried out. Reasons for the limited use of analyses included difficulties in choosing candidate times and volatility in the object system: in many tasks time related measures were considered inaccurate because of wide deviations in the processing time, and because flow times were not specified. As a result, the analyses were considered unreliable. The solution suggested was to add frequency information to the 'task'. Although this information was not supposed to be modeled during activity modeling, but rather during analysis, it was added to the modeling technique, to help gather frequency data while modeling time properties.

2) **Cost analysis** was not carried out because gathering costs via task structures was difficult, and the project lacked the necessary resources. Hence, all cost-based modeling constructs, including the cost-cycle time chart, were not applied. Because of these difficulties the consultants examined accounting-based approaches which could be used with current modeling methods. In ABC-based accounting (Morrow 1992) the resources would have the cost data and cost drivers. Moreover, tasks would then be linked to resources (as in our models) and to task specific cost structures. Hence, instead of relying on task costs, the cost analysis would be based on resources costs. ABC-based accounting would require linkages to external tools, such as a spreadsheet application.

3) **Value adding** was not related directly to the analyses because its use was not possible because of the limited cost analyses. Instead, reports of value adding capability were applied to identify removable tasks, i.e. non-value-adding tasks. During modeling, however, the value-added features had been understood so strictly that less than 10% of tasks were specified to add value. Moreover, internal sales had more non-value adding tasks than Finnboard sales, indicating that the mill should perform the minimum possible outbound logistics by itself and leave the rest to the export association. The value-adding was considered to be improved by relating it to the cost-cycle time chart: cost and delay analysis would then support analysis of value-adding activities.

4) **Simplification of processes** was performed by streamlining the delivery process. To this end the effort focused on exception management and the redesign of sales processes. Most of the simplification possibilities were detected during the modeling step, but the automated analysis allowed comparison of item-based workflows between different sales channels (i.e.

internal sales vs. Finnboard sales, and internal sales to different types of customers based on delivery terms). Because cost data was not available this analysis relied on elapsed time only and had the same difficulties with inaccurate results.

5) **Organize around processes.** At the level of individual workers the communication matrix did not find strong bindings between workers in different organizational units. Hence, the organizational structure seemed to follow the task structure already. At the level of organizational units the communication matrix was more useful: it allowed the inspection of differences between internal sales and Finnboard sales. In the former case, the mill had a lot of connections with other parties, e.g. haulage, harbor, and customer, whereas in the latter case, the export association managed most of the negotiations with other parties. However, because the project focused on the mill, no suggestions were made about how to organize the responsibilities in the network.

6) **Minimize re-work and duplication of work.** Candidate tasks to be removed were sought using the architecture matrix and the item workflow. The architecture matrix showed tasks which created or updated the same data and thus pointed out tasks to be removed or combined. Item workflows described iterations in the process and thus clarified the repetition of work. During the analysis the architecture matrix revealed possibilities for re-designing processes based on access rights (i.e. create, use). Item workflows did not reveal why work needed to be repeated.

To summarize, the architecture matrix was the only analysis which directly enabled the generation of designs. The candidate designs could be made by changing the data access rights for tasks. Other analysis reports measured the current situation, but did not include any built-in possibilities to suggest candidate designs. These reports were supported with what-if analyses, i.e. by changing the values in the analysis tool and running the analysis again.

6.3.5.3.2 Review support

Most method use was concerned with validating models with the domain experts. Hence, the review support was of great importance. In a CASE tool, review support implies the production of documents for different stakeholders to validate the models.

Validation was performed in two phases: first related to the general task structure and organization structure, and second in relation to the details of the models (i.e. to properties used in analyses).

In the first phase, the review was carried out using graphical models. The main difficulties while reviewing the models concerned dividing flows and specifying volumes. Initially, the method included only a ‘condition’ property type for describing dividing flows. The domain experts suggested that dividing flows should be specified in more detail, e.g. by describing logical operators or a ratio. An example of such a situation is shown in Figure 6-8 in which information about production time (ID 4.2.6) is used in two tasks. The use of logical operators (and/or), as proposed by Goldkuhl (1989), would allow the modeling of situations where the information object is used in both tasks or in one of the tasks. Moreover, users suggested a percent-based specification showing, for example, that in 40% of the cases the information was used by only one of the tasks. Moreover, the condition values were not shown in graphical models and thus they suggested a notational change. The users also suggested that volume information should be shown graphically. This addition required a new property type for the 'information' and 'material object' types, with a new notational element (i.e. a text field close to the rectangular symbol of the 'information' and 'material' object types).

Although these additions were simple, their influence on the model analyses (e.g. item workflow) was unclear. It was suggested that each analysis case be handled separately either by modeling all conditions separately, or by omitting the conditions during the transfer of data to the analysis tool. In the latter case, the conditions should be entered while making a what-if analysis.

In the second phase, the review focused on validating the property values. For this task we developed a report tool for documenting the tasks of each individual, who could then review the information. These documentation reports were also included into the final report. In addition to personal reviews, the method users proposed state modeling to collect and integrate workers' views into state models. This was believed to help inspect the dynamic behavior of order management independently of workers' tasks. It could therefore offer a behavior-oriented view to help validate task structures (i.e. the process oriented view).

6.3.6 Method experiences and refinements

Method evaluation provided a good amount of experiences of the method and suggested several method modifications. Method development focused mainly on analysis needs and emphasized modeling constructs which were needed by the analyses.

The method refinements suggested were a direct outcome of the method evaluation. The evaluation clarified that the most important changes related to modeling life-cycles of information or material objects, managing variation in time, and describing volumes. These are reflected in the metamodel illustrated in Figure 6-10. It should be noted that not all metamodel constraints, such as scopes, are captured in the metamodel because neither the metamodeling language nor the tool supported them adequately.

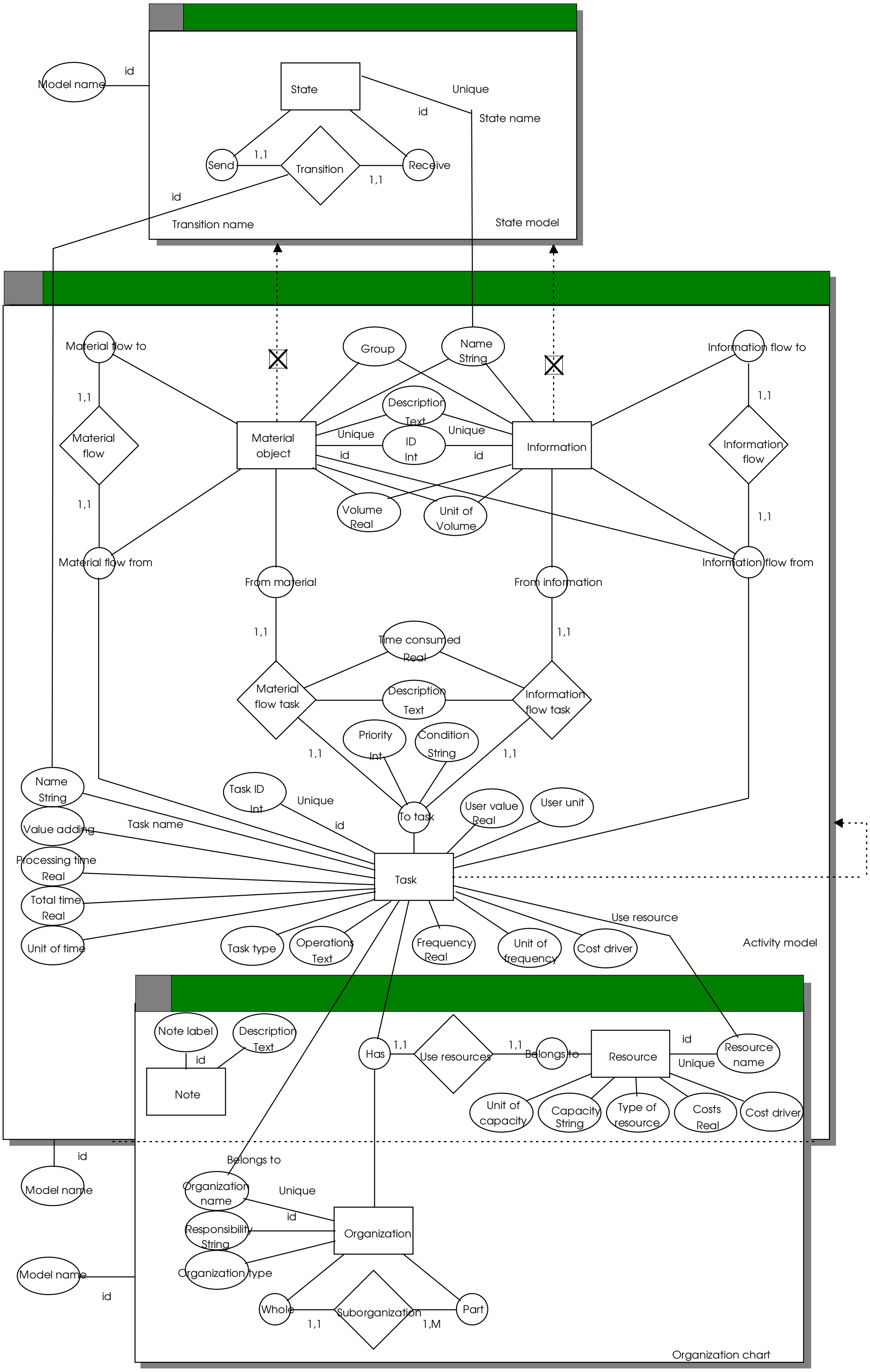


FIGURE 6-10 Metamodel of the refined method.

A simplified state model was considered adequate to model the life-cycle of information and material objects. The simplification meant that events and conditions typical in state models (cf. metamodels in Section 4.3) were excluded. Instead, the state model was integrated to the activity model through explosion and polymorphism. Explosion meant that each 'information' and 'material object' instance was linked to a state model. Although the cardinality of the explosion could not be specified in the metamodel the explosion should be mandatory for 'information' and 'material object' instances and "floating" state models should not be possible (i.e. the cardinality of the explosion should be one-to-one for the source and one-to-many for the target state model). Checking of cardinality constraints is passive because we wanted to leave unspecified whether activity models or state models should be created first. This metamodeling choice also influenced the dependency of polymorphism structures.

Polymorphism was defined between two techniques: values of the 'name' property type characterizing the 'information' and 'material object' types were shared with 'state name' values. Similarly, 'task name' values were shared with 'transition name' values. While using the method this method specification would allow the modeler to refer to existing property values instead of entering the same values twice or more. As a result, modeling becomes faster and less error-prone, and model changes are reflected automatically in the tool. Another possibility would be to refer to the whole information or material object instead of a single property. This possibility was not used because the tool did not support it. The polymorphism allows inspection and checking of models. For example, each transition should be represented for a task in an activity model, and all states should be required as information or material objects in some activity model. It must be noted that the polymorphism could not be defined to be dependent because the explosion cardinality did not expect that either of the techniques should be used first. Hence, the polymorphism was checked passively at the user's request.

Activity modeling was simplified by removing some unused property types: 'money', 'copy' and 'costs'. To enable calculation of delays and costs, the 'information' and 'material objects' were supposed to be characterized with volume information. The 'task' object type was refined by relating property types for specifying frequency and user-defined aspects. Although the ISD effort indicated that error rates could be specified with their own property type, it was considered to be specific to the cardboard mill only. Hence, user-defined values were expected to be more flexible in future. Moreover, to specify more detailed descriptions about activity models a new property type 'description' was attached to information and material object types and flows.

The modeling experiences showed that costs are difficult to collect in a similar manner to other workflow characteristics. Therefore, the cost analysis was changed totally: instead of adding cost information to individual tasks and items (i.e. material or information) they were related to resources. The cost structures were calculated through Activity Based Counting (Morrow 1992). Because the modeling tools used were not well-suited to accounting, the tool would export cost data into a spreadsheet. For this purpose, the 'type of resource' was supposed to refer to the kind of cost, and the 'capacity' to a cost driver. Information about the resource use of each task could already be modeled with the method.

To support model review we considered it necessary to show more design information graphically. Because the tool could not show properties related to relationship types, the 'condition' was moved to the 'to task' role type.

In addition, the evaluation suggested changes to the tool. First, the tool should allow graphical selection of a task chain and transfer it into the analysis tool. Second, the predefined reports for documenting and checking were suggested to be improved, enabling the use of passive constraints (e.g. cardinality of explosion). Alternatively it was suggested to automate passive checking while transferring the models into the analysis tool. This option was abandoned because it would slow the transfer of models into the analysis tool. Third, the numbering of identifiers should be automated.

The method evaluation also allowed improvements in activity modeling, method related contingencies, and automated analyses. Activity modeling was considered to be easy to use, its models were understandable, and communication with end-users improved. As already mentioned, the main difficulties were related to maintaining task hierarchies and identifying codes when models changed.

Second, because *a priori* method selection did not follow any contingency selection framework, the relevance of method selection criteria could not be measured. Instead, during method construction the compatibility with earlier experiences with the logistics data model were emphasized. After all refinements it was interesting to notice that the refinements included no major changes which conflicted with the underlying data model. Instead, the original data model was extended with some behavior-related concepts.

Third, the automated analyses were disappointing when compared with the original objectives. The analysis reports did not originally allow the generation of candidate solutions, and the analysis results often looked doubtful. Maybe the case was too complex for the required analysis, and the given measuring properties too inaccurate because of the variation in the process studied. It was therefore suggested that the analyses would be tried out in smaller, more bounded business systems. Accordingly, principles should be sought for choosing between alternative workflow scenarios (e.g. product based, customer based, worst case, etc.).

6.4 Lessons learned

In this chapter we described two method engineering cases. The cases were carried out as action research studies. The action research method offers possibilities for learning in three areas (Checkland 1991): the area of an application, the methodology applied, and the particular ideas promoted. In our studies the application deals with developing local methods for specific ISD environments. In Checkland's (1991) terminology, the methodology denotes the general principles applied in inspecting an application area. Hence, for us this means method engineering along with related methods and tools such as metamodeling and metaCASE tools. The ideas promoted are the metamodeling language constructs and the *a posteriori* view of ME.

The following subsections discuss the evaluation part of action research. First, we describe in Section 6.4.1 general findings about local method development. Since our action research studies were not focused on all aspects of local method development (like costs or management principles) we shall only inspect the development of tool-supported methods. Second, we shall inspect differences between the proposed ideal ME principles and the cases (Section 6.4.2). Finally, we describe in Section 6.4.3 findings related to the use of incremental ME principles. Because we also participated in the ISD process, some findings could be presented about how to develop inter-organizational ISs in wholesale and improve the delivery systems of a cardboard mill. Our studies, however, were designed to operate at the ME level, not at the ISD level (cf. Section 3.3.1). Some general solutions for ISD, however, were already discussed as part of the cases.

6.4.1 Local method development

The studies show that organizations develop their own methods. In the wholesale case, the ME effort was targeted to support a specific BPR project. The mill case included some features characteristic of developing a more universally applicable method: the consultants wanted to develop a method which would be independent of object systems, and appropriate for analyzing a variety of workflows related to logistic ISs. Because the method developed was intended to be used in all projects the consulting company engaged in, the second case followed an organization based ME.

Local methods were developed because of the limitations found in the existing methods used, inadequate tool support, and the lack of knowledge about other methods. In the wholesale case, the need to distinguish between organizations involved in the delivery chain and to characterize inter-organizational processes led to the establishment of the ME project. In the mill case, the need to automate analyses of workflows necessitated the development of specific tool functionality and as a by-product allowed the development of a propriety modeling method. Hence, modeling capabilities were addressed primarily in the wholesale case, and automated analysis capabilities (i.e. tool support) were emphasized in the mill case. It should be noted that in both cases the existence of a specific ISD project clearly influenced how the method development and evaluation effort was carried out. For example, if the wholesale company had not been in the middle of a major business process re-engineering effort, a local method would not have been developed.

The methods developed in the two cases had similarities: they addressed the modeling of business processes, described material objects or flows, defined organizational units, and characterized these modeling elements with some similar properties (e.g. volume, capacity). The main differences between the methods was the granularity of analysis. The wholesale company tried to understand its order entry and purchasing processes better in relation to its business environment, to support the move to a two level hierarchy. The mill case focused on individual workers and aimed to analyze the structures of tasks. More detailed differences between the methods can be identified by comparing the metamodels.

Both organizations found the developed method useful: interviews showed that the method developed was considered to work better than those used earlier. For example, in the mill case the consultants estimated that with the methods and tools used earlier, they could perform only half of the modeling and analysis tasks supported by the engineered method and tool. In general, models based on the developed method were considered to be easier to read and understand, to support communication better, and to allowed the combination and analysis of views of multiple stakeholders, or even of multiple organizations. Moreover, in the mill the connection of the models to the quality system was important.

Satisfaction with the developed method does not mean that no problems existed. In fact, the method refinements clearly showed that the methods had to be improved. The main difficulties included different interpretations of conceptual structures and analysis. Most of the effort in method development related to agreeing and confirming an understanding about the method and tool functionality. In the mill case, the consultants also considered the objectives for the engineered method to be too ambitious. Meeting these objectives in turn required significant resources and time.

Satisfaction with the tool was surprisingly different among user types. Method engineers considered the metaCASE functionality limited (i.e. all metamodeling constraints could not be supported). As a result, a lot of time was consumed while trying to find roundabout ways to build method-tool companionship. People in the organizations using the tool, however, were highly satisfied with the tool, although they requested several new features which did not directly address the method-tool companionship. These included importing available process maps into the tool (e.g. from the documentation of a quality system), providing links to external documentation tools, and improving method-independent reports.

6.4.2 Method engineering

The method engineering process was quite similar in both cases and all ME tasks were carried out. One reason for the similarities was the tool adaptation which required detailed method specifications. The similarities in the ME process were also a consequence of planning the action research, since the ME process needed to include the *a posteriori* ME tasks postulated.

During ME, *a priori* method selection (cf. Section 3.2.1) was made among relatively few methods, and included methods which were known or had been used earlier. However, in the second case selection was supported by a relatively large review of methods (cf. Tolvanen and Lyytinen 1994) and tools (cf. Lindström and Raitio 1992). During ME neither of the cases applied contingency frameworks, because such frameworks were not available. Those reviewed were considered to be too broad since they did not help to distinguish between modeling techniques based on identified characteristics. Similar observations were also made in Section 3.2.3 while analyzing other ME cases.

The main differences in the ME processes were the metamodeling languages used, the emphasis on different ME tasks, and the types of stakeholders involved. First, in the wholesale case the only metamodeling language applied was that used by the metaCASE tool, whereas in the mill case the early phases of method development had been carried out with another metamodeling language (i.e. the ER-based logistics data model). However, here a metamodel of the activity model method already included in the metaCASE tool was taken as a starting point during tool adaptation (i.e. reuse of an existing metamodel).

Second, the use of resources and duration of tasks differed greatly. In the mill case, the method engineering took more time and resources. One obvious reason was the objective of the ME project to develop a general purpose method. In other words, the method was expected to be applicable for solving logistic problems in other areas too. Moreover, the mill case stressed tool adaptation because it required implementation of the analysis functionality. About 1/3 of the resources were spent on tool adaptation, whereas in the wholesale case the tool adaptation was the least resource-consuming task.

Third, the participation of method users differed in the cases: the IT personnel of the wholesaler participated actively in the method construction and evaluation, whereas the personnel of the mill did not directly participate in the ME project. One reason for the difference was the two-party setting between the consulting company and the mill.

6.4.3 Principles of incremental method engineering

As the objectives of action research indicated, our interest was to demonstrate the viability of incremental ME principles. First we analyze whether the situational methods were possible to describe with meta-data models and the proposed metamodeling constructs. Second, we analyze whether the *a posteriori* view was appropriate as a mechanism of method evaluation and refinement.

6.4.3.1 Modeling situational methods

In both cases, methods were modeled with metamodeling languages embedded in a metaCASE tool. This was needed to provide modeling tools for the ISD projects. In the mill case the metamodeling also included ER-based modeling, but only for outlining the concepts and relationships used in the method. Moreover, the ER model was used for metamodeling before the selection of the metaCASE tool.

Not all method knowledge, however, could be fully supported, because of the limitations in the metamodeling language. These limitations concerned modeling property types with unique, mandatory, and data type constraints. Other limitations related to defining that an object must participate in at least one connection, must have a specific number of instances, and can not participate in connections which are cyclic. Most limitations were related to integrating modeling techniques: constraints related to complex objects (i.e. exclusive component objects and aggregated relationships) and polymorphism structures (i.e. sharing of several property types at a time, and dependency on other instances) could not be defined.

The proposed metamodeling constructs allowed methods to be specified more completely than was adapted into the tools. In this sense, they can be considered sufficient for engineering the methods in the cases. Only one limitation in capturing method knowledge was found. In both cases a derived property value had to be specified. An example of the derived data type is calculating the processing time of a task from its subprocesses. Because this type of dynamic calculation is difficult to capture into a static data model, this requirement suggested the use of metamodeling languages other than those based on semantic data models (see also Section 4.5.3).

It must be emphasized that not all proposed metamodeling constructs were fully applied since not all possible rules of methods were needed in the cases. For example, an explosion structure was used (refined method in Figure 6-4) but cardinality of explosions were not used. Similarly not all scopes and checking modes related to the metamodeling constructs were applied. One reason for the limited use of scopes may be because of the relatively simple structure of the methods in the cases, i.e. the fact that they included only a few modeling techniques. Thus, the viability of every metamodeling construct was not demonstrated via the metamodels. During ME, however, awareness of method knowledge which is not specified into metamodels is valuable because it allows one to understand alternative method configurations and metamodeling choices.

The limited use of metamodeling constructs also suggests that specific metamodeling constructs are needed while modeling specific methods. For example, the requirement for modeling derived data types was not detected during the modeling of text-book methods (Chapter 4) which were mostly IS analysis and design methods. In contrast, our case studies required business modeling methods which often incorporated numerical values.

Although our aim was not to evaluate the metaCASE tool used, their limitations and capabilities influenced the metamodels. First, because the metaCASE tool did not support matrix-based representations tool adaptations were not made for all parts of the method. Second, checking reports allowed us to overcome limitations of the metamodeling languages through passive checking. Passively checked metamodeling constructs related to mandatory properties, unconnected object types, multiplicity of roles, and multiplicity of types.

6.4.3.2 Refining situational methods

The case studies followed the principles of incremental ME: in addition to constructing methods we also evaluated their applicability. This evaluation led to several refinements to the methods initially constructed. This finding supports our re-evaluation of method use (cf. Section 2.5.4): it seems to be difficult, if not impossible, to construct a situation-dependent method by following solely *a priori* tasks of ME.

The requirements for method refinements were obtained by following the *a posteriori* steps of ME: collecting experience, analyzing method use, and improving methods. The reporting of the ME cases followed the three evaluation mechanisms of incremental ME: 1) type-instance matching, 2) modeling capabilities, and 3) problem solving capabilities.

Type-instance matching suggested a large number of method refinements. These dealt with removing less frequently used property types, dividing object types and classifying relationship types, and creating linkages between types of the method which can refer to the same model data (instance values). The analysis of constraints leads to modifying methods in more detail. The changes introduced influenced all constraints except those dealing with uniqueness, cardinality, and inclusion. In other words, these constraints were defined adequately in the *a priori* constructed methods. It must, however, be noted that not all constraints could be fully evaluated since they were not allowed in metamodels. Although this part of the evaluation was carried out by method engineers, all the refinements were validated with the method users.

The modeling capability evaluation seeks to abstract relevant aspects of object systems and keep the resulting models consistent. The cases contained several situations in which methods were considered inadequate to model the object systems or parts of them. Individual differences were not analyzed, because only a few users actually modeled with the method: other method stakeholders were involved in reviewing the models and conducting analyses. This part of the evaluation proposed new types and related constraints, and even new modeling techniques. These extensions were partly the same as those found during type-instance matching.

The checking analysis revealed possibilities to improve the consistency of models via an integrated and more strictly defined metamodel. Better support for consistency was achieved by defining complex objects and polymorphism structures. These refinements decreased the need for manual maintenance and improved the consistency of models. The evaluation also revealed the need for derived property type values, and the use of a single tool. These changes, however, could not be carried out because such method specifications could not be captured in the metamodels.

The evaluation principles focused on analyzing the role of methods in solving the business problems. This part of the evaluation was divided into finding candidate solutions through form conversion, and supporting model validation by producing documents. During form conversion, new features relevant for the generation and analysis of design solutions were suggested, together with new analysis algorithms. In the second case, larger extensions to the analysis needs were also made, i.e. the addition of Activity Based Counting. Support for exporting design data and analysis results into external tools were considered adequate.

The integration and validation of models from different parts of the object systems required a new modeling technique. Other refinements suggested dealt with notations and reports: notations were modified by graphically showing information about model data instead of using model related textual reports. The contents and layout of reports were also changed.

The required method refinements were made into the corresponding method specifications (i.e. metamodels), or were achieved using the tool (i.e. checking reports). Not all the required changes, however, could be made because either the metamodeling language or the tool did not support them. In this sense, not all suggested method improvements could be taken into use. It should be noted that none of the required changes could be predicted. Moreover, because the refinements were found to improve the method, the *a posteriori* approach to ME is clearly viable.

Finally, while evaluating the results of action research it must be noted that the use of incremental ME was limited to one cycle (less than a year). For example, contingencies in the ISD object system did not change during that time. Therefore, new method refinements would be needed if the evaluation were to be carried out again. At the same time, the contribution of individual evaluation mechanisms would mostly likely be different.

7 CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

This thesis suggests principles that support local method development. The research is motivated by earlier findings in method research: the majority of ISD organizations which use methods develop their own variants instead of solely applying external methods as specified. Organizations, however, lack proven guidelines to develop or adapt their methods. Though method modeling languages and customizable modeling tools, such as metaCASE tools, have become available, there is a paucity of (meta)methods for method development. Our aim was to systematize a set of local method development principles that would improve the method engineering process and increase metamodeling support. Accordingly, this thesis concentrates on principles for engineering modeling techniques for ISD tools.

The ME principles developed can help organizations develop local methods and manage changes in modeling techniques. The principles are founded on extended metamodeling capabilities and on an *a posteriori* view of ME. In this view, experiences of method use are utilized to improve methods. The metamodeling capabilities will be summarized in Section 7.1, and the *a posteriori* mechanisms in Section 7.2. Finally, in Section 7.3 we propose directions for future research in this area, and complementary research topics, such as the study of implications for building tools and languages for ME.

7.1 Modeling languages for method engineering

The first contribution of the thesis deals with extended support for metamodeling. Before proposing metamodeling constructs we surveyed method knowledge and described method-tool companionship. Survey into method knowledge was needed to understand various types of method knowledge and explain the focus of method engineering taken in the thesis:

modeling conceptual structures behind modeling techniques. The concept of method-tool companionship was applied to describe how tools can support method use through abstraction, checking, form conversion and review. Although we view metamodeling in the context of modeling tools our interest is on metamodeling. The reason why we have studied metamodels in tools is that metamodels which do not influence ISD have a limited impact. Conversely, modeling of ISs is more beneficial if the models can be applied for implementing ISs. Among the categorical choices in metamodeling languages we have focused on semantic data models. They are most widely applied in large metamodeling efforts and as metametamodels of metaCASE tools and repositories.

Extended support for metamodeling was developed by proposing new constructs for metamodeling languages. We found a limited set of constructs which can help method engineers to specify relevant aspects of modeling techniques. It is clear that unlimited support for all types of methods is impossible. Unlike with many other metamodeling languages we did not try to build a language which is as powerful as possible. Rather we sought to balance power and ease of use so that it can be used with minimal effort but the language is at the same time powerful enough to model methods. In fact, ME has the same objective: to develop methods which help users to perceive and specify relevant aspects of object systems rather than model "everything".

The examination of metamodeling constructs was carried out by conducting a content analysis of a chosen sample of the method literature. This type of an inductive approach has not been generally used in such an extent to develop metamodeling languages. In all, we analyzed 17 methods consisting of 72 modeling techniques of which 3 methods and 19 modeling techniques are reported in detail in this thesis. The methods covered different ISD phases including business modeling, requirements engineering, system analysis and system design methods. The literature described the concepts, rules, notations and possible requirements in building tool support for methods. The data was classified into distinct types allowing us to simplify and systematize conceptual structures of methods. In our case, the classification of method knowledge was based on the metamodeling languages supported by metaCASE tools. The metamodeling approach allowed us to capture and understand method knowledge and the metaCASE tools allowed building, checking, and querying on metamodels, as well as to make tool adaptations. Hence, as a by-product of the analysis, we developed a modeling tool for each method.

The content analysis allowed us to categorize method knowledge and identify constructs which are needed to model methods more completely. The metamodeling constructs were divided into those necessary for modeling single techniques and those needed to integrate multiple techniques. Furthermore, we identified varying scopes in method name spaces and crafted checking modes for each construct.

The proposed metamodeling constructs lead us to assess available metamodeling languages. This assessment revealed that current methodical support for method modeling is modest. While in recent years some progress has been made in outlining conceptual and theoretical principles for metamodeling and metamodel based tools (for a survey see Kelly 1997, Tolvanen et al. 1996) we found that available metamodeling languages do not provide adequate ME support. Having evolved from general purpose data models or data modeling languages, existing ME languages are capable of expressing specific semantic constraints imposed by the business data modeling domain. Many of the languages promoted for ME are applied on ISD as well and the required metamodeling constructs are quite different than those needed in IS modeling. As a result of the assessment, we identified structures of method knowledge which could not be represented adequately with the studied metamodeling languages. For example, metamodeling languages offered limited constructs for modeling interconnected techniques and the identified scopes of method knowledge were inadequate. In this sense, the content analysis of method literature contributed to the current understanding of detailed method knowledge. It revealed several aspects of method knowledge which have not been identified by the developers of metamodeling languages. The evaluation also lead us to examine the limitations of semantic data models as a foundation for metamodeling. With respect to modeling tools, the semantic data models are powerful in describing static aspects, but poor in describing dynamic rules applied in modeling techniques.

The results of the evaluation can be applied by researchers and practitioners alike: developers of metamodeling languages can use them for analyzing and extending their languages; tool vendors can apply them for extending their metametamodel based tools; and method engineers can use them to identify method knowledge which is neglected. Extended metamodeling constructs allow organizations to better specify, understand, analyze and refine methods.

7.2 Experience based method refinements

The second contribution of the thesis lies in an improved understanding of the method engineering process. These improvements were obtained by extending method engineering to cover method evaluation and refinement. The proposed incremental approach does not cover only the selection and construction of methods for a given situation (*a priori*), but also the evaluation of the applicability of methods and method improvements (*a posteriori*). Earlier research into the ME process has so far focused on constructing methods in an "one -shot" manner, as described in Sections 3.2 and 5.2. We however regard ME as an incremental process. We believe that the method is rarely defined at once, and written down as a complete metamodel. The process of arriving at a method is fragmented, evolutionary and largely intuitive. Though we can identify some refined pieces of ISD methods, the reality of ME tends to be meandering towards a solution, as situations change and stakeholders learn more. This means that any ME approach focusing only on the initial method construction is incomplete and ME principles need to be extended to cover improvements of the methods based on their use.

The incremental approach complements ME principles by proposing steps for *a posteriori* ME. These steps deal with collecting, analyzing, refining, and sharing methodical knowledge. The steps are based on explicit metamodels as well as on decisions and rationale behind method development. Metamodels are needed to understand methods in use and method refinements, and method rationale is needed to describe why methods were specified as they were. Throughout these steps we applied three mechanisms of method evaluation and refinement. Like the metamodeling constructs, these mechanisms examine modeling techniques in the context of modeling tools. The first mechanism — the type-instance matching — compares differences between a modeling technique's intended use (as seen from the metamodel) and actual use (as seen from models). The second mechanism — analysis of modeling power — examines the capability of the method to represent the desired aspects of the object system in models and to maintain the consistency of the models. The third mechanism — the analysis of the role of a method in problem solving — focuses on a method's capability to generate alternative solutions and support subsequent decision making. The latter two mechanisms address experiences and learning of method stakeholders, such as designers, end-users, domain experts, ISD tool experts, and method engineers.

These mechanisms are suggested so that they can focus on method aspects which need refinement. As a consequence, if the analysis phase suggests a method modification, it reveals that the *a priori* constructed method was not fully applicable. The refinements extend, modify or remove parts of the method knowledge. The refinements can be gradual and small (in comparison with other method development strategies). This explains the title of the proposed approach. The term gradual suggests that method refinements are made to the currently used method, rather than by selecting a new method. Small changes are a consequence of gradual changes; applicability is achieved by modifying parts of the method.

The incremental approach was examined through an action research intervention in two cases. The cases covered all the major steps of ME. Our discussion focused primarily on the *a posteriori* view and method evaluation mechanisms. The evaluation led to several refinements of the constructed methods. The refinements added, modified, and removed parts of method knowledge. Through a tool implementation the new method was taken into use. This finding provides evidence that *a priori* method construction alone does not always provide adequate support. In the cases, the suggested identification principles and method improvement mechanisms were found to be useful. The metamodeling approach used also revealed some extensions to the metamodeling constructs. Moreover, the use of metamodels was found to be useful while specifying local methods and analyzing their evolution. In this sense, the metamodels supported not only the local method development, but also the action research studies of detailed method knowledge and its evolution.

During the action research interventions the method refinements were performed only once. An incremental approach to method engineering, however, would necessitate several "reflection" cycles. Method engineers must obtain data from several situations to yield a metamodel repository with information about the applicability of methods (and their parts). With effective use of this incremental approach, methods can be constructed and improved based on their demonstrated ability to support specific situational factors.

The ME principles developed can be applied in an organization which is developing its own methods and needs methodical guidelines. Moreover, the principles of incremental ME are suitable not only for local method development, but also for development of standardized methods (as shown in case B).

7.3 Directions for future research

In reflecting on the research questions addressed, we observed several interesting research topics. These would allow us to further evaluate and expand the findings of this thesis.

For any modeling language, functionality and usability form central issues: ME languages are no exception. Future research on ME languages should concentrate on these aspects, either by extending existing ME languages or by creating new ones. With respect to the functionality aspect, the sufficiency of the metamodeling constructs could be analyzed by modeling more methods. The selected sample should include other types of methods than those modeled here. The sufficiency of the proposed metamodeling constructs can also be examined by modeling organizations' in-house methods, rather than methods described in the literature. These examinations can confirm (or raise doubts about) the relevancy of the proposed metamodeling constructs, and most likely find new constructs.

The proposed metamodeling constructs can be used to evaluate other metamodeling approaches. They can be used as a set of requirements to develop new metamodeling languages, or extend existing ones. Research on metamodeling should be extended to cover other types of method knowledge, in addition to the conceptual structures behind modeling techniques. Candidate types of method knowledge to be modeled include processes, participation, and decision making.

When analyzing the functionality of a metamodeling language, its usability should not be forgotten. This suggests investigating the use of ME languages using different research methods. In fact, to proceed in ME research we need empirical studies about the use of metamodeling more than proposals of new metamodeling languages. Surveys and field studies must be made to analyze what metamodeling languages are used in practice; laboratory studies are needed to investigate user preferences for different visual representational paradigms (e.g. Kelly and Rossi 1997); and case studies are needed to assess the usability of metamodeling languages in a ME project.

Empirical research is also relevant to the study of the ME process. Because ME is a relatively new research field, complementary research efforts and the use of various research methods are needed to improve the quality of research conclusions (see Tolvanen et al. 1996). As pointed out in this thesis, more case studies and action research are needed to analyze local method development in detail. These research methods should be applied to examine what factors contributed to success or failure in local method development, how frequently and to what extent methods are changed, and how methods evolve. These questions presuppose longitudinal research efforts, as well as close interaction between method use and method development situations. In addition to longitudinal studies, larger scale ME efforts, in terms of the number of stakeholders and method size, should be inspected. Studies should also address methods other than business modeling, apply different metamodeling languages, and implement method-tool companionship with different metaCASE tools.

Within empirical research, other research methods, such as field studies and surveys, must be used. Although several surveys of method use and to some extent also of method development have been performed, there is still a need for new ones. One reason is that existing studies have obtained different results, and several key questions of ME remain unanswered. Surveys should analyze how common in-house methods are, and whether stakeholders are satisfied with local methods. Field studies allow the examination of the ME process in more detail. They should examine the circumstances under which local methods are developed, whether the ME process consists of "radical" or incremental changes, and how ME projects are organized and managed.

Finally, the incremental ME principles should be taken into account while developing metamodeling languages and metaCASE tools. In addition to extending metamodeling languages with the proposed constructs, they should also be applied in metaCASE technology. MetaCASE tools should offer functionality to modify and version metamodels, to update models when a method already in use is changed, to support the collection and structuring of experiences about the use of the method, and to automate the mechanisms of method evaluation. In particular, metrics for type-instance matching should be implemented into metaCASE tools. Design rationale models should also be taken into use for recording and explaining metamodeling decisions. Tool support for these functionalities would allow the proposed principles of incremental ME to be used to full advantage.

REFERENCES

Aaen, I., (1992) CASE tool Bootstrapping - how little strokes fell great oaks. In: *Next Generation CASE tools* (eds. K. Lyytinen, V.-P. Tahvanainen), IOS Press, Amsterdam, Netherlands, pp. 8-17.

Aaen, I., Siltanen, A., Sørensen, C., Tahvanainen, V.-P., (1992) A Tale of Two Countries - CASE Experiences and Expectations. In: *The Impact of Computer-Supported Technologies on Information Systems Development*, (eds. Kendall et al.) Elsevier North-Holland, pp. 61-94.

Aalto, J.-M., (1993) Experiencies on Applying OMT to Large Scale Systems. In: *Proceedings of the seminar on Conceptual Modeling and Object-Oriented Programming*, (eds. A. Lehtola, J., Jokiniemi), Finnish Artificial Intelligence Society, pp. 39-47.

Aalto, J.-M., Jaaksi, A., (1994) Object-Oriented Development of Interactive Systems with OMT++. *Proceedings of Technology of Object-Oriented Languages and Systems* (TOOLS 14) (eds. R. Ege, M. Singh, B. Mayer), Prentice-Hall, pp. 205-218.

Ahituv, N., (1987) A metamodel of information flow: a tool to support information systems theory, *Communications of the ACM* 30(9), pp.781-791.

Alderson, A., (1991) Meta-CASE Technology. In: *Proceedings of European Symposium on Software development environments and CASE technology* (eds. A. Endres, H. Weber), Springer-Verlag, No. 509, pp. 81-91.

Alegic, S., (1988) *Object-Oriented Database Programming*. Springer-Verlag, New York.

Andersen Consulting, (1991) *Foundation-Plan/1*: Object access and design (version 1.1).

Argyris, C., Schön, D., (1978) *Organizational Learning, A theory of action perspective*. Addison-Wesley.

Armense, P., Bandinelli, S., Ghezzi, C., Morzenti, A., (1993) A survey and assessment of software process representation formalism, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 3, No. 3, pp. 401-426.

Auramäki, E., Leppänen, M., Savolainen, V., (1987) Universal framework for information activities. *Data Base*, Fall/Winter, pp. 11-20.

Avison, D., (1996) Information Systems Development: A Broader Perspective. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke), Chapman-Hall, London, pp. 263-277.

Avison, D., Wood-Harper, T., (1990) *Multiview - an exploration in information systems development*. McGraw-Hill, Maidenhead.

Awad, M., Kuusela, J., Ziegler, J., (1996) *Object-Oriented Technology for Real-Time Systems - A Practical Approach Using OMT and Fusion*, Prentice-Hall.

Baskerville, R., (1996) Structural Artifacts in Method Engineering: The Security Imperative. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 8-28.

Baskerville, R., Travis, J., Truex, D., (1992) Systems Without Method: The Impact of New Technologies on Information Systems Development Projects. In: *The Impact of Computer Supported Technologies on Information Systems Development*. (eds K. Kendall, K. Lyytinen and J. DeGross), IFIP Transactions A8, North-Holland, pp. 241-269.

Baskerville, R., Wood-Harper, T., (1996) A critical perspective on action research as a method for information systems research. *Journal of Information Technology*, July.

Batani, C., Lenzerini, M., Navathe, B., (1992) *Conceptual database design: An entity relationship approach*. Benjamin-Cummings Publishing Company, Redwood City.

Benjamin, R., Blunt, J., (1992) Critical IT Issues: The Next Ten Years. *Sloan Management Review*, Summer, pp. 7-19.

Bennetts, P., Wood-Harper, T., (1996) A traditional methodology: Its context and future. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 97-106.

Bergsten, P., Bubenko Jr, J., Dahl, R., Gustafsson, M., Johansson, (1989) L.-Å., RAMATIC - a CASE shell for implementation of specific CASE tools. TEMPORA report T6.1, SISU, Stockholm.

Bergstra, J., Jonkers, H., Obbink, J., (1985) A software development model for method engineering. *ESPRIT´84: Status report of ongoing work* (eds. J. Roukens, J. Renuart), Elsevier Science Publishers B.V., North-Holland.

Bidgood, T., Jelley, B., (1991) Modeling corporate information needs: fresh approaches to the information architecture. *Journal of Strategic Information Systems*, Vol. 1, No. 1.

Bititci, U., Carrie, A.S., (1990) Information material flow mapping. *Logistics Information Management*, March, pp. 31-36.

Blum, B., (1994) A taxonomy of software development methods. *Communications of the ACM*, Vol. 37, No. 11, pp. 82-94.

Boehm, B.W., Papaccio, P.N., (1988) Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, Vol. 14, 10 (October), pp. 1462-1477.

Bommel, P. van, ter Hofstede, A. H. M., van der Weide, Th.P., (1991) Semantics and verification of object-role models, *Information Systems* 16(5), pp. 471-495.

Booch, G., (1991) *Object-Oriented Desing with Applications*. The Benjamin/ Cummings Publishing Company Inc., Redwood City.

Booch, G., (1994) *Object-Oriented Analysis and Design with Applications*, Benjamin/Gummings Publishing Company Inc.

Booch, G., Jacobson, I., Rumbaugh, J., (1996) *Unified Modeling Language (version 0.9)*, Rational Software Corporation.

Booch, G., Jacobson, I., Rumbaugh, J., (1997) *Unified Modeling Language: version 1.0*, Rational Software Corporation.

Booch, G., Rumbaugh, J., (1995) *Unified Method for Object-Oriented Development*, Documentation Set (version 0.8), Rational Software Corporation.

Brinkkemper, S., (1990) *Formalisation of Information Systems Modelling*. Thesis Publishers.

Brinkkemper, S., (1996) Method engineering: Engineering of information systems development methods and tools. *Information and Software Technology*, 38, pp. 275-280.

Brinkkemper, S., de Lange, M., Looman, R., van der Steen, F.H.G.C., (1989) On the derivation of method companionship by metamodelling. In: *Proceedings of CASE'89, Third International Workshop On Computer-Aided Software Engineering*, London, July 17-21, 1989 (ed. J. Jenkins), IEEE Computer Society Press, pp. 266-286.

Brodie, M., (1984) On the development of data models. *Perspectives from artificial intelligence, databases, and programming languages* (eds. M. Brodie, J. Mylopoulos,J. Schmidth), Springer-Verlag, pp. 19-47.

Bronts, G., Brouwer, S., Martens, C., Proper, H., (1995) A unifying object role modelling theory, *Information Systems* 20(3), pp. 213-235.

Brooks, F., (1975) *The mythical man-month: Essays on software engineering*. Addison Wesley Publishing Company.

Bubenko jr, J., (1986) Information System Methodologies -A Research View. In: *Information System Design Methodologies: Improving the Practise*. Proceeding of the IFIP WG 8.1 Working Conference, Noordwijkerhout, the Netherlands, 5-7 may, 1986. (eds. T.W. Olle, H.G. Sol, A.A. Verrinj-Stuart) North Holland Publishing Company, Amsterdam, pp. 289-318.

Bubenko jr, J., (1988) Selecting a Strategy for Computer-Aided Software Engineering (CASE). SYSLAB-report Nr. 59, University of Stockholm, Sweden.

Bubenko, J., Langerfors, B., Sølvberg, A., (eds.) (1971) *Computer-Aided Information Systems Analysis and Design*, Studentlitteratur, Lund.

Bubenko, J., Wangler, B., (1992) Research directions in conceptual specification development. In: *Conceptual Modeling, Databases and CASE: An Integrated view of Information System Development* (eds. P. Loucopoulos, R. Zicari), John Wiley, New York.

Buchanan, D., Boddy, D., McCalman, J., (1988) Getting in, getting on, getting out and betting back. In: *Doing Research in Organizations* (ed. A. Bryman), Routledge.

Cameron, J., (1989) JSP&JSD: *The Jackson approach to software development* (2nd edition), IEEE Computer Society Press.

CASE Outlook (1989) Where Do Repositories Come From? *CASE Outlook*, 4, pp. 20-29.

CCTA (Central Computer and Telecommunication agency) (1995) *SSADM4+: reference manual*, NCC Blackwell.

CDIF (1997): http://www.cdif.org/

Charette, R.N., (1989) *Software Engineering Risk Analysis and Management*, Intertext Publications, McGraw-Hill Book Company.

Checkland, P. B., (1981) *Systems Thinking, Systems Practice,* J. Wiley, New York.

Checkland, P., (1991) From framework through experience to learning: the essential nature of action research. *Information systems research: contemporary approaches and emergent traditions* (eds. H.E. Nissen, H.K. Klein, R. Hircschheim), Elsevier Science Publishers B.V, North-Holland, pp. 397-403.

Chen, M., (1988) *The integration of organization and information system modeling: a metasystem approach to the generation of group decision support systems and computer-aided software engineering*. Dissertation, University of Arizona.

Chen, M., Nunamaker, J.F., Weber, E.S., (1989) Computer-aided software engineering: present status and future directions. *Data Base*, Spring, pp. 7-13.

Chen, P., (1976) The entity-relationship model - toward a unify view of data. *ACM Transactions on Database Systems*, 1, 1, pp. 9-36.

Chikofsky, E., (1988) Software Technology People Can Really Use. *IEEE Software*, March, pp. 8-10.

Chikofsky, E., Rubenstein, B., (1988) CASE: Reliability engineering for information systems. *IEEE Software*, March, pp. 11-16.

Ciborra, C.U., (1987) Research agenda for a transaction costs approach to information systems. In: *Critical Issues in Information Systems Reseach* (eds. R.J. Boland, R.A. Hirschheim), John Wiley & Sons Ltd.

Clemons, E., Row, M., (1991) Sustaining IT Advantage: The Role of Structural Differences. *MIS Quaterly*, September, pp. 275–292.

Coad, P., Yourdon, E., (1991a) *Object-Oriented Analysis*. Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey.

Coad, P., Yourdon, E., (1991b) *Object-Oriented Design*. Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremes, P., (1994) *Object-Oriented Development - The Fusion Method*. Prentice-Hall, Englewood-Cliffs.

Conway, B., Hunter, R., Light, M., (1995) *The AD Management Continuum: Integrated Methods, Process and Project Management*, Gartner Group, Strategic Analysis Report, R-480-127.

Cotterman, W., Senn, J. (eds) (1992) Challenges and strategies for research in system development, John Wiley & Sons Ltd.

Cronholm, S., Goldkuhl, G., (1994) Meanings and motivates of method customization in CASE environments - observations and categorizations from an empirical study. In: *Proceedings of the 5th Workshop on the Next Generation of CASE Tools*, (eds. B. Theodoulidis), University of Twente, pp. 67-79.

Curtis, B., (1992) The CASE process. Proceedings of the IFIP WG 8.1 Working Conference on *Impact of Computer Supported Techniques on Information Systems Development*. pp. 333-343.

Curtis, B., Kellner, M., Over, J., (1992) Process modeling, *Communications of the ACM*, Vol. 35, No. 9, pp. 75-90.

Curtis, B., Krasner, H., Iscoe, N., (1988) A field study of the software design process for large systems. *Communications of the ACM*, 31, 11, pp. 1268-1287.

Danzinger, M., Haynes, (1989) P., Managing the CASE environment, *Journal of Systems Management*, Vol. 40, 5, pp. 29-32.

Davenport, T.H., Eccles, R.G., Prusak, L., (1992) Information Politics. *Sloan Management Review*, Fall, pp. 53-65.

Davenport, T.H., Short, J.E., (1990) The New Industrial Engineering: Information Technology and Business Process Redesign. *Sloan Management Review*, Summer, pp. 11-27.

Davis, F., (1989) Perceived usefulness, perceived easy of use, and user acceptance of information technology, *MIS Quaterly*, vol. 13, 3, pp. 319-339.

Davis, G. B., Collins, R. W., Eierman, M., Nance, W. D., (1991) *Conceptual Model For Research On Knowledge Work*. University of Minnesota, Minneapolis MN 55455, MISRC-WP-91-10.

Davis, G., Olson, M., (1985) *Management Information Systems: Conceptual Foundations, Structure and Development*. McGraw-Hill, New York.

Davis, G.B., (1982) Strategies for information requirements determination. *IBM Systems Journal*, Vol. 21, No. 1, pp. 4-30.

De Troyer, M., Meersman, R., Ponsaert, F., (1984) *RIDL User Guide*, International Centre for Information Analysis Services, Control Data Belgium, Brussels, Belgium.

DeMarco, T., (1979) *Structured Analysis and Systems Specification*. Englewood Cliffs, N.J., Prentice-Hall.

Dur, R., (1992) *Business Reengineering in Information Intensive Organizations*. Dissertation, the Netherlands.

Earl, M., (1989) *Strategies for Information Technology*, Prentice-Hall.

Ebert, J., Sϋttenbach, R., (1997) *An OMT Metamodel*, Research report 13/97, Institut fϋr Informatik, Universität Koblenz -Landau.

Ebert, J., Sϋttenbach, R., Uhe, I., (1996) *MetaCASE in practice: A Case for KOGGE*, Research report 22/96, Institut fϋr Informatik, Universität Koblenz -Landau.

Elmasri, R., Weeldreyer, J., Hevner, A, (1985) The category concept: an extension to the entity-relationship model, *Data and Knowledge Engineering*, Vol. 1, pp. 75-116.

Embley, D., Kurtz, D., Woodfield, S., (1992) *Object Oriented Systems Analysis, A model-driven approach*, Yourdon Press, Prentice-Hall, Englewood Cliffs.

Essink, L., (1988) A conceptual framework for information systems development methodologies. In: *Information Technology for Organizational Systems* (eds. Bullinger et al.) Elsevier-Science Publishers B.V, pp. 354-362.

Eynde Van, D., Bledsoe, J., (1990) The changing practice of organization development, *Leadership and organization development journal*, Vol. 11, No. 2, pp. 25-30.

FIPS, (1993a) Integration definition for function modeling (IDEF0), *Federal Information Processing Standards Publication*, 183 (FIPS 183).

FIPS, (1993b) Integration definition for function modeling (IDEF1X), *Federal Information Processing Standards Publication*, 184 (FIPS 183).

Firesmith, D., Henderson-Sellers, B., Graham, I., Page-Jones, M., (1996) *OPEN Modeling Language (OML) Reference Manual*, OPEN Consortium.

Fitzgerald, B., (1995) *The use of system development methods: a survey*. Paper ref 9/95, University College Cork.

Fitzgerald, B., (1996) Formalized systems development methodologies: a critical perspective. *Information systems journal*, 6, pp. 3-23.

Fitzgerald, G., (1991) Validating new information systems techniques: a retrospective analysis. In: *Information Systems Research: Contemporary Approaches and Emergent Traditions* (eds. H.-E. Nissen, H.K. Klein, R. Hirschheim), Elsevier Science Publishers B.V, pp. 657-672.

Flood, R., (1993) *Beyond TQM*, Wiley, Chichester.

Floyd, C., (1987) Outline of the paradigm change in software engineering. *Computers and Democracy: A Scandinavian Challenge* (eds. G. Bjerknes, P. Ehn, M. King) Avebury Gower, Brookfield Vermont.

Flynn, D., Goleniewska, E., (1993) A survey of the use of strategic information systems planning approaches in UK organizations. *Journal of Strategic Information Systems*, Vol 2, No 4, pp. 292-319.

Forte, G., Norman, R.J., (1992) A self-assessment by the software engineering community. *Communications of the ACM*, Vol. 35, No. 4 (April), pp. 28-32.

Fraser, M., Kumar, K., Vaishnavi, V., (1991) Informal and formal requirements specification languages; Bridging the gap. *IEEE Transactions on Software Engineering*, 17, 5, pp. 454-466.

Friedman, A., Cornford, D., (1989) *Computer Systems Development: History, Organization and Implementation*. John Wiley & Sons Ltd.

Frost, S., (1994) *The Select perspective: Extending Rumbaugh's OMT for client/server system development*. Select Software Tools.

Galliers, R.D., (1985) In search of a paradigm for information systems research. *Research methods in information systems* (eds. E. Mumford, R. Hirschheim, G. Fitzgerald, A.T. Wood-Harper), Elsevier Science Publishers, North-Holland, pp. 281-297.

Galliers, R.D., (1992) Choosing Information Systems Research Approaches. *Information Systems Research: Issues, methods and practical guidelines* (ed. R. Galliers), Blackwell Scientific Publications, pp. 144-162.

Galliers, R.D., Land, F.F., (1987) Choosing Appropriate Information Systems Research Methodologies, *Communications of the ACM*, Vol. 30, 11, pp. 900-902.

Gane, C., Sarson, T., (1979) *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, Englewood Cliffs, New Jersey.

Gigch van, J.P., (1991) *System design modeling and metamodeling*. Plenum Press, New York.

Gladden, G.R., (1982) Stop the life cycle, I want to get off. *Software Engineering Notes*, 7, 2, pp. 35-39.

Goldkuhl, G., (1989) *Datasystem och verksamhetsutveckling* (in Swedish). Intention.

Goldkuhl, G., (1990) *Kontextuell verksamhetsanalys med handlingsgrafer* (in Swedish), Intention AB.

Goldkuhl, G., (1993) *Verksamhets utveckla datasystem*. Intention (in Swedish), Affärslitteratur AB, Linköping.

Goldkuhl, G., (1992) Contextual activity modeling of information systems. In: *Proceeding of the Third International Working Conference on Dynamic Modelling of Information Systems*, Noordwijkerhout, June 9-10.

Goldkuhl, G., Cronholm, S., (1993) Customizable CASE Environments: A Framework for Design and Evaluation, In: *Proceedings of COPE IT'93 / NordData*.

Goldkuhl, G., Cronholm, S., Krysander, C., (1992) Adaptation of CASE tools to different systems development methods. In: *Proceedings of 15th IRIS*, (eds. G. Bjerknes, T. Bratteteig, K. Kautz), Department of Informatics, University of Oslo, pp. 142-156.

Grant, D., Ngwenyama, O., Klein, H., (1992) *Validating ISD Methodologies Within The Organizational Context: An Action Research Case Study*. Working paper series, Binghampton, State University of New York, 92-215.

Griethuysen, J., (1982) *Concepts and terminology for the conceptual schema and the information base*. Publication nr. ISO/TC97/SC5-N695.

Grundy, J., (1993) *Multiple textual and graphical views for interactive software development environments*, Dissertation, University of Auckland.

Grundy, J., Venable, J., (1996) Towards an integrated environment for method engineering. *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke), Chapman-Hall, London, pp. 45-62.

Hackathorn, R., Karimi, J., (1988) A Framework for comparing information engineering methods. *MIS Quaterly*, June.

Hammer, M., Champy, J., (1993) *Reengineering the Corporation- A manifesto for business revolution*. Nicholas Brealey publishing Ltd., London.

Hardy, C., Edwards, H., Thompson, J., (1996) The unification of method engineering approaches. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 439-450.

Hardy, C., Thompson, J., Edwards, H., (1995) The use, limitations and customization of structured systems development methods in the United Kingdom. *Information and Software Technology*, 37 (9), pp. 467-477.

Harel, D., (1988) On visual formalism, *Communications of the ACM*, 31, 5 (May), pp. 514-530.

Harmsen, F., (1997) *Situational Method Engineering*. Dissertation, Moret Ernst & Young Management Consultants, the Netherlands.

Harmsen, F., Brinkkemper, S., Oei, H., (1994a) A language and tool for the engineering of situational methods for information systems development, *Proceeding of the 4th International Conference Information Systems Development* (ISD'94), (eds. J. Zupancic, S. Wrycza), Moderna Organizacija, pp. 206-214.

Harmsen, F., Brinkkemper, S., Oei, H., (1994b) Situational Method Engineering for Information System Project Approaches. In: *Methods and Associated Tools for the Information Systems Life Cycle* (A-55), (A. Verrijn-Stuart, T. Olle), Elsevier Science B.V., North-Holland.

Harmsen, F., Saeki, M., (1996) Comparison of four method engineering languages. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 209-231.

Harrington, H.J., (1991) *Business Process Improvement*. McGraw-Hill, Inc., USA.

Henderson-Sellers, B., (1992) *A book of object-oriented knowledge : object-oriented analysis, design and implementation : a new approach to software engineering* Prentice-Hall.

Henderson-Sellers, B., Bulthuis, A., (1996a) COMMA: Sample metamodels. *Report on Object-Oriented Analysis and Design, JOOP*, Nov-Dec, pp. 44-48.

Henderson-Sellers, B., Bulthuis, A., (1996b) The COMMA Project: Final report. COTAR technical report 95/35, University of Techology, Sydney.

Henderson-Sellers, B., Edwards, J., (1994) *The working object — Object-oriented software engineering: methods and management*. Prentice-Hall.

Heym, M., (1993) *Methoden-Engineering Spezification und Integration von Entwicklungsmethoden für Informationssysteme* (in German), Dissertation, Hochschule St.Gallen, Switzerland.

Heym, M., Österle, H., (1992) A Reference Model of Information Systems Development. In: *The Impact of Computer Supported Technologies in Information Systems Development* (eds. K.E. Kendall, K. Lyytinen, J.I. DeGross), Amsterdam, North-Holland, pp. 215-240.

Heym, M., Österle, H., (1993) Computer -aided methodology engineering. *Information and Software Technology*, Vol. 35, 6/7 (June/July), pp. 345-354.

Hidding, G., Gwendolyn, J., Freund, M., Joseph, J., (1993) *Modeling Large Processes with Task Packages*, Workshop on Modeling in the Large, AAAI Conference, Washington, D.C.

Hillegersberg van, J., (1997) *Metamodeling-based integration of object-oriented systems development*. Dissertation, Thesis Publishers, Amsterdam.

Hillegersberg van, J., Kumar, K., Welke, R.J., (1998) Using metamodeling to analyze the fit of object-oriented methods to languages. *Proceedings of the 31st Hawaii International Conference on System Sciences*, Volume V, (eds. R. Blanning, D. King) IEEE Computer Society, pp. 323-332.

Hobby, J., (1993) Spending on case will raise. *Computing*, September.

Hochstettler, W., (1986) *A model for supporting multiple software engineering methods in a software environment*, Dissertation, The Ohio State University.

Hoef van de, R., Harmsen, F., (1995) Quality requirements for situational methods. In: *Proceedings of the 6th Next Generation of CASE tools* (ed. G. Grosz), Jyväskylä, Finland.

Hofstede ter, A., (1993) *Information modeling in data intesive domains*. Dissertation, University of Nijmegen, the Netherlands.

Hofstede ter, A., Proper, H., Weide van der, P., (1993) Formal definition of a conceptual language for the description and manipulation of information models, *Information Systems*, Vol. 18, 7, pp. 489-523.

Hofstede ter, A., Verhoef, T., (1996) Meta-CASE: Is the game worth the candle?, *Information Systems Journal*, 6, pp. 41-68.

Hofstede, A. H. M. ter, Nieuwland, E. R., (1993) Task structure semantics through process algebra, *Software Engineering Journal*, (8), pp.14-20.

Hofstede, A. H. M. ter, van der Weide, Th. P., (1993) Expressiveness in data modeling, *Data & Knowledge Engineering* (10), pp. 65-100.

Hong, S., van den Goor, G., Brinkkemper, S., (1993) A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies. In: *Proceedings of the 26th Hawaii International Conference on Systems Science* (eds. J. Nunamaker, R. Sprague), Vol. 4., IEEE Computer Society Press, Los Alamitos.

Hopkins, T., Horan, B., (1995) *Smalltalk : an introduction to application development using VisualWorks*, Prentice Hall, London.

Huber, G., (1991) Organizational learning: the contributing process and the literatures. *Organization Science*, Vol. 2, No. 1., pp. 88-115.

Hughes, J., Reviron, E., (1996) Selection and evaluation of information system development methodologies: The gap between the theory and practice. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 309-319.

Hull, R., King, R., (1987) Semantic Database Modeling Survey, Applications, and Research Issues, *ACM Computing surveys* 19(3), pp. 201-260.

Humphrey, W.S., (1988) Characterizing the Software Process: A Maturity Framework. *IEEE Software*, March, pp. 73-79.

Humphrey, W.S., (1989) *Managing the software process*. The SEI series in Software Engineering. Addison-Wesley.

IBM (1984) *Business Systems Planning — Information Systems Planning Guide*. Application Manual, IBM Corporation, July.

IFPUG (1994) International Function Point Users Group*: Function Point Counting Practices Manua"*, Release 4.0, IFPUG Standards.

Iivari, J., (1992) Relationships, aggregations and complex objects. In: *Information Modeling and Knowledge Bases III: Foundations, Theory and Applications* (eds. S. Ohsuga, H. Kangassalo, H. Jaakkola, K. Hori, N. Yonezaki), IOS Press, pp. 141-159.

Iivari, J., Kerola P., (1983) A sociocybernetic framework for the feature analysis of information systems development methodologies. In: *Information Systems Development Methodologies: A Feature Analysis* (eds. T.W. Olle. H.G. Sol, C.J. Tully), Elsevier Science Publishers B.V., North-Holland.

Iona, (1997) *Orbix Programming Guide*. IONA Technologies PLC.

Isakowitz, T., Stohr, E., Balasubramanian, P, (1995) RMM: A methodology for structured hypermedia design. *Communications of the ACM*, Vol. 38, 8, pp. 34-44.

Isazadeh, H., Lamb, D., (1997) A Comparative Review of MetaCASE Tools. In: *Proceedings of the Information System Development Conference*, Plenum Press.

ISO (1990) ISO-IEC 10027. Information technology - Information Resource Dictionary System (IRDS) - Framework, ISO/IEC International standard.

ISO, (1991) *EXPRESS Language Reference Manual*, ISO TC184/SC4/WG5, Document N14, Owner: Philip Spiby, CADDETC, 171 Woodhouse Lane, Leeds LS2 3AR, UK.

Jaaksi, A., (1997) *Object-oriented development of interactive systems*, Dissertation, Tampere University of Technology, Publications 201, Tampere, Finland.

Jackson, M.A. (1976) Constructive Methods of Program Design. In: *Proceedings of First Conference of the European Cooperation in Informatics*, Vol. 44.

Jacobson, I., (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, Addison-Wesley.

Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S., (1995) Concept Base: A deductive object base for meta data management. *Journal for intelligent information systems*, 4, 2, pp. 167-192.

Jarke, M., Pohl, K., Rolland, C., Schmitt, J.-R. (1994) Experience-Based Method Evaluation and Improvement: A process modeling approach, In: *Proceedings of the IFIP WG8.1 Working Conference CRIS'94*, (eds. Olle, T.W., Verrijn Stuart, A.A.), North-Holland Publishers, Amsterdam, pp. 1-27.

Jarke, M., Pohl, K., Weidenhaupt, K., Lyytinen, K., Marttiin, P., Tolvanen, J.-P., Papazoglou, M., (1998) Meta modeling: A formal basis for interoperability and adaptability. In: *Information Systems Interoperability* (B. Krämer, M. Papazoglou), John Wiley Research Science Press.

Jayaratna, N., (1994) *Understanding and Evaluating Methodologies: NIMSAD, A Systemic Approach*, McGraw-Hill.

Jeffrey, D.R., (1987) Software engineering productivity models for management information system development, In: *Critical Issues in Information Systems Research* (eds. R. Boland, R. Hirschheim), John Wiley & Sons Ltd, pp. 113-134.

Johansson, H., McHugh, P., Pendlebury, A., Wheeler, W., (1993) *Business process re-engineering: Breakpoint strategies for market dominance*, John Wiley and Sons.

Jönsson, S., (1991) Action Research. *Information systems research: contemporary approaches and emergent traditions* (eds. H.E. Nissen, H.K. Klein, R. Hircschheim), Elsevier Science Publishers B.V, North-Holland, pp. 371-396.

Joosten, S., Schipper, M., (1996) Validation of the Workflow Analysis Technique "Trigger Modeling", In: *Proceedings of the 2nd Americas Conference on Information Systems*, Phoenix, Arizona, pp. 626-628.

Jordan, E., Evans, J., (1992) The simulation of IS strategy using SIMIAN. In: *Dynamic Modeling of Information Systems II* (eds. H.G. Sol, R.L.Grosslin), Elsevier Science Publishers B.V., The Netherlands, pp. 131-144.

Kaasboll, J., Smordal, O., (1996) Human work as context for development of object oriented modeling techniques. *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke), Chapman-Hall, London, pp. 111-125.

Katz, R. L., (1990) Business/enterprise modeling. *IBM Systems Journal*, Vol 29, No. 4.

Kelly S., Lyytinen K., Rossi M., (1996) MetaEdit+: A Fully configurable Multi-User and Multi-Tool CASE and CAME Environment. In: *Proceedings of the 8th Conference on Advanced Information Systems Engineering* (eds Y. Vassiliou, J. Mylopoulos), Springer Verlag.

Kelly, S., (1994) A matrix editor for a metaCASE environment, *Information and Software Technology*, Vol. 36, No. 6, pp. 361-171

Kelly, S., (1995) What's in a Relationship? On distinguishing property holding and object holding. In: *Proceeding of the IFIP conference Information System Concepts: Towards a consodilation of views* (eds. E. Falkenberg, W. Hesse, A. Olive), Chapman & Hall, pp. 144-159.

Kelly, S., (1997) *Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+*, (Dissertation). Jyväskylä Studies in Computer Science, Economics and Statistics, No. 41, University of Jyväskylä.

Kelly, S., Rossi, M., (1997) Differences in Method Engineering Performance with Graphical and Matrix Tools: A Preliminary Empirical Study. *Proceedings of the second CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design* (eds. K. Siau, Y. Wand, J. Parsons), Barcelona, Spain, University of Nebraska at Lincoln.

Kelly, S., Smolander, K., (1996) Evolution and Issues in MetaCASE, *Information and Software Technology*, 38, pp. 261-266.

Kelly, S., Tahvanainen, V-P., (1994) Support for Incremental Method Engineering and MetaCASE. In: *Proceedings of the fifth workshop on the next generation of CASE tools*, B. Theodoulidis (ed.), Memoranda Informatica 94-25, University of Twente, NL, pp. 140-148.

Kerner, D.V., (1979) Business Information Characterization Study. *Data Base*, Spring.

Kim, W., Bertino, E., Garza, J.F. (1989) Composite objects revisited. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (eds. J. Clifford, B. Lindsay, D. Maier), SIGMOD Record, Vol. 18, No. 2.

Kim, Y., March, S., (1995) Comparing data modeling formalism. *Communications of the ACM*, Vol. 38, No. 6, pp. 103-115.

Kinnunen, K., Leppänen, M., (1994) O/A-Matrix and a Technique for Methodology Engineering, *Proceeding of the 4th International Conference Information Systems Development* (ISD'94) (eds J. Zupancic, S. Wrycza), Moderna Organizacija, pp. 113-126.

Kitchenham, B., Pickard, L., Pfleeger, S., (1995) Case studies for method and tool evaluation. *IEEE Software*, July, pp. 52-62

Klein, H., (1983) A verbal rejoinder to Colter's paper, *4th American Conference on Information Systems*, Houston.

Konsynski, B., (1993) Strategic Control in the Extended Enterprise. *IBM Systems Journal*, vol. 32, no. 1, pp. 111-142.

Kotteman, J., Konsynski, B., (1984) Information Systems Planning and Development: Strategic Postures and Methodologies. *Journal of Management Information Systems*, Vol. 1, No. 2, pp. 45-63.

Krasner, G., Pope, S., (1988) A Cookbook for using the model-interface-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, August/September.

Krogstie, J., Sølvberg, A., (1996) A classification of methodical frameworks for computerized information systems support in organizations. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 278-295.

Kronlöf, K., (1993) ed, *Method Integration: Concepts and Case Studies*. John Wiley & Sons, Chichester.

Kumar, K., Welke, R., (1984) Implementation failure and system developer values: assumptions, truisms and empirical evidence. In: *Proceedings of the 5th International Conference on Information Systems (ICIS)*, (eds. L. Maggi, J. King, K. Kraemer), pp. 1-13.

Kumar, K., Welke, R.J., (1992) Methodology engineering: a proposal for situation-specific methodology construction. In: *Challenges and Strategies for Research in Systems Development* (eds. W.W. Cotterman, J.A. Senn), John Wiley & Sons Ltd, pp. 257-269.

Kurki, T., (1996) Object-TT: and object-oriented system development model of TT-Group Inc (in Finnish), *Syteemityö*, Helsinki, 1/96, pp. 22-24.

Lee, H., Billington, C., (1992) Managing Supply Chain Inventory: Pitfalls and Opportunities. *Sloan Management Review*, Spring, pp. 65-73.

Lieberherr, K. J., Ignacio, S., Cun, X., (1994) Adaptive Object-Oriented Programming using Graph-Based Customization, *Communications of the ACM*, May, (4), pp. 94-101.

Lindströn, H., Raitio, P., (199 2) *Logistics software - a survey and an analysis framework*, (in Finnish), Report J-12, Technical Reseach Centre of Finland, Helsinki.

Loh, M., Nelson, R., (1989) Reaping CASE Harvests. *Datamation*, July, 1, pp. 31-34.

Low, G.C. Jeffrey, D.R. (1990) Function Points in the Estimation and Evaluation of the Software Process. *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 1, Jan, pp. 64-71.

Lundeberg, M., (1982) The ISAC approach to specification of information systems and its applications to the organization of an IFIP working conference. In: *Information System Design Methodologies: A Comparative Review*, (eds. T.W. Olle, H.G. Sol, A.A. Verrinj-Stuart), North Holland Publishing Company.

Lundeberg, M., (1992) A framework for recognizing patterns when reshaping business processes. *Journal of Strategic Information Systems*, Vol 1, 3 (June).

Lundeberg, M., Goldkuhl, G., Nilsson, A., (1981) *Information Systems Development: A Systematic Approach.* Prentice-Hall, Englewood Cliffs.

Lyytinen, K., (1986) *Information systems development as social action: framework and critical implications*. Dissertation, University of Jyväskylä.

Lyytinen, K., (1987) A taxonomic perspective of information systems development: theoretical constructs and recommendations. In: *Critical issues in information systems research,* (eds. R.J.Boland, R.A.Hirschheim), John Wiley & Sons Ltd., pp. 3-41.

Lyytinen, K., Hirschheim, R., (1987) *Information System Failures - a survey and classification of the empirical literature*. Oxford Surveys in Information Technology, Oxford University Press, Vol. 4, pp. 257-309.

Lyytinen, K., Marttiin, P., Tolvanen, J.-P., Jarke, M., Pohl, K., Weidenhaupt, K., (1998) CASE Environment Adaptability: Bridging the Islands of Automation, Submitted.

Lyytinen, K., Smolander, K., and Tahvanainen, V.-P., (1989) Modelling CASE Environments in Systems Development, In: *Proceedings of the first Nordic Conference on Advanced Systems*, SISU, Stockholm.

Lyytinen, K., Tahvanainen, V.-P., Smolander, K., (1991) *Computer Aided Methodology Engineering (CAME) - A Research Proposal*. University of Jyväskylä, Department of Computer Science and Information Systems, Jyväskylä.

Macdonald, K.H., (1991) The Value Process Model. In: *The Corporation of the 1990s: information technology and organizational transformation* (ed. M. Scott Morton). Oxford University Press, pp. 299–309.

March, J. G., Simon, H.A., (1958) *Organizations*, Wiley, New York.

Martin, J., Finkelstein, C., (1981) *Information Engineering*, Vols 1 and 2, Prentice Hall, Englewood Cliffs, New Jersey.

Marttiin, P. (1994) Towards Flexible Process Support with a CASE shell, In: *Advanced Information Systems Engineering, Proceedings of the Third International Conference CAiSE'94, Utrecht, The Netherlands, June 1994*, G. Wijers, S. Brinkkemper and T. Wasserman (Ed.), Springer-Verlag, Berlin, pp. 14-27.

Marttiin, P., (1998) *Customizable Process Modeling Support and Tools for Design Environment*, (Dissertation). Jyväskylä Studies in Computer Science, Economics and Statistics, No. 43, University of Jyväskylä.

Marttiin, P., Harmsen, F., Rossi, M., (1996) A functional framework for evaluating method engineering environments: the case of MaestroII/Decamerone and MetaEdit+. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 63-86.

Marttiin, P., Lyytinen, K., Rossi, M., Tahvanainen, V. -P., Tolvanen, J. -P., (1995) Modeling requirements for future CASE: issues and implementation considerations, *Information Resources Management Journal* 8(1) pp. 15-25.

Marttiin, P., Rossi, M., Tahvanainen, V.-P., Lyytinen, K., (1993) A comparative review of CASE shells - A preliminary framework and research outcomes. *Information & Management*, 25, pp. 11-31.

Mathiassen, L., Munk-Madsen, A., Nielsen, P., Stage, J, (1995) *Object oriented Design* (in Danish) Marko, Aalborg.

Mathiassen, L., Munk-Madsen, A., Nielsen, P., Stage, J, (1996) Method engineering: Who's the customer. *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke), Chapman-Hall, London, pp. 232-245.

Mayer, B., (1992) *Eiffel: The Language*. Prentice-Hall, Hemel Hempstead.

McClure, C., (1989) *CASE is software automation*. Prentice Hall, Englewood Cliffs, New Jersey.

McLeod, G., (1997) Method Points: Towards a metric for method complixity. In: *Proceedings of the Second CAiSE/IFIP8.1 International Workshop on Evaluation of modeling techniques in systems analysis and design* (EMMSAD'97), (eds. K. Siau, Y. Wand, J. Parsons).

Mercurio, V., Meyers, B.F., Nisbet, A.M., Radin, G., (1990) AD/Cycle strategy and architecture, *IBM Systems Journal*, 29, 2, pp. 170-188.

Meta Systems, (1989) *QuickSpec - User's Guide* (*version 1.0.)*, Meta Systems Ltd., Ann Arbor, Michigan, USA.

MetaCase Consulting, (1994) *MetaEdit 1.2 User's Guide*, Micro-Works, Jyväskylä, Finland.

MetaCase Consulting, (1996a) *MetaEdit+ (version 2.5): User's Guide*, Micro-Works, Jyväskylä, Finland.

MetaCase Consulting, (1996b) *MetaEdit+ Method Workbench (version 2.5): User's Guide*, Micro-Works, Jyväskylä, Finland.

Miner, A., Mezias, S., (1996) Ugly duckling no more: Past and futures of organizational learning research. *Organization Science*, Vol. 7, No. 1, pp. 88-98.

Morrow, M., (ed.) (1992) *Activity-based management: New approaches to measuring performance and managing costs*, Woodhead-Faulkner, New York.

Moynihan, E., Taylor, M., (1996) A comparative examination of historical and current business systems development. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 29-40.

Nandhakumar, J., Avison, D., (1996) Information systems development methodology in use: An empirical study. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 205-214.

Naumann, J., Davis, G., McKeen, J., (1980) Determining information requirements: A contingency method for selection of a requirements assurance strategy. *The Journal of Systems and Software*, 1, pp. 273-281.

Necco, C.R., Gordon, C.L., Tsai, N.W. (1987) Systems Analysis and Design: Current Practices, *MIS Quarterly*, December, pp. 461-475.

Neches R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., Swartout, W.R., (1991) Enabling Technology for Knowledge Sharing. *AI Magazine*, Fall, pp. 36-55.

Nijssen, G., Halpin, T., (1989) *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*. Prentice-Hall, Sydney.

Nilsson, E.G., (1989) CASE Tools and Software Factories. In: *Proceedings of CASE89 The First Nordic Conference on Advanced Systems Engineering*, Stockholm, Sweden, May 9-11 (eds. B. Steinholtz, A. Sølvberg, L. Bergman) SISU, Stockholm, Sweden, pp. 42-60.

Nissen, H., Jeusfeld, M., Jarke, M., Zemanek, G., Huber, H., (1996) Managing multiple requirements perspectives with metamodels. *IEEE Software*, March, pp. 37-48

Nissen, H.E., (1996) Responsible action in the use, management and development of information systems. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 1-16.

Nonaka, I., (1994) A dynamic theory of organizational knowledge creation. *Organization Science*, Vol. 5., No. 1, pp. 14-37.

Norman, R., Chen, M., (1992) Editor's introduction, *IEEE Software*, March, pp. 13-16.

Nuseibah, B., Finkelstein, A., Kramer, J., (1996) Method engineering for multi-perspective software development. *Information and Software Technology*, 38, pp. 267-274.

Oakland, J., (1993) *Total quality management: the route to improving performance* (2nd ed.), Butterworth Heinemann, Oxford.

Odell, J., (1996) A primer to method engineering. Presentation at the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support, Atlanta, August, 1996.

Oei, J. L. H., van Hemmen, L. J. G. T., Falkenberg, E. D., Brinkkemper, S., (1992) *The Meta Model Hierarchy: A Framework for Information for Information Systems Concepts and Techniques*, University of Nijmegen, Nijmegen.

Oei, J., Falkenberg, E., (1994) Harmonisation of information systems modelling and specification techniques, In: *Methods and Associated Tools for the Information Systems Life Cycle*, (eds. A. A. Verrijn-Stuart and T. W. Olle) No. A-55, Elsevier Science publishers, pp. 151-168.

Oei, J.L.H. (1995) A meta model transformation approach towards harmonisation in information system modelling. In: *Information System Concepts - Towards a consolidation of views*, (eds. Falkenberg, W. Hesse and A. Olivé), Chapman & Hall, London, pp. 106-127.

Oinas-Kukkonen, H., (1996) Method rationale in method engineering and use. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 87-93.

Olle, T.W., (1994) Expanding methodologiess to handle distributed systems. In: *Proceeding of IFIP TC8 conference Business Process Re-engineering: Information Systems Opportunities and Challenges*, (eds. Glasson et al), Elsevier Science B.V., North-Holland, pp. 443-457.

Olle, T.W., Hagelstein, J., Macdonald, I.G., Rolland, C., Sol., H.G., Van Assche, F., Verrijn-Stuart, A.A., (1991) *Information Systems Methodologies - A Framework for Understanding*. (2^{nd} edition) Addison-Wesley Publishing Company, The Bath Press, Avon.

Olle, T.W., Sol, H.G., Bhabuta, J., (eds.) (1988) *Proceeding of the IFIP WG 8.1 Working Conference on Computerized Assistance During the Information Systems Life Cycle*, Engham, England, 19-22, September, 1988. North-Holland, Amsterdam, The Netherlands.

Olle, T.W., Sol, H.G., Tully, C.J., (eds.) (1983) *Proceeding of the IFIP WG 8.1 Working Conference on Feature Analysis of Information Systems Design Methodologies*, York, United Kingdom, 5-7, July, 1983. North-Holland, Amsterdam, The Netherlands.

Olle, T.W., Sol, H.G., Verrijn-Stuart, A.A., (eds.) (1982) *Proceeding of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies*, Noordwijkerhout, The Netherlands, 10-14, May, 1982. North-Holland, Amsterdam, The Netherlands.

Olle, T.W., Sol, H.G., Verrijn-Stuart, A.A., (eds.) (1986) *Proceeding of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies: Improving the Practise*, Noordwijkerhout, The Netherlands, 5-7, May, 1986. North-Holland, Amsterdam, The Netherlands.

OMG(1997): http://www.omg.org/

Österle, H., Brenner, W., Hilbers, K., (1990) The implementation of Information Systems Architectures. Information Management 2000 Research Program, University of St. Gallen, Report CCIM2000/6.

Oz, E., (1994) When Professional Standards are Lax: The CONFIRM Failure and its Lessons. *Communications of the ACM*, Vol. 37, No. 10, pp. 29-36.

Parkinson, J., (1996) *60 minute software - strategies for accelerating the information systems delivery process*. John Wiley and Sons, New York.

Parsons, J., Russo, N., Tolvanen, J.-P., (1997) Trainging methodology researchers: Foundational research methods. In: *Proceedings of the 5th BCS Conference on Training and Education of Methodology Practioners and Reseachers* (eds. N. Jayaratna, T. Wood-Harper, B. Fitzgerald) Springer.

Patton, M., (1990) *Qualitative Evaluation and Research Methods*, Newbury Park, Sage, 2nd edition.

Pohl, K., (1996) *Process-Centered Requirements Engineering,* Research Studies Press, John Wiley & Sons.

Porter, M., (1985) *Competitive Advantage*. New York: Free Press.

Punter, T., Lemmen, K., (1996) The MEMA-model: towards a new approch for Method Engineering. *Information and Software Technology*, 38, pp. 295-305.

Pyburn, P., (1983) Linking the MIS Plan with Corporate Strategy: An Exploratory Study, *MIS Quaterly*, June, pp. 1-14.

Ramackers, G., (1994) Model integration and model execution. In: *Methods and associated tools for information systems life-cycle*. (eds. A.A. Verrijn-Stuart, T. Olle), Elsevier-Science B.V., pp. 223-239.

Ramesh, B., Edwards, M., (1993) Issues in the development of requirements traceability model. In: *Proceedings of IEEE Symp. Requirements Engineering,* San Diago, California.

Rapoport, R., (1970) Three dilemmas of action research. *Human relations*, 23, pp. 499-513.

Rask, R., Laamanen, P., Lyytinen, K., (1993) Automatic Derivation and Comparison of Specification Level Software Product Metrics in a CASE environment, *IEEE Transactions on Software Engineering*, vol. 19 no. 7, pp. 661-671.

Rockart, J., (1979) Chief Executives Define Their Own Data Needs. *Harward Business Review*, Vol. 57, No. 2.

Rockart, J.F., Hofman, J.D., (1992) Systems Delivery: Evolving New Strategies. *Sloan Management Review*, summer, pp. 21-31.

Rolland, C., Prakash, N., (1996) A proposal for context-specific method engineering. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 191-208.

Rolland, C., Souveyet, C., Moreno, M., (1995) An approach for defining ways-of-working, *Information Systems*, 20(4), pp.337-359.

Ross, T.R., Schoman, K.E., (1977) Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering,* Vol. 3, No. 1.

Rossi, M., (1995) The MetaEdit CAME environment, In: *Proceedings of the MetaCase 95,* University of Sunderland press, Sunderland.

Rossi, M., (1998) *Advanced Computer Support for Method Engineering - Implementation of CAME Environment in MetaEdit+*, (Dissertation). Jyväskylä Studies in Computer Science, Economics and Statistics, No. 42, University of Jyväskylä

Rossi, M., Brinkkemper, S., (1996) Complexity Metrics For Systems-Development Methods And Techniques, *Information Systems*, 21, 2, pp. 209-227.

Rossi, M., Gustafsson, M., Smolander, K., Johansson, L.-Å., Lyytinen, K., (1992) Metamodeling Editor as a Front End for CASE shell, *Advanced Information Systems Engineering*, (ed. P. Loucopoulos), Springer-Verlag, Lecture Notes in Computer Science #593, Berlin, pp. 546-567.

Rossi, M., Tolvanen, J-P., (1995) Using Reusable Frameworks in Development of a Method Support Envionment, In: *Proceedings of The WITS 1995*, Amsterdam, The Netherlands, (eds. M. Jarke, S. Ram), pp. 240-249.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., (1991) *Object-Oriented Modeling and Design*. Prentice-Hall.

Russo, N., Hightower, R., Pearson, J., (1996) The failure of methodologies to meet the needs of current development environments. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 387-394.

Russo, N., Klomparens, R., (1993) *Formal application development methdologies in the 90's*. Working paper, Northern Illinois University, DeKalb, Illinois.

Russo, N., Wynekoop, J., Walz, D., (1995) The use and adaptation of system development methodologies. *Proceedings of International Conference of IRMA (International Resources Management Association*, Atlanta, May 21-14.

Ryan, K., Kronlöf, K., Sheehan, A., (1996) Method integration. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 235-246.

Saeki, M., Wenyin, K., (1994) Specifying software specification and design methods. In: *Proceedings of 6th International Conference on Advanced Information Systems Engineering* (eds. G. Wijers, S. Brinkkemper, T. Wasserman), Springer-Verlag, pp. 353-366.

Sauer, C., Lau, C., (1997) Trying to adopt system development methodologies - a case-based exploration of business users' interests. *Information Systems Journal*, 7, pp. 255-275.

Savolainen, V., (1992) A dynamic framework of reference of information systems development. *Dynamic modelling of information systems II* (eds. H.G. Sol, R.L. Grosslin), Elsevier Science Publishers, pp. 285-307.

Schipper, M., Joosten, S., (1996) A validation procedure for information systems modeling techniques. In: *Workshop on Evaluation of Modeling methods in Systems Analysis and Design*, 8th Conference on Advanced Information Systems Engineering (CAISE'96).

Schön, D., (1983) *The Reflectice Practitioner*, Basic Books Inc., New York.

Scott Morton, M., (1985) The state of the art of research. In: *The Information Systems Research Challenge* (eds. F. McFarlan et al.), Boston: Harward Business School Press.

Seppänen, V., Kähkönen, A. -M., Oivo, M., Perunka, H., Isomursu, P., Pulli, P., (1996) *Strategic Needs and Future Trends of Embedded Software*. Technology Development Centre, Technology review 48/96, Sipoo, Finland.

Shlaer, S., Mellor, S.J., (1992) *Object Lifecycles: Modeling the world in states.* Prentice-Hall, Yourdon Press Computing Series.

Slooten van, C., Schoonhoven, B., (1994) Towards Contingent Information Systems Development Approaches: In: *Proceedings of ISD´94: Methods and Tools, Theory and Practice*.

Slooten van, K., (1995) *Situated Methods for Systems Development*, Dissertation, University of Twente, the Netherlands.

Slooten van, K., Hodes, B., (1996) Characterizing IS development projects. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 29-44.

Smith, H.A., McKeen, J.D., (1993) Re-engineering the Corporation: Where Does I.S. Fit In: *Proceedings of the 26 th Hawaii International Conference on Systems Science*, (eds. J.F. Nunamaker, R.H. Sprague), Vol. 3., IEEE Computer Society Press, USA.

Smith, J.M., Smith, D.C.P., (1977) Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, Vol. 2, No 2.

Smith, P., Stobart, S.C., Thompson, J.B., (1990) Potential Benefits and Problems of Customizable CASE tools. In: *Fourth International Workshop on Computer-Aided Software Engineering*, (eds. R.J. Norman, R. Van Ghent) IEEE Computer Society Press, Los Alamitos.

Smolander, K., Tahvanainen, V.-P., Lyytinen, K., Marttiin, P., (1991) MetaEdit - a flexible graphical environment for methodology modeling. In: *Advanced Information Systems Engineering* (eds. R. Andersen, J. Bubenko, A. Sølvberg), Berlin, Germany, Springer-Verlag, pp. 168-193.

Smolander, K., (1991) GOPRR - a proposal for a metamodelling method, MetaPHOR-project, internal paper.

Smolander, K., (1991) *Metamodels in CASE Environments*. Licenciate thesis, Computer Science Reports, University of Jyväskylä, Jyväskyl ä, Finland.

Smolander, K., (1992) OPRR - A Model for Modeling Systems Development Methods. In: *Next Generation CASE Tools* (eds. K. Lyytinen, V.-P. Tahvanainen) IOS Press, Amsterdam, Netherlands, pp. 224-239.

Smolander, K., Lyytinen, K., Tahvanainen, V-.P., (1989) Family tree of methods (in Finnish), Syti-project (internal-paper), University of Jyväskylä, Finland.

Smolander, K., Tahvanainen, V.-P., Lyytinen, K., (1990) How to Combine Tools and Methods in Practise - a Field Study. In: *Lecture Notes in Computer Science, Second Nordic Conference CAiSE'90*, (eds. B. Steinholtz, A. Sølvberg, L. Bergman) Stockholm, Sweden, May, pp. 195-211.

Smyth, D., Checkland, P., (1976) Using a systems approach: the structure of root definitions. *Journal of applied systems analysis*, Vol 5, 1.

Song, X., Osterweil, L., (1992) Towards Objective and Systematic Comparisons of Software Design Methodologies, *IEEE Software* 18 (5), pp. 43-53.

Sorenson, P., Tremplay, J.-P., McAllister, A., (1988) The Metaview System for Many Specifications Environments. *IEEE Software*, March, pp. 30-38.

Spurr, K., Layzell, P., Jennison, L., Richards, N., (eds.) (1994) *Software assistance for business re-engineering*, John Wiley & Sons.

Standish Group International (1995) *Chaos*. http://www.standishgroup.com/chaos.html.

Stegwee, R.A., Van Waes, R.M., (1993) Flexible CASE tools for Information Systems Planning. In: *Computer-Aided Software Engineering - Issues and Trends for the 1990s and Beyond*, (ed. T. Bergin), Idea Group Publishing, pp. 248-292.

Sullivan, C.H., (1985) Systems Planning in the Information Age. *Sloan Business Review*, Vol. 26, 2, pp. 3–11.

Susman, G., Evered, R., (1978) An assessment of the scientific merits of action research. *Administrative Science Quaterly*, 23, Dec, pp. 582-603.

Süttenbach, R., Ebert, J., (1997) *A Booch Metamodel*, Research report 5/97, Institut für Informatik, Universität Koblenz -Landau.

Tagg, B., (1990) Implementing tool support for box structures. *IBM Systems Journal*, Vol. 29, 1, pp. 79-89.

Taivalsaari, A., Vaaraniemi, S., (1997) TDE: Supporting geographically distributed software design with shared collaborative workspaces. *Proceedings of CAiSE'97, Advanced Information Systems Engineering* (eds. A. Olive, J. Pastor), Springer, Heidelberg, pp. 309-408.

Teichroew, D., Hershey III, E., (1977) PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, January.

Teichroew, D., Macasovic, P., Hershey III, E., Yamato, Y., (1980) Application of the Entity-Relationship Approach to Information Processing Systems Modeling, ISDOS-project, The University of Michigan.

Teng, J., Kettinger, W., Guha, S., (1992) Business process redesing and information architecture: establishing the missing links. In: *Proceedings of the 13th International Conference on Information Systems*, (eds. J. DeGross, J. Becker, J. Elam), pp. 81-89.

Teorey, T., (1990) *Data base modeling and design: The entity-relationship approach.* Morgan Kaufmann Publishers, Calif., USA.

Tollow, D., (1996) Experiences of the pragmatic use of structured methods in public sector projects. In: *Lessons learned from the use of methodologies* (Proceedings of the 4th Conference on Information System Methodologies), (eds. N. Jayaratna, B. Fitzgerald), British Computer Society, pp. 177-186.

Tolvanen, J.-P., (1995) Incremental method development for business modeling: an action research case study. In: Proceedings of the 6th Next Generation of CASE tools (ed. G. Grosz), Jyväskylä, Finland.

Tolvanen, J.-P., Lyytinen, K., (1993) Flexible method adaptation in CASE - the metamodeling approach. *Scandinavian Journal of Information Systems*, Vol. 5, pp. 51-77.

Tolvanen, J.-P., Lyytinen, K., (1994) Modeling Information Systems in Business Development: Alternative perspectives on business process re-engineering. In: *Proceeding of IFIP TC8 conference Business Process Re-engineering: Information Systems Opportunities and Challenges*, (eds. Glasson et al), Elsevier Science B.V., North-Holland, pp. 567-579.

Tolvanen, J.-P., Marttiin, P., Smolander, K., (1993) An Integrated Model for Information Systems Modeling. In: *Proceedings of the 26th Annual Hawaii International Conference on Systems Science*, (eds. J. Nunamaker, R. Sprague), IEEE Computer Society Press, Los Alamitos, pp. 470-479.

Tolvanen, J.-P., Rossi, M., (1996) *A metamodeling approach to method comparison: A survey of a set of ISD methods*, WP-34, Department of Computer Science and Information Systems, University of Jyväskylä.

Tolvanen, J.-P., Rossi, M., Liu, H., (1996) Method engineering: current research directions and implications for future research. In: *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, pp. 296-317.

Turner, W. S., R. P. Langerhorst, G. F. Hice, H. B. Eilers and A. A. Uijttenbroek (1988) *SDM: system development methodology*, North-Holland.

Venable, J., (1993) *CoCoA: A Conceptual Data Modeling Approach for Complex Problem Domains*. Dissertation, State University of New York at Binghampton, United States.

Vepsäläinen, A., (1988) A Relational view of Activities for Systems Analysis and Design. *Decision Support Systems*, 4.

Verheijen, G., Van Bekkum, J., (1982) NIAM: An information analysis method. In: *Information Systems Design Methodologies: A Comparative Review* (eds. T. Olle, H. Sol, A. Verrijn-Stuart) North-Holland Publishing Company, pp. 537-589.

Verhoef, T., (1993) *Effective Information Modeling Support*. Dissertation, Delft University of Technology, The Netherlands.

Verhoef, T.F., ter Hofstede, A.H.M., Wijers, G.M. (1991) Structuring modeling knowledge for CASE shells. In: *Proceeding of CAiSE'91 Advanced Information Systems Engineering*, (eds. R. Andersen, J. Bubenko, A. Sølvberg) Berlin, Germany, Springer-Verlag, pp. 502-524.

Vitalari, N., Dickson, G., (1983) Problem solving for effective systems analysis: an experimental exploration. *Communications of the ACM*, 26, 11, pp. 948-956.

Vlasblom, G., Rijsenbrij, D., Glastra, M., (1995) Flexibilization of the methodology of system development, *Information and Software Technology*, Elsevier-Science B.V., 37, No. 11, pp. 595-607.

Walden, K., Nerson, J.-M., (1995) *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice-Hall.

Wand, Y., (1996) Ontology as a foundation for meta-modeling and method engineering. *Information and Software Technology*, 38, pp. 281-287.

Wangler, B., Wohed, R., Öhlund, S. -E., (1993) Business Modelling and Rule Capture in a CASE Environment. In: *Proceeding of the Fourth Workshop on The Next Generation of CASE Tools* (eds. S., Brinkkemper, F. Harmsen), University of Twente, The Netherlands, pp. 189-204.

Ward, P., Mellor, S., (1985) *Structured Analysis for Real-Time Systems*. Prentice-Hall, Englewood Cliffs, New Jersey.

Wasserman, A., (1980) Software tools and the user software engineering project, Technical report#46, University of California San Francisco, Medical Information Science.

Wastell, D., (1996) The fetish of technique: methodology as a social defence. *Information Systems Journal*, 6, pp. 25-40.

Waters, S.J., (1974) Computer-aided methodology of computer systems design. *The Computer Journal*, Vol. 17, 3, pp. 211-215.

Weber, R., Zhang, Y., (1996) An analytical evaluation of NIAM's grammar for conceptual schema diagrams. *Information Systems Journal*, 6, pp. 147-170.

Welke, R., Konsynski, B., (1980) An examination of the interaction between technology, methodology and information systems: A tripartite view. *Proceedings of the first International Conference on Information Systems*, pp. 32-48.

Welke, R.J., (1981) IS/DSS: DBMS support for information systems development, ISRAM WP-8105-1.0, McMaster University, Hamilton.

Welke, R.J., (1988) The CASE Repository: More than another database application, MetaSystems Ltd., Ann Arbor.

Welke, R.J., Forte, G., (1989) Meta Systems on Meta Models. *CASE Outlook*, 4, December, pp. 35-45.

Wijers, G., (1991) *Modeling Support in Information Systems Development*, Thesis Publishers Amsterdam.

Wijers, G., ter Hofstede, A., van Oosterom, N., (1992) Representation of information modeling knowledge. *Next Generation of CASE tools* (eds. K. Lyytinen, V.-P- Tahvanainen), IOS Press, Amsterdam, the Netherlands, pp. 167-223.

Wijers, G., van Dort, H., (1990) Experiences with the use of CASE tools in The Netherlands. In: *Advanced Information Systems Engineering,* (eds. B. Steinholtz, A. Sølvberg, L. Bergman), Springer-Verlag, Germany, pp. 5-20.

Wood-Harper, T., (1985) Research Methods in Information Systems: Using Action research. In: *Research Methods in Information Systems* (eds. E. Mumford, R. Hirschheim, G. Fitzgerald, A.T. Wood-Harper, Elsevier Science Publishers B.V. (North-Holland), pp. 169-191.

Wood-Harper, T., Antill, L., Avison, D., (1985) *Information Systems Definition: The Multi-View Approach*. Blackwell Publishers, Oxford.

Wynekoop, J., Conger, S., (1991) A review of computer aided software engineering research methods. In: *Information Systems Research: Contemporary approches and emergent traditions*, (eds. H.-E. Nissen, H.K. Klein, R. Hirchheim), Elsevier Science Publishers B.V., pp. 301-325.

Wynekoop, J., Russo, N., (1993) System Development Methodologies: Unanswered questions and the research-practice gap. In*: Proceeding of the 14th International Conference on Informtion Systems* (eds. J.J. DeGross, R.P. Bostrom, D. Robey), pp. 181-190.

Wynekoop, J., Russo, N., (1997) Studying system development methodologies: an examination of research methods. *Information System Journal*, 7, pp. 47-65.

Wynekoop. J., Senn, J., Conger, S., (1992) The implementation of CASE tools: an innovation diffusion approach. In: *The impact of computer supported technologies on information systems development*, (eds. K. Kendall, K. Lyytinen, J. DeGross), North-Holland, pp. 25-41.

Yin, R., (1993) *Applications of case study research*. Sage Publications.

Yourdon, E., (1986) What ever happened to structured analysis? *Datamation*, June, pp. 133-138.

Yourdon, E., (1989a) *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey.

Yourdon, E., (1989b) Pacific Bell, *American Programmer*, 2, No. 9 (September), pp. 3-6.

Yourdon, E., (1992) *The Decline and Fall of the American Programmer*, Prentice-Hall, Englewood Cliffs, NJ.

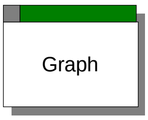
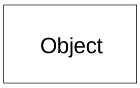
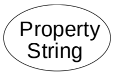
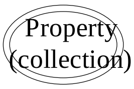
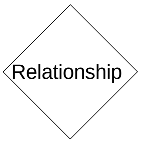
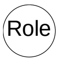
Yourdon, E., Constantine, L., (1989) *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, N.J., Prentice-Hall.

APPENDIX

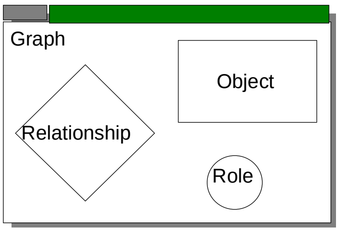
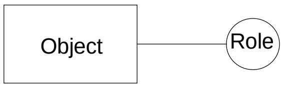
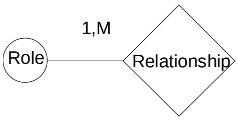
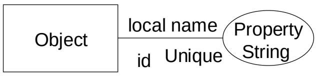
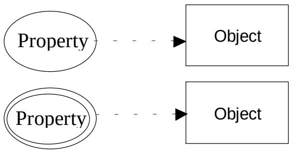
GOPRR metamodeling language

This appendix describes the GOPRR metamodeling language in two ways: by describing the notation for graphical metamodels, and by describing the textual set format used for reporting the metamodels here. A more detailed description of GOPRR can be found from the MetaEdit+ Method Workbench User's Guide (MetaCASE 1996b) and from (Kelly 1997).

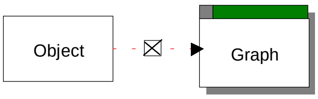
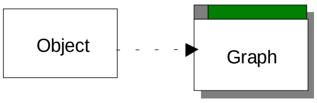
Graphical metamodeling constructs

Concept & representation	Description
Graph type 	A graph type is a collection of object, relationship and role types, and bindings describing how these can be connected. A graph type usually denotes a modeling technique, such as data flow diagrams or class diagrams.
Object type 	An object describes a thing that can exist on its own. Object type names are typically nouns. Examples include process, class, and attribute.
Property type  	Properties describe characteristics of instances of other types. Property type names are generally nouns or adverbs. Examples include class name, cardinality, and attributes. Each property type has a basic data type (e.g. number, string, Boolean, text, another type (graph, object, role or relationship), or a collection of one of these). A collection data type is represented with a double ellipse.
Relationship type 	A relationship can exists between objects. It connects objects through roles. Semantically, relationships are usually verbs, but relationship type names are sometimes also nouns or adverbs. Examples include inheritance, call, and usage.
Role type 	A role specifies how an object participates in a relationship. Semantically, roles are adverbs. Role type names are often prepositional phrases or verbs. Examples include subclass, from, and receives.

(continues)

Inclusion 	An inclusion relationship can exist between a graph type and its components (i.e. object, relationship, and role types). Inclusion is used to combine all the main components of a technique. Inclusion is many-to-many, so that the same type can belong to many graph types.
Participation 	An object type can participate in zero to many role types. In a graph type, a role type must be related to at least one object type.
Composition 	Relationship types are related with at least two compositions to roles. Together with a participation, this forms a binding (cf. Kelly 1997). Each role type in a binding is characterized with a cardinality constraint describing how many instances of this role type must (minimum) or may (maximum) occur in an instantiation of this binding.
Property of 	A property can characterize instances of other types (i.e. non-properties). This relation is described in a metamodel with the property of relationship. Each property of relationship is specified further with three constraints: 1) id to describe if the property type is used as a naming property (a non-property type can have only one id), 2) uniqueness to specify if there is no duplication of property values allowed among instantiations of this 'property of' relationship, and 3) local name to define a name for this use of the property type. Hence, two non-property types can refer to the same property type but with different labels; the labels are visible for example in dialogs for editing the properties of this non-property.
Property link 	The data type of a property type can be itself a non-property type. This is defined with a property link relationship from the property type to a non-property type.

(continues)

Explosion  	An object can be linked to one or more graphs via an explosion. Explosion is typically used between different graph types. Examples include that a process in a data flow diagram can be related to a state diagram and to process specifications.
Decomposition  	An object can be decomposed into a new graph. This feature is known as functional decomposition in data flow diagrams, or leveling of graphs to form a hierarchy. The decomposition target is typically of the same graph type as the source's containing graph. Note that only one decomposition is allowed for each object instance, and it applies in all graphs containing that object. In contrast, there may be a set of graph types specified as possible decomposition targets for an object type, and this set may be different in each graph type where this object type is used.

Set format of metamodels

The metamodels reported in Section 4 are made by querying the repository of MetaEdit+. The set format has been applied because MetaEdit+ does not use graphical metamodels for tool adaptation.

In the set format all types are described as sets, e.g. the object types of the whole SA/SD method are represented thus:

```
Object types = {Process, Store, External, Module, State, Entity}
```

The 'property of' relationship is described as a mapping of the set of property types which are associated with each non-property type (roles, objects, relationships). For example:

```
<organization, {organization name, Owner}>
```

Participation and composition are described as a binding: each binding stores a relationship, two or more roles, and for each role, one or more objects. Because a graph type can include several bindings they form a set.

```
Process/Entity Matrix={<Data usage,{<Used,{Entity}>,
                                    <Uses,{Business Process}>}>}
```

Inclusion is described for each technique only implicitly through the bindings, i.e. the non-property types included in the graph type can be found from the union of all binding members for that type.

Property links referring to non-property types which are not directly in an inclusion relationship in any graph type of this method are described by a pair containing a non-property and a set of properties.

```
<Attributes, {<Attribute, {Attribute name, Data type, Attribute
                           type, Initial value, Constraints,
                           Visibility}>}>
```

Explosions are described as set of pairs of an object type an d a set of graph types the object type may explode to.

```
Explosions ={<Process, {Structure Chart, State Diagram}>}
```

Decompositions are described as a set of pairs of an object type and a set of graph types the object type may decompose to.

```
Decomposition ={<Process, {Data Flow Diagram}>}
```

GOPRR metamodels

The metamodels made and adapted into MetaEdit+ and MetaEdit are available from MetaCase Consulting, http://www.metacase.com.

YHTEENVETO (FINNISH SUMMARY)

Tämän väitöskirjatyön tavoitteena on parantaa tietojärjestelmien suunnittelumenetelmien soveltuvuutta. Verrattuna moneen muuhun insinöörialaan tietojärjestelmien suunnittelumenetelmien yksi erityispiirre on niiden tilannekohtaisuus. Erilaisiin kehitysympäristöihin ja erilaisten tietojärjestelmien suunnitteluun soveltuvat erityyppiset menetelmät: esimerkiksi matkapuhelimien suunnittelussa menetelmätarpeet ovat erilaisia kuin suunniteltaessa www-sovelluksia tai logistisia prosesseja tukevia tietojärjestelmiä. Tilannekohtaisten menetelmien tarpeellisuutta korostavat uusien tietojärjestelmätyyppien ja käytettävissä olevien teknisten ratkaisuiden lisääntyminen. Empiiristen tutkimusten mukaan onkin varsin tavallista että tietojärjestelmien kehitystä harjoittavat organisaatiot ja yksittäiset kehitysprojektit muokkaavat menetelmiä omia käyttötilanteita varten.

Tässä väitöskirjatyössä esitetään periaatteita organisaatioiden suorittaman menetelmäkehityksen tukemiseksi. Menetelmiä tarkastellaan osana tietokoneavusteisia suunnitteluohjelmistoja. Nämä ohjelmistot tarjoavat tuen valitun menetelmän mukaiselle tietojärjestelmän kuvaamiselle, kuvausten ylläpidolle ja analysoinnille sekä tietojärjestelmän määritysten tuottamiselle. Työn ensimmäisessä osassa tarkastellaan olemassa olevia menetelmäkehityksen periaatteita ja menetelmäkehityshankkeita. Tarkastelu osoittaa heikkouksia menetelmäkehityksen periaatteissa jotka liittyvät menetelmien yksityiskohtaiseen määrittelyyn ja niiden tilannekohtaisen soveltuvuuden arviointiin.

Työn toisessa osassa keskitytään menetelmien mallintamiseen, eli metamallintamiseen, tutkimalla menetelmien yksytyiskohtaisen kuvaamisen kannalta tarpeellisia mallinnuskielten käsitteitä. Metamallinnuskielten käsitteitä etsitään analysoimalla joukkoa suunnittelumenetelmiä, kuvaamalla ne metamallinnuskielten avulla ja sovittamalla menetelmät muokattaviin suunnitteluohjelmistoihin. Löydettyjä käsitteitä sovelletaan määritettäessä menetelmiä ja arvioitaessa olemassa olevia metamallinnuskieliä.

Työn kolmannessa osassa tarkastellaan menetelmien tilannekohtaista soveltuvuutta ja esitetään periaatteita organisaatioiden menetelmätietämyksen luomiseksi ja ylläpitämiseksi. Esitetyt periaatteet perustuvat tietojärjestelmiä kuvaavien mallien ja menetelmiä kuvaavien metamallien väliseen vertailuun. Näiden periaatteiden käyttökelpoisuus havainnollistetaan kuvaamalla niiden käyttöä kahdessa tietojärjestelmien kehityshankkeessa: tukkukaupan ja metsäteollisuuden logististen tietojärjestelmien suunnittelussa. Molemmissa tapaustutkimuksissa käytettyjen suunnittelumenetelmien soveltuvuutta pystytään parantamaan työssä esitettyjen periaatteiden avulla. Työn tuloksia voivat hyödyntää kaikki suunnittelumenetelmiä käyttävät organisaatiot menetelmien soveltuvuuden parantamiseksi ja menetelmäosaamisen kehittämiseksi. §