How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com

Steven Kelly
MetaCase
Jyväskylä, Finland
stevek@metacase.com

ABSTRACT

Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution directly with domain concepts. Within product lines domain-specific approaches are applied to specify variability and then generate final products together with commonality. Such automated product derivation is possible because both the modeling language and generator are made for a particular product line — often inside a single company. In this paper we examine which kinds of reuse and product line approaches are applied in industry with domain-specific modeling. Our work is based on empirical analysis of 23 cases and the languages and models created there. The analysis reveals a wide variety and some commonalities in the size of languages and in the ways they apply reuse and product line approaches.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Model-driven software engineering**; **Domain specific languages**; *Abstraction, modeling and modularity*; System modeling languages; feature modeling.

KEYWORDS

Domain-specific language, domain-specific modeling, product line variability, product derivation, code generation

ACM Reference Format:
Juha-Pekka Tolvanen and Steven Kelly. 2019. How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3336294.3336316

1 INTRODUCTION

Domain-specific languages and models raise the level of abstraction beyond programming by specifying the solution directly with domain concepts. This is particularly suitable in product line development, as variability and related rules are part of the language used to create specifications of variants. The final products are then generated from these high-level specifications. This automation is possible because both the language and generators need fit the requirements of only this product line. A language-based approach for product lines manages a larger variability space, enables more freedom and flexibility and allows creating variations not possible with other approaches like parameter tables or feature models [4].

While Domain-Specific Modeling (DSM) solutions are widely applied [17, 19] and industrial cases using them are also included in the Product Line Hall of Fame[1], there are few studies analyzing which kind of languages have been created and applied in industry. Studies analyzing languages within product lines tend to focus on feature modeling languages, their extensions [3, 18, 25] and comparisons of them [1, 5, 22]. When domain-specific languages are addressed, the evaluation and comparison is focused on considering different languages structures for a given situation and product line (e.g. [13, 15]). Studies analyzing larger numbers of domain-specific languages have focused on approaches to finding language constructs [19] or identification of worst practices [9].

In this study we investigate which kinds of languages companies have created for their product lines. In particular we focus on how these DSM languages define variants and reuse these variant definitions. We take an empirical approach, analyzing 23 industry cases and the DSM languages created in them. Our analysis investigates the language definitions (metamodels), their target output (e.g. program code generated), the people involved in language creation, and the reuse enabled by the language both outside of and within the models. The analysis shows that the sizes of the languages vary greatly (the largest being 14 times larger than the smallest) and that the cases use a wide variety of language structures for managing variation.

In the next section we describe the product line cases and how the languages were analyzed. Section 3 describes and gives examples of the different approaches (modeling language structures and processes) for supporting product line development that we identified. Section 4 analyzes the data and evaluates this categorization of approaches, and Section 5 summarizes our conclusions.

2 ABOUT THE ANALYZED PRODUCT LINES

This study is based on analyzing 23 industry cases of domain-specific modeling applied to product lines. They were selected from cases over the last 15 years where the authors have had access to the language definition — often in a consultant role supporting language and generator creation, but not having the sole responsibility for their creation. All modeling languages and generators were thus implemented in MetaEdit+ [7, 12]. The language patterns recognized, however, are not limited to any particular tool. To avoid

[1]http://splc.net/hall-of-fame/

Table 1: Product line DSM cases analyzed

#	Domain	Targets	By	Size	Use	Approach
1	Consumer electronics	C, HTML, Docs	1		3	1
2	Industrial automation	PLC, GUI, DB schema, net config, deploy	1	165	5	1
3	Enterprise applications	C#, DB schema	1		6	1
4	Railway signaling	Simulation, XML	1	291	6	1
5	Signal Processing Systems	Matlab, simulation, XML	1		4	1
6	Oil drilling	Cost calculation, documentation	1		3	1
7	Big data applications	Java, JSON, CQL, SPARQL, SQL	2	397	4	1
8	Printing process	Ruby, XML, Docs	3	55	4	1
9	System performance	Gherkin, HTML, Docs	1	145	3	1
10	Consumer electronics	JSON	3	72	2	2
11	Telecom service	XML	1	61	2	2
12	Medical	XML, audit documents, change history	1	63	1	2
13	High-level synthesis	System C	1	450	3	2
14	Radio network	TTNC-3, simulation/animation	1		5	2
15	An automotive system	System specification	3	62	3	2
16	Database applications	Java	3	46	1	2
17	Consumer electronics	C, localization, docs	1	403	6	3
18	Automotive architecture	Simulink, ISO26262 documents, AUTOSAR	3	652	6	3
19	Telecom	C, build automation	2	109	3	3
20	Insurance	Cobol, DB schema	3	234	6	4
21	Aerospace	C#, XSD, JSON, API	2	121	1	5
22	Automotive ECU	Python, JSON, Test document, change history	3	64	5	5
23	Software testing	Propriety format of state machines	3	317	6	6

repeating ourselves, we chose only cases not already covered in our 2005 article [19].

All the language definitions were created freely by metamodeling, rather than being limited to being customizations, extensions or profiles of existing languages. Complete freedom was thus available when defining the language, and tooling did not restrict the language structures applied or created. Other approaches have also been seen in the literature on modeling and product lines: e.g. already available support for feature modeling could be applied directly or be extended in a preferred way, as suggested in [1, 5, 22]. Similarly, UML could be extended with stereotypes and profiles, or companies could consider defining both metamodel-based and profile-based as in [13].

The cases were chosen to cover different kinds of product lines from various industries: from consumer electronics through database applications to automotive and industrial automation systems. Table 1 summarizes these cases by their problem **Domain** and other features, in a roughly increasing order of model-level focus on product line aspects.

In all cases, DSM was applied not simply for planning, design or to support communication, but to automate development within a product line by performing model checking and generating expected output from the models: typically code but in some cases also other formal models, configurations, audit specifications and documentation. The main outputs generated are listed in Table 1 column '**Targets**'.

Languages were created with varying amounts of domain and language creation experience, as shown in Table 1 column '**By**':

(1) The organization in-house
(2) An external consultant with language creation expertise
(3) Both

The size of the DSM languages varied significantly: the smallest consisting of 46 language constructs and the largest 652 constructs. These values (where known) are shown in Table 1 column '**Size**'. As a comparison, UML 2.5 [14] has 247 constructs when implemented with the same GOPPRR meta-metamodel [7, 12].

The cases also varied in how far along the adoption path the language had progressed (shown in Table 1 column '**Use**'):

(1) Language still under definition
(2) Sample models made with stable language
(3) Significant modeling of real cases as test
(4) Real pilot project
(5) Production use
(6) Long term production use

Data on the use of the languages, such as how many variants within the product line had been developed, was not available for all cases. Among the cases where it was known, there was a wide variation: from just a few to hundreds of variants. Where data on the effort for creating and maintaining the languages in questions was available, it has been published in [21], and cross-tool comparisons of effort also exist, e.g. [11]. For details of the success and impact of using DSM in practice see [20, 24].

The final column in Table 1, '**Approach**', is described in the following section.

3 HOW LANGUAGES ADDRESSED VARIABILITY

The classification of the approaches for specifying variation was gathered mostly from the language definitions available to authors. The language definitions could thus be investigated directly in MetaEdit+, and sample variant models created. For the investigation we also analyzed the variant models created, and verified our understanding from the consultants or in-house developers who were involved in creating the languages and generators.

Common to all languages was that they described the variability and left most of the commonality to existing legacy, platforms, components etc. Often a domain framework is created alongside the language, extracting further commonality. Those commonality parts are then integrated via generators.

The analysis of the cases and their languages led us to identify six approaches for addressing variability and reuse. The categorization is mostly based on language definitions but also extends into questions of process, and some approaches may require particular tool support features. The main criteria for the identification were if the created models are reused among variants, and how that reuse was established and maintained. If certain commonly reused parts of variants were maintained centrally and provided to all modelers when creating a new variant, these were considered 'core' models.

Data from the cases indicated the following spectrum and categorization of '**Approach**' (the final column in Table 1):

(1) Each model and its elements are for a single variant
(2) Reuse of models or model elements across multiple variants
(3) Mark/filter/modify reused models or elements for variants
(4) Core models and variant models
(5) Core models and languages for restricted variation of core
(6) Multilevel: model elements become language elements

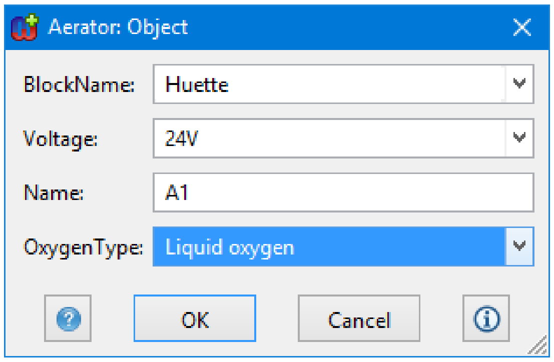
This list of approaches is not complete but reveals the typical approaches applied, some more common than others. The ordering follows our interpretation of how great a focus the approaches place on explicit product line support. In our experience the later approaches also often contain earlier ones, and as a language is developed and used, it may move further along the scale of approaches. In the following subsections we describe each approach in more detail and illustrate their use with practical examples (using real examples from cases where allowed).

3.1 Each model and its elements are for a single variant

The classical approach for applying domain-specific languages is described by [23]: The language focuses on describing the varying parts, whereas the common parts, defined in the framework, components etc., are outside the scope of the language. During the variant derivation phase the generator reads the models and integrates the variant design with the common parts in legacy code, frameworks etc.

A key feature of languages following this approach is that modelers focus on developing one variant at once: all changes made to the models are done within the context of a single variant. This approach is organizationally clear: the product development team owns all aspects of variability, has sole responsibility on testing, versioning etc. Such languages are typical in cases such as providing telecom services per operator, an industrial automation system per factory or a service per customer.

Fig. 1 and Fig. 2 illustrate this with a practical example [16]. A company manufacturing automation systems for fish farms creates a separate specification for each customer and their system. For every Aerator being delivered as a part of the system, the language allows the modeler to set specific values for the variant (or use defaults), e.g. for the voltage and what kind of oxygen is used. The language sets this variation space and with a simple example like in Fig. 1, this part of the language can be considered to be a configurator over the parameters of the whole product line offering.


Aerator: Object

BlockName: Huette

Voltage: 24V

Name: A1

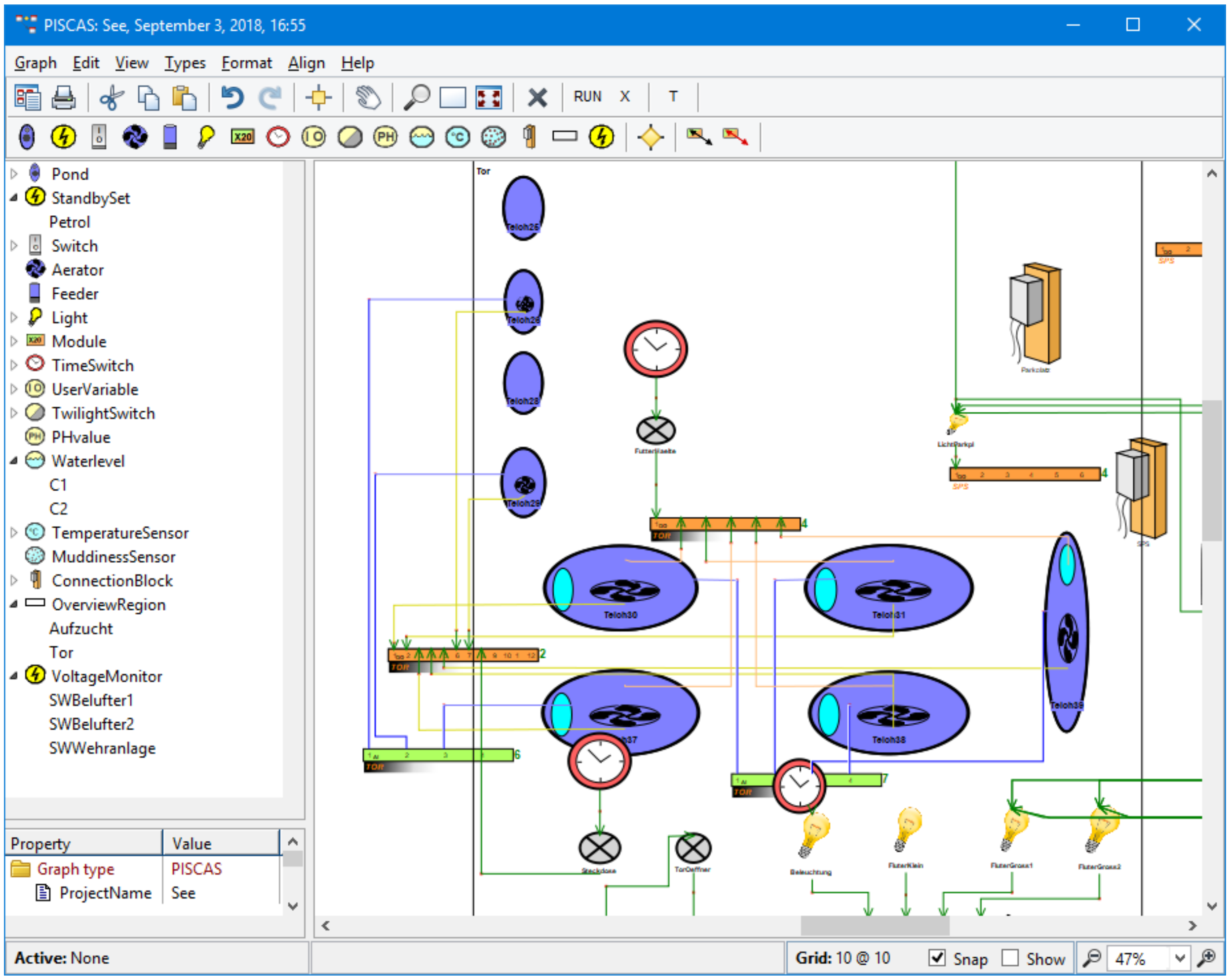
OxygenType: Liquid oxygen

Figure 1: Setting values for a variant of Aerator

In practice, all variation is not so simple. The unit of variability can be any aspect of the product line that can be expressed with a language. For instance, additional variation related to an Aerator can be its location, relation to fish ponds, network and power supply etc. DSM languages enable capturing this richer variation too, as illustrated in Fig. 2, which shows a portion of an automation system in which a particular Aerator is one varying part. For example, the blue ellipses are ponds for the fishes and one aspect of variability is their location. The variant also specifies other customer-specific features (lights, feeding, monitoring Ph level, muddiness, temperature etc.) and a network with related configuration. Location, order, connection and behavior are all kinds of variation that can be difficult or impossible to express with wizards, parameter tables or feature models, but fit well to DSM.

From the variant models like Fig. 2 the company produces PLC code for an automation system, database definitions for storing persistent data, network configuration for the given setup, installation guidelines, material needs etc. The location of the ponds is used to produce a user interface to monitor and control the automation system.

Languages focusing on a single variant are useful when variant development teams work independently and there is no need to share functionality created for one variant with others. If such a need arises without proper language support, often the only choice left is clone and own — and keeping track of changes made in the original variant model and copies.



Property	Value
Graph type	PISCAS
ProjectName	See

Active: None

Grid: 10 @ 10 [✓ Snap] Show 47%

Figure 2: Variant of fish farm automation for a given customer

3.2 Reuse of models or model elements in multiple variants

When more variants are being developed, often a need arises to reuse existing work: Functionality that is already defined for one variant is found useful for others too. To avoid clone and own, a sub-model or element is allowed to be referred to by more than one variant. (Having a hierarchy of models, and even allowing reuse of a sub-model, are also found in Approach 1, but there the reuse is always within a single variant.) As in Approach 1, the top-level model's name is often effectively the variant name.

Fig. 3 and Fig. 4 illustrate examples of reuse in the CPL language used to specify how calls are processed for a telecom operator's customers. Fig. 3 specifies one customer's call redirection service which uses voicemail redirection (subaction block at the bottom) based on location. Similarly, another customer's service for location-based redirection defined in Fig. 4 reuses the same voicemail redirection service in a more complex rule. The reused service is defined only once and applied in several variants. The variant models then detail

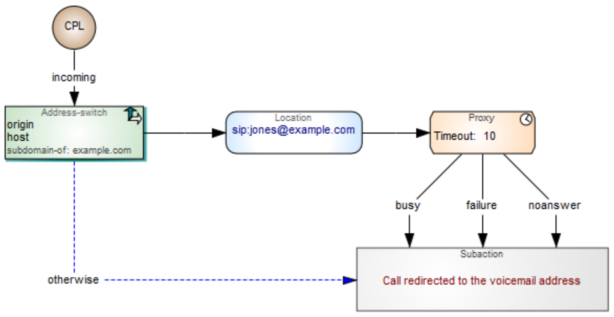


Figure 3: Location-based call redirection to voice mail

the contexts (e.g. busy, noanswer, etc.) in which the reused subaction is used. In other words, the language knows the correct ways to reuse the subaction and can guide and check variant creation during specification.

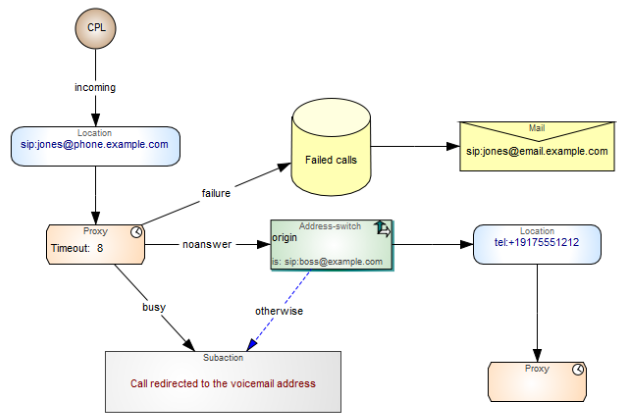


Figure 4: Call redirection reusing voice mail redirection

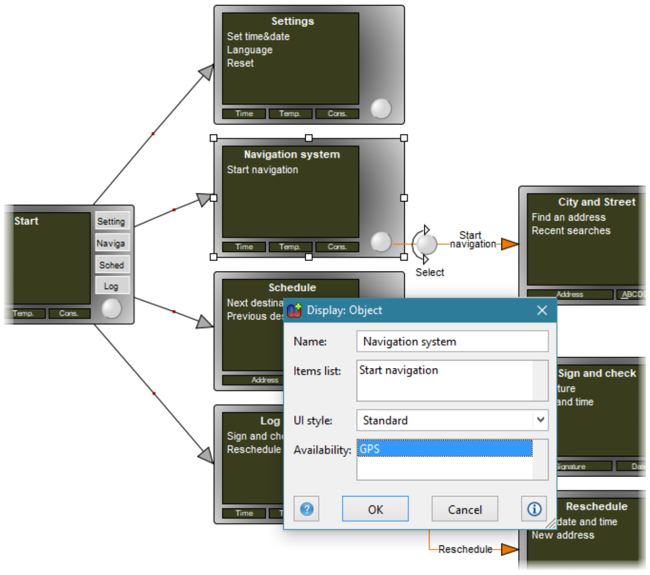
The reused service can be provided as a whitebox or a blackbox component depending on whether its details are made visible. Introducing model reuse raises similar questions as any other kind of reuse: Can reusers see the details, can they change them, is this reuse by reference or copy, etc. Also questions arise about the different lifecycles of reused parts and variant-specific parts: Are updates and bug fixes to reused parts delivered automatically to their users? Modeling tools may also offer support for governance and management with access rights, review policy etc. We discuss the role of tools for variability later in Section 3.4.

While this small example is based on the same language, i.e. reusable services are specified with the same language as all other services, it is possible to have different languages for different users (e.g. one for service creators at operator side and another for implementing the services, as presented in [6]). When languages are different among teams this also clarifies the responsibility of core team and variant development teams.

3.3 Explicit ability to mark, filter or modify models or elements for variants

Once elements can be reused as in Approach 2, there often arises a need to change those elements somewhat in particular variants. For example, model elements can be marked as only being present or active in certain variants (or other conditions), or a higher-level model can specify to include a lower-level model with certain filtering or configuration. The former can be considered bottom-up variant specification, and the latter top-down. In both cases, new concepts to explicitly specify variation are added to the language (in contrast to Approach 2, where the language remained the same.)

An example of bottom-up variant specification is seen in Fig. 5, which shows an excerpt of the UI and interaction flow of a car infotainment system. A definition for display 'Navigation system' is selected and its property dialog is shown. In the dialog, the Availability property value defines that this 'Navigation system' display is provided only when the GPS module is available. Also all other functionality related to the Navigation display like its menus, knobs etc. are excluded as well as any other displays reachable only via the Navigation display. While the variability is set here based


Display: Object

Name: Navigation system

Items list: Start navigation

UI style: Standard

Availability: GPS

Figure 5: Variation of infotainment system for GPS sensor

on the sensors available, it could be also based on product names (e.g. available only for infotainment system 'CarInfo3000'), features (e.g. available only for Autopilot) etc. A downside of this approach is that the properties like availability need to be entered — and later possibly modified — in multiple places (e.g. displays in this example).

Rather than adding variant information to individual language elements, the top-down approach extracts variant information to a new, higher model layer: A dedicated language is used to configure existing models defined for variants. Fig. 6 illustrates this top-down approach, showing a watch product family consisting of three members: Delicia, Ace and Sporty. While here one model specifies all three variants, each product could have a separate configuration model. In this example product line, each watch contains a display (on the left) and a logical behavior (on the right) whose submodel contains a number of applications, such as alarm or stopwatch.

One application, Stopwatch, is illustrated in Fig. 7. It allows the user to start the stopwatch running, showing the elapsed time and an icon while in that Running state; stop the stopwatch; and reinitialize the elapsed time. All three variants use the same specification of the Stopwatch application, yet in a different manner as specified in the higher-level family model. For example, Sporty has an icon for Timer but none for Stopwatch, so the stopwatch icon is not shown in Sporty when Stopwatch is running.

With this kind of language, variant developers can decide if existing variant models already contain the needed functionality and add a configuration using them. New products can thus be created quickly by referencing and configuring existing functionality. Alternatively, the development team can extend an existing functionality for the new variant but in a manner that the functionality for existing products is maintained.

With languages following this approach, the variant development teams take responsibility for the whole product line, not just their variant. The models created then need to be versioned and

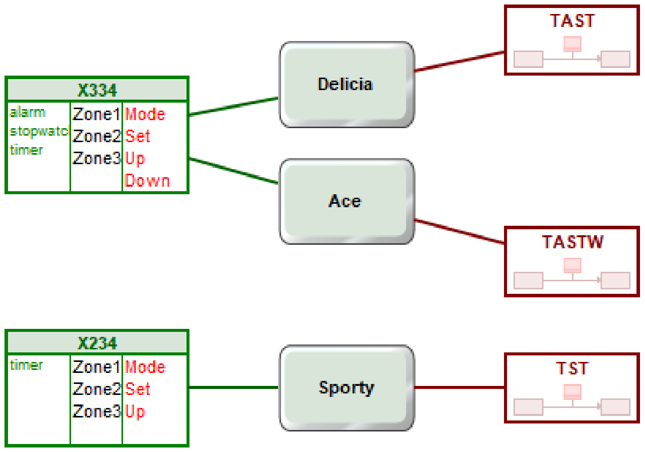


Figure 6: Watch family

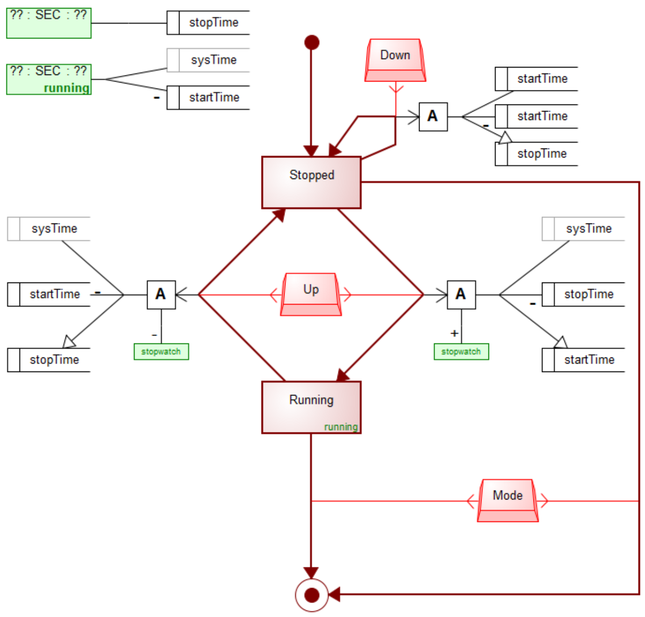


Figure 7: Stopwatch application supporting several variants (Ace, Delicia, Sporty)

tested together. This may well require establishing different policies for variant development teams than when using languages whose models each address only one variant.

3.4 Core models and variant models

Product line companies generally differentiate the platform and other commonalities, developed by the 'core team', and the individual variants developed by 'product development teams'. While DSM has generally already moved the core commonalities out of the modeling language, modeling several variants generally reveals some parts of variant models that are first reused in another variant (as in Approach 2), and later recognized as being useful to all variants. These 'core models' can then be supplied to each team starting a new variant, and can be maintained separately by a 'core modelers' team. This explicit recognition, decision and separation of core models is what distinguishes Approach 4 from Approach 2.

At least initially there are generally no clear differences in the domain itself on which parts can be core and which parts can be specified for a given variant, and so the modeling language can be the same for both teams. When languages do not address variation it is then left to the tooling and processes. Such tool-based approaches then vary based on tool functionality and their integration with other data and the other tools used.

Tools may provide in-built support for working with core models and variant models. For example, models can be defined in different repositories or projects within the modeling tool and their access can be restricted for different teams. Typically product development teams may not be allowed to change core models or their individual elements but just use them. For example, in a MetaEdit+ multi-user repository, a project's model elements can be set to be read-only, so that only members of the core team can change them and others may only refer to them in their designs. When the core changes, the variant development teams see the updates automatically.

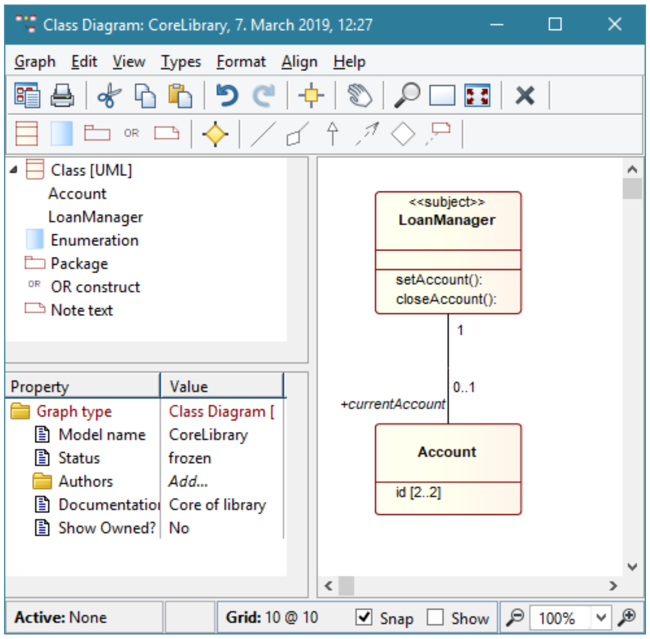
Alternatively, it is possible that core models are maintained separately and the core team can release them with dedicated content and versions. For example, different versions of the core models can be released to different variant development teams or different sets of core models can be released to different variant teams. Variant development teams may then also decide when to update to the next version of the core.

Tool support is needed to manage sharing and updating core models for teams applying them in product development. For example, in MetaEdit+, a core team can release models to be imported in separate repositories used by product development teams — and different rights can be set if changes to them are allowed or not. When the core models change, the core team can release the changed models and choose that they will automatically update previously delivered core models. This approach is applicable if there are subcontractors involved that do not get access to all designs, or development teams are working remotely in different locations, or there is otherwise a need to keep variant models made for different customers separate (as in Approach 1).

3.5 Core models and languages for restricted variation of core

Among the cases analyzed we also found examples where modifications to the reused core were allowed but in a restricted way. The need for modification and the type of modifications allowed were identified during domain analysis and language definition. This approach is illustrated with a class diagram using an example of library variants from [2]. Fig. 8 shows a small core model in MetaEdit+ containing just two classes that are common for all library systems. This model is created by the core development team.

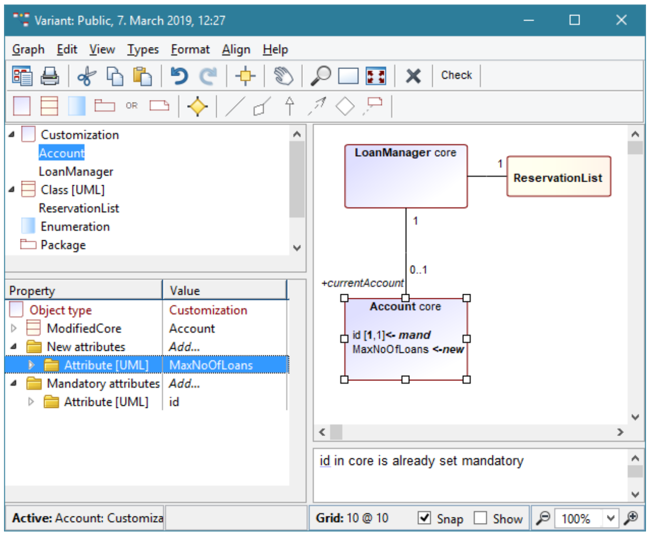
A variant development team then creates a product using a modeling language that restricts the modifications to the core models. Fig. 9 illustrates the use of this language. A variant of library system for public libraries uses both 'LoanManager' and 'Account' from the core, but allows adding new functionality created just for the



Property	Value
Graph type	Class Diagram [
Model name	CoreLibrary
Status	frozen
Authors	Add...
Documentatio	Core of library
Show Owned?	No

Figure 8: A small example of core model for a library

variant, like 'ReservationList' in the example. More importantly, the 'Account' class from the core has been modified in two ways — as allowed by the modeling language. A new attribute for maximum number of loans has been added and an existing attribute for identification is made mandatory. The notation of the language makes clear the parts used from the core and the type of modifications that have been made to them. Both the notation and the kinds of modifications allowed are set by the language creators.



Property	Value
Object type	Customization
ModifiedCore	Account
New attributes	Add...
Attribute [UML]	MaxNoOfLoans
Mandatory attributes	Add...
Attribute [UML]	id

id in core is already set mandatory

Figure 9: Creating a variant by customizing the core

With this approach the core parts can evolve and be updated, yet at the same time their modification in variant projects is possible.

The tooling can also check for possible inconsistencies and report them as illustrated in Fig. 9. Here the variant development team has made the 'id' attribute mandatory but it has later been changed to be mandatory in the core too. MetaEdit+ checks the variation rules and reports on possible inconsistencies: See the error text below the diagram stating 'id in core is already set mandatory'. Variability rules defined are not only applied when creating the models but also while editing, generating variants etc.

While this example is very small and shows just three ways to extend the core, the same principle for setting restrictions on how core models can be modified can be applied for other kind of extension rules and for other, more domain-specific, languages than the class diagrams shown here.

3.6 Multilevel: model elements become language elements

Finally, in one particular product line DSM case we identified a pattern known as multilevel modeling: A model created in one role or by certain developers is used to form language elements for the others. This is illustrated below with a small example on sensors available for a variant. Fig. 10 shows a model of possible sensors and their characteristics, such as what they measure and how they can be applied, e.g. with continuous polling or low energy usage. In the multilevel approach, a model specifies here capabilities of the device been used. For example, there is a polling sensor for orientation providing a compass direction.

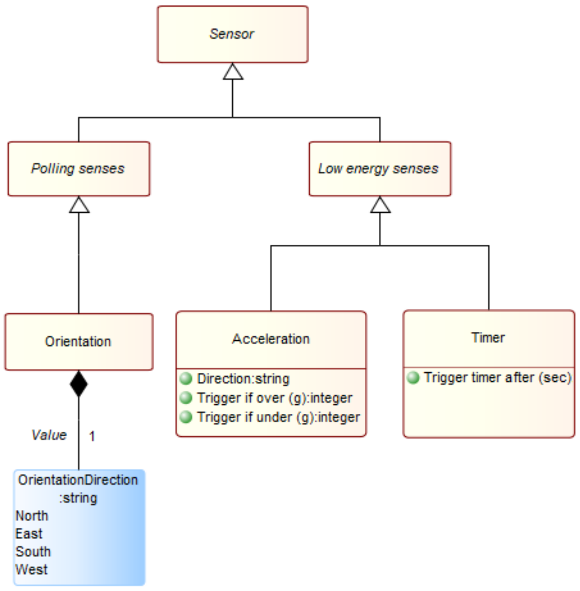
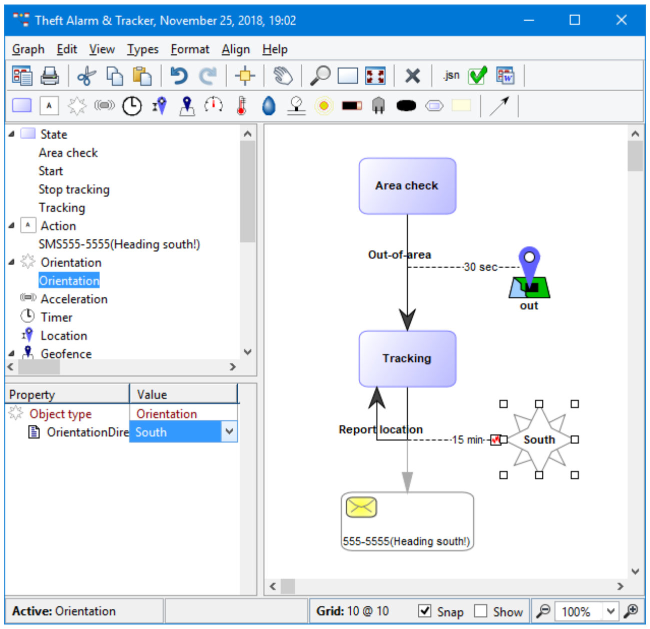

Sensor
Polling senses
Low energy senses
Orientation
Acceleration
Direction:string
Trigger if over (g):integer
Trigger if under (g):integer
Timer
Trigger timer after (sec)
Value
1
OrientationDirection :string
North
East
South
West

Figure 10: Adding an Orientation sensor language concept

This model then forms a part of the language used to specify applications for the given device variant. Fig. 11 illustrates this language being used to specify tracking using an Orientation sensor. As it is polling-based, the model states that data from the Orientation sensor is read every 15 minutes. If the compass direction is

South, the bottom transition is triggered and a message is sent to the given phone number. All available language concepts — including the newly-added Orientation sensor — are visible in the toolbar of the modeling tool. Refining the sensors in Fig. 10 thus updates the language being used for variant development in Fig. 11.



Property	Value
Object type	Orientation
OrientationDire	South

Figure 11: Using an Orientation sensor in a variant

As in the previous two approaches, there is a clear possibility for having different kinds of users or roles. Changing the language also opens up the normal tool support questions of language evolution: fortunately one of the strengths of the tool in this case [10].

4 EVALUATION OF THE CASE DATA AND PRELIMINARY ANALYSIS

Among the cases analyzed, the commonest approaches were Approach 1: languages for a single variant (9 cases) and Approach 2: languages designed for reusing models or model elements among variants (7 cases). The large number of languages focusing on single variants is understandable as they are the simplest in terms of process, and proven practices for creating such languages have long been available (e.g.[23]). The entire support for product line engineering is provided by the language and generators, with no explicit mention of variants, and no cross-variant reuse in the models.

Our experience that the Approaches form an ordering received some support and no refutation from the data. Two cases which originally focused on single variants (Approach 1) later encountered the need for reuse and moved to Approach 2 (#10, #11), and two others to Approach 3 (#17, #19).

When analyzing how the languages were developed, we identified that almost all languages developed in-house, without support from external consultants, fell into Approaches 1 and 2. A one-tailed Pearson correlation of 0.52 was found between increasing explicitness of product line approach and increasing categories of language creator expertise involved, statistically significant with $P < 0.01$. No correlation (0.04) was found between language creators and how far the language progressed along the path to full production use.

It is clear from Table 1 that the sizes of the languages vary considerably: the largest is 14 times the size of the smallest. The data also showed that among the largest languages were those that have been applied for a long time (#17, #18, #20, #4). A one-tailed Pearson correlation of 0.60 was found between Size and Use, statistically significant with $P < 0.005$. No correlation (0.07) was found between Size and Approach.

We also investigated whether the type of modeling language chosen had an effect on Size, Use or Approach. Language types were classified based on their main focus or model of computation as Location, Costs, UI, Query, Structure, State machine, UI flow, Data flow, Action flow, Sequence and Data structure. A single case often included two language types. Although all cases using the simplest four language types at the start of this list also used the simplest single-variant Approach 1, the effect was not statistically significant because of the small sample size for those language types (only one or two cases for each of those language types).

5 CONCLUSIONS

In this study we analyzed 23 industry cases where domain-specific modeling languages were created for product line development. In particular we investigated how domain-specific languages handle reuse of the variant specifications. The cases were selected to cover different kinds of product lines from various industries. The oldest languages included in the analysis have been used for over 10 years and the newest only just created.

The analysis of languages focused on their definitions (metamodels), along with broader information about the area targeted and the creation and adoption process. We applied the languages in a tool to investigate how variant specifications can be created and reused.

In the cases studied, we identified six different approaches to supporting product line development in DSM. With even the most basic DSM language already supporting product line development implicitly, it was interesting to see that over half the languages were developed to provide further product line support. A third of the languages added reuse of variant models or model elements, and another third offered various kinds of explicit modeling of variation. The list of approaches is obviously not complete, but it gives an indication of the richness of language structures that can be applied to support product lines with DSM.

The majority of these languages were created in-house, without support from external consultants. In the cases where other language definition approaches were applied external consultants were involved. This can be explained that these cases called for more experience on language design (like adding configuration for variability) or even on the particular tool, at least at the highest end (support for multi-level modeling).

More research work is needed to analyze the benefits of each approach, its costs in terms of the creation and maintenance effort, and what aspects within the case have led to the chosen approach. Future work can also extend the number of cases analyzed, and cover DSM solutions created by other tools. We are not aware of studies investigating numerous industry cases that used other tools, and welcome such studies addressing language creation with other tools and technologies. Other research methods could also be applied, e.g. giving more detailed analysis of language evolution within an individual case. Conversely, the scope could be widened by a survey.

Perhaps the most encouraging aspect of this research was to see how many cases were able to use MetaEdit+ to successfully implement a product line approach with DSM, even with only in-house resources. With so much development still taking place by clone and own, any approaches and tools that can offer organizations a reliable path to product line development are welcome.

REFERENCES

[1] Acher M., Heymans P., Collet P., Quinton C., Lahire P., Merle P., Feature Model Differences. In: Ralyté J., Franch X., Brinkkemper S., Wrycza S. (eds) Advanced Information Systems Engineering. CAiSE 2012. Lecture Notes in Computer Science, vol 7328. Springer, 2012

[2] Atkinson, C., Bunse, C., Bayer, J., Component-based Product Line Engineering with UML, Pearson Education, 2002

[3] Cognini, R, Corradini, F., Polini, A., Re, B., Extending Feature Models to Express Variability in Business Process Models, In proceedings of Advanced Information Systems Engineering Workshops, Springer, 2015

[4] Czarnecki, K., Eisenecker, U., Generative Programming, Methods, Tools, and Applications, Addison-Wesley, 2000

[5] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A., Cool features and tough decisions: a comparison of variability modeling approaches. In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12). ACM, 2012

[6] Hulshout, A., Service Creation with MetaEdit+: A telecommunications solution. Presentation at Code Generation Conference, Cambridge, May 19th, 2007

[7] Kelly, S., Lyytinen, K., Rossi, M., MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos P., Mylopoulos J., Vassiliou Y. (eds) Advanced Information Systems Engineering. CAiSE 1996. Lecture Notes in Computer Science, vol 1080, Springer, 1996

[8] Kelly, S., Tolvanen, J.-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press, 2008

[9] Kelly, S., Pohjonen, R., Worst Practices for Domain-Specific Modeling, IEEE Software, Vol. 26, 4, 2009

[10] Kelly, S., Tolvanen, J.-P., Collaborative creation and versioning of modeling languages with MetaEdit+. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '18), Babur, Ö., Strüber, D., Abrahão, S., Burgueño, L., Gogolla, M., Greenyer, J., Kokaly, S., Kolovos, D., Mayerhofer, T., Zahedi, M. (eds.). ACM, pp. 37–41, 2018

[11] El Kouhen, A., Dumoulin, C., Gerard, S., Boulet, P., Evaluation of Modelling Tools Adaptation. CNRS HAL hal-00706701, 2012, http://tinyurl.com/gerard12

[12] MetaCase, MetaEdit+ Workbench 5.5 User's Guide, www.metacase.com, 2018

[13] Mewes, K., Domain-specific Modeling of Railway Control Systems with Integrated Verification and Validation, Ph.D. thesis, University of Bremen, 2009

[14] Object Management Group, Unified Modeling Language, Version 2.5.1, 2017

[15] Preschern, C., Kajtazovic, N., Kreiner, C., Evaluation of Domain Modeling Decisions for Two Identical Domain Specific Languages, Lecture Notes on Software Engineering 2, 1, 2014

[16] Preschern, C., Leitner, A., Kreiner, C., Domain-Specific Language Architecture for Automation Systems: An Industrial Case Study, In: Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications (eds. Störrle et al.) DTU Informatics, 2012

[17] Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D., What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, Vol. 26 , 4, 2009

[18] Sousa, G., Rudametkin, W., Duchien, L., Extending feature models with relative cardinalities, Proceedings of the 20th International Systems and Software Product Line Conference, Beijing, China, ACM, 2016, https://doi.org/10.1145/2934466.2934475

[19] Tolvanen, J.-P., Kelly, S., Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. Proceedings of the 9th International Software Product Line Conference, Obbink, H., Pohl, K. (eds.). Springer-Verlag, LNCS 3714, 2005

[20] Tolvanen, J-P. and Kelly, S. Model-Driven Development Challenges and Solutions - Experiences with Domain-Specific Modelling in Industry. In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, SCITEPRESS Science and Technology Publications, Lda, 2016

[21] Tolvanen, J.-P., Kelly, S., Effort Used to Create Domain-Specific Modeling Languages. In ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS 18), ACM, New York, NY, USA, 2018

[22] Wahyudianto, Budiardjo, E., Zamzami, E., Feature Modeling and Variability Modeling Syntactic Notation Comparison and Mapping, Journal of Computer and Communications, 2014

[23] Weiss, D., Lai, C.T.R., Software Product-line Engineering, Addison Wesley, 1999

[24] Whittle, J., Hutchinson, J., Rouncefield, M., The State of Practice in Model-Driven Engineering, IEEE Software, 31, 3, 2014

[25] Zaid, L, Kleinermann, F., Troyer, O., Feature Assembly: A New Feature Modeling Technique, Proceedings of 29th International Conference on Conceptual Modeling, Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.). Springer, 2010