Visual Modeling of Self-Adaptive Systems

Saivignesh Sridhar Eswari
Software Designer at Nobleo
Eindhoven, Netherlands
s.e.saivignesh@gmail.com

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

Emil Vassev
SEDEV Consult Ltd
Sofia, Bulgaria
emil@vassev.com

***Abstract*—When developing autonomous systems, designers employ different kinds of knowledge to specify systems. We present a visual modeling approach created for specifying self-adaptive systems. The approach uses model-based approach to specify the system context and ontology addressing both structural and behavioral parts. The resulting models are used with code generation with knowledge reasoning frameworks and tools. The presented approach supports collaboration and communications within the safety design team, improve productivity of the team and reduce the cost of software certification.**

Keywords—autonomous systems; self-adaptive systems; visual modeling; model-based development; domain-specific languages; KnowLang; MetaEdit+

I. INTRODUCTION

Autonomous vehicles have to be safe and reliable. While certification programs and safety standards such as ISO 26262 provide guidance, safety design and development of safe and reliable functionality is time-consuming and costly. Moreover, the integration and promotion of autonomy in vehicles is an extremely challenging task, and although autonomous cars are already seen on our streets, the first severe accidents prove that they are not as secure as we had hoped them to be.

We present a visual modeling solution developed to meet recommendations for using model-based approaches in safety design and the trend on automotive industry on using code generation from visual models. The presented approach is based on the KnowLang framework [1] developed for Knowledge Representation and Reasoning (KR&R) in self-adaptive systems. The modeling approach consists of a set of integrated visual models, each providing a particular view of the system, such as its overall context, structures and their relationships, along with specific behavior. The visual modeling approach makes the system's description a relatively easy task where models support communication and gathering feedback within the team. The visual modeling approach also relies on capabilities to manage complexity, such as partition structure from behavior by providing different user-adapted views of the system (e.g. overall and detailed) and by providing possibility to have different views to the systems (e.g. view only inheritance among structural elements). Another key part is tooling that enables collaborative model-based development: several engineers can edit the same specifications simultaneously with continuous integration.

From the model-based specifications the implemented generator produces code for the KnowLang framework for further analysis and execution. This automates the routine and makes the KnowLang framework better accessible for engineers so that they can focus on safety design. The models can also be used for reporting, providing different views for different stakeholders and for documenting the system. By using visual modeling along with automatic code generation, we practically reduce both development time and effort, decrease certification costs and improve development productivity.

In this paper, we present KnowLang framework along with the developed visual modeling approach and code generator. This work was done to meet needs of an automotive company. We describe the process of creating the tooling and show practical cases and examples of using the modeling approach when developing various self-adaptive systems.

II. SELF-ADAPTATION AND KNOWLEDGE REPRESENTATION

Autonomous systems, such as automatic lawn mowers, smart home equipment, driverless train systems, or autonomous cars, perform their tasks without human intervention.

A. KnowLang

KnowLang [1,2,3,4] is a framework for KR&R that aims at efficient and comprehensive knowledge structuring and awareness [5] based on logical and statistical reasoning. Knowledge specified with KnowLang takes the form of a Knowledge Base (KB) that outlines a Knowledge Representation (KR) context. A key feature of KnowLang is a formal language with a multi-tier knowledge specification model (see Fig. 1) allowing integration of ontologies together with rules and Bayesian networks [6].

The language aims at efficient and comprehensive knowledge structuring and awareness. It helps us tackle [2]: 1) explicit representation of domain concepts and relationships;

2) explicit representation of particular and general factual knowledge, in terms of predicates, names, connectives, quantifiers and identity; and 3) uncertain knowledge in which additive probabilities are used to represent degrees of belief. Other remarkable features are related to knowledge cleaning (allowing for efficient reasoning) and knowledge representation for autonomic behavior.

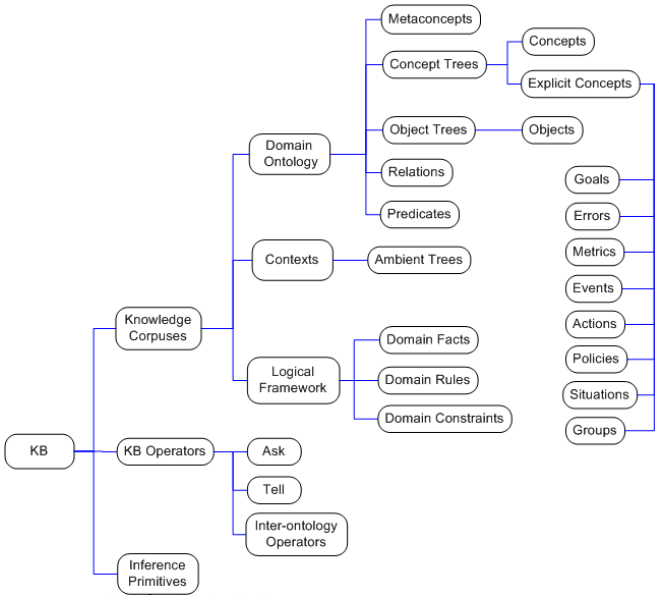

Fig. 1. KnowLang Specification Model.

By applying KnowLang's multi-tier specification model (see Fig. 1) we build a Knowledge Base (KB) structured in three main tiers [1, 2]: 1) Knowledge Corpuses; 2) KB Operators; and 3) Inference Primitives. The tier of Knowledge Corpuses is used to specify KR structures. The tier of KB Operators provide access to Knowledge Corpuses via special classes of *"ask"* and "tell*"* Operators where "ask" Operators are dedicated to knowledge querying and retrieval and "tell" Operators allow for knowledge update. When we specify knowledge with KnowLang, we build a KB with a variety of knowledge structures such as ontologies, facts, rules and constraints where we need to specify the ontologies first in order to provide the vocabulary for the other knowledge structures.

A KnowLang ontology is specified over *concept trees*, *object trees*, *relations* and *predicates*. Each concept is specified with special properties and functionality and is hierarchically linked to other concepts through "parents" and "children" relationships. For reasoning purposes every concept specified with KnowLang has an intrinsic "state" attribute that may be associated with a set of possible state values the concept instances may be in. The concept instances are considered as objects and are structured in object trees - a conceptualization of how objects existing in the world of interest are related to each other. The relationships in an object tree are based on the principle that objects have properties, where the value of a property is another object, which in turn also has properties. Moreover, *concepts* and *objects* might be connected via *relations*. Relations are binary and may have *probability-distribution* attribute (e.g., over time, over situations, over concepts' properties, etc.). *Probability distribution* is provided to support probabilistic reasoning and by specifying relations with probability distributions we actually specify Bayesian networks connecting the concepts and objects of an ontology.

B. Knowledge Representation

When developing autonomous systems, designers employ different kinds of knowledge to derive models of specific domains of interest. There's no standard classification system - the problem domain determines what kinds of knowledge designers might consider and what models they might derive from that knowledge [7]. Designers can use different elements to represent different kinds of knowledge. Knowledge representation (KR) elements could be primitives such as *rules*, *frames*, *semantic networks* and *concept maps*, *ontologies* and *logic expressions* [7]. These primitives might be combined into more complex knowledge elements. Whatever elements they use, designers must structure the knowledge so that the system can effectively process it and humans can easily perceive the results.

In the dynamically changing automotive industry, designers need to achieve optimized designs and successful validation earlier in the automotive engineering process. Many adopt advanced automation technologies based on model-driven development to meet this challenge. Note that various model-based approaches provide automotive engineering software for design, simulation, verification, and manufacturing, allowing one to create a digital model that drives the entire product development process. Advanced analysts and designers can use analysis and simulation solutions for kinematics, dynamics, structural, thermal, flow, motion, multi-physics, and optimization in a single environment. Seamless sharing of model data between design and analysis delivers results quickly, to impact critical design decisions. The use of visual models was also a requirement from the company for a tool for specifying self-adaptive behavior. The use of visual models is also backed by empirical research in particular when investigating studies on quality of the specification, effectiveness and efficiency. As an example, Jakšić et al. [8] performed a statistical analysis for comparing the quality, efficiency and productivity between textual representation and graphical model-based representation. They focused on feature trees applied in product lines which resemble perhaps the closest the concept trees of KnowLang. The result of the empirical study was that graphically created specification was more complete and of better quality than the textually specified ones. Also graphical modeling took less time than creating the same feature model with textual specification.

C. Visual Modeling Tools

While it is possible to create tools from the scratch, we applied Language Workbench approach providing most of the needed functionality automatically: Only support for KnowLang language, its model-based visualization, checking correctness of the specifications, code generation and integration with other tools was added. MetaEdit+ [10] was applied as the tooling as it satisfied the requirements of the automotive company. These included support for collaborative

modeling, version control, integration with relevant tools applied in automotive (e.g. Simulink, HiP-HOPS), updating both models and metamodels, as well as availability of supporting services. Naturally tool support was expected for visual modeling and implementing the generators for integration with KnowLang and other targets.

MetaEdit+ provides tools that enable developing modeling support iteratively without programming. Language definition and language use happens in the same environment allowing immediate testing of the language definition and update it based on using the language. The language definition follows the process of:

1) Defining the language concepts used to create the models.
2) Setting the rules for these concepts allowing preventing creating illegal or unwanted specifications.
3) Defining the visual notation that is used when editing and reading the models. Notation can also show information on incompleteness, model references etc. that are not directly related to specification itself.
4) Implementing the generators that produce the required artifacts like code, simulation data, tests etc.

At any step of this process the language definition can be applied and tried out. This is also possible with multi-user version of MetaEdit+: Language engineers can define the language and others may at the same time use the modeling language. The feedback loop between language definition and language use helps to reduces errors, minimize the risks of creating unwanted language features and improve user acceptance.

III. Model-Based Development for KnowLang

The visual modeling support for KnowLang was implemented by one person. The implementation was done incrementally and the results were reviewed by three persons. The implementation was done during spring 2017 within a period of three calendar months.

A. Defining and Formalizing Concepts

The language definition started by identifying the different visual views for KnowLang specifications: *Concept trees* expressing domain ontology, *predicates* to express complex system states, *contexts* to specify environment or situation in which the concepts are, and *behavior* expressed with Boolean expression.

Since the concepts of KnowLang were already defined (see Section II.A) the metamodeling process largely means mapping the KnowLang concepts to the visual modeling concepts, such as to objects, their relations, roles and properties. Fig. 2 shows a definition of Concept trees and its modeling elements. These include various concepts used as modeling objects, their inheritance and probability based relations shown as relationships, and roles for defining how objects participate with the relationships.

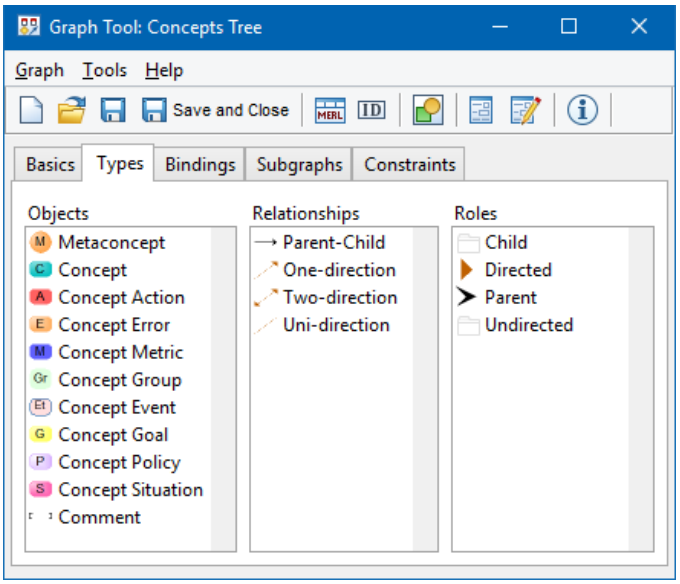

Fig. 2. Definition of Concept tree in KnowLang

The elements modeled for representing Concept Trees are *Metaconcept*, *Generic Concept*, *Explicit Concepts* and *Relations*. Each of the modeling elements shown in Fig 2 are defined in further detail. Fig. 3 shows one such definition: the Concept with its properties. The description in the bottom of window is used in the help system available for the modeler using KnowLang.

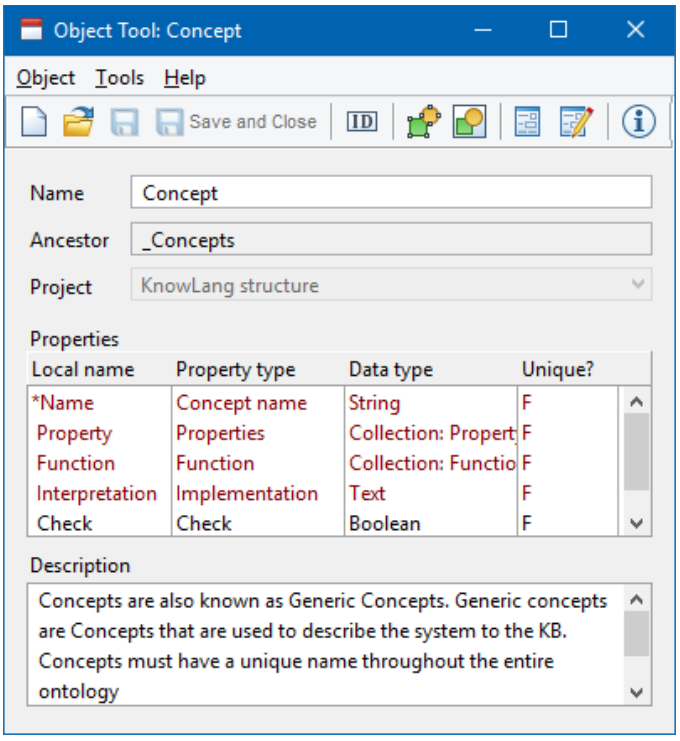

Fig. 3. Definition of Concept of KnowLang

Once defined, each part of the language specification was tried out to specify reference systems. Other language concepts of Concept trees are defined similarly. Also other

views of KnowLang, such as behavior and complex states, were defined similarly.

Since we were defining a visual language also the resulting language definition differs from the grammar definition used for textual languages. In visual models a particular element, such as 'Passenger' Concept can be entered only once and referred elsewhere from the specification - also from other places than in the same visual diagram. Thus change in one diagram is reflected to everywhere without the need for find and replace or using refactoring tools as when working with plain text. Similarly automated trace, such as where a particular 'Passenger' concept is used is directly available. This helps traceability and providing documentation reports. Visual language also provides views and separation of concerns for knowledge representation as well as possibility to view and filter specification in different level of detail or for different audience. For example, one might be interested to view plain concept inheritance whereas others their connections. This notation part is discussed in Section III.C.

B. Defining Rules

Each modeling element, such as *Concept Trees* or *Concepts*, illustrated above, may have rules and constraints. For instance, names of concepts may be unique with the concept tree, or inheritance between the concepts may allow multiple inheritance. If such rules are defined into the metamodel they can be checked at the modeling time preventing creation illegal or unwanted specifications. As it is cheaper to prevent errors happen rather than correct them later, we added to the metamodel also various model checking rules. For the metamodel definition MetaEdit+ provides ready rule templates as applied for KnowLang definition in Fig. 4 and 5.

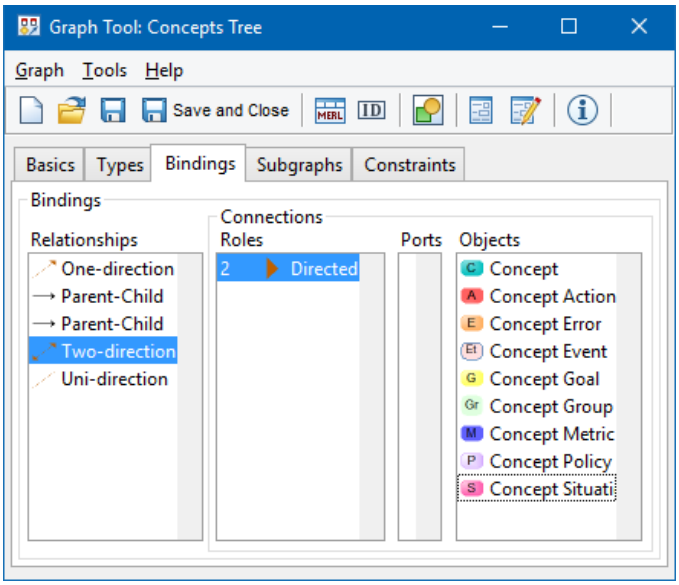

Fig. 4. Binding rule for directed relationship.

Fig. 4 illustrates the definition of directed relationship between a set of objects. *Two-directed* relationship must have always at least two *Directed* roles when connecting any of the defined objects listed. Fig. 5 shows a definition of uniqueness constraint that each concept must have a unique name within a concept tree. Similarly rules for mandatory naming, legal connections, number of connections etc. were added to the metamodel.

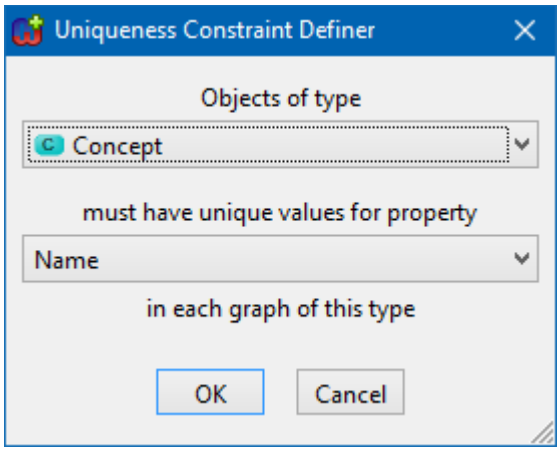

Fig. 5. Uniqueness rule for naming

Rules for all other parts of KnowLang were added similarly. As we divided the visual presentation to different views the metamodel was finalized with interlinking the views. In most cases such linking appeared automatically as in MetaEdit+ the model elements can be reused and linked between the views. For other cases, like organizing the model hierarchically, the metamodel definition was extended with linking rules. Fig. 6 illustrates some linking rules, such as that *Concept* may have a *State chart* and *Action Concept* may have *Pre-conditions* and *Post-conditions* been defined in own views.

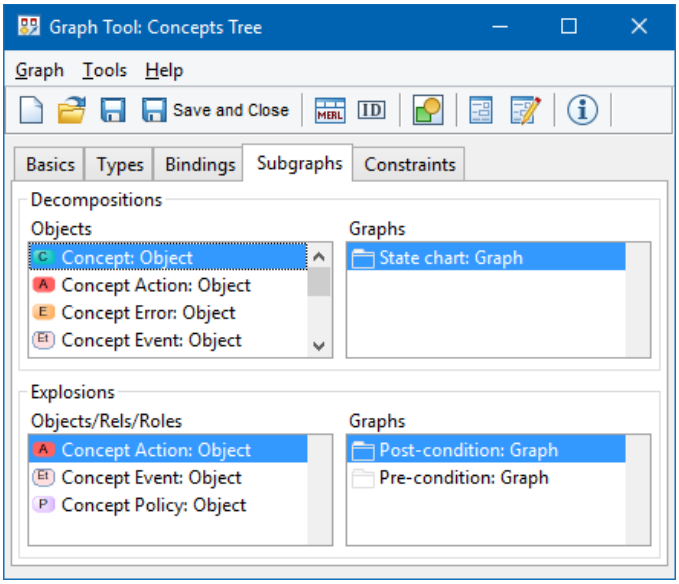

Fig. 6. Explicit rules for linking modeling elements with views

C. Defining Notation

Visual modeling requires a concrete syntax. We defined the syntax following the KnowLang presentation material and extended it with visual properties to gather summary

information, modeling linking data and error annotation. We applied various visual variables for the notation, such as shapes, colors, fonts, as guided by [9] to improve readability, understanding and working with the specification models. Fig. 7 illustrates definition of the notation for a *Concept* element. The notation is defined with Symbol Editor of MetaEdit+. Alternatively existing visualizations could be imported and applied.

Initially the notation provided just the basics: A green rectangle showing the unique name of the concept. To manage different views, the definition was extended with a visual clue on the upper right corner to indicate if concept has associated subgraph. This visualizes the rule of the language defined in Fig 6.

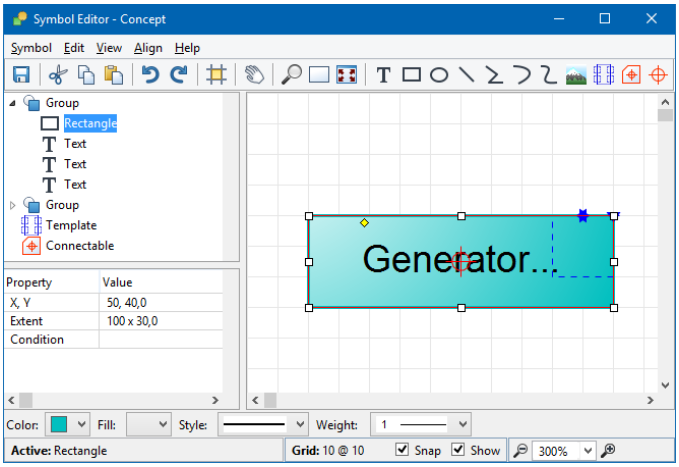

Fig. 7. Symbol definition for Concept

While the elements have richer structure than just name, an alternative representation approach was added. A modeler may want to see visually further details of the element. For this purpose two possible representations were added - both been shown in Fig 8. The symbol on the left shows the minimal view and the symbol on the right characteristics that were considered important to visualize. Note that part of the data like properties and functions are directly took from the metamodel of *Concept* (see metamodel in Fig 3.) whereas the states are retrieved from the *State chart* linked to the concept (see metamodel for this part in Fig 6.).

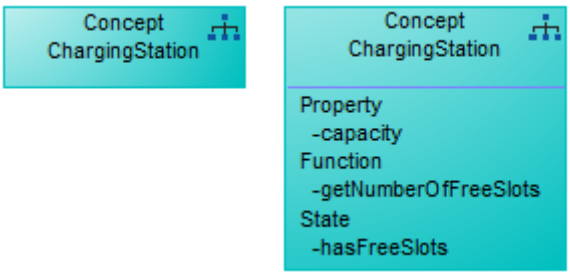

Fig. 8. Two possible visualizations for the Concept - as chosen at modeling time

The definition of the notation was done similarly to other views and their modeling elements - not just for main modeling objects but also for their relationships and roles. A more complete illustration of the visualization aspects is given in the example section.

D. Implementing KnowLang generators

After having defined the notation we created simultaneously models representing the knowledge for KnowLang. While these models were used to test the language they also served as basis for generators. After having models we implemented generators producing the knowledge in KnowLang and calling it for compilation.

The generator was implemented with Generator Editor of MetaEdit+. Fig. 9 shows the main structure of the generator on the upper left corner. A generator called *Code* starts from the *Concept tree* and for each concept produces information on its inheritance relationships with other concepts, as well as with defined properties, functions, and states. For each of these parts there is own subgenerator: These subgenerators match to the concepts expressed in the metamodel. In the bottom of the screen, one subgenerator is shown handling the generation of states. It calls again other generators producing behavioral logic in Boolean expression given for the concept.

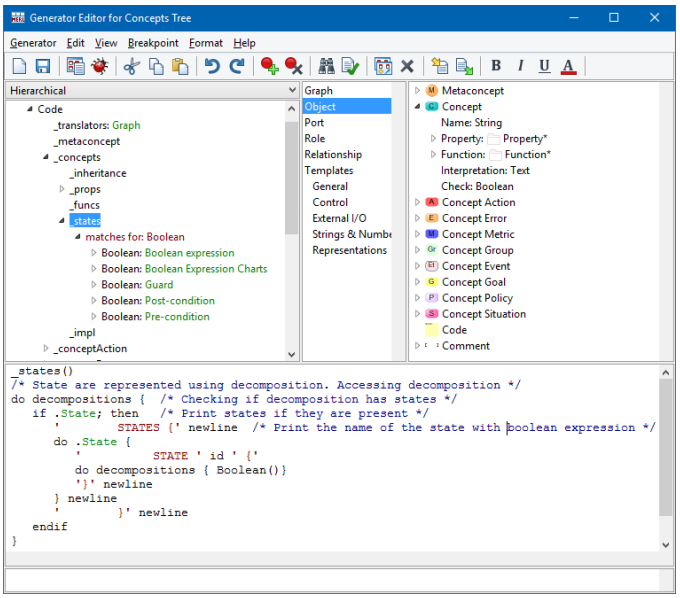

Fig. 9. Definition of the generator (example)

The other parts of the Generator Editor provide access to the metamodel (top right) and to the generator commands (top middle). This allows writing the generator within the context of the given metamodel, aka KnowLang concepts.

While the main part of the generator is navigating the visual model to produce the code, it also integrates with KnowLang reasoner by calling it at the end with the generated output. In this way the developer can move easily from the visual model to see the results been executed in KnowLang.

In addition to generating the KnowLang code the same generator system was used to provide model checking rules not included in the metamodel. These included model guidance (e.g. created Boolean expression is partial), reporting (e.g. document generation), queries on models and producing metrics.

F. On the implementation process

The implementation was done by one person during spring 2017 within a period of three months. Half of the effort was on defining the metamodel (Section III.A-C) and second half on implementing the generator (Section III.D). During implementation phase 3 persons provided feedback to the work done. The implementation was tested and verified by using the created modeling language to specify various kinds of systems and by comparing to the reference test cases.

IV. EXAMPLE

The modeling solution is applied to specify safety functionality in different application areas, such as autonomous cars, unmanned space explorer and surveillance drones. We use next car safety as an example and for the sake of brevity show parts of the key models only. The aim of car safety project is to compute a set of alternative routes for its current destination, to ensure that the vehicle always runs on sufficient battery and to drive safely around crosswalks.

The modeling process starts with defining the ontology of the system with concept trees. For each concept, its properties, functions and states are defined. Fig. 10 shows the ontology with concept trees in MetaEdit+ modeling tool, and Fig. 11 shows a portion of this model with details.

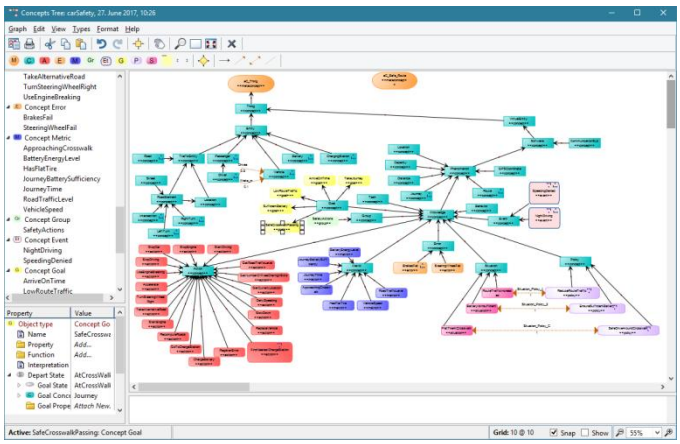

Fig. 10. Concept tree of car safety

The notation uses different colors and shapes for the model elements to assist reader identify the different KnowLang concepts. Fig. 11 shows details of the concept tree dealing with Software Phenomenon on Journey and Route as well as related knowledge on errors, situations and policies. If there is a need to exemplify the ontology with examples, corresponding instances of these concepts can be defined as object trees. Object trees were specified in the metamodel as part of the KnowLang support.

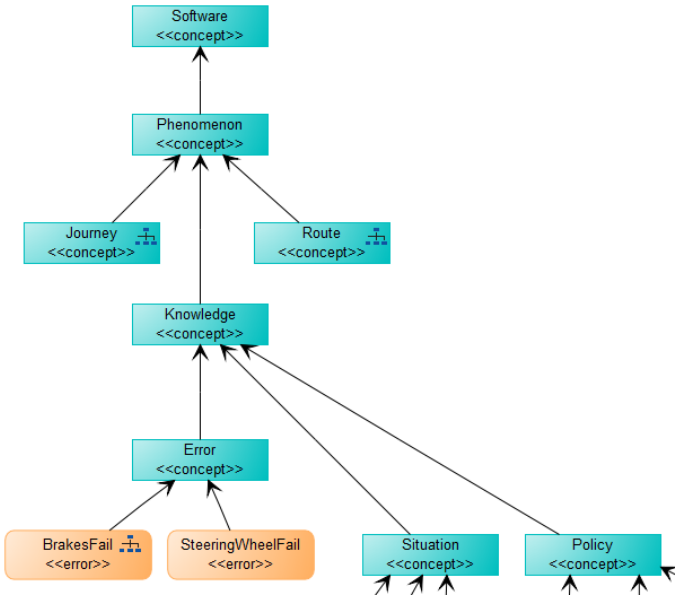

Fig. 11. Concept tree for software phenomenon (partial)

Behaviour is described with states using Boolean expression. A Boolean expression for *AvoidCollision* in shown in Fig. 12. This differentiates *InLowTraffic* and *InHighTraffic* conditions: in a high traffic *NeedFix* and *FlatTireAtCrosswalk* are not allowed. All these states refer to other Boolean expressions defined for other concepts: Traffic concepts of Route, NeedFix for Brakefailure, and FlatTire on Journey concept. These concepts were defined in the Concept tree (Fig 11).

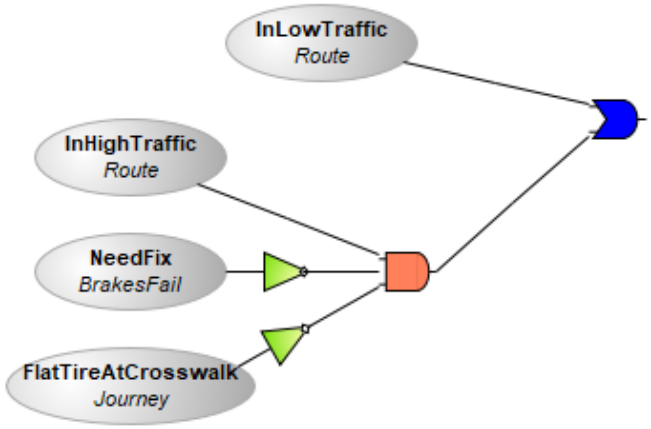

Fig. 12. Behavior expression for avoiding collision

Predicates in KnowLang are considered as complex system states because their evaluation depends on the evaluation of the involved concept states. Fig 13 illustrates such predicate dealing with three concepts on collision avoidance.

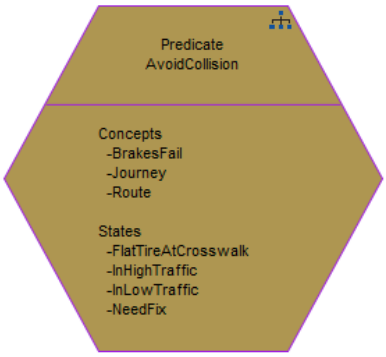

Fig. 13. Predicate or Complex State for avoiding collision

Created models can be transformed at any point of time to the analyzer, reasoner, or simulation applied - given that generator is available. Since we applied KnowLang the generator produces KnowLang code.

Fig. 14 shows the portion of the KnowLang code as generated from the visual models. The part highlighted is related to concept Journey (Fig. 11) and its related states, like that dealing with flat tire used to define behavior in Fig 12. KnowLang and the generated code is running on top of the system developed. When the system wants to take decisions it consults with KnowLang providing self-adaption.

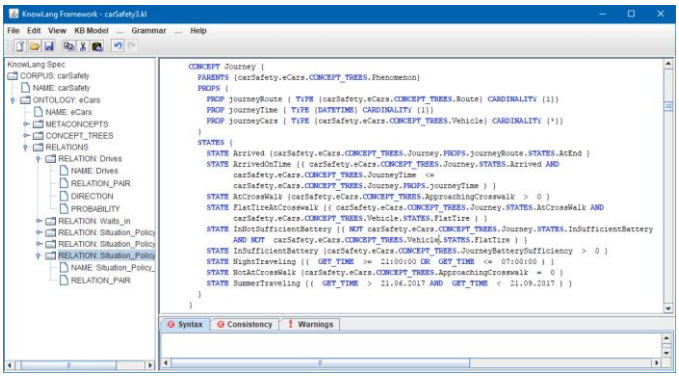

Fig. 14. Generated code in KnowLang

V. CONCLUSIONS

We presented a visual modeling language for developing self-adaptive systems. The developed approach provides several benefits for development teams:

- Support communication and collaboration within a team. Different users may take a different view to the specifications and can edit the same specification simultaneously.
- The model can be applied for generating the code improving the productivity and removing the need for learning particular syntax and debug coding errors.
- The modeling language guides developers by partitioning the system specification into different concerns, like concepts, dependencies, behavior, etc.
- Reduce the cost of software certification.
- Reduce the time to market a product.

We presented also the actual language creation process covering the metamodel with rules, visual notation and code generator. The actual language implementation work was done in the period of 3 calendar months by one person. Because the investment is modest, it pays off quickly as all the other developers can then model with the language, and run the generators creating the code. As both the modeling language and generators are freely accessible, the presented approach also gives full control for the company for making possible extensions in the future.

ACKNOWLEDGEMENT

We would like to thank dr. ir. Ion Baroson and prof dr. Mark van den Brand at Eindhoven University of Technology for collaboration and for their continuous support and guidance throughout this project. We would also like to thank Baesis Automotive for initiating this project and supporting us.

REFERENCES

[1] Vassev, E., Hinchey, M., Knowledge Representation for Adaptive and Self-aware Systems. In Software Engineering for Collective Autonomic Systems, Volume 8998 of LNCS. Springer, 2015.

[2] Vassev, E., Hinchey, M.., Knowledge Representation for Adaptive and Self-Aware Systems. In Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, 2015.

[3] Vassev, E., Hinchey, M., KnowLang: Knowledge Representation for Self-Adaptive Systems. In: IEEE Computer 48 (2), 81–84, 2015.

[4] KnowLang Framework for Knowledge Representation and Reasoning for Self Adaptive Systems. http://www.knowlang.engineeringautonomy.com (accessed Jan 2018).

[5] Vassev, E., Hinchey, M., Awareness in Software-Intensive Systems. In IEEE Computer 45(12), 84–87, 2012.

[6] Neapolitan, R., Learning Bayesian Networks. Prentice Hall, 2013.

[7] Vassev, E., Hinchey, M., Knowledge Representation and Reasoning for Intelligent Software Systems. In IEEE Computer 44 (8), 96–99, 2011.

[8] Jakšić, A., France, F., Collet, P., Ghosh, S., Evaluating the usability of a visual feature modeling notation. International Conference on Software Language Engineering. Springer. 2014.

[9] Moody, D., The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering, IEEE Transactions on Software Engineering, Volume: 35, Issue: 6, 2009

[10] MetaEdit+, http://www.metacase.com (accessed Jan 2018)