Requirements Engineering by Generator-Based Prototyping

Andreas Metzger

AG VLSI Design and Architecture
Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
metzger@informatik.uni-kl.de

Abstract: Efficiency plays an important role in modern development methods. Especially activities that do not promote the actual software product need to be executed efficiently, because otherwise these activities would be neglected. Examples for these activities are establishing consistency between development documents and the creation of throw-away prototypes. In this paper, an approach for the automation of such activities during requirements engineering is presented, focusing on the creation of prototypes for reactive systems. The automation, and thereupon the efficient execution of these activities is achieved through the application of generators that are created on the basis of a formal product model. Such a product model describes all types of development artefacts and the relations between these products. In addition to the realization of document generators, this product model allows the systematic creation of tools for automating other development activities such as checking completeness of the specification or detecting conflicting requirements.

Introduction[1]

Modern reactive systems like automotive controllers, telecommunication devices, or building control systems become more and more complex, as the number of functional and quality-of-service requirements increases. Such systems typically realize between 200 [MeQ02] and 8000 [RaJ01] requirements. Therefore, a systematic and efficient development method is needed for timely delivering such systems, while guaranteeing final products of high quality.

Prototyping is an important technique in the requirements engineering phase of such a development method. Through prototyping, an executable model of the software product can be achieved before the final product has been created. With this executable model, the *validation* of user requirements becomes feasible because users can easily communicate with developers by applying and experimenting with the prototype nearly as they would do with the final product [BKK92].

Besides such a validation of requirements, the prototypes can be applied for test-based *verification* of development products by developers. To emphasize the benefits of prototyping, the building automation case study that was performed by a development team of the SFB 501 shall be stated [QuZ99], in which a total number of 32 errors was identified by inspection, while prototyping led to the discovery of 33 additional errors [MMZ02].

It is a well established fact that the later in development an error is discovered, the higher is the cost of its removal. Therefore, if prototyping can be applied early during the requirements engineering phase, the number of errors that lead to a costly removal in later phases can be reduced. However, for prototyping to be adopted as part of an efficient requirements

1. This paper has been published in H. Alt, M. Becker (Eds.) *Software Reuse – Requirements, Technologies and Applications*. Proceedings of the International Colloquium of the SFB 501. Department of Computer Science. University of Kaiserslautern. 2003. pp. 25–35

engineering method, the prototypes must be attainable with reasonable effort. As mostly semi-formal documents are predominant at the beginning of development, the challenge therefore is to efficiently create prototypes from this limited information. We think efficiency can best be achieved with the generator-based concept that is introduced in this paper.

As code generators for operational specification languages like *SDL* (*Specification and Description Language* [OFM97]) are available, an intermediate step in our approach is the transformation of the semi-formal[1] development documents into such an operational notation. This is obviously a less complex task than generating programming language code directly.

To provide a systematic solution for that transformation, we propose employing a precise and complete model of all types of development products. This so-called *product model,* which can be available as part of a precise definition of the used development method, reflects each type of development artefact and its relations to other types of artefacts. By parsing the existing semi-formal development documents and extracting the formal parts, an instantiation of this model is achieved. Then, from such an instantiation, operational specifications are automatically created by generating the operational documents that make up the specification. Additionally, method-specific templates are instantiated during the generation process. This has the eminent benefit of producing documents that can directly be read and modified by developers.

To formalize a concise product model, the specialization of the development method to a specific application domain is of great benefit, as it allows the precise definition of development activities and products. In this paper, the requirements engineering method *PROBAnD,* which was developed for specifying building automation systems [Que02], is utilized for this purpose. System development with this method begins with the semi-formal description of development data in HTML tables. From such HTML documents, operational models of the system are specified in SDL documents. In the initial form of the method, prototyping could be employed *after* the system or certain parts of it had operationally been specified in SDL. The generator-based approach that is presented in this paper can be regarded as an extension of this requirements engineering method. Through employing the detailed product model of this method, the efficient generation of prototypes *before* the system or its parts have operationally been specified becomes feasible.

After a short overview of related work, the product model, on which our approach is based, is depicted in the following section. Then, the concepts for generating SDL documents, from which prototypes are created, are described. Finally, the application to other development activities is outlined.

Related Work

Our approach provides the transformation of semi-formal documents into SDL specifications in such a way that the transparent integration of the generated documents into the development process is possible. This is in contrast to approaches like the ones proposed by Mansurov et al. [MaV00] or Khendek et al. [KGB98]. Where Mansurov et al. administer a universal transformation from *MSCs* (*Message Sequence Charts* [ITU99]) to SDL, Khendek et al. allow more control over this transformation by providing the basic structure of the output documents as a constraint for the generation process. To derive MSCs from early requirements documents, an approach for generating MSCs from *Use Case Maps* (*UCMs* [BuC96]), which is introduced in [MAB01], can be applied. The information that is contained in the Use Case Maps diagrams can be considered as being similar to the information stored in the semi-

1. In accordance to [WiD94], we regard semi-formal documents as documents that contain information in a structured form, which allows manipulation of the data conforming to its intended semantics.

formal documents of our PROBAnD method (e. g., traceability between requirements is modeled in UCMs through paths that connect responsibilities).

The product model that we use for our approach can be seen as a meta-model of the development documents. This meta-model has several similarities to the meta-model of the well-known *Unified Modeling Language, UML* [BJR99]. However, compared to the UML meta-model, our product model allows the forward as well as the backward traceability of requirements. Further, where the meta-model in UML is mostly used for exchanging data between tools (model interchange [OMG01]), we utilize this model for automating development activities, such as the checking of consistency and completeness or the construction of stubs for partial prototypes.

Product Model

When defining a detailed product model that regards all necessary development products, the model itself might become very complex. In our case, the complete model currently contains 80 different entities. Therefore, we structure our model by classifying the different types of entities according to Fig. 1 [MeQ02].

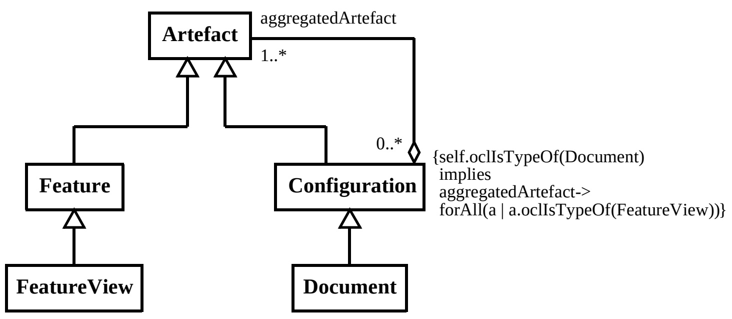

Fig. 1. Classification of Development Products

The most general type of entity is called *artefact*, which can be identified with any development product. Each artefact has a unique name and can optionally be described by informal text. More specialized types of entities are *features*, which represent *atomic* development products, and *configurations*, which aggregate less complex artefacts[1].

Features contain the actual development information independent of its representation. Thus, for each feature different *feature-views* exist, which contain feature information in a concrete representation; e.g., in the HTML or the SDL notation. This is why feature-views can be regarded as a specialization of features. Usually, developers will be working on *documents*, which are composed of feature-views. Therefore, documents are classified as special configurations and as such are described by the constraint in Fig. 1.

We postulate that each feature-view for any feature can be generated solely from the information that is stored in the respective feature. Accordingly, we assume that each document can be created from a respective configuration. As a consequence, the types of features that are depicted in the following subsections accurately reflect all types of atomic information that can be contained in the development documents.

The PROBAnD requirements engineering method [MeQ02][Que02] starts with the specification of *needs* (see Fig. 2), which are stated by the users or customers in natural language.

1. Note, that our notion of a feature is different from the general meaning of the term feature, which is a self-contained functional part or aspect of a specification or system [PuS01].

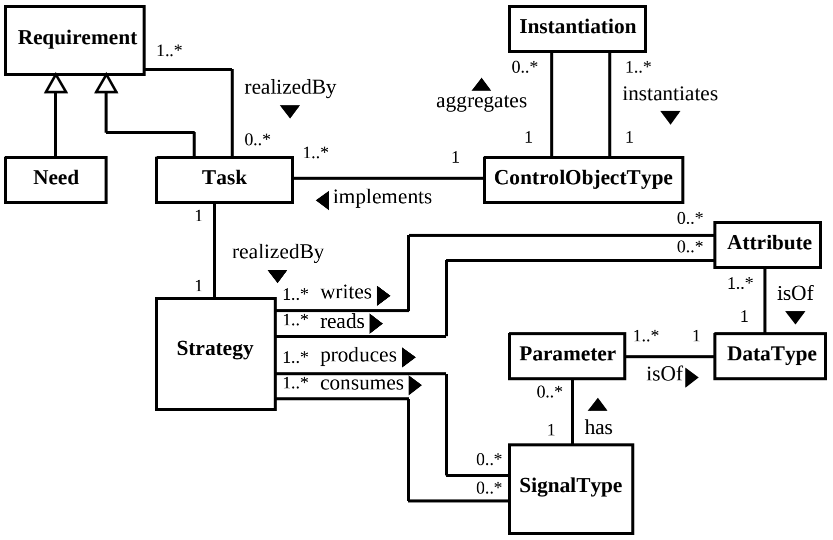

Fig. 2. Types of Features of the Product Model

These needs are realized by various *tasks*, which themselves might be realized by other tasks. One or more of these tasks are implemented by a *control object type*, where each control object (i.e., instance of a control object type) can be regarded as an active object with its own independent flow of control.

As an important guideline in the PROBAnD method, control objects always have to form an aggregation tree. This is reflected by the one-to-many aggregation relation between the feature control object type and the feature *instantiation*.

For each control task, a *strategy* has to be specified. Such a *strategy* is responsible for realizing the actual behavior of the task. Therefore, each strategy can read or write *attributes*, and produce or consume signals that are of globally defined *signal types*. These signal types can possess additional *parameters*. Where attributes are used for communication between strategies of a single control object type, signals are employed for the asynchronous communication between different control objects.

In the following section, we show how the features that have been introduced above are employed for generating documents, from which prototypes can automatically be created.

Document Generation

As it was pointed out at the beginning of the previous section, documents are composed of various feature-views. To be able to generate documents from features, configurations of features that accurately reflect the aggregation of feature-views by the respective documents are introduced. With this information, the creation of documents is performed as it is shown in Fig. 3. First, a feature-view is generated for each feature that is aggregated by the respective configuration. This is followed by composing these feature-views to the required document, which is allowed to be incomplete depending on the set of available features.

Additionally, document templates are applied at this point for producing documents that conform to the modeling guidelines, thus allowing the developers to use a generated document like any other document that might have been created manually.

Because our document generation approach works on features and configurations that reflect the current state of development, existing documents need to be parsed to extract the features and configurations represented in the product model. This parsing activity can simply be performed by applying the concepts that have been presented for generating documents in a reverse order. First, the feature-views of a specific document are extracted. In addition to the view information that is explicitly contained in the development documents, the aggrega-

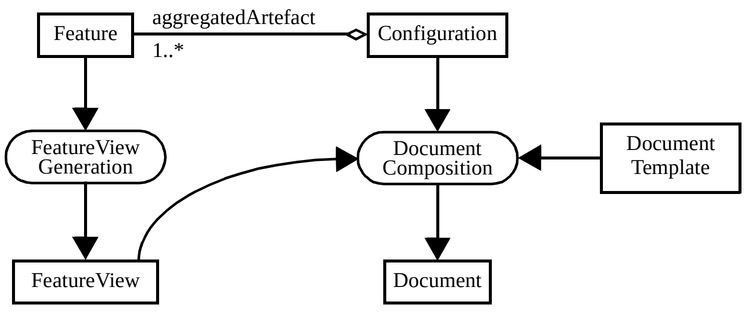

Fig. 3. Generation Steps on an Abstract Level

tion of feature-views, which is implicitly reflected in the development documents as they are composed of feature-views, needs to be identified. Then, from this feature-view data and the aggregation information, features and the according configurations can be retrieved. This involves resolving textual references contained in the feature-views to establish relations between features. Hence, inconsistencies in the development documents can be identified at this point, and developers can be notified to correct the respective artefacts.

Automatic Prototype Creation

To be able to generate prototypes, SDL specifications, which are made up of individual SDL documents, are generated according to the above approach. Therefore, a *control object type configuration* (c.f. Fig. 4), which correlates to the respective SDL control object type document, is defined. Further, a configuration is established that describes which control object types make up the final SDL system. Thereupon, this *system configuration* aggregates several control object type configurations. Also, the top-level control object type is specially tagged to describe the root of the aggregation tree.

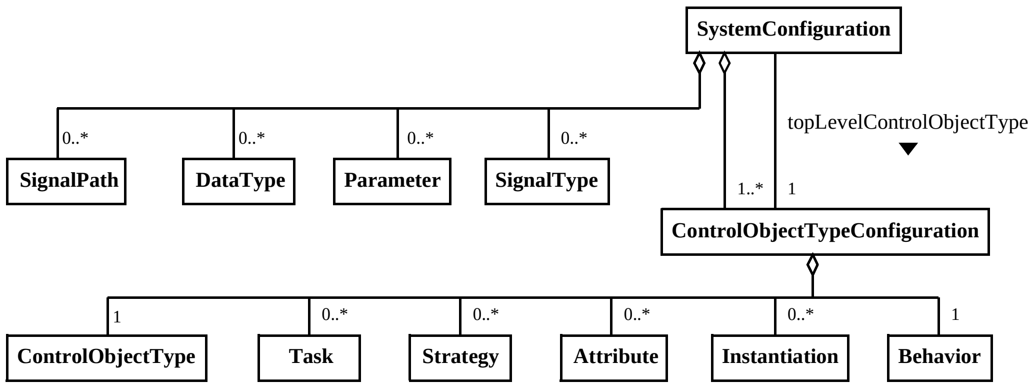

Fig. 4. Hierarchical Configuration for Generating SDL Documents

The control object type configuration aggregates only those features that are visible locally to the respective control object type (e.g., attributes or strategies). Globally visible features (like various data types) are aggregated by the system configuration as all control object types can reference these features.

To illustrate the above concepts, we will use the instantiation of the product model for a simple lighting control example (see Fig. 5). This simple control system controls the illuminance in a single room using two luminaires.

In the exemplary system, the control object type *LightingZone* aggregates one instance of the control object type *PushButton* as well as one instance of the object type *Luminaire*. The

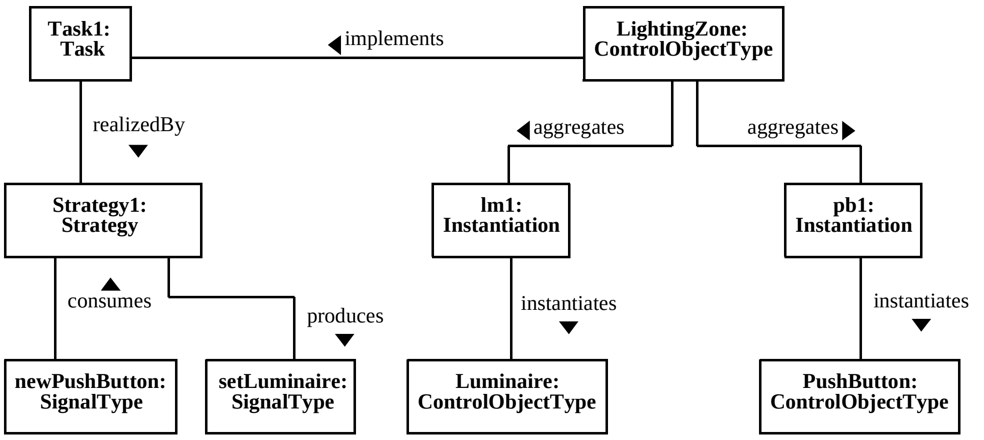

Fig. 5. Partial Product Model Instantiation for a Small Lighting Control System

control object type *LightingZone* implements task *Task1*, which is realized by the strategy *Strategy1*. As *Task1* is responsible for turning on the luminaires if the push-button is being pressed, the respective strategy (*Strategy1*) consumes signals of the type *newPushButton* (which indicates that the button has been pressed) and it produces signals of the type *setLuminaire* (which is used to toggle the luminaires between on and off).

To generate an SDL document for the control object type *LightingZone*, first the required feature-views are generated; e.g., the SDL view of one of the instantiations of the control object type *PushButton* is "*pb1* : *PushButton*", which corresponds to the block type instantiation syntax of SDL. Then, these feature-views are composed to form the desired document (e.g., an SDL package or block type document) by applying the particular SDL template of the PROBAnD method; in Fig. 6, the feature-view example from above can be found in the second SDL block from the top (the variable parts of the template that are replaced by feature-views are underlined).

Besides such instantiation information, the communication channels are determined from the produced/consumed relations between strategies; e.g., as shown in Fig. 6, the channel *b.i1* is realized, which conveys signals of the types defined in the signal list *bi1*, which in turn contains the signal type *newPushButton*.

Further, the behavior description of the control object types could be created; in the example in Fig. 6, the block *LightingZoneCtrl* contains this behavior. However, the strategies that are specified in the semi-formal documents are currently described in natural language, and are therefore not suited for being used in the generator-based approach. For this reason, our research is directed towards the specification of these strategies by means of 'partial' (extend-

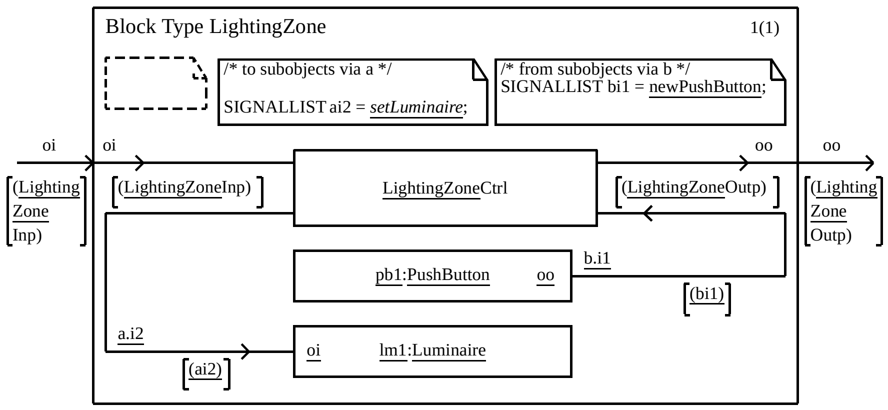

Fig. 6. Generated SDL Block Type Document

ed) finite state machines, which are composed to complete state-machines and then transformed into SDL state diagrams during the generation of the SDL control object type documents.

Because SDL documents can now be generated from product model data, the creation of partial prototypes becomes possible with only little effort. For this, control object types that represent stubs and drivers are established by directly modifying instantiations of the product model, followed by generating SDL documents from this modified instantiation. In addition to structural changes, the automatic creation of stubs that expose simple types of behavior is possible; e. g., there can be stubs that only consume signals, where other stubs respond to received signals. On top of that, stubs that randomly or periodically produce signals can be conceived.

To obtain an executable prototype, the generated SDL documents are extended with a generic prototyping library, which is specified in SDL together with extensions in C. This library establishes the communication interface of the control objects to the environment, which can be realized by a building performance simulator [Zim02][Rie02] or a physical testbed [Met01]. From such an extended SDL specification, C-code, which is compiled into executable binaries, is generated. We have successfully applied Telelogic's Tau SDL Suite [LEH00] for that step.

Support of Additional Activities

Besides the application for the above activities of prototype generation, our approach enables the automatic execution of other development activities. One of these activities is the establishment of consistent documents. This is achieved by parsing the most recent versions of documents, verifying completeness and consistency on the model-level, and generating new and consistent versions of these documents from the corrected data. As an example, if the implementation of a task is assigned to a different control object type, the affected control object type documents as well as the list of all tasks (task list document) are updated. Thus, an efficient iterative development process becomes feasible. Feature information can be changed more easily in some types of documents than in other types of documents. Therefore, modifications during development can efficiently be accomplished by applying changes to the documents for that this operation can more effortlessly be performed, followed by parsing these documents, and generating the remaining documents from the modified information.

The automatic creation of consistent documents can only be carried out reasonably, if the verification has revealed no serious faults in the requirements specification documents. Otherwise, the developers need to correct these first. Therefore, invoking the verification tools after modifications of these documents provides an automated inspection, and thus can be used to enforce that only correct documents are committed.

Besides verification activities, further analyses of development data become possible. For example, complexity metrics for development artefacts [Que02] can automatically be computed, resulting in data for decision support by the project management; e. g., if a control object type poses to become too complex it should be split into smaller ones. Another important analysis is the detection of interactions between requirements (also known as feature interaction [MaC00]). Such an analysis has methodically been established by employing the requirements traceability relations (*realizedBy*) in the product model [MeW03].

As browsing through the multitude of development documents on the search for a specific piece of development information can be very tiresome, a computerized support for this activity is of great help. With the availability of all development artefacts and relations on the model-level this can easily be achieved. We have implemented a browsing tool for that purpose that allows the navigation through the product model with simple textual commands, similar to queries to a data-base.

Finally, an important activity in system development is reuse. On the model-level reuse can be supported by merging two instantiations of the product model into one final model, from which a more complex specification can be generated. An example for such an activity could be the extension of a heating control system by illuminance control parts.

Results

First implementations of tools [Met02] for the above activities have shown the feasibility and systematic nature of our approach. These tools have been strictly developed on the foundation of the product model in such a way that each type of artefact is represented by a Java class and a relation between artefacts is realized through Java methods and attributes. Class-frames that contain these methods and attributes have automatically been attained from a UML input model using iLogix's Rhapsody Modeler tool [iLo02]. At the time of writing, the complexity of these class-frames was 4700 lines of code.

Where feature and configuration classes provide a common basis for exchanging development data between tools as well as a foundation for the model-based automation of other development activities, the feature-view and document classes contain tool specific generation and parsing methods. In Table 1 the complexity for extending the class-frames for realizing our tools is given.

Tool-Supported Activity	Lines of Code	Tool-Supported Activity	Lines of Code
parse semi-formal documents	4100	verify completeness and consistency	900
generate documents	5300	create stubs and drivers	800
analyze requirements interaction	1900	browse through artefacts	800

Table 1. Manually Implemented Lines of Code

As can be seen from these figures, all tools have been attained with reasonable effort. The most effort is spent on the tools that perform the transformation from documents to features and configurations and vice versa. As during parsing activities many consistency checks are already performed, the complexity of the verifier tool is very small. A good example for attaining efficiency through a generic approach is the browsing tool, which—through employing Java's reflection mechanism—has been implemented in one person day.

The above tools have successfully been applied for a complex building control system example [QuZ99][QTB02]. The semi-formal documents were parsed, leading to an instantiation of the product model that contains 1109 features and 36 configurations. From this data, a skeleton SDL system has automatically been created that contains all necessary signals and instantiations, such that developers can focus on the interesting parts, i.e., the specification of the behavior. Also, the creation of consistent documents has been applied for creating a correct task list from 28 control object type documents.

Finally, this system specification has been examined for interactions between requirements. As a result, 29 such interactions have been identified in the system that was developed by the development team of the SFB 501 [QuZ99]. A comparison with an extension of this system [QTB02] has shown that six new interactions have been introduced.

The run-time and memory complexities of our tools are very moderate. Each tool exposes a run-time complexity that is linear in the number of artefacts; e.g., 28s of processing time on a 440MHz HP-PA RISC workstation were needed for verifying and transforming the semi-formal documents of the above example, which comprise 12600 lines of HTML code, into a skeleton SDL system. Additionally, each tool invocation required 12 s for reading, resp. writing, the instantiation information. Also, the working memory requirements linearly depend on the number of artefacts. For the above system, 23 MB were needed at run-time.

Conclusion and Perspectives

Modern development processes have to produce high quality products in a reasonable amount of time. Therefore, each development activity should be examined, and its improvement potential should be utilized. In this paper, a generator-based approach has been presented as a foundation for such improvements, where a common product model allows the systematic creation of the required tools and the exchange of data between these tools.

In addition to the early generation of prototypes, this approach enables an efficient iterative development process. Further, activities like verification, analysis and browsing of development data are enabled.

The underlying requirements engineering method, which was initially developed for specifying building automation systems, has been successfully extended and modified for other domains, like building performance simulators [Zim02] and railway crossing controllers [QuM02]. Therefore, we think that the model-based approach presented in this paper can be extended accordingly. Further, we believe that whenever a precise-enough product model and suitable modeling templates are available for a development method, the presented concepts can be applied.

An extension of these concepts to the design phase is possible. Based on operations on the instantiation of the product model, a re-partitioning (i. e., re-assignment of tasks to control object types) becomes possible. The selection of the partitions can be supported by the extension of the product model with results of the analysis of the dynamic behavior of the system (e. g., maximum signal bandwidth) [QST99]. Other research activities are directed towards the field of reliable distributed embedded systems, where an advanced failure behavior analysis that yields graded results is being developed [TST02].
Still, in the current state of the *PROBAnD* method, there is a rather large gap between the behavior description in the semi-formal development documents, where natural language is used, and the behavior description in the SDL documents, where extended finite state machines are specified. We will be evaluating the use of 'partial' finite state machines and their composition to EFSMs for which our product model has to be extended accordingly. The UML meta-model shows how this can be done for hierarchical state machines. We plan on using similar concepts with a strong focus on traceability from the strategies to the realizing parts of the state machine.

References

[BJR99] G. Booch, I. Jacobson, J. Rumbaugh. *The Unified Modelling Language User Guide*. Reading, Mass.: Addison-Wesley. 1999

[BKK92] R. Budde, K. Kautz, K. Kuhlenkamp et al. *Prototyping: An Approach to Evolutionary System Development*. Berlin; Heidelberg: Springer-Verlag. 1992

[BuC96] R.J.A. Buhr, R.S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall. 1996

[iLo02] iLogix. *Rhapsody*. Web Site. Andover, Mass. 2002
http://www.ilogix.com/products/rhapsody/

[ITU99] ITU-T. *Message Sequence Chart (MSC)*. Recommendation Z.120. Geneva, Switzerland: International Telecommunication Union. 1999

[KGB98] F. Khendek, G. Robert, G. Butler et al. "Implementability of Message Sequence Charts" in *Proceedings of the SDL Forum Society Workshop on SDL and MSC (SAM '98)*. Berlin, Germany. 1998

[LEH00] P. Leblanc, A. Ek, T. Hjelm. "Telelogic SDL and MSC tool families" in *Telektronikk*, Vol. 96, No. 4. Telenor Research & Development. 2000. pp. 156–63

[MAB01] A. Miga, D. Amyot, F. Bordeleau et al. "Deriving Message Sequence Charts from Use Case Maps Scenario Specifications" in *Tenth SDL Forum (SDL '01)*. Copenhagen, Denmark. 2001

[MaC00] E. Magill, M. Calder (Eds.) *Feature Interactions in Telecommunications and Software Systems VI*. Amsterdam: IOS Press. 2000

[MaV00] N. Mansurov, D. Vasura. “Approximation of (H)MSC semantics by Event Automata” in *Proceedings of the SDL Forum Society International Workshop on SDL and MSC (SAM 2000)*. Grenoble, France. 2000

[MeQ02] A. Metzger, S. Queins. “Specifying Building Automation Systems with PROBAnD, a Method Based on Prototyping, Reuse, and Object-orientation” in P. Hofmann, A. Schürr (Eds.) *OMER – Object-Oriented Modeling of Embedded Real-Time Systems*. GI-Edition, Lecture Notes in Informatics (LNI), P-5. Bonn: Köllen Verlag. 2002. pp. 135–140

[Met01] A. Metzger. *Ein flexibles Testfeld für Experimente im Bereich der Gebäudeautomation und -simulation*. Report 4/01. SFB 501. University of Kaiserslautern, Germany. 2001

[Met02] A. Metzger et al. *PROTAGOnIST – Tools for Iterative Software Development*. Web Site. Department of Computer Science. University of Kaiserslautern

[MeW03] A. Metzger, C. Webel. “Feature Interaction Detection in Building Control Systems by Means of a Formal Product Model” submitted to *Feature Interaction Workshop FIW ‘03*. Ottawa, Canada. June, 2003

[MMZ02] A. Mahdavi, A. Metzger, G. Zimmermann. “Towards a Virtual Laboratory for Building Performance and Control” in R. Trappl (Ed.) *Cybernetics and Systems 2002*. Vol. 1. Vienna: Austrian Society for Cybernetic Studies. 2002. pp. 281–286

[OFM97] A. Olsen, O. Færgemand, B. Møller-Pedersen et al. *System Engineering Using SDL-92*. 4th Edition, Amsterdam: North Holland. 1997

[OMG01] OMG. “UML Model Interchange” Chapter 5 in *Unified Modeling Language v1.4*. Object Management Group, Inc. 2001
http://cgi.omg.org/docs/formal/01-09-76.pdf

[PuS01] E. Pulvermüller, A. Speck, J.O. Coplien et al. “Feature Interaction in Composed Systems” in E. Pulvermüller, A. Speck, J.O. Coplien et al. (Eds.) *Feature Interactions in Composed Systems ECOOP 2001*. Workshop #8. 2001. pp. 1–6

[QST99] S. Queins, B. Schürmann, T. Tetteroo. *Bewertung des dynamischen Verhaltens von SDL-Modellen*. Report 9/99. SFB 501. University of Kaiserslautern. 1999

[QTB02] S. Queins, M. Trapp, T. Brack et al. *Floor 32*. On-line Development Documents. Department of Computer Science. University of Kaiserslautern. 2002
http://wwwagz.informatik.uni-kl.de/d1-projects/ResearchProjects/Floor32/

[Que02] S. Queins. *PROBAnD – Eine Requirements-Engineering-Methode zur systematischen, domänenspezifischen Entwicklung reaktiver Systeme*. Ph.D. Thesis. Department of Computer Science. University of Kaiserslautern. 2002

[QuM02] S. Queins, A. Metzger. “The PROBAnD Railway Crossing Specification”. Contribution to the SDL-2000 Design Contest of the 3rd SAM Workshop. Aberystwyth, Wales. June, 2002
http://wwwagz.informatik.uni-kl.de/d1-projects/ResearchProjects/RailwayCrossing/

[QuZ99] S. Queins, G. Zimmermann. *A First Iteration of a Reuse-Driven, Domain-Specific System Requirements Analysis Process*. SFB 501 Report No. 13/99. University of Kaiserslautern. 1999

[RaJ01] B. Ramesh, M. Jarke. “Towards Reference Models for Requirements Traceability” in *IEEE Transactions on Software Engineering*. 27(1). 2001. pp. 58–93

[Rie02] J.P. Riegel. “Flexibler Entwurf von Gebäudesimulatoren” in D. Tavangarian, R. Grützner (Eds.) *Simulationstechnik – ASIM 2002*. Erlangen: Gruner Druck GmbH. pp. 576–578

[TST02] M. Trapp, B. Schürmann, T. Tetteroo. “Failure Behavior Analysis for Reliable Distributed Embedded Systems” in *Proceedings of IPDPS – Workshop on Parallel and Distributed Real-Time Systems*. Fort Lauderdale, Fla. April, 2002

[WiD94] R. Wieringa, E. Dubois. “Integrating Semi-formal and Formal Software Specification Techniques” in *Information Systems*. 19(4). Elsevier Science Ltd. 1994. pp. 35–54

[Zim02] G. Zimmermann. “Efficient Creation of Building Performance Simulators Using Automatic Code Generation” in *Energy and Buildings*. 34. Elsevier Science B.V. 2002. pp. 973–983