

Early Prototyping of Reactive Systems Through the Generation of SDL Specifications from Semi-formal Development Documents

Andreas Metzger, Stefan Queins

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

Abstract: Prototyping of software systems can significantly cut down the number of errors that are introduced by misunderstandings between users and developers. Further, it allows the detection of errors that can hardly be found through inspection. Therefore, if prototyping can be made available early in the software development process, the costly removal of errors in later stages of development can be reduced. In this paper, a prototyping approach for supporting the development of reactive systems is presented. In this approach, prototypes are generated from semi-formal development documents, which are predominant during the requirements engineering phase. To perform this generation step in a systematic manner, a formal model is employed that describes all types of development products and their relations for a given requirements engineering method.

Keywords: prototyping, reactive systems, SDL generation, requirements engineering

1 Introduction¹

Modern reactive systems (like automotive controllers, communication devices, or building automation systems) become more and more complex, as the functional and quality-of-service requirements to these systems increase. Further, the complexity is heightened as the number of distributed components rises; e.g., the automatic control of a 100 room office building might require as much as 1000 physical components to be considered.

Therefore, a systematic and efficient development method is needed for timely delivering such systems, while guaranteeing final products of a high quality. Prototyping is one technique that can play an important role 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 realized. 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. These tests are ideally suited for verifying the inherent dynamic behavior of reactive systems, which is a challenging task to be performed with

1. This paper has been published in *Proceedings of the 3rd SAM (SDL and MSC) Workshop*, Aberystwyth, Wales: SDL Forum Society; University of Wales. June, 2002

static techniques such as inspection or formal analysis. To emphasize the benefits of prototyping, the building automation case study introduced in [QuZ99] shall be stated, where a total number of 32 errors was identified through inspection, while prototyping lead to the discovery of 33 additional errors (mostly of dynamic nature) [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 engineering method, the prototypes must be attainable with reasonable effort. As informal or at most semi-formal development documents are predominant at the beginning of development, the challenge therefore is to efficiently create prototypes from this limited information. We think this can effectively be approached with the generator-based concept that is introduced in this paper.

Code generators for formal specification languages like *SDL (Specification and Description Language* [ITU99]) are already available. Therefore, an intermediate step in arriving at the desired prototype is the transformation of the semi-formal development documents into such a formal notation, as this is a far less complex task than generating programming language code directly. To systematically perform this 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 product and its relations with other types of products. Parts of this product model are instantiated using the set of available semi-formal development documents. Then, from such an instantiation, formal specifications are automatically created by generating the respective formal documents.

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, a requirements engineering method that we initially developed for specifying building automation systems is presented [MeQ01]. This method bases on the application of object-orientation to handle complexity, reuse to gain efficiency as well as product quality, and prototyping to utilize the benefits that have been introduced above. System development with this method operates on semi-formal *HTML documents (HyperText Markup Language* [W3C97]), which are transformed into formal *SDL documents* as development proceeds. In the initial form of the method, prototyping could be employed *after* the system or certain parts of it had formally been specified in *SDL*. The prototyping approach presented in this paper can be regarded as an extension of this requirements engineering method. Through employing the method's fine-grained product model, the efficient generation of prototypes *before* the system or its parts have formally been specified becomes feasible.

The following sections introduce our requirements engineering method and present the respective product model in detail. Then, basic concepts for creating prototypes from instantiations of this model are described, ways for instantiating the product model are outlined, and finally the application of the above concepts in the context of an iterative development process is shown.

2 The Requirements Engineering Method

The first phase in system development is the requirements engineering phase. In this phase, the often vague needs of the users are transformed into a set of documents that should precisely and completely define the requirements to the system. As an efficient example of a requirements engineering approach, we introduce the *PROBAnD method (Prototyping-, Reuse-, and Object-based Building Automation Development)*, which has been developed for specifying complex

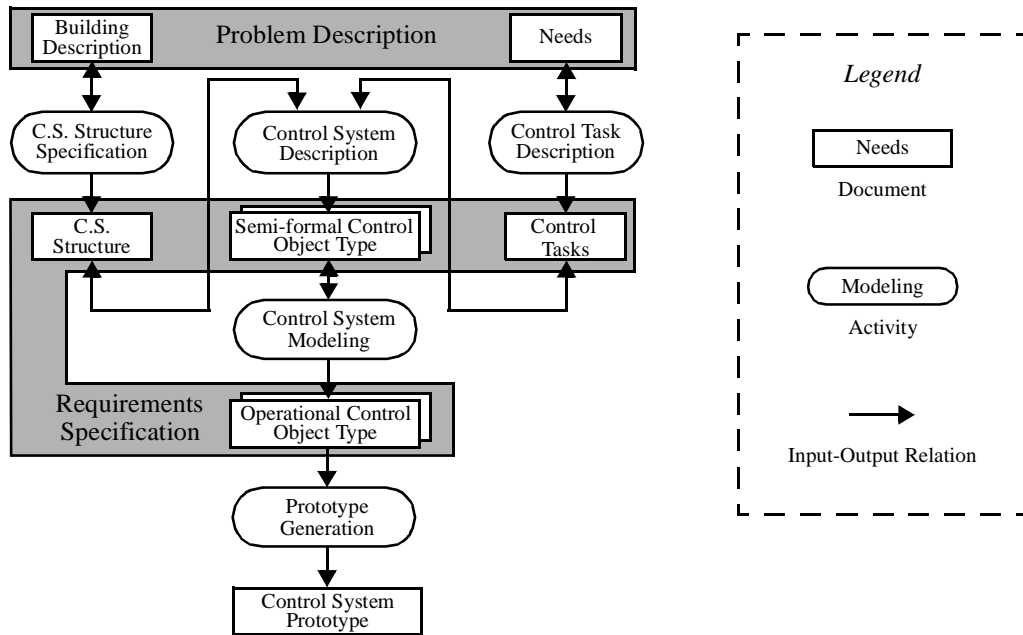


Fig. 1: Overview of the PROBAnD Method

building control systems software [MeQ01][Que02]. To introduce this method, a rather informal description of development activities and documents will be given, illustrated by a small lighting control example that will be used throughout the remainder of this paper.

The PROBAnD method takes the **problem description** as an input, which is divided into the **building description** and a collection of **needs** (c.f. Figure 1).

The building description contains a description of the building’s structure; e.g., a floor-plan that shows that the building is made up of one floor with three rooms. Further, the building’s installation is depicted; e.g., in an informal text that states: “There are two luminaires, two push-buttons, and one motion-detector per room”.

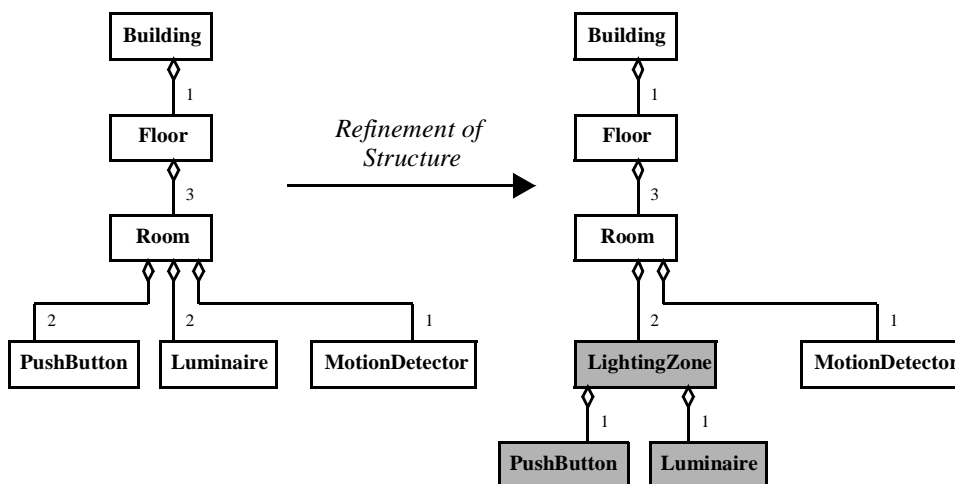


Fig. 2: Possible Structures of a Simple Lighting Control System

From this building description, an initial **control system structure** (as shown in the example on the left-hand side of Figure 2) can easily be derived because the **control objects** (the objects of the control system) are most often identical to the building’s objects or are a sub-set of these;

e.g., lighting needs to be controlled only within the boundaries of one room, and therefore the control object for lighting can be identified with this room.

To handle the huge number of control objects that need to be regarded for large building control systems, **control object types** are formed, which are aggregated according to the hierarchy of the building’s objects; e.g., the control object type **Floor** aggregates the control object type **Room**. As requirements engineering proceeds, this control system structure can be refined, as long as the strict aggregation hierarchy of control object types is maintained; e.g., **PushButton** and **Luminaire** can be aggregated by the additional control object type **LightingZone** if each **PushButton** should only control one **Luminaire** (see right-hand side of Figure 2).

The other part of the problem description is made up by a collection of needs (see Figure 1). These needs informally describe the control system from the point of view of the users; e.g., a typical need for the small lighting control example can be “It should be possible to turn on the lights in a room” (**need_1**). The needs are split into less complex **control tasks** such that they can be assigned to single control object types; e.g., for the above need, there could be the control task: “Turn on luminaire if push-button is pressed” (**task_1**) assigned to **LightingZone**, and the control task: “Notify of the push-button being pressed” (**task_2**) assigned to **PushButton** (c.f. Figure 2).

As a guideline for the above steps, we suggest that the responsibilities of a control object type at a certain level should match the control tasks that can be performed at that level, which allows minimizing the flow of information. This distribution of responsibilities, which can easily be applied in the domain of building automation systems, has similarities to an *organizational hierarchy* [Zim98]. Communication between control objects is realized through the exchange of signals (which may have parameters) that are only allowed to travel along the aggregation hierarchy. This helps maintaining the easy comprehensibility of the specification, and allows the creation of documents from a set of few *templates* to simplify recurring activities (like specifying communication channels between control object types).

After the control object types have been identified and control tasks have been assigned, **strategies** for realizing the control tasks are informally given in natural language, leading to a collection of **semi-formal control object types**, which are expressed in HTML documents. An example for an informal strategy for the above control task for LightingZone could be: “If the signal *newPushButton* is received, send the signal *setLuminaire(on)* to turn on the luminaire” (**strategy_1**).

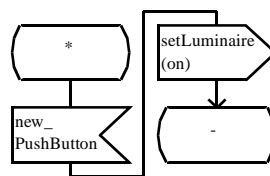


Fig. 3: Strategy for one Control Task of LightingZone (in SDL notation)

From these control object types, **operational control object types** are specified in SDL documents. Together these form an executable SDL specification, from which control system prototypes can be generated. As an example for the behavioral aspects of such documents, the SDL realization of **strategy_1** from above is shown in Figure 3.

After this abstract overview of the PROBAnD method, each development product that is needed for prototype generation will be introduced in more detail in the succeeding section.

3 The Product Model

Before a refined model of the development products is presented, a general classification of the entities of the product model seems suitable. In Figure 4, this classification is shown in the *Unified Modeling Language (UML)* [BJR99], which is also used for describing the remaining class diagrams in this paper.

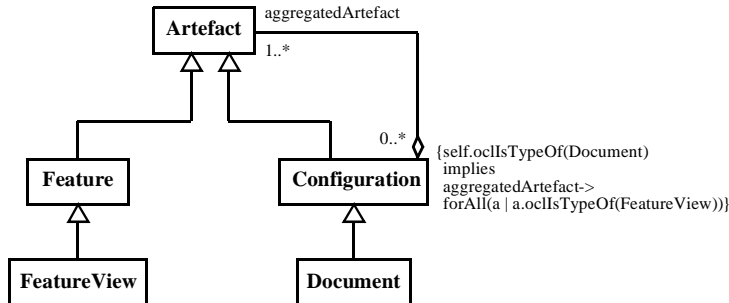


Fig. 4: Classification of Development Products

We introduce the notion of **artefacts**, which can be identified with development products. Each artefact has a unique name and an optional textual description. Artefacts are specialized to atomic artefacts, called **features**, and more complex artefacts, called **configurations**, which aggregate less complex artefacts.

Features can be regarded as ‘virtual’ entities, which contain development information in a form that is not readable or directly modifiable by developers. The real world manifestations of features are called **feature views**, which express feature information in a human-readable form, e. g., in the HTML or the SDL notation. Because feature views contain the same information as features do—only in another representation—, feature views can be regarded as a specialization of features.

As feature views correspond to features, which are *atomic* artefacts, configurations of feature views are needed, on which the developers can work efficiently. These special configurations, which are described by the constraint in Figure 4, can be regarded as **documents**.

To reduce the number of artefacts to be stored during development, and to simplify establishing consistency between artefacts, we require that each feature view for any feature can be generated from the information stored in the respective feature alone. Accordingly, we require that each document can be created from a respective configuration (see Section 4). For the depicted PROBAnD method, this reduces the number of different types of artefacts to approximately 45% (the complete product model contains 53 types of artefacts, of which only 24 types of features and configurations need to be stored). Further, this requirement eliminates the need to check the features against their views for consistency. Therefore, the types of features that are depicted in the following subsections as part of the product model accurately reflect all types of atomic information that can be contained in the development documents.

In Figure 5, the types of features that are regarded during system development with the PROBAnD method are shown (the refinement of **data type** and **signal path** has been omitted for readability reasons).

As it has been outlined above, the PROBAnD requirements engineering process starts with the specification of **needs**, which are stated by the users or customers in natural language.

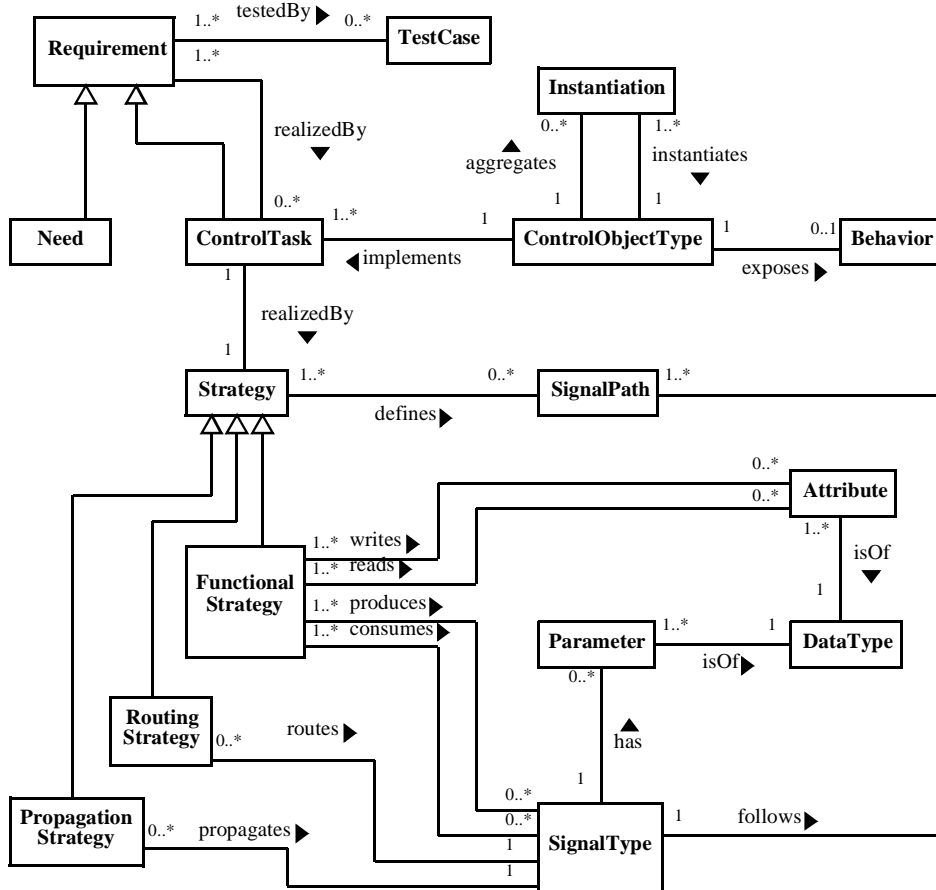


Fig. 5: Product Model of the PROBAnD Method (Types of Features)

These needs are realized by various **control tasks**, which themselves might be realized by other control tasks. Each **control object type** implements one or more of these control tasks. In addition, control object types can **instantiate** other control object types, which realizes the strict aggregation hierarchy of the control system structure.

As soon as prototypes of the system are available, all **requirements** (i.e. needs and control tasks) should be tested by **test cases**. In addition to their use for verification, test cases can play an important role for validation together with customers, as they can be guided by these test cases when experimenting with the system.

For each control task, a **strategy** has to be specified. There are three types of strategies that we distinguish:

A **functional strategy** is responsible for realizing the actual behavior of the task (an example is **strategy_1** in Section 2). Therefore, each functional 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**. Attributes are used for communication between strategies of a single control object type, whereas signals are employed for communication between different control objects.

As it has been explained above, communication is allowed along the aggregation hierarchy only. Therefore, **routing** and **propagation strategies** are needed for correctly exchanging signals between control objects that are not directly connected. Where a propagation strategy propagates a received signal independent from any of the signal's parameters or the signal's sender, a routing strategy uses these data to determine the path to the receiver (or a set of receivers) of

the signal. The distinction between these two types of strategies is important for generating the respective behavior descriptions in the formal SDL documents.

Several routing and propagation strategies might be required to establish communication between remote control object types. Further, the communication routes between control objects might depend on the actual instantiation context of the control object types. Typically this routing/propagation information is distributed over the affected control object types and thus tedious to comprehend and modify. Therefore, we propose specifying this information in a compact form in so called **signal paths** (c. f. Figure 5).

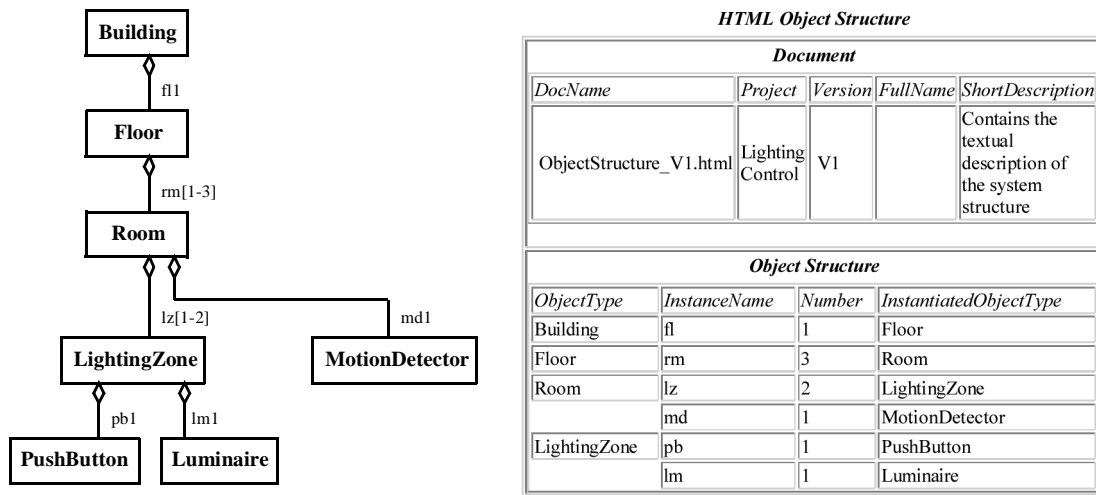


Fig. 6: Structure of a Simple Lighting Control System

For specifying such signal paths, we have successfully applied a regular-expression-like syntax [Fri97]. To illustrate the use of these regular expressions, Figure 6 again shows the control system structure of the small lighting control example in a slightly modified UML representation as well as in the semi-formal HTML representation. The role names of the aggregation relation specify the ‘local’ part of the instance names (the range in brackets represents the number of instantiations). With the assumption that the top-level control object type is instantiated exactly once, these names are concatenated along the aggregation hierarchy to form complete instance names; e. g., there are three instances of the control object type **MotionDetector** with the instance names **md1rm1fl1**, **md1rm2fl1**, and **md1rm3fl1**. To specify that the **MotionDetector** instance **md1** of each **Room** instance sends signals to both **LightingZone** instances **lz1** and **lz2** of the respective Room instance, the following signal path expression can be used:

md1:MotionDetector/rm(*=\$1):Room/fl1:Floor
> lz[1-2]:LightingZone/rm\$1:Room/fl1:Floor

The dollar sign ‘\$’ denotes a placeholder, the asterisk ‘*’ denotes the wildcard expression, and the ‘>’ character shows the direction of the signal. The expressions left and right of ‘>’ precisely describe a single instance or a set of control objects by stating a sequence of signal path segments along the aggregation hierarchy. Each signal path segment consists of the ‘local’ part of the instance name, followed by the name of the control object type and the separator ‘/’.

Because the strategies of one control object type might show interrelationships that cannot be expressed through the independent description of each of the strategies, the overall **behavior** of the respective control object type might have to be specified explicitly (see Figure 5). A typical example for that can be given, if the SDL feature views for strategies are regarded. Each strategy would naturally be specified as a small (and most often partial) **extended finite state machine**

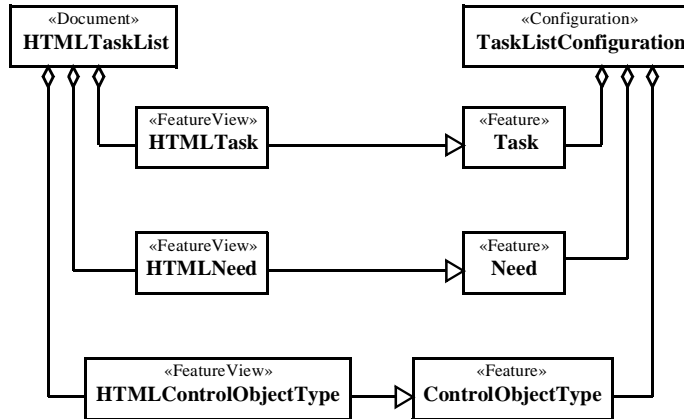


Fig. 7: Example for Relation Between Document and Respective Configuration

(EFSM). When all strategies have to be composed to form one EFSM for the control object type, more information might have to be added because states might have to be multiplied or the action of one transition might have to be inserted into the transition of the EFSM of another strategy. Also, storing the information of a control object type's behavior is indicated, if the respective SDL object type document has already been specified. This typically occurs later in the development process or when reuse activities have been performed.

All documents that are regarded in the PROBAnD method are composed of feature views that are specializations of the features that have been presented in this section. In the following section, we show how these features are employed for generating documents, from which prototypes can automatically be created.

4 SDL Specification Generation and Prototype Creation

As it was pointed out at the beginning of Section 3, documents are an aggregation of various feature views. To be able to generate documents from features, configurations of features that accurately reflect the aggregation relation of the respective documents have to be introduced.

As an example, Figure 7 shows the **task list configuration**, which correlates to the **HTML task list** document. Therefore, this configuration contains all control tasks together with their depen-

dependencies to needs and control object types as found in the respective document (Figure 8 shows an excerpt of such an HTML document for the lighting control example).

Task List

<i>Document</i>				
<i>DocName</i>	<i>Project</i>	<i>Version</i>	<i>FullName</i>	<i>ShortDescription</i>
TaskList_V1.html	Lighting Control	V1		Contains a list of all control tasks.

<i>Tasks</i>				
<i>ModelName</i>	<i>Description</i>	<i>realized by</i>	<i>realizes</i>	<i>ControlObjectType</i>
task_1	Turn on luminaire if push-button is pressed	task_2	need_1	LightingZone
task_2	Notify of the push-button being pressed		task_1	PushButton
...

Fig. 8: Composition of a Document from Feature Views

The simple configuration presented above was used to illustrate the relationships between the different classes of artefacts. In the remainder of this section, the configurations that are used for creating SDL specifications are introduced.

Each control object type that is identified during requirements engineering is formally specified in an SDL document. Therefore, to generate such SDL documents from feature information, configurations that correlate to the respective SDL documents are needed. Further, a configuration is needed that describes what control object types make up the final SDL system.

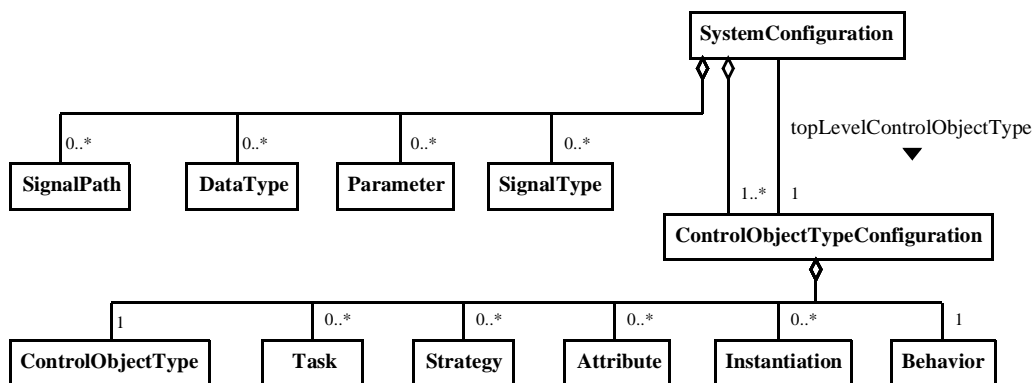


Fig. 9: Hierarchical Configuration for Generating SDL Documents

Figure 9 shows the **system configuration** that was introduced for that purpose. This configuration consists of several **control object type configurations**, where the top level control object type is specially tagged as this represents the root of the aggregation hierarchy.

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 they can be referenced by all control object types.

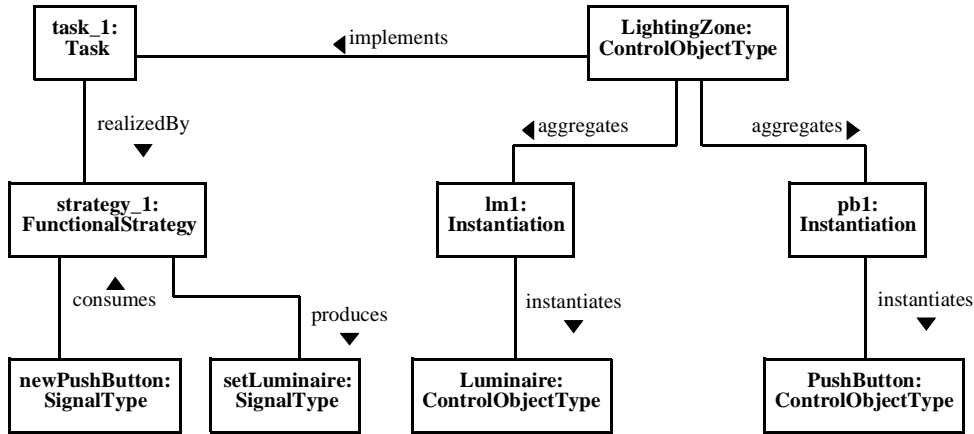


Fig. 10: Excerpt from a Concrete Feature Instantiation

The basic concept for generating documents from a configuration of features is to generate feature views for each feature that is regarded. Then, these feature views are composed to the required documents by applying the respective document templates, in which the variable parts are replaced by the feature view data.

To illustrate this concept, we show how a set of concrete features of the simple lighting control example (depicted in Figure 10 as an object diagram) is used for creating an SDL document. (Note, that the concrete features have been attained by parsing the respective semi-formal development documents as outlined in Section 5).

The control object type **LightingZone** implements the control task **task_1**, which is realized by the functional strategy **strategy_1** (see introduction of the example in Section 2). This strategy consumes signals of the type **newPushButton** (which indicate that the button has been pressed), and it produces signals of the type **setLuminaire** (which are used to toggle the luminaires between on and off). In addition, **LightingZone** aggregates one instance of the control object type **PushButton** as well as one instance of the object type **Luminaire**, which realizes the respective part of the control system structure (c.f. Figure 6).

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 Figure 11, 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 highlighted in italics).

Besides purely structural information (like the instantiation of control object types), the communication channels are determined from the produced/consumed relations between strategies; e.g., as shown in Figure 11, the channel **b.i1** is realized, which conveys signals of the types defined in the signal list **bi1**, which contains the signal type **newPushButton**.

Further, the behavior description of the control object types can be created; in the example in Figure 11, the block **LightingZoneCtrl** contains this behavior. In addition to the SDL view of the feature behavior, the generated behavior description contains generic initialization actions, definitions of attributes derived from the feature attribute, and signal propagation as well as routing code deduced from signal paths. As an example, Figure 12 shows the EFSM that was

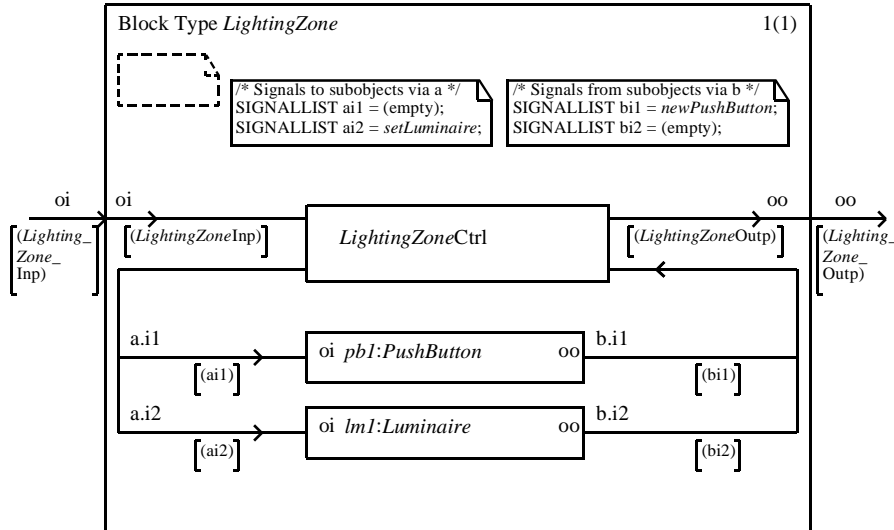


Fig. 11: SDL Block Type Document Created from the Example in Figure 10

generated for the control object type **Room**, which contains the code for propagating the **newMotionDetector** signals as specified in the signal path example in Section 3.

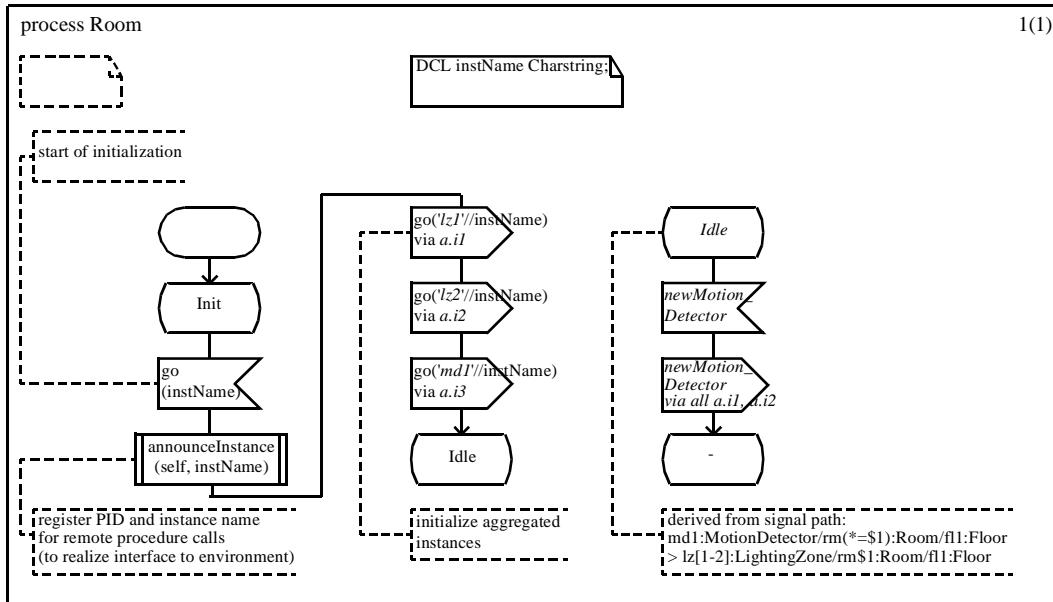


Fig. 12: Generated SDL Process for Control Object Type Room

Besides their use for prototyping purposes, the generated SDL documents have to be read and modified by developers to refine the overall specification (also see Section 5). Therefore, documents in the graphical representation of SDL, *SDL-GR*, need to be generated in order to maintain readability and to enable the application of graphical editors. As *SDL-GR* documents are stored in formats that vary between tools, intermediate documents in the *Common Interchange Format (CIF)* [ITU00] are generated that contain the complete **system specification** (see

Figure 13). From this system specification, tool-specific **SDL-GR documents** can be generated with **CIF to SDL-GR converters**.

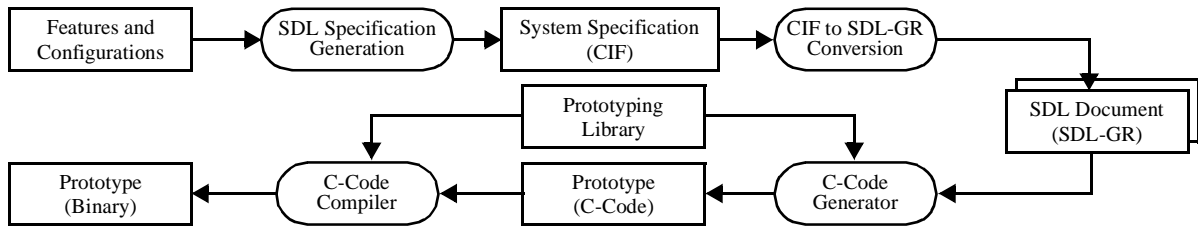


Fig. 13: SDL Specification and Prototype Generation (Activities and Documents)

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 (usually a building simulator or a physical test-bed) [MMZ02], and further provides a set of utility procedures for trigonometric calculations, type conversions, or text output.

From this 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.

5 Iterative Development

As it has been presented above, features and configurations that reflect the current state of development are needed for prototype generation. Because developers work on documents only, these documents need to be parsed to extract the features and configurations represented in the product model. Figure 14 provides an overview of the **generation** and **parsing activities** and the affected artefacts.

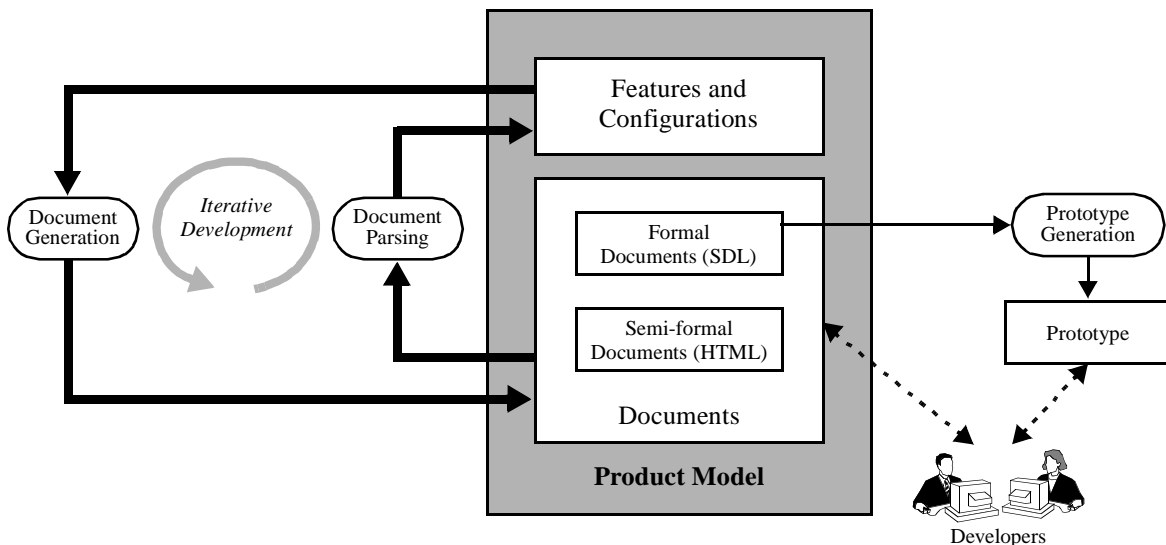


Fig. 14: Iterative Development with Generator Support

Basically, the parsing activity can 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; e.g., in HTML documents, this requires reading data from tables. This task can easily be performed with simple parsers that know of the structure of the underlying docu-

ment templates. In addition to the view information that is explicitly contained in the development documents, the aggregation 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 the textual references contained in the feature views into relations between features. At this point, inconsistencies in the development documents can be identified, and developers can be notified to correct the respective artefacts.

With the availability of document generation and the possibility to parse documents, an efficient *iterative development* becomes feasible. Feature information can be changed more easily in some documents than in other documents; e.g., changes in the semi-formal control object type documents are easier to perform than in the respective formal documents. Therefore, modifications during development can efficiently be accomplished by applying changes to the document for that this operation can more easily be performed. This is followed by parsing this document and finally by generating the remaining documents using the modified information. To illustrate these benefits, two examples will be presented in the following paragraphs.

A relatively simple change that can occur during development is a refinement of the system structure (an example is shown in Section 2). In the semi-formal document that describes the control system structure (see Figure 6 for an example), this requires changing a few textual entries at the place where the number and types of instances are specified. However, this modification has wide reaching consequences in the formal control object type documents. SDL blocks might have to be added or removed. Further, the channels and signal lists might have to be changed accordingly. Finally, initialization code might have to be adapted for the new number of instances.

Far more complicated is a change in the assignment of tasks to control object types. Where this change requires the effortless modification of the task list document (see Figure 8 for an example), the adaptation of the semi-formal control object type documents as well as their formal counterparts is more tedious. Most importantly, the specified communication between the strategies of the affected tasks has to be established. This can require a different routing of signals, i.e. the change of signal paths. Because the new signal paths may contain control object types that have not propagated or routed the respective signals before the change, new propagation or routing strategies for that purpose might have to be created. The situation becomes even more difficult, if the communication between tasks has been performed via attributes (this occurs if the communicating tasks are assigned to the same control object type), and one of these tasks is assigned to another control object type. This might require the creation of new signal types. However, it has to be considered carefully, if this operation should be automated, or the responsibility for these changes should fall in the hands of the developers.

6 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 basis for such improvements. With the ability to perform prototyping very early in the development process, verification and validation activities are supported. Few would disagree that this will reduce the number of errors in later phases, which will cut down the overall correction effort.

In addition, this approach enables an efficient iterative development process, in which changes that would be tedious to be carried out manually can automatically be performed with the application of document generators. Further, the consistency of all documents is guaranteed.

To show the feasibility of the approach, tools for parsing and generating documents are being realized. These tools are systematically developed on the foundation of the classes of the product model. Each type of artefact is represented by a Java class. Where the feature and configuration classes are used for exchanging the development data between document parsers and generators (see Figure 14), the feature view and document classes contain generation and parsing methods. A first implementation of a tool for handling HTML documents has successfully been realized, which supports the soundness of our approach.

To demonstrate the benefits of early prototyping and the advantages of an iterative development based on document generation, case studies are planned, and results will be compared with the data of previous case studies [FMQ99][QuZ99].

The underlying requirements engineering method, which was initially developed for specifying building automation systems, has been successfully extended and modified for creating building performance simulators [Zim01] as well as for designing a railway crossing controller together with its environment [QuM02]. Therefore, we think that the generative 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 approach can be applied effectively.

On the basis of the formal model of the development artefacts, special aspects of the system can be analyzed. Currently, research activities are directed in the field of reliable distributed embedded systems, where an advanced failure behavior analysis that yields graded results is being developed [TST02].

Still, we experience 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. Especially for efficiently specifying stubs that should mimic the behavior of the final objects only roughly, a specification technique of a higher level of abstraction than that provided by EFSMs is needed. Investigations on the applicability of MSCs [MaV00][KGS99], Petri Nets [Ebe92], and Use Case Maps [ALB99] have shown potential solutions. However, most of these notations have to be extended to fit in the control object type concept of our method.

References

- [ALB99] D. Amyot, L. Logrippo, R.J.A. Buhr, et al. "Use Case Maps for the Capture and Validation of Distributed Systems Requirements" in *Fourth International Symposium on Requirements Engineering (RE '99)*. Limerick, Ireland. June, 1999.
- [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
- [Ebe92] C. Ebert. "Experiences with Colored Predicate-transition Nets for Specifying and Prototyping Embedded Systems" in *IEEE Transactions on Systems, Man and Cybernetics*. Part B (Cybernetics). Vol. 28. No. 5. October, 1998. pp. 641–652
- [Fri97] J. Friedl. *Mastering Regular Expressions: Powerful Techniques for Perl and other Tools*. Cambridge; Cologne: O'Reilly. 1997

- [FMQ99] R. Feldmann, J. Münch, S. Queins, et al. *Baselining a Domain-Specific Software Development Process*. SFB 501 Report No. 02/99. University of Kaiserslautern. 1999
- [ITU99] ITU-T. *Specification and Description Language (SDL)*. Recommendation Z.100 (11/99). Geneva, Switzerland: International Telecommunication Union. 1999
- [ITU00] ITU-T. *Common Interchange Format for SDL*. Recommendation Z.106 (11/00). Geneva, Switzerland: International Telecommunication Union. 2000
- [KGS99] I. Krüger, R. Grosu, P. Scholz, et al. “From MSCs to Statecharts” in Franz J. Ramming (Ed.). *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers. 1999
- [LEH00] P. Leblanc, A. Ek, T. Hjelm. “Telelogic SDL and MSC tool families” in *Elektronikk*, Vol. 96, No. 4. Telenor Research & Development. 2000. pp. 156–63.
- [MaV00] N. Mansurov, D. Vasura. “Approximation of (H)MSC semantics by Event Automata” in *Proceedings of the SAM 2000 Workshop*. Grenoble, France. 2000
- [MeQ01] A. Metzger, S. Queins. “A Reuse- and Prototyping-based Approach for the Specification of Building Automation Systems” in A. Schürr (Ed.). *OMER-2 Workshop Proceedings*. Herrsching; Munich: University of the Federal Armed Forces. 2001. pp. 3–9
- [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
- [Que02] S. Queins. *PROBAnD – Eine Requirements-Engineering-Methode zur systematischen, domänenspezifischen Entwicklung reaktiver Systeme*. Dissertation. 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
- [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
- [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
- [W3C97] W3C. *HTML 3.2 Reference Specification*. W3C Recommendation. World Wide Web Consortium. Cambridge, Mass. 1997
- [Zim98] G. Zimmermann. “Domain Specific Model Architecture for Complex Embedded Systems: A Building Automation Case Study” in H. Franke, B. Kleinjohann, J. Sztiapanovits (Eds.). *Adaptation and Evolution in Embedded Information Systems*. Seminar No. 98441. Report No 226. Dagstuhl. 1998
- [Zim01] G. Zimmermann. “A New Approach To Building Simulation Based on Communicating Objects” in R. Lamberts, et al. (Eds.) *Seventh International IBPSA Conference Proceedings*. Vol. 2. Rio de Janeiro, Brazil. 2001. pp. 707–714

Links to documents that are available online can be found at:
<http://www.wagz.informatik.uni-kl.de/staff/metzger/SAM>