Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009)

Stefan Van Baelen[1], Thomas Weigert[2], Ileana Ober[3], Huascar Espinoza[4], and Iulian Ober[5]

[1] K.U.Leuven - DistriNet, Belgium, Stefan.VanBaelen@cs.kuleuven.be
[2] Missouri University of Science and Technology, USA, weigert@mst.edu
[3] University of Toulouse - IRIT, France, Ileana.Ober@irit.fr
[4] CEA - LIST, France, hdeo@hotmail.com
[5] University of Toulouse - IRIT, France, Iulian.Ober@irit.fr

Abstract. The second ACES-MB workshop brought together researchers and practitioners interested in model-based software engineering for real-time embedded systems, with a particular focus on the use of models for architecture description and domain-specific design, and for capturing non-functional constraints. Eleven presenters proposed contributions on domain-specific languages for embedded systems, the Architecture Analysis and Design Language (AADL), analysis and formalization, semantics preservation issues, and variability and reconfiguration. In addition, a lively group discussion tackled the issue of combining models from different Domain Specific Modeling Languages (DSMLs). This report presents an overview of the presentations and fruitful discussions that took place during the ACES-MB 2009 workshop.

1 Introduction

The development of embedded systems with real-time and other critical constraints raises distinctive problems. In particular, development teams have to make very specific architectural choices and handle key non-functional constraints related to, for example, real-time deadlines and to platform parameters like energy consumption or memory footprint. The last few years have seen an increased interest in using model-based engineering (MBE) techniques to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models in a layered construction of systems.

The Second Workshop on *Model Based Architecting and Construction of Embedded Systems* (ACES-MB 2009) brought together researchers and practitioners interested in all aspects of model-based software engineering for real-time embedded systems. The participants discussed this subject at different levels, from model specification languages and analysis techniques to model-based implementation, deployment and reconfiguration. The workshop was attended by 47 registered participants coming from 19 different countries.

2 Workshop Contributions

The keynote [2] was given by Prof. Marco Di Natale from the Scuola Superiore Sant'Anna in Pisa, Italy, who discussed semantics preservation issues in the design and optimization of software architectures for automotive systems.

Architecture selection and design optimization are critical stages of the Electronics/Controls/Software (ECS)-based vehicle design flow. In automotive systems design, complex functions are deployed onto the physical hardware and implemented in a software architecture consisting of a set of tasks and messages. Several optimizations of the software architecture design were presented, including the definition of the task periods, the task placement and the signal-to-message mapping. In addition, the assignment of priorities to tasks and messages were automated in order to meet end-to-end deadlines and to minimize latencies.

Architecture selection can be accomplished by leveraging worst case response time analysis within an optimization framework. Suggestions on how to use stochastic or statistical analysis to improve the approach were provided. Semantics preservation issues impose additional constraints on the optimization problem, but also reveal very interesting tradeoffs between memory and time/ performance. The need to deal with heterogeneous models and standards, such as AUTOSAR (AUTomotive Open System ARchitecture) in the automotive industry, further complicates the scenario.

7 full papers and 3 short papers had been accepted for the workshop, see [1]. A synopsis of each presentation is given below. Extended versions of articles [11] and [12] are included in this workshop reader.

[3] presents SOPHIA, a modelling language that formalizes safety-related concepts (e.g., accident models), and explores strategies to implement SOPHIA as a complementary modelling language to SysML.

[4] describes a prototype modeling language for the design of networked control systems using passivity to decouple the control design from network uncertainties. The resulting designs are by construction more robust with respect to platform effects and implementation uncertainties.

[5] presents a formal specification of the design models used in COMDES-II, a component-based framework for distributed control systems, featuring an open architecture and predictable operation under hard real-time constraints.

[6] proposes a systematic method for Worst-Case Execution-time (WCET) analysis that make Simulink/Stateflow model information (e.g., infeasible executions) available to static timing analysis tools.

[7] presents a methodology and tool for building and translating AADL systems into a distributed application. This allows runtime analysis to fully assess system viability, and to refine and correct the behavior of the system using BIP.

[8] investigates the iterative model-driven design process for complex embedded systems, which suffers from a delay between applying changes to the model and obtaining information about the resulting properties of the system, and identifies opportunities for a novel, integrating research domain on AADL.

[9] presents a methodology for the specification and analysis of critical embedded systems, based on an architectural design language that enables modeling

of both software and hardware components, timed and hybrid behavior, faulty behavior and degraded modes of operation, error propagation, and recovery.

[10] focuses on finding performance problems when use cases are added to (a family of) products, and proposes an automated algorithm that generates performance models by combining use case models, and an approach for performance optimization by adding flow control elements into the system design.

[11] integrate variability approaches from Software Product Lines (SPL) into Domain-Specific Languages (DSLs) like Matlab / Simulink, in order to specify variability, configure products and to resolve element dependencies. High-Order Transformations derive the variability mechanisms (as a generated model transformation) from the meta-model of the DSL.

[12] supports the combination of robustness and flexibility in automotive systems through the integration of reconfiguration capabilities into AUTOSAR, supporting the management of complex system at the architectural level.

3 Summary of the Workshop Discussions

The workshop was divided into 4 sessions: domain-specific languages for embedded systems, the Architecture Analysis and Design Language (AADL), analysis and formalization, and variability and reconfiguration. After each session, a group discussion was held on topics and issues raised during the session presentations. The following integrates and summarizes the conclusions of the discussions.

AUTOSAR

The automotive industry is challenged to distribute an increasing number of functions, which are enabled through an increasing number of smart sensors and actuators, onto a descreasing number of Electronic Control Units (ECUs). The AUTOSAR architecture is a key element for the automotive industry to decouple the software functions from the ECUs. The main challenges for the automotive embedded system industry are (1) how to perform the mapping of functions onto ECUs, and (2) how to preserve the semantics of these functions. The focus here is on the timing analysis to obtain an adequate mapping of the tasks to the resources. Currently no structured hierarchical methodology is used, but the allocation is performed using a bottom-up approach through a number of iterations. Regarding communication between ECUs, the CAN (Controller Area Network) bus is rather easy to deal with, while the FlexRay protocol is more complex. In most cases no formalized architectural frameworks are used on top of AUTOSAR. The trickiest part for the timing analysis is to get the correct Worst Case Execution Time (WCET) bound for safe and precise resource optimization, hereby avoiding overestimations due to infeasible paths. Further research on obtaining accurate WCET information is therefore necessary.

Combining models from different DSMLs

There is a need for using different DSMLs during the development of embedded systems. The main challenges hereby are (1) to combine such models and

map identical representations onto a single concept, (2) to obtain semantic compatibility, and (3) to guarantee and preserve consistency between these models. The UML-MARTE profile seems to be too complex for ordinary developers and end users for expressing non-functional system properties; they require a simple language that is easily understandable. In addition, there can be a need to use several DSMLs, and the results expressed in these domain-specific models must be compatible and consistent. Three approaches for connecting domain-specific models were discussed. First, explicit semantic interfaces can be defined to connect heterogeneous models. The link between AUTOSAR, Simulink, and Modelica is an example. Second, an underlying semantic framework can be used to map the domain-specific models onto a universal model. Such approach also has to deal with concepts present in the DSML that cannot easily be mapped onto the underlying framework. UML-MARTE could be a suitable candidate as an underlying supporting semantic framework for building domain-specific models that could be mapped and integrated on top of UML-MARTE. Finally, a third approach is to restrict a general model to a specific subset, for instance by defining a restrictive UML profile. SysML can be seen as an example of defining a restrictive DSML on top of UML. In practice, we will have to combine all three approaches during embedded systems development. There is a need to utilise heterogenerous models that either are subsets of UML-MARTE, will be mapped onto it, or will be interfaced with it.

The correctness by construction approach is very powerful, but very difficult to realise when using heterogeneous models. If one can define the right abstractions and refine them, then one no longer needs to reopen the component box when connecting to them. This works fine for some combinations (e.g., from a synchronous reactive system description to a time-triggered architecture), but not for all combinations (e.g., to an event-driven architecture). One solution is to use only time-triggered architectures, and take the potential inefficiency as a fact, since reliability and composability might be more important than efficiency. This may be a good approach at least in some domains, such as aerospace and automotive, but may not apply in other domains, such as consumer electronics. This approach is best suited for a one-pass development process, while in practice incremental development is often used. The developer models part of the functionality, defines an architecture based on estimates, analyses and simulates the application, and then tries to obtain more concrete and precise estimates during later stages of the development. Iteratively, the developer can go back to change the behavior and functional and/or physical architecture at higher levels based on this refined information. As such, one starts with a simplified version of the system that gradually is adjusted and reshaped until a suitable system description is obtained.

Reconfiguration

Reconfiguration is often a key element even in safety-critical embedded systems. It is used to obtain a graceful degradation in an FDIR (Fault Detection,

Isolation and Recovery/Reconfiguration) approach, enabling continuous operation under degraded conditions. Often the failing system cannot be shut down but must reach a stable state as soon as possible. It is not clear if such safe state can always be reached, and if so in which timeframe. Therefore the aerospace industry heavily relies on redundancy in order to avoid system failures. Similarly, the introduction of steer-by-wire in the automotive industry is not a matter of computational power or price but rather of reliability and culture, since there is not yet the same level of culture about safety and reliability as in the aerospace industry.

Acknowledgements

We thank all workshop participants for the lively discussions. We especially thank our keynote speaker, Prof. Marco Di Natale, for his inspiring presentation. We also thank the workshop steering committee and program committee members for their support. This workshop was supported by the ARTIST2 Network of Excellence on Embedded Systems Design (http://www.artist-embedded.org) and by the EUREKA-ITEA project SPICES (http://www.spices-itea.org).

References

1. Van Baelen, S., Weigert, T., Ober, I., Espinoza, H., eds.: Second International Workshop on Model Based Architecting and Construction of Embedded Systems. CEUR Workshop Proceedings Vol. 507, CEUR, Aachen, Germany (2009)
2. Di Natale, M.: Semantics preservation issues in the design and optimization of SW architectures for automotive systems. In: [1]. pp. 9–9
3. Cancila, D., Terrier, F., Belmonte, F., Dubois, H., Espinoza, H., Gérard, S., Cuccuru, A.: SOPHIA: a modeling language for model-based safety engineering. In: [1]. pp. 11–25
4. Eyisi, E., Porter, J., Hall, J., Kottenstette, N., Koutsoukos, X., Sztipanovits, J.: PaNeCS: A modeling language for passivity-based design of networked control systems. In: [1]. pp. 27–41
5. Angelov, C., Sierszecki, K., Guo, Y.: Formal design models for distributed embedded control systems. In: [1]. pp. 43–57
6. Tan, L., Wachter, B., Lucas, P., Wilhelm, R.: Improving timing analysis for Matlab Simulink/Stateflow. In: [1]. pp. 59–63
7. Chkouri, M.Y., Bozga, M.: Prototyping of distributed embedded systems using AADL. In: [1]. pp. 65–79
8. Langsweirdt, D., Vandewoude, Y., Berbers, Y.: Towards intelligent tool-support for AADL based modeling of embedded systems. In: [1]. pp. 81–85
9. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Model-based codesign of critical embedded systems. In: [1]. pp. 87–91
10. Ersfolk, J., Lilius, J., Muurinen, J., Salomäki, A., Fors, N., Nylund, J.: Design complexity management in embedded system design. In: [1]. pp. 93–106
11. Botterweck, G., Polzer, A., Kowalewski, S.: Using higher-order transformations to derive variability mechanisms for embedded systems. In: [1]. pp. 107–121
12. Becker, B., Giese, H., Neumann, S., Schenck, M., Treffer, A.: Model-based extension of AUTOSAR for architectural online reconfiguration. In: [1]. pp. 123–137

Using Higher-order Transformations to Derive Variability Mechanism for Embedded Systems

Goetz Botterweck[1], Andreas Polzer[2], and Stefan Kowalewski[2]

[1] Lero, University of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie
[2] Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
{polzer|kowalewski}@embedded.rwth-aachen.de

Abstract. The complexity of embedded systems can partly be handled by models and domain-specific languages (DSLs) like Matlab / Simulink. If we want to apply such techniques to families of similar systems, we have to describe their variability, i.e., commonalities and differences between the similar systems. Here, approaches from Software Product Lines (SPL) and variability modeling can be helpful. In this paper, we discuss three challenges which arise in this context: (1) We have to integrate mechanisms for describing variability into the DSL. (2) To efficiently derive products, we require techniques and tool-support that allow us to configure a particular product and resolve variability in the DSL. (3) When resolving variability, we have to take into account dependencies between elements, e.g., when removing Simulink blocks we have to remove the signals between these blocks as well. The approach presented here uses higher-order transformations (HOT), which derive the variability mechanisms (as a generated model transformation) from the meta-model of a DSL.

1 Introduction

Embedded systems are present in our everyday life, for instance in washing machines (to save energy and water), in mobile devices (to simplify our lives) and in cars (to guarantee our safety).

In many cases, the engineering of embedded systems has to fulfill conflicting goals, such as reliability and safety on the one hand, and economic efficiency on the other hand. Moreover, the complexity of such systems is increasing due to the extension of functionality and increased communication with other systems.

One possibility to deal with the complexity, requirements and the cost pressure is to use model-based development techniques like Matlab / Simulink. The advantages of such an approach are that connections between system components are expressed in an intuitive way on a higher abstraction level, which hides implementation details. Other benefits are support for simulation and increased reuse due to the component-oriented approach.

In the context of this paper, we regard a “system” as a Matlab / Simulink model that contains components (represented as blocks) and provides a certain functionality.

Nowadays, such systems are reused with small, but significant changes between different applications. Such variability causes additional complexity, which has to be handled. Some well-known techniques for this are suggested by Software Product Lines (SPL) [1,2]. In the research presented here, we discuss how SPL techniques can be applied and adapted for the domain of model-based development of embedded systems.

The main contributions of this paper are (1) an analysis of Matlab / Simulink mechanism for variability, (2) concepts for realizing this variability with model transformations, (3) a mapping mechanism that adjusts the model according to configuration decisions (extension of [3]), and (4) concepts for "pruning", i.e., the cleanup of components that are *indirectly* influenced by configuration decisions.

First, we will give an overview of our approach (in Section 2). After this, we will explain methods of modeling and managing variability with Matlab / Simulink (Section 3). Subsequently, we explain the additional pruning methods (Section 4) and the implementation with model transformations (Section 5).

2 Overview of the Approach

We address the challenges described in the introduction with a model-driven approach that relies on higher-order transformations. Before we go into more details, we will give a first overview of our approach (see Figure 1). The approach is structured into two levels (1) *Domain Engineering* and *Application Engineering*, similar to other SPL frameworks [1,2]. Please note that we mark processes with numbers (❶ to ❼) and artefacts with uppercase letters (A and C; B will be used in later figures). In addition, we use indexes (e.g., A_d and A_a) to distinguish artefacts on Domain Engineering and Application Engineering level.

2.1 Domain Engineering

Domain Engineering starts with the consideration of the context and requirements of a product line. This *Feature Analysis* ❶ leads to the creation of a *Domain Feature Model* A_d, which defines the scope and available configuration options. Subsequently, in *Feature Implementation* ❷ a corresponding implementation is created. Initially, this implementation is given in the native Simulink format (`*.mdl` files). To access this native implementation in our model-based approach, we have to convert it into a model. To this purpose this we use techniques based on Xtext [4]. (see [5,6] for more details). As a result we get the corresponding *Domain Implementation Model* C_d.

To prepare the derivation of products, a higher-order transformation (HOT) ❸ is executed, which reads the DSL meta-model and generates the derivation transformation ❻. The generated transformation will later be used during *Application Engineering*.

2.2 Application Engineering

The first step in *Application Engineering* is *Product Configuration* ❹, where, based on the constraints given by the *Domain Feature Model* A_d, configuration decisions are made, which defines the particular product in terms of selected or eliminated features.

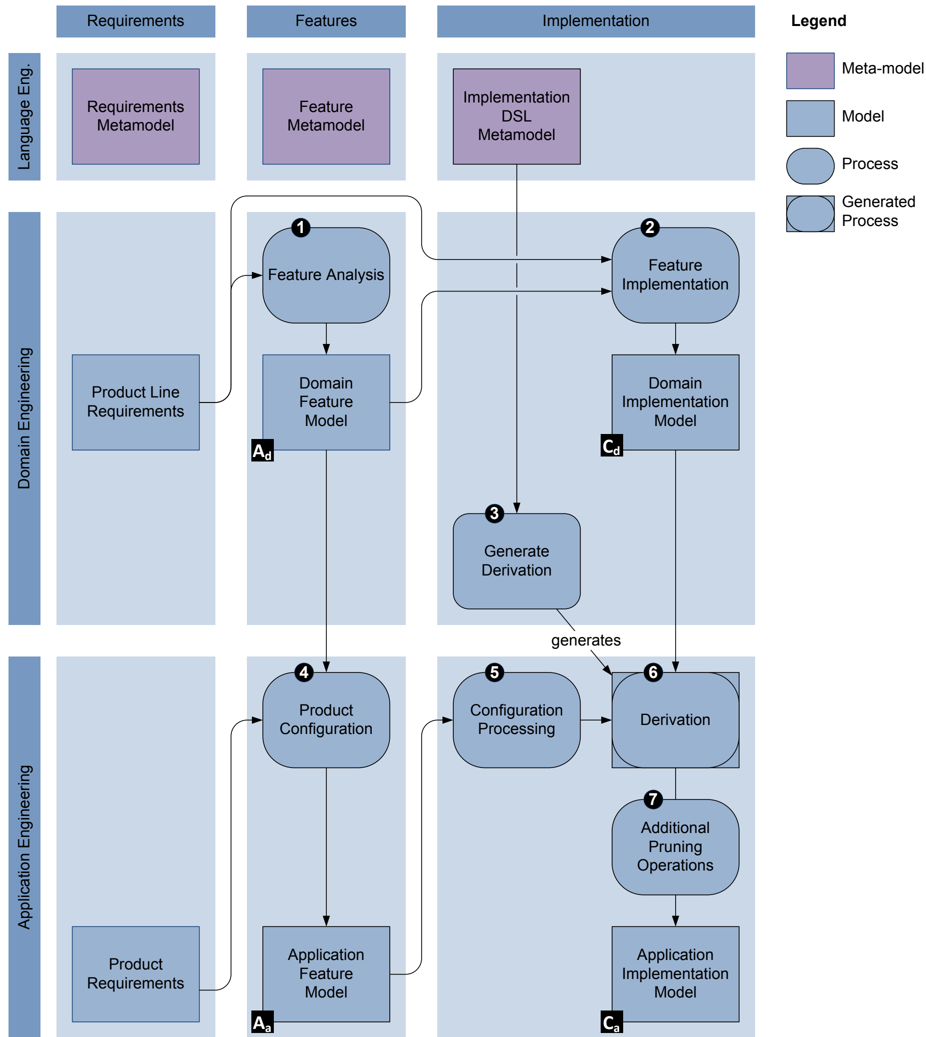


Fig. 1. Overview of the approach.

This results in a product-specific configuration which is saved in the *Application Feature Model* A_a. After some further processing ❺, this configuration is used in the *Product Derivation* ❻ transformation generated earlier by the HOT. This derivation reads the *Domain Implementation Model* C_d and – based on the product-specific configuration – derives a product-specific implementation. After additional pruning operations ❼, the result is saved as the *Application Implementation Model* C_a, which can be used in further processing steps (e.g., Simulink code generation) to create the final executable product.

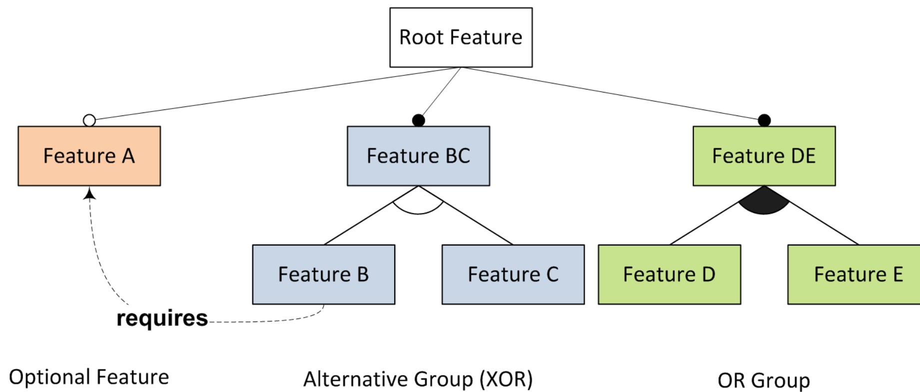


Fig. 2. Feature Model for the variant Matlab / Simulink model shown in Figure 3.

In the following sections, we will describe this in more detail. We start with a discussion of variability mechanisms during *Feature Implementation* ❷ and *Derivation* ❻ Section 3. Subsequently, the *Pruning Operations* ❼ are introduced in Section 4.

3 Managing Variability with Matlab / Simulink

In this section, we introduce the variability mechanisms which we used in Simulink models and how they are influenced by configuring a corresponding feature model. As an example see the feature model shown in Figure 2, which contains typical notational elements of feature models. This feature model has an optional `Feature A`, XOR-grouped `Feature B` and `Feature C`, and OR-grouped `Feature D` and `Feature E`. Additionally the `requires` relation indicates that `Feature B` needs `Feature A`, i.e., whenever B is selected A has to be selected as well. This model defines a set of legal configurations. Each of these configurations contains a list of selected features and represents a product of the product line.

Matlab / Simulink is a modeling and simulation tool provided by *The Mathworks*. The tool provides *Blocks* which can be represented graphically and *Lines* which indicate communication signals. Blocks can contain other blocks, which allows to abstract functionality in special blocks called *Subsystems*. A similar technique is used by summarizing multiple communication signals into *buses*.

In many cases, a feature can be implemented in Matlab / Simulink by one cohesive subsystem containing the feature's functionality. All necessary communication signals consumed by the subsystem have to be provided by an interface, represented by *Input ports*. These input ports are normally connected to buses. The information provided by a component are made available via *Output ports*, which are again organized as buses.

We identified different possibilities to introduce variability in Matlab / Simulink for the required *Feature Implementation* ❷. Predominantly, we use the technique called *negative variability*. When applying this technique a model is created, which contains the *union* of all features across all possible configurations. When deriving a product-

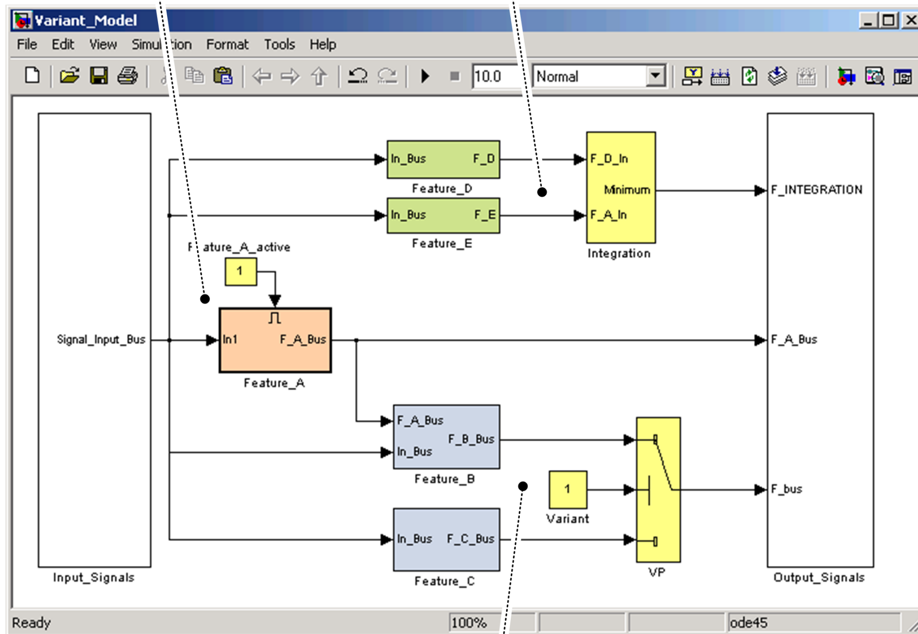


Fig. 3. Implementing Variability within a Matlab / Simulink model.

specific model this “union” model is modified such that features that were selected are activated and features that were not selected are deactivated or deleted.

To realize such a model in Simulink we can use two different approaches: (1) embedding the variability mechanisms *internally* (i.e., within the Simulink model) or (2) manipulating the model, based on variability information described *externally* (i.e., outside of the model). We will describe these two different techniques by explaining how common structures of feature models (Optional feature, Or Group, Alternative Group) can be implemented. For more information see [7,8]. An overview of feature diagrams and their formal semantics can be found in [9].

3.1 Variability mechanisms within the Simulink model

It is possible to realize variability by inserting artificial elements *into* the Simulink model. These elements do not implement functionality of a feature, but the variability mechanism described before. See the example model in Figure 3 where the blocks that

implement features (*Feature_A*, *Feature_B*, *Feature_C*, ...) have been augmented with additional elements for variability (*Feature_A_active*, *Integration*, *Variant*, *VP*).

Mandatory features are – by definition – present in all variants. Hence, there is nothing to do for the implementation of mandatory features when deriving a product. Please notice that there is no mandatory feature in the example.

Optional features can be realized in Matlab / Simulink models as a *triggered subsystem*, which is a special block that can be activated using a boolean signal (see *Feature A* in Figure 3). By using these mechanisms we are able to activate or deactivate a feature implementation.

When modeling *Alternative (XOR) group* and *Or group* we can use similar variability mechanisms. However, in addition we have to take care of the resulting signals and how they are fed into subsequent blocks. For alternative features we apply a *Switch* block (see the block *VP* in Figure 3) to choose the right output signal.

For *OR-related features* the integration of the results cannot be described in general, but has to be implemented depending on the particular case. In particular, we have to take into the account the case when more than one feature of the group is selected and present in the implementation. We can implement an integration block for such cases in different ways. One example, is the case where a limit is calculated for a certain dimension. Then the integration can be done by using a minimum (or maximum) function. In doing so, the lowest (highest) value is used by the system. As an example for this see *Feature D* and *Feature E* which are combined using an *Integration* block.

The *Feature Model* can contain additional constraints, e.g., the *Requires* line in Figure 2. These relations are sometimes introduced because of strategic or economical reasons, sometimes caused by technical dependencies in the implementation model.

3.2 Variability mechanisms outside of the Simulink model

A second possibility, besides variability within the model, is the direct manipulation of the model with an *external* tool. To this end, during product derivation it is necessary to analyze the structure of a model file and delete the blocks representing deactivated features in such a way that a desired configuration is obtained.

When creating the union model (covering all potential products) it might be necessary to create a model, which could not be executed within Simulink since the execution semantics of the created structure is undefined. This is caused by variability decisions, which are still to be made and cause conflicting elements to remain. Only when we further remove elements, then we get an executable model.

As an example consider the case when we want to combine signals of two alternative features in an XOR group. In the union model, we might create the blocks for these two features side-by-side and feed their results into the same inport of the next block. As long as we configure this model (i.e., when the variability is applied), some parts will be removed, such that we get a valid and executable Simulink model. However, if we leave the union model (i.e., no variability applied) we do not realize the exclusion of the two features ("the X in XOR"). This leads to two signals feeding into one input port, which is an invalid Simulink model with undefined execution semantics.

In the example in figure Figure 3 this would correspond to connecting the *F_B_Bus* output port of block *Feature_B* directly to the port *F_bus* of the bus *Output_Signals*,

while at the same time connecting the *F_C_Bus* output port of Block *Feature_C* directly to the same port *F_bus*. If we would try to execute such a model, Simulink would respond with an error message.

The advantage of this kind of external method is that we do not have to pollute the domain model with artificial variability mechanisms.

Analyzing both possibilities, we came to the conclusion that a combination of both is an appropriate way of introducing variability. There are two major requirements which have to be fulfilled introducing variability methods. On the one hand, we have to keep the characteristics of model based development (e.g., easy testing and simulation, capturing of dependencies possible), on the other hand the derived product should no longer contain any variability mechanisms. The mechanisms, which we introduced to realize this are explained in the next section.

3.3 Managing Variability

The mechanism that implements the variability of a feature tree has to fulfill certain requirements. In particular, it is important to keep the ability of simulating and testing the model. Therefore, it is necessary to build a model which has correct syntax, even after the internal variability mechanisms have been introduced. Additionally, the developer must have the possibility to introduce new features directly in the model. Due to the fact that there are space and performance requirements the mechanisms have to be removed if a product-specific model is derived form a configuration.

To this end, in general we used the approach of modeling variability *within* Simulink but with blocks which are specially marked as variability blocks. These special blocks are adopted afterwards according to the desired configuration. This means for obtaining a configuration, features that are not necessary will be deleted. Additionally signals between blocks are rerouted to be able to eliminate the variability blocks, which are not necessary in the derived product. The exact methods for a *Switch*-block, *Triggered subsystem* and, integration mechanism is given in the following paragraphs.

The *Switch*-block (see Figure 3 *VP*) is used to express the relation between alternative grouped features. Therefore only one of them will give their contribution to the system. Using this, the developer of the Simulink model is able to execute all features simply by choosing a configuration of the switch, which selects the corresponding feature implementation block.

When deriving the final executable product, it is necessary to delete those features that are not selected in the feature configuration. The output ports of each selected feature has to be connected with the port the switch block points to. All other corresponding signals have to be removed. In the end, no variability mechanism should be left in the model.

The situation is a bit simpler for *triggered* subsystems (see Figure 3 *Feature_A*), which implement optional features. During simulation these blocks can be selected easily using *trigger*-signals. If the corresponding feature is selected then the *trigger*-port has to be activated. When deriving a product-specific model a subsystem has to be deleted if the corresponding feature is deactivated. If a feature is activated nothing has to be done.

A general mechanism to join the signals of features is not so easy to adopt. When simulating the system the developer has to activate the desired features. This can be done either by using triggered subsystems to implement the features or just by disconnecting their signals with the block which joins the signals.

In case of deriving a real product two cases have to be distinguished. If only one feature is selected the subsystems realizes other feature and the variability block (see Figure 3 *Integration*) joining the signals can be deleted. If more than one feature is selected the subsystems implementing the features which are not selected can be deleted. But in this case the integration mechanism is still necessary for the remaining subsystems.

4 Pruning

After handling feature implementation and applying the basic variability decisions there are more adaptations that have to be done. In particular, we have to configure components that are *indirectly* influenced by the variability decisions.

One example for such adaptation are interfaces for input/output signals. As soon as features (i.e., the block implementing that feature) are removed from the model during product derivation some signals are no longer required. Hence, the interfaces have to be updated accordingly to (1) heal inconsistencies and/or (2) optimize the implementation. Such adaptation have to be done recursively on all levels of the hierarchy of subsystems.

Figure 4 shows an example with the two subsequent model transformations for product derivation (negative variability) and pruning. First, during *Product Derivation* ❻ some block are removed as a direct consequence of configuration decisions. Then, during *Pruning* ❼ dependent elements, which are indirectly influenced by the operation before, are cleaned up.

5 Implementation

The implementation discussed here (see Figure 5) is a technical realization of the approach shown earlier (see Figure 1). The technical implementation follows the same structure, with Domain Engineering (Processes ❶ to ❸) and Application Engineering (processes ❹ to ❼).

5.1 Generating the derivation transformation

The higher-order transformation (HOT) is the only transformation which is generated completely manual. It is realised by `Metamodel2Derivation.atl` ❸, and reads the meta-model of the DSL and generates the model transformation ❻, which is able to copy instances of this DSL.

Some excerpts of `Metamodel2Derivation.atl` are shown in Listing 1. The transformation is generated as an ATL `query` which concatenates the output as one large string. The helper function `generateCopyRules()` (see Listing 1, lines 4–8) generates copy rules for all classes in the meta-model. Details of each copy rule

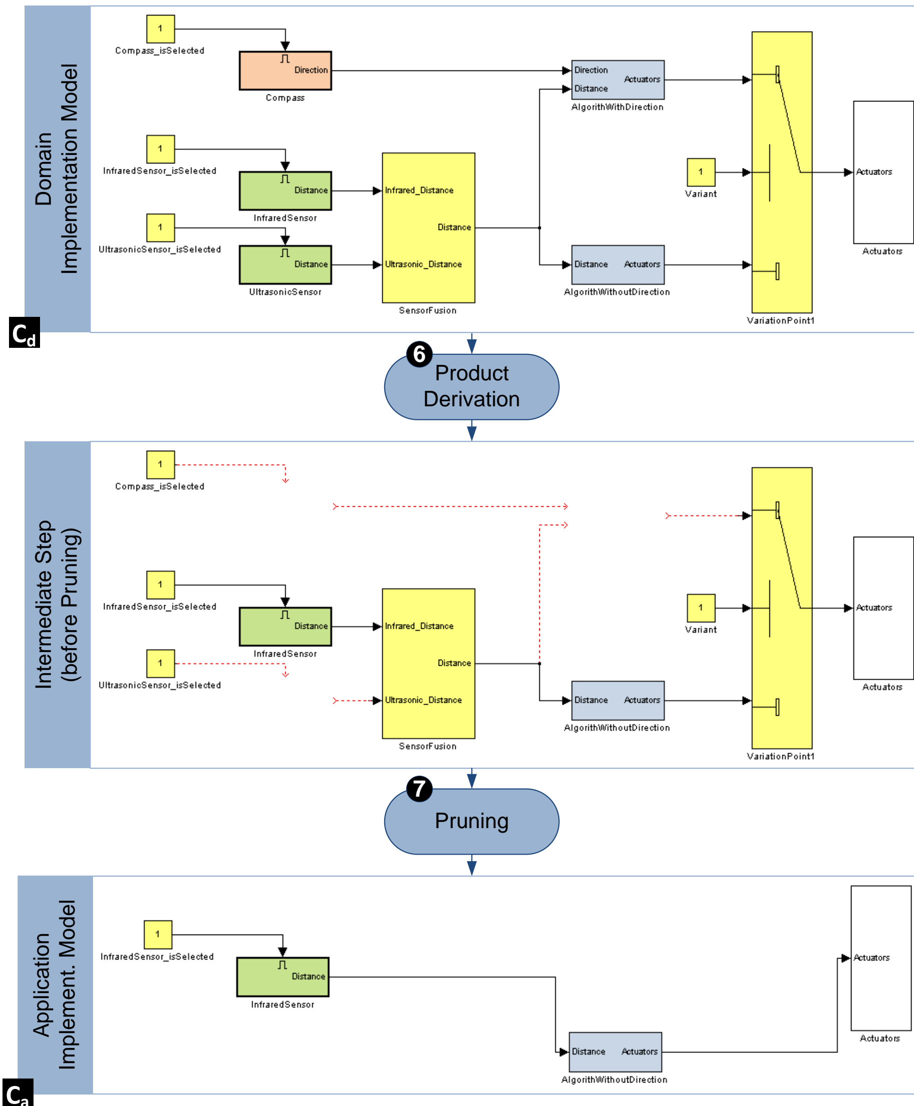


Fig. 4. Example of pruning.

are generated by the function `Class.toRuleString()` (see Listing 1, lines 10–21). The resulting strings form the transformation file for the model transformation ❻, which realizes the "negative variability". It is explained in the next section.

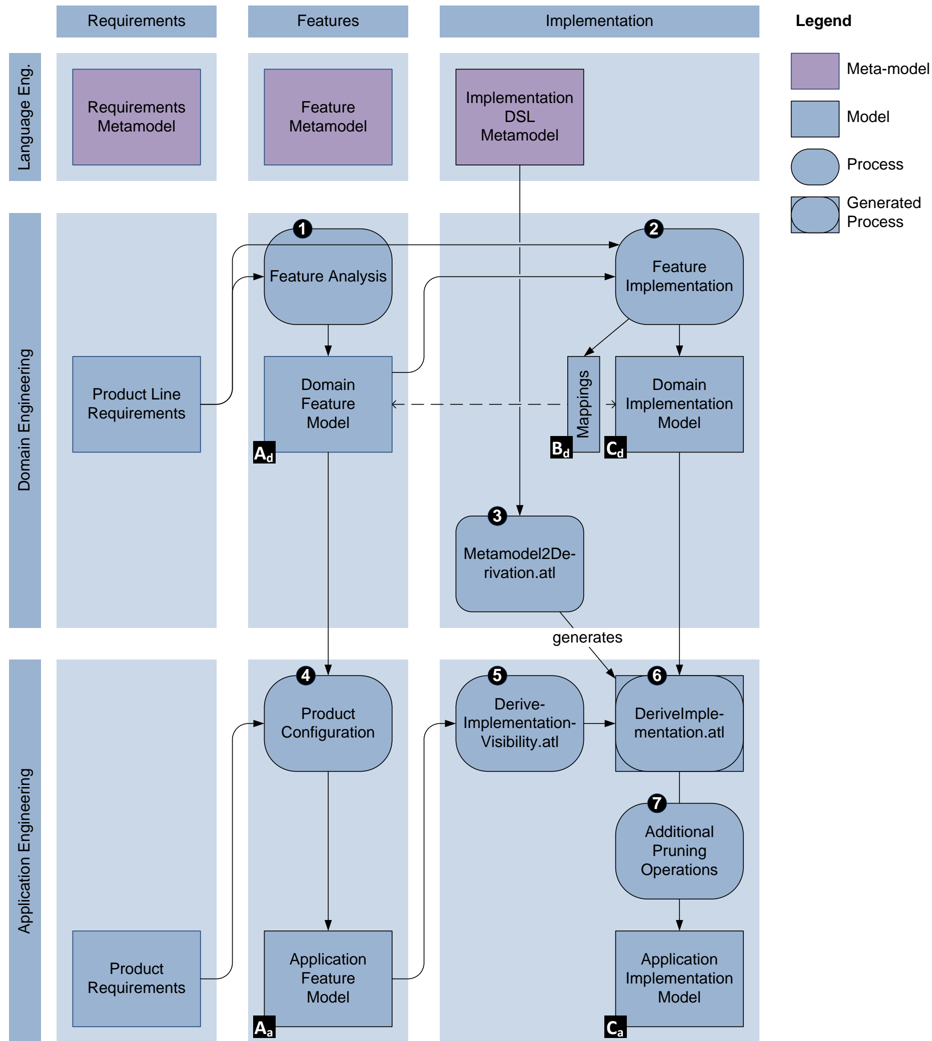


Fig. 5. Technical model workflow.

5.2 The derivation transformation

The generated derivation transformation `DeriveImplementation.atl` ❻ realizes a principle called "negative variability" [10] (also known as a "150% model"), where the *Domain Implementation Model* C_d contains the *union* of all product-specific models. Hence, the derivation transformation has to *selectively* copy only those elements, which will become part of the product-specific model, here the *Application Implementation Model* C_a. This is realized using the following mechanisms:

```
query Metamodel2Derivation =
[..]

helper def: generateCopyRules() : String =
  ECORE!EClass
    ->allInstancesFrom('IN')
    ->sortedBy(o|o.cname())
    ->iterate(e; acc : String = '' | acc + e.toRuleString());

helper context ECORE!EClass def : toRuleString() : String =
  if not self."abstract" and self->inclusionCondition() then
    'rule ' + self->cname() + ' {\n' +
    '  from s : SOURCEMETA!' + self->cname() +
            self->inputConstraint() + '\n' +
    '  to t : TARGETMETA!' + self->cname() +
            ' mapsTo s (' +
            self->contentsToString() + ')\n' +
    '}\n\n'
  else
    ''
  endif;
[..]
```

Listing 1. Metamodel2Derivation.atl, transformation (3), excerpt.

- For each meta-class in the DSL there is one copy rule. For instance, for the class `Block` a rule is created which copy instances of the class `Block`.
- Each copy rule contains a condition that refers to an `.isVisible()` helper function, which controls whether an element is "visible" for the particular product and, hence, is copied or not.
- Moreover, to avoid inconsistencies, whenever references are processed, it is checked if the referenced elements are visible, as well.
- The decision whether instances of a certain meta-class will be copied are implemented in the visibility functions. For instance, `Block.isVisible()` calculates this for all instances of `Block`. In the initial version of `DeriveImplementation.atl`, which is automatically generated from the meta-model, all visibility functions are set to true by default.
- In a second transformation, `DeriveImplementationVisibility.atl` ❺ Listing 2 these visibility functions are manually redefined and overloaded. These functions access the product configuration and determine, which elements go into the product and which do not. Later on, these will be overloaded over the default visibility functions, by using ATL's superimpose mechanisms.

During the executing of the product derivation transformations the visibility functions read the `Application Feature Model` Aa and decide how this influences the filtering of elements in the `Domain Implementation Model`.

```
module DeriveArchitectureDetermineVisibility;

create TARGET : TARGETMETA from SOURCE : SOURCEMETA, CONFIG :
    CONFIGMETA;

-- true if Block is referenced by a selected feature
helper context SOURCEMETA!Block def : isSelected() : Boolean =
[..]

-- true if Block is referenced by an eliminated feature
helper context SOURCEMETA!Block def : isDeselected() : Boolean =
[..]

helper context SOURCEMETA!Block def : isVisible() : Boolean =
  if self.isSelected() then
    if self.isDeselected() then
      true.debug('feature conflict for block' + self.name)
    else
      true
    endif
  else
    if self.isDeselected() then
      false
    else
      true -- default to visible
    endif
  endif;
[..]
```

Listing 2. DeriveImplementationVisibility.atl, transformation (5), excerpt.

To come to this decision, it is necessary to know how the various features in the feature model (A_d and A_a) are related to the corresponding elements in the Matlab / Simulink implementation model.

This is defined by the *Mappings* B_d between the *Domain Feature Model* A_d and the *Domain Implementation Model* C_d. With that mechanism we are able to link features (given in the configuration model) to the corresponding blocks (given in the implementation model).

To implement pruning operations we are currently experimenting with new user-defined methods. These methods adapt the copy rules such that the pruning methods briefly introduced in Section 4 are realized. These pruning operations are influenced by the configuration and the mapping of features. However, they affect model components which are only *indirectly* referenced by the mapping.

6 Related Work

Several projects deal with Product Derivation. The ConIPF project provides a methodology for product derivation [11]. ConIPF concentrates on the formalization of derivation knowledge into a configuration model. Deelstra et al. provide a conceptual framework for product derivation [12].

When dealing with variability in domain-specific languages a typical challenge is the mapping of features to their implementations. Here, Czarnecki and Antkiewicz [13] used a template-based approach where visibility conditions for model elements are described in OCL. In earlier work [14,15], we used mapping models and model transformations in ATL [16] to implement similar mappings. Heidenreich et al. [17] present FeatureMapper, a tool-supported approach which can map features to arbitrary EMF-based models [18].

Voelter and Groher [10] used aspect-oriented and model-driven techniques to implement product lines. Their approach is based on variability mechanisms in openArchitectureWare [19] (e.g., XVar and XWeave) and demonstrated with a sample SPL of home automation applications.

In earlier work [20,5], the authors have experimented with other mechanisms for negative variability (pure::variants Simulink connector [21] and openArchitectureWare's XVar mechanism [19]) to realize variability in Embedded Systems. The mechanisms were applied to microcontroller-based control systems and evaluated with a product line based on the Vemac Rapid Control Prototyping (RCP) system.

The approach presented here can be seen as an integration and extension of work from Weiland [22] and Kubica [23]. Both presented mechanisms to adopt Matlab / Simulink-models based on feature trees. Weiland implemented a tool which influences certain variability points in a Simulink model. However, variability mechanisms are not removed during variability resolution. The approach given by Kubica constructs a new Simulink model for a derived product.

Tisi et al. provide an literature review on higher-order transformations [24] including a classification of different types of HOT. Oldevik and Haugen [25] use higher-order transformations to implement variability in product lines. Wagelaar [26] reports on composition techniques for model transformations.

7 Conclusion

In this paper we presented an approach to introduce and adopt variability in a model-based domain specific language (Matlab / Simulink) for developing embedded systems.

All model transformations were implemented in the ATLAS Transformation Language (ATL) [16]. The original version was developed with ATL 2.0. We are currently experimenting with ATL 3.0 and its improved support for higher-order transformations.

With our approach we are able to simulate and test variable Simulink-models by introducing mechanisms to manage variability and additionally derive models which contain only the product specific components. This provides us with (models of) systems that are more efficient, e.g., with respect to memory and computation time.

8 Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/. The higher-order transformation ❸ was inspired by a case study by Dennis Wagelaar.

References

1. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. The SEI series in software engineering. Addison-Wesley, Boston, MA, USA (2002)
2. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering : Foundations, Principles, and Techniques. Springer, New York, NY (2005)
3. Beuche, D., Weiland, J.: Managing flexibility: Modeling binding-times in Simulink. [27] 289–300
4. Eclipse-Foundation: Xtext http://www.eclipse.org/Xtext/.
5. Polzer, A., Botterweck, G., Wangerin, I., Kowalewski, S.: Variabilitaet im modellbasierten Engineering von eingebetteten Systemen. In: 7. Workshop Automotive Software Engineering, collocated with Informatik 2009, Luebeck, Germany (September 2009)
6. Botterweck, G., Polzer, A., Kowalewski, S.: Interactive configuration of embedded systems product lines. In: International Workshop on Model-driven Approaches in Product Line Engineering (MAPLE 2009), colocated with the 12th International Software Product Line Conference (SPLC 2008). (2009)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute (1990)
8. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison Wesley, Reading, MA, USA (2000)
9. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International. (2006) 136–145
10. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan (September 2007)
11. Hotz, L., Wolter, K., Krebs, T., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J.: Configuration in Industrial Product Families - The ConIPF Methodology. IOS Press (2006)
12. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. The Journal of Systems and Software **74** (2005) 173–194
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, Tallinn, Estonia (September 29 - October 1 2005)
14. Botterweck, G., Lee, K., Thiel, S.: Automating product derivation in software product line engineering. In: Proceedings of Software Engineering 2009 (SE09), Kaiserslautern, Germany (March 2009)
15. Botterweck, G., O'Brien, L., Thiel, S.: Model-driven derivation of product architectures. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007), Atlanta, GA, USA (2007) 469–472
16. Eclipse-Foundation: ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/.
17. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: ICSE Companion '08: Companion of the 13th international conference on Software engineering, New York, NY, USA, ACM (2008) 943–944

18. Eclipse-Foundation: EMF - Eclipse Modelling Framework http://www.eclipse.org/modeling/emf/.
19. openarchitectureware.org: Official open architecture ware homepage http://www.openarchitectureware.org/.
20. Polzer, A., Kowalewski, S., Botterweck, G.: Applying software product line techniques in model-based embedded systems engineering. In: 6th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2009), Workshop at the 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada (May 2009)
21. Pure::systems: pure::variants Connector for Simulink
22. Weiland, J., Richter, E.: Konfigurationsmanagement variantenreicher simulink-modelle. In: Informatik 2005 - Informatik LIVE!, Band 2, Koellen Druck+Verlag GmbH, Bonn (September 2005)
23. Kubica, S.: Variantenmanagement modellbasierter Funktionssoftware mit Software-Produktlinien. PhD thesis, Univ. Erlangen-Nürnberg (2007) Arbeitsberichte des Instituts für Informatik, Friedrich-Alexander-Universität Erlangen Nürnberg.
24. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. [27] 18–33
25. Oldevik, J., Haugen, O.: Higher-order transformations for product lines. In: 11th International Software Product Line Conference (SPLC 2007), Washington, DC, USA, IEEE Computer Society (2007) 243–254
26. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: ICMT. Volume 5063 of Lecture Notes in Computer Science., Springer (2008) 152–167
27. Paige, R.F., Hartman, A., Rensink, A., eds.: Model Driven Architecture - Foundations and Applications, Proceedings of the 5th European Conference (ECMDA-FA 2009). Volume 5562 of Lecture Notes in Computer Science., Enschede, The Netherlands, Springer (June 2009)

Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration

Basil Becker[1], Holger Giese[1], Stefan Neumann[1],
Martin Schenck[2] and Arian Treffer[2]

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
[1]forename.surname@hpi.uni-potsdam.de
[2]forename.surname@student.hpi.uni-potsdam.de

Abstract. In the last few years innovations in the automotive domain have been realized by software, leading to a dramatically increased complexity of such systems. Additionally, automotive systems have to be flexible and robust, e.g., to be able to deal with failures of sensors, actuators or other constituents of an automotive system. One possibility to achieve robustness and flexibility in automotive systems is the usage of reconfiguration capabilities. However, adding such capabilities introduces an even higher degree of complexity. To avoid this drawback we propose to integrate reconfiguration capabilities into AUTOSAR, an existing framework supporting the management of such a complex system at the architectural level. Elaborated and expensive tools and toolchains assist during the development of automotive systems. Hence, we present how our reconfiguration solution has been seamlessly integrated into such a toolchain.

1 Introduction

Today most innovations in the automotive domain are realized by software. This results in a dramatically increasing complexity of the developed software systems.[1] The objective of the AUTOSAR framework is to deal with this complexity at the architectural level. Additionally, these systems need to deal with many situations concerning the context in which the software is operating. Such systems and especially the software, which is realizing essential functionalities of the overall system, need to be flexible to react to changes of its context. Regardless of whether such a system needs to react to failures or to other contextual situations[2], flexibility and robustness plays an important role in today's automotive applications.

[1] The complexity concerning the size of the developed software, the functionality realized by the software system and so on.

[2] An example of such a situation is in the case the car is connected to diagnostic devices.

Reconfiguration is one possibility to facilitate the flexibility and robustness of such systems. Different possibilities exist to realize reconfiguration within automotive software. One is to realize reconfiguration mechanisms at the functional level. Because the AUTOSAR framework primarily provides mechanisms to deal with the complexity at the architectural level, also the reconfiguration aspects should be available at the same level. Deriving architectural information from the functional level could be difficult or even impossible and hence we propose to specify reconfiguration aspects at the architectural level and to automatically derive the needed reconfiguration functionality based on the architectural information.

Further, in a typical development scenario one has to deal with a black-box view of components provided by third parties for which elaborated information about the included functionality is not available, what also hampers the management of reconfiguration aspects at the functional level. Another possible solution is to introduce a new approach inherently facilitating reconfiguration aspects in the context of automotive systems. Today, standard methods and tools already exist for supporting the development process of AUTOSAR. Because adapting existing tools or developing new ones is very costly, the propagation of such a new approach would hardly be suitable in practice. Summarizing, we have identified the need for a development approach that is able to provide reconfiguration capabilities at the architectural level, which can be seamlessly integrated into an existing development solution and that can also include third party components in the reconfigurable architecture. In this work[3] we show how reconfiguration capabilities, which are currently not included in the existing AUTOSAR approach can be supported at the architectural level without degrading existing development solutions, tools or the standard itself. We further show how the needed functionality for realizing the reconfiguration logic can be automatically generated based on the architectural information describing the reconfiguration. The used application example for our evaluation is related to the field of fault-tolerant systems since from our perspective such systems are one possible field to which reconfiguration as discussed in the remainder of this work can be applied.

The paper is organized as follows. In Section 2 we discuss existing approaches supporting reconfiguration relevant for automotive systems and especially those approaches providing reconfiguration capabilities at the architectural level. In Section 3 we briefly introduce the existing toolchain, which builds the technological foundation for our investigation concerning the developed extension for on-line reconfiguration within the AUTOSAR framework. Subsequently, in Section 4 we show how such a system is usually modeled with the given tools and how the additional reconfiguration aspects could be formulated based on the input/output of the existing toolchain. In Section 5 we show how these created additional reconfiguration aspects are automatically merged back into the original architecture and how the merged result fits into the existing tools without discarding or degrading parts of the original toolchain. Finally we give a short discussion concerning the current results of our work in Section 6.

[3] The work presented in this paper is an extension of [2].

2 Related Work

In several different areas of computer science ideas have been presented that are related to the approach we are going to present in this paper. In the field of software product lines and especially dynamic software product lines the topic of variable software has been addressed. The software architecture community has presented some work on the reconfigurability of robotic systems. Work tailored to the automotive domain has been carried out in the DySCAS project. We did some research on self-optimizing mechatronic systems.

In previous work we have presented a modeling technique called Mechatronic UML (mUML) that is suitable for the modeling of reconfigurable and self-optimizing mechatronic systems [1, 3]. However, the mUML approach differs from the one that will be presented in this paper, in the fact that mUML uses an own code generation mechanism and thus can hardly be integrated into existing development tool chains.

In the DySCAS[4] project dynamically self-configuring automotive systems have been studied [4, 5]. DySCAS does not provide a model based development approach tailored to the specification of reconfiguration. Reconfiguration is specified with policy scripts, which are then evaluated by an engine at runtime [6].

Software Product Line Engineering (SPLE) [7] aims at bringing the assembly line paradigm to software engineering. Typically a software product line is used to develop multiple variants of the same product. However, as the classical SPLE approach targets the design-time variability of software it is not comparable to the approach we are going to present in this paper. Recently a new research branch has emerged from SPLE called Dynamic Software Product Line Engineering [8]. In Dynamic Software Product Lines the decision of which variant to run has moved from design time to run time. Such an approach is presented in [10], where the authors describe a dynamic software product line, which is suitable for the reconfiguration of embedded systems. In contrast to our approach. this one is restricted to the reconfiguration of pipe-and-filter architectures and the reconfiguration has to be given in a textual form.

In [11] a framework for the development of a reconfigurable robotic system has been presented. However, the presented approach – in contrast to ours – does not support the model-driven development of reconfiguration. A policy-based reconfiguration mechanism is described in [13]. The authors present a powerful and expressive modeling notation for the specification of self-adaptive (i.e. reconfigurable) systems, but their approach requires too much computational power and is thus only remotely applicable to embedded systems. In [14] an approach based on mode automata has been presented. However, mode automata only support switching between different behaviors internal to a component and do not cover architectural reconfiguration.

[4] http://www.dyscas.org

3 Existing Development Approach

For the development of embedded systems – especially in the automotive domain – several tools exist that provide capabilities for model-based development of such systems. Tools used by companies are typically mature, provide reliable and optimized code generation mechanisms and are as expensive as complex. Hence, any technique that claims to be usable in the domain of embedded / automotive systems must be integrated into the existing toolchain. We will use this section to exemplary describe a toolchain, which might be used in the context of the AUTOSAR domain specific language.

3.1 AUTOSAR

The AUtomotive Open System ARchitecture (AUTOSAR) is a framework for the development of complex electronic automotive systems. AUTOSAR provides a layered software architecture consisting of the Application layer, the Runtime Environment and the Basic Software layer. Figure 1 shows the different layers of the architecture. The Basic Software layer provides services concerning HW access, communication and Operating System (OS) functionality [15]. The Basic Software provides several interfaces in a standardized form to allow the interaction between the Basic Software layer and the application layer routed through the Runtime Environment. The Runtime Environment handles the communication between different constituents of the application layer and between the application layer and the Basic Software layer (e.g., for accessing Hardware via the Basic Software [16]). The Application layer consists of Software Components, which can be hierarchically structured and composed into so-called Compositions. Software Components and Compositions can have ports and these ports can be connected via Connectors (see [17] for more details). The real communication is realized through the Runtime Environment in cases of local communication between Software Components (Compositions) on the same node (Electronic Control Unit) or through the Runtime Environment in combination with the Basic Software in cases of communication between different nodes.

The main focus of AUTOSAR is the modeling of architectural aspects and of structural aspects. The behavior modeling (e.g., needed control functionality for reading sensor values and setting actuators) is not the main focus of the AUTOSAR framework. For modeling such behaviour, existing approaches and tools can be integrated into the development process of AUTOSAR. One commonly used tool for the model based development of behavior is MATLAB/Simulink (as described in Section 3.2). For executing such functionality AUTOSAR provides the concept of Runnables, which are added as a part of the internal behavior of a Software Component. Developed functionality could be mapped to Runnables and these Runnables are mapped to OS tasks. Additionally, events can be used to decide inside an OS task whether specific runnables are executed at runtime (e.g., runnables could be triggered by events if new data has been received via

a port of the surrounding Software Component). See [18] for more details about the OS provided by the AUTOSAR framework.

Once the modeling and configuration hass been done, in the current release 3.1 of AUTOSAR changes at runtime concerning the structure of the application layer (e.g., restructuring connectors) are not facilitated by the framework.

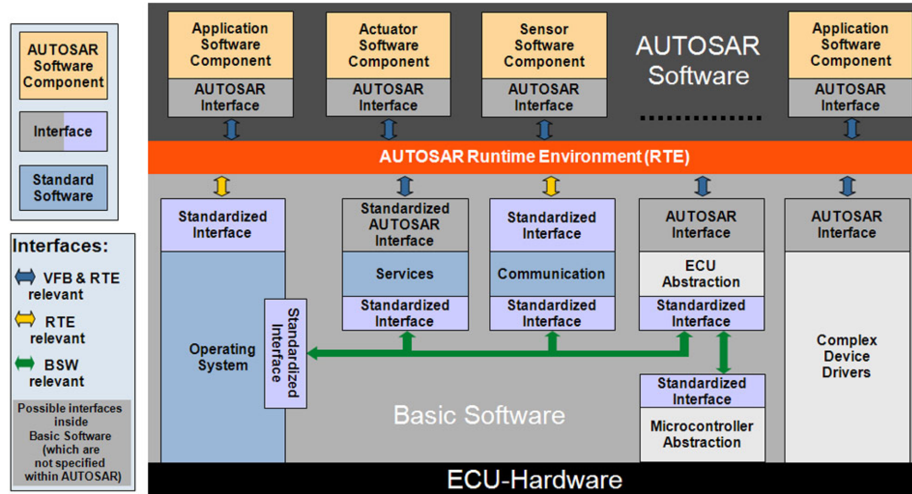


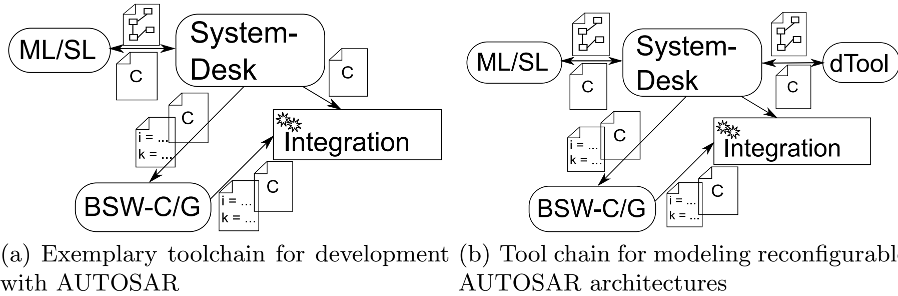
Fig. 1. The AUTOSAR layered architecture[5]

3.2 Existing Toolchain

The scheme in Figure 2(a) shows one possible toolchain for the development of AUTOSAR systems. Rectangles with rounded corners represent programs; rectangles with cogwheels stand for processes. The arrows indicate exchange of documents; the type of the document (i.e. models, C code or parameters) is annotated to the arrows. The system's architecture (i.e. components, ports and connectors) is modeled in SystemDesk.[6] Together with the architecture SystemDesk also supports the modeling of the system's deployment to several ECUs. The components behavior is specified using MATLAB with the extension Simulink. For MATLAB/Simulink (ML/SL) special AUTOSAR block sets exist, which allow the import of AUTOSAR components specified in SystemDesk into MATLAB. Within MATLAB the functionality (e.g., control functionality) of the internal behavior of the AUTOSAR components then can then be realized.

[5] Picture taken from http://www.autosar.org/gfx/media_pictures/AUTOSAR-components-and-inte.jpg.

[6] http://www.dspace.de



(a) Exemplary toolchain for development with AUTOSAR (b) Tool chain for modeling reconfigurable AUTOSAR architectures

Fig. 2. The current and the extended toolchain for the development with AUTOSAR

Further SystemDesk supports the generation of optimized C code, which conforms to the AUTOSAR standard concerning the Runtime Environment (see Section 3.1). Together with the C implementation of the software components modeled in SystemDesk, the generated output also contains a configuration for the basic software layer. This layer is generated from specialized tools (e.g., Tresos by ElectroBit, abbreviated as BSW-C/G in Figure 2) and is specific to the system modeled in SystemDesk and the available hardware. At the integration step a build environment compiles the generated C code and builds the software running on each ECU.

3.3 Evaluation Example

The used application example for showing the reconfiguration capabilities that are supplemented to the existing AUTOSAR framework in our approach is the reconfiguration of a set of adjacent-aligned distance sensors. The discussed evaluation example allows sensor failures to be reacted to in the manner that the failure of individual sensor instances is compensated.[7]

Such adjacent-aligned sensors are commonly used in a modern car, e.g., in the case of a parking distance control. Such a parking distance control uses sensors (e.g., ultrasonic sensors) embedded in the front or rear bumper for measuring the distance to nearby obstacles.

Additionally in Section 5.3 we discuss the evaluation results of experiments we have made on an evaluation platform using the techniques described in Section 4.

4 Modeling Reconfiguration

In order to make an AUTOSAR system architecture reconfigurable, some additional concepts are needed. The toolchain needs to be extended in a certain way that extensions do not invalidate the existing toolchain. From our perspective

[7] For our application example we assume that a sensor failure can be observed at the level of Software Components.

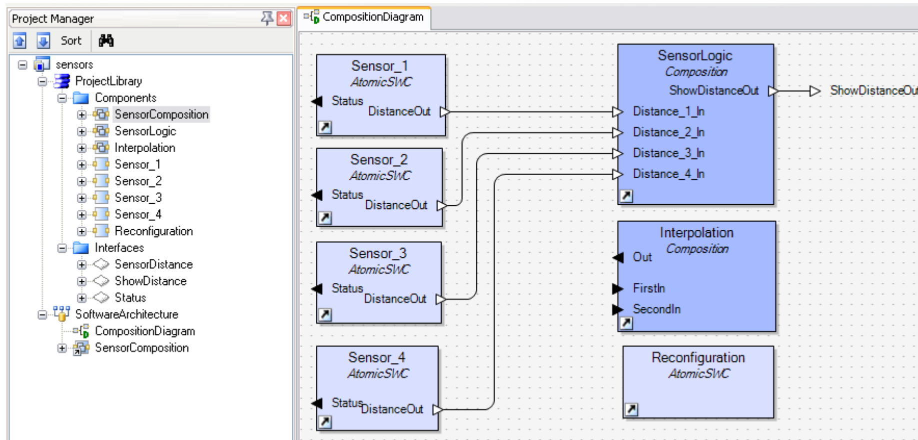


Fig. 3. SystemDesk screenshot of the evaluation example configuration

the best way is to integrate an optional tool that can be plugged into the existing toolchain.

4.1 Extended Toolchain

Our modeling approach is currently restricted to the modeling of AUTOSAR software architectures. The toolchain in Figure 2(b) shows our approach of extending the existing toolchain by another tool without degrading existing ones. By using this proposal the developer is free to choose whether he wants to use our given enhancement or not. He can either model an architecture that does not provide any reconfiguration or he can use our tool in addition and empower himself to specify and realize reconfiguration aspects. The advantages are obvious: better control and overview due to the diagrammatic depiction. A detailed description of the extended toolchain can be found in [2].

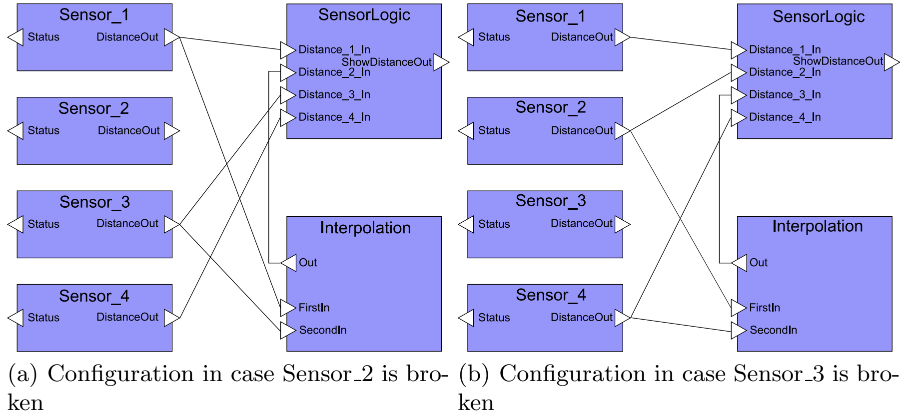
Figure 3 shows the relevant part of the software architecture concerning our application example modeled in SystemDesk. Like depicted on the right side of Figure 3 the composition consists of four Software Components representing the distance sensors[8] connected to another composition *SensorLogic* evaluating the sensor values to a single value provided by the port *ShowDistanceOut*.[9]

The above mentioned elements (Software Components, ports and connectors) are used to describe the software when no reconfiguration is intended. Some additional elements shown in Figure 3 are described in more detail in the following section. These elements (*Interpolation*, *Reconfiguration* and the unused ports of the sensors) are used later to realize the reconfiguration functionality.

---

[8] The ports accessing the HW via the Runtime Environment and Basic Software are not shown here because they are not object of reconfiguration.

[9] To allow a better understanding *SensorLogic* calculates a single output value based on the different input values. Potentially also several output values can be computed.



(a) Configuration in case Sensor_2 is broken (b) Configuration in case Sensor_3 is broken

Fig. 4. Two configurations of the architecture for two different scenarios

4.2 dTool

The usual modeling procedure is not altered until the modeling in SystemDesk[10] is initially done as described above. After the model from SystemDesk is exported in form of an XML file[11] and loaded into the *dTool* the constituents concerning the reconfiguration can be specified. Using the dTool we are now able to model two different aspects relevant for the reconfiguration. On the one hand, our tool allows new configurations to be created that differ from the initial one. Such differences are alternative connections (in the form of connectors) between components and/or compositions. Which parts of the architecture are relevant concerning reconfiguration is indicated by the Software Component *Reconfiguration* included in the original SystemDesk model. Alternatively, the dTool allows relevant parts of the imported architecture to be chosen manually. On the other hand, our dTool allows an automaton to be modeled, which specifies how to switch between the modeled configurations.

Figure 4(a) depicts the configuration (modeled in the dTool) associated with the state that sensor two is broken. In the shown configuration the value of the port *DistanceOut* from the broken sensor *Sensor_2* is not available. Consequently the value sent to the port *Distance_2_In* of the composition *SensorLogic* is interpolated from the two sensor values of the first and the third sensor via the additional composition *Interpolation*.

Figure 4(b) shows the configuration associated with the state that sensor three is broken and the value sent to the port *Distance_3_In* of the composition *SensorLogic* is interpolated based on the sensor values of the second and the fourth sensor.

---

[10] http://www.dspace.de/ww/en/ltd/home/products/sw/system_architecture_software/systemdesk.cfm

[11] The AUTOSAR framework specifies XML-Schemes for exchanging AUTOSAR models in a standardized form.

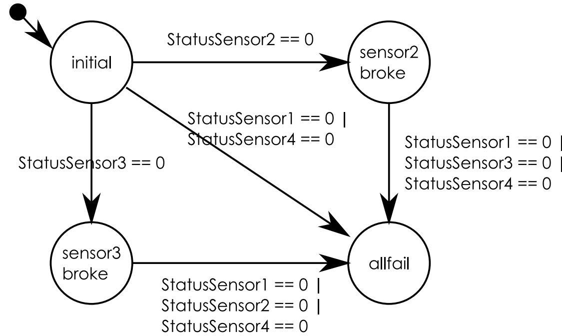


Fig. 5. Reconfiguration automaton in the dTool

The composition *Interpolation* used here provides some functionality for interpolating two different sensor values. This functionality has been added specifically for our application example.[12] This interpolation functionality is used to approximate the value of a broken sensor based on the values of two adjacent sensors. It is potentially possible to integrate this functionality into an existing Software Component, but for a better understanding we decided to introduce a new Software Component for this purpose.

The second relevant part for the reconfiguration that can be modeled in the dTool, is the automaton shown in Figure 5 specifying how to switch between different configurations. The automaton consists of the initial state *initial* where all four sensors work correctly, the state *sensor2broke* where the second sensor is broken, the state *sensor3broke* where the third sensor is broken, and the state *allfail* where the first or the fourth sensor or more than one sensor is broken. Transitions between these states specify which reconfiguration is applied at runtime. The transitions are further augmented with guards. These guards are expressions over the values provided by components within the reconfigurable composition, which provide information relevant for the reconfiguration (in our case this information is provided via the *Status*-ports of the four Sensor-Software Components). An example of such a guard is shown at the transition from state *initial* to state *sensor2broke* requiring that the status port of the Software Component *Sensor_2* provides the value 0 (indicating a broken sensor).

For the application example we assume that such status ports of the Software Components representing the sensors exist as we otherwise were not able to observe each sensor's status.[13]

---

[12] In our application example this functionality has been realized using MATLAB/Simulink to provide more vaguely pesimistic distance values.

[13] Alternatively an observer could be realized in the form of an additional Software Component evaluating the sensor values over time and providing the status ports. If the measured values of consecutive points in time repeatedly have improper values (too big differences) a malfunction can be deduced.

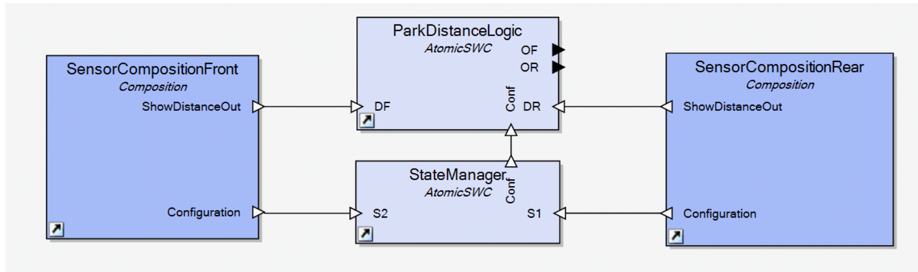


Fig. 6. An example for a hierarchy of compositions

AUTOSAR allows a hierarchy of compositions to be modeled. Our approach supports this AUTOSAR feature seamlessly. In the dTool a hierarchy of compositions is represented as two or more nested reconfiguration compositions. It is possible that one reconfigurable composition contains more than one inner reconfigurable composition. In Figure 6 an example for a nested composition is shown. The example is a combination of two adjacent-aligned distance sensor arrays (represented through *SensorCompositionFront* and *SensorComposition-Rear*) like shown in Figure 3. The composition on the left represents the parking sensors of the front bumper, and the composition on the right the sensors of the rear bumper. Depending on the current configuration of the used compositions (*SensorCompositionFront* and *SensorCompositionRear*) the *StateManager* in Figure 6 decides which configuration to choose resulting in different functional behavior realized by *ParkDistanceLogic*.[14]

5 Merge

In its current version the AUTOSAR standard does not support reconfiguration as a first-class modeling element. While reconfiguration can be realized at the functional level when using the AUTOSAR framework, this won't be visible at the modeling elements provided by the framework. Thus, SystemDesk also does not support modeling of diagrams that represent different variations of one composition. Hence the direct import of the reconfiguration, which we have modeled in the *dTool*, is impossible. Nevertheless we want to make use of SystemDesk's elaborated and AUTOSAR standard conform code generation capabilities. We had to find a way to translate the reconfiguration behavior into a SystemDesk/AUTOSAR model. This is done by merging all configurations into one final model. In the final model, the reconfiguration logic will be encapsulated by two components, the RoutingComponent and the StateManager.

[14] Although in the shown example no architectural reconfiguration is realized, our approach supports such reconfiguration for hierarchies in the same way as shown above for the non-hierarchical case.

5.1 Merging configurations

Our modeling approach only allows the reconfiguration of connections between components, and is not suitable for the addition and removal of components at runtime.[15] Hence, a merged configuration consists of all components that have been modeled in SystemDesk at the early stages. Connections that do not exist in all configurations are redirected via a special component, called RoutingComponent. Therefore, the first step is to build the intersection of all configurations. Connections found here are directly inserted into the merged model. Next the RoutingComponent is added.

Generating the RoutingComponent The RoutingComponent intersects every connection that is not invariant to the reconfigurable composition. Therefore, the RoutingComponent has to know at each point in time which configuration is currently the active one. Which configuration is active, is determined by the evaluation of the current configuration and the valuation of the variables used in the guards of the reconfiguration automaton (cf. Figure 5). Because evaluating at each point in time the actual value of the automaton and repeatedly sending this value to the RoutingComponent is much too expensive, we have implemented a different strategy.

The configurations modeled in the dTool each get a unique number. The RoutingComponent receives the number of the currently active configuration via a special input port. Using this information the RoutingComponent can be implemented as a sequence of switch statements. The computation of the current active configuration is done in a second component – the StateManager. The dTool automatically generates a runnable for the RoutingComponent containing the described behavior.

StateManager The StateManager – as briefly mentioned above – is responsible for the computation of the currently active configuration. Therefore, it has to be connected with all ports that provide values, which are used in the guards of reconfiguration automaton. Each time the StateManager receives an update on its ports, it has to evaluate the automaton again and change the value of the currently active configuration accordingly.

Updates to the StateManager's ports are signaled by events, which then trigger the StateManager's evaluation function.[16]

The StateManager is the component in the merged project that has to deal with the hierarchy of compositions. The nested reconfigurable composition's StateManager therefore provides a port that publishes its current configuration.

[15] Please note that the dTool allows to modeling configurations, that do not contain all components. The semantics is that the components are hidden. A dynamic loading of components is not supported by AUTOSAR.

[16] Event mechanisms in form of AUTOSAR Runtime Environment events (for more information see [16]) have been used to trigger the runnable realizing the functionality of the StateManager.

This port's value is then used by the StateManager of the surrounding reconfigurable composition. Please note that compositions in AUTOSAR can be used at multiple places. Hence in one situation a composition can be used in a nested way and once at the topmost level. Therefore, using the configuration of the surrounding reconfigurable composition's StateManager is not allowed. Referring to Figure 6 the StateManagers included in the compositions SensorComposition-Front and SensorCompositionRear cannot use the status of the StateManager of the surrounding composition, but the StateManager of the surrounding composition can use the StateManagers included in the sub-compositions.

5.2 Final SystemDesk project

Figure 7 shows the Sensor composition after exporting the merged model to SystemDesk again. The components for the distance sensors are all connected to the RoutingComponent, which is named *Reconf* in this diagram. The system modeled in our application example does not allow an interpolation for the sensor components one and four. Subsequently, these components are always directly connected with the SensorLogic component and are not handled by the Routing-Component. Nevertheless, they also have to be connected to the RoutingComponent as the sensor values are used to interpolate the second, respectively third sensor in the case of a failure.

The StateManager is depicted below the RoutingComponent and is connected to the RoutingComponent through the *Conf* ports, which provide information about the currently active configuration. As defined in the reconfiguration automaton (cf. Figure 5), the decision of which configuration to use depends on the values of the sensor components' status ports. Therefore the StateManager is connected to those ports. As the reconfiguration automaton does not rely on any values provided by the Interpolation or SensorLogic component, the StateManager is not connected with them.

5.3 Evaluation Results

The above described approach for the modeling and realization of reconfiguration aspects has been evaluated within a project arranged at the Hasso Plattner Institute in collaboration with dSPACE GmbH.

As an evaluation platform for the shown approach the Robotino robot[17] has been used, which provides an open platform for running C/C++ programs (among others) on a Real-Time Operating System (RTOS). The RTOS is provided in form of RTAI[18], which is a real-time extension for the Linux operating system. To be able to evaluate the developed concepts on this platform an execution environment has been realized based on the existing RTAI Linux, which allows the outcome of the above described extended toolchain, including the resulting parts of the reconfiguration functionality, to be compiled and executed.

[17] http://www.robotino.com

[18] For more details see https://www.rtai.org.

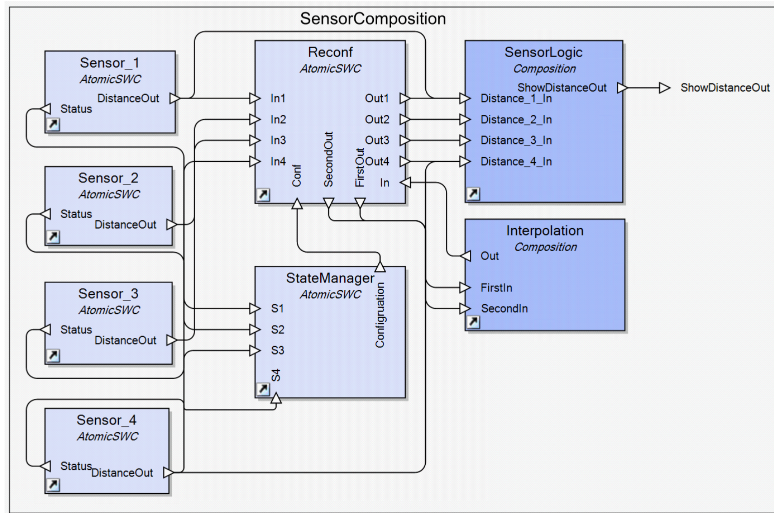


Fig. 7. Resulting merged software architecture in SystemDesk

The robot provides nine distance sensors uniformly distributed around its chassis. In the context of our evaluation experiments we modeled the reconfiguration of distance sensors accordingly to the above used evaluation example using nine instead of four sensors.[19]

The generated source code of the different tools has been compiled and executed on the platform to show the applicability of our approach. In addition, we analyzed the overhead resulting from the reconfiguration functionality added by our approach in comparison to the original functionality without any reconfiguration capabilities. For this purpose we measured the execution time of the generated reconfiguration automaton included in the added StateManager in combination with the parts resulting from the routing functionality realized in the additional RoutingComponent (both components are shown in Figure 7).

In the case of the nine sensors provided by the robot we measured execution times of the relevant parts concerning the reconfiguration functionality between 20 and 100 microseconds depending on the type of reconfiguration (react to the defect of one or several sensors at the same point in time). The tests have been realized on the equivalent execution platform on which the real functionality has been executed when running the application example on the robot.[20] While the robot provides a more powerful processor like is the case for the most Electronic Control Units (ECUs) used within a modern car, even by using a platform or processor that has only a tenth of the computation power we will not reach

[19] For a better understanding we decided to only show four sensors in the previous sections.

[20] The robot is equipped with 300 MHz processor.

an overhead concerning the reconfiguration leading to an execution time much greater than one millisecond.

6 Conclusion & Discussion

In this paper we have presented an approach to extend AUTOSAR architectures with reconfiguration capabilities. Like discussed in [12] there exist two different strategies for how and when to identify the appropriate configuration. First, identification of possible system configurations is completely done at runtime or second, the identification is specified by the developer at design time. Our approach has to fit into the existing AUTOSAR framework and has to meet certain performance requirements. Therefore, the second strategy has been chosen. Thus, the possible configurations as well as the decisions for when to switch between them are specified at design time. In such a way a set of possible variants in the form of configurations is defined before runtime and one of these is selected at runtime accordingly. In doing so the overhead added to the resulting reconfigurable architecture has been shown to be negligible, but the developer rewards an easier development of reconfiguration logic, which otherwise has to be done manually at the functional / implementation level. We have successfully shown that it is possible to use high-level architectural modeling techniques without generating massive runtime overhead. As a result of the chosen application domain, the used AUTOSAR framework and the usage of a highly optimized code generator, modeling artifacts (like components, ports and connectors) are no longer visible at runtime. To be able to apply the first type of strategy described in [12] and reason at runtime about possible configurations as well as the decision when to switch between them our approach needs to be extended. The architectural information and the current state concerning the chosen configuration need to be available at runtime, as well as some mechanism realizing the decision regarding in which situation to switch between them. The tradeoff between different aspects like flexibility, efficiency and safety when applying the different strategies needs to be seriously considered, especially for safety-critical systems like automotive systems. Although our approach has only been evaluated in the context of AUTOSAR it should be applicable to almost any component-based development approach, e.g., for AADL [9].

For the future we plan to also support the reconfiguration of distributed compositions. From an architectural point of view a distributed composition does not differ from a local one, as AUTOSAR completely hides the communication details in the Runtime Environment layer from perspective of the application layer. Anyway, a distributed scenario contains enough challenges such as timing delays, Basic Software configuration, deployment decisions concerning Routing-Components, just to name a few. Further the high-level architectural modeling we have introduced in this paper also allows the verification of the modeled systems. First attempts in these directions have been very promising and we are looking forward to looking into the details.

Acknowledgment We would like to thank dSPACE GmbH and especially Dirk Stichling and Petra Nawratil for their support in setting up the project. We also whish to thank: Christian Lück, Johannnes Dyck, Matthias Richly, Nico Rehwaldt, Thomas Beyhl, Thomas Schulz and Robert Gurol.

References

1. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. Intl. Journal on Software Tools for Technology Transfer **10**(3) (2008) 207–222
2. Becker,B., Giese, H., Neumann, S., Treffer, A., Schenck, M.: Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration. In: Proc. of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009) 105–112
3. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular design and verification of component-based mechatronic systems with online-reconfiguration. In: Proc. SIGSOFT '04/FSE-12, New York, NY, USA, ACM Press (2004) 179–188
4. Feng, L., Chen, D., Törngren, M.: Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In: Proc. of 47th IEEE Conference on Decision and Control. (2008) 3737–3742
5. Anthony, R., Ekeling, C.: Policy-driven self-management for an automotive middleware. In: HotAC II: Hot Topics in Autonomic Computing on Hot Topics in Autonomic Computing, Berkeley, CA, USA, USENIX Association (2007)
6. DySCAS Project: Guidelines and Examples on Algorithm and Policy Design in the DySCAS Middleware System, Deliverable D2.3 Part III. (February 2009) Available online: http://www.dyscas.org/doc/DySCAS_D2.3_part_III.pdf.
7. Pohl, K., Böckl, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer, Berlin Heidelberg New York (2005)
8. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic Software Product Lines. Computer **41**(4) (2008) 93–95
9. Feiler, Peter H., Gluch, David P., Hudak, John J.: The Architecture Analysis & Design Language (AADL): An Introduction. Techreport no. CMU/SEI-2006-TN-011 - Software Engineering Institute, Carnegie Mellon University (2006)
10. Kim, M., Jeong, J., Park, S.: From product lines to self-managed systems: an architecture-based runtime reconfiguration framework. In Proc. of Workshop on Design and Evolution of Autonomic Application Software, ACM (2005) 1–7
11. Kim, D., Park, S., Jin, Y., Chang, H., Park, Y.S., Ko, I.Y., Lee, K., Lee, J., Park, Y.C., Lee, S.: SHAGE: a Framework for Self-managed Robot Software. In: Proc. SEAMS '06, Shanghai, China, ACM (2006) 79–85
12. Trapp, M., Adler, R., Förster, M., Junger, J.: Runtime adaptation in safety-critical automotive systems. In: SE'07: Proc. of the 25th conference on IASTED
13. Georgas, J.C., Taylor, R.N.: Policy-based Self-adaptive Architectures: A Feasibility Study in the Robotics Domain. In: Proc. SEAMS '08, ACM (2008) 105–112
14. Talpin, J.P., Brunette, C., Gautier, T., Gamatié, A.: Polychronous Mode Automata. In: EMSOFT '06: Proc. of the 6th ACM & IEEE International Conference on Embedded software, ACM (2006) 83–92
15. AUTOSAR GbR: List of Basic Software Modules. Version 1.3.0.
16. AUTOSAR GbR: Specification of RTE. Version 2.1.0.
17. AUTOSAR GbR: Specification of the Virtual Functional Bus. (2008) Version 1.0.2.
18. AUTOSAR GbR: Specification of Operating System. (2009) Version 3.1.1.