

Generative Design of Hardware-in-the-Loop Models

Uwe Ryssel, Joern Ploennigs,
Klaus Kabitzsch
Department of Computer Science
Dresden University of Technology
Dresden, Germany
{uwe.ryssel, joern.ploennigs,
klaus.kabitzsch}@inf.tu-dresden.de

Michael Folie
ITK Engineering GmbH
Munich, Germany
michael.folie@itk-engineering.de

ABSTRACT

Embedded software is used nowadays in many applications. To ensure the function and reliability of the software, hardware-in-the-loop methods are commonly used to test it in a simulated environment. Due to the rising complexity of the implemented function, performance limitations and practicality reasons, the simulations are often specialized to test few aspects of the software and reach therewith a high diversity. This diversity is difficult to manage for a user and results in wrong selected components and compatibility problems. This paper presents a generative programming approach that handles the diversity and includes an interface concept to evaluate the compatibility. To use this approach profitably in real-world applications, a migration approach is presented using a model analyzer. The evaluation of the presented approach is exemplified in the automotive domain using MATLAB/Simulink.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software;
D.2.9 [Software Engineering]: Management—*Software configuration management*; I.6.4 [Simulation and Modeling]: Model Validation and Analysis; I.6.5 [Simulation and Modeling]: Model Development

General Terms

Design, Management, Reliability

Keywords

Hardware-in-the-loop, Generative programming, Model design, Model migration

1. INTRODUCTION

Various application areas use embedded software, which performs control tasks in process and building automation,

and in the automotive domain. The complexity and diversity of the implementations steadily increase with new features added to the already complex control tasks like ABS or ESP. The challenge of the engineering process is to ensure the reliability and safety of the software. Especially in the automotive domain safety guarantees are crucial for all future x-by-wire technologies.

To increase and guarantee the safety and reliability of the created software components they are tested repeatedly during their development process. Most of these tests are performed within a simulated car environment, as cars and hardware are unavailable in early stages or the tests are too risky. These so-called *in-the-loop* tests are introduced and classified by the development stage of the system under test in Section 2.1.

A car is a very complex physical system, where the simulation models of simple subsystems already get large, complex, and cumbersome to simulate. To increase the simulation performance, the designers use different simulation models that are tailored to the individual test case and function of interest. This results in a high number of simulation models for the components with their optional functions and the environment in different conditions and levels of detail.

The simulation models are often built of components itself. This simplifies and speeds up the development process not only due to the reuse of components. All simulation models and components are stored in a *component library*. These libraries often develop a *library scaling problem* [1], as each feature added to a family of simulation models results in multiple copies of these models. The explosion of objects humbles first the designer, who has to select the correct simulation model or component out of many variants. As a result, he spends a long time on searching and if he cannot find the correct model, he recreates model variants or connects the wrong ones. Especially with the increasing complexity of the models and number of interfaces, this compatibility problem intensifies and creates not only costs but also counteracts the aim to increase reliability and safety.

Generative programming can solve this problem by aggregating variants of the same simulation model or component in one parameterizable component. The user specifies the individual realization, which is then generated to his demands. This solution is comparable to a monolithic, parameterizable component, but these components contain many unused, specialized code, which causes a problem for resource limited real-time hardware used in hardware-in-the-loop test. Generative programming instead leaves only the relevant code fragments and results in better performing models therefore.

However, the best concept to reduce the number of models in a library is hard to establish, if the benefit applies only to new models and the number of existing models remains the same. Hence, it is very important to migrate existing models to the generative programming approach, to cut the number of variants.

This paper addresses these three problems. First, the diversity of simulation models and components is managed by a generative programming approach. Second, the compatibility problem is addressed with an extended component model in Section 4 that leads to the extended library concept in Section 5. Third, the automatic migration of existing models to the library is explained in Section 6. The following Sections 2 and Section 3 introduce the related work and the example domain of this approach.

2. RELATED WORK

2.1 In-the-loop Testing

In-the-loop test methods can be used in embedded software development and production to ensure reliability and safety. These methods connect the *system under test* (SUT) to a simulation model of the environment (see Figure 1). For instance, if the brake system controller of a car have to be tested, the environment will be the brake system itself, whose simulation consists of models of hydraulic elements like pumps and tubes.

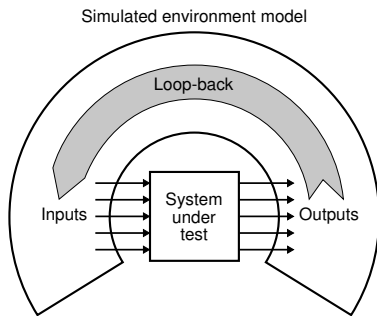


Figure 1: In-the-loop test method

The structure of in-the-loop methods is basically always equal. The simulated environment passes input signals to the SUT, which processes the input and creates output signals. The simulation uses these output signals to calculate its new state and new input signals, which close the loop. As long as both the environment simulation and the SUT are deterministic, the test scenarios are reproducible. This makes it possible to specify test conditions and retest the SUT during all development stages in a test-driven development. Depending on the development stage of the SUT the in-the-loop methods are classified in:

- *Model-in-the-loop (MIL)*
In this level the tested component is specified as a model. This executable model can be used to check if the specification complies with the requirements.
- *Software-in-the-loop (SIL)*
In the second level the tested component is available as code, e. g. in C language. This code can be, for instance, generated automatically from the model spec-

ified before. White box methods can be additionally used to test the code.

- *Processor-in-the-loop (PIL)*
In the third optional level the compiled code is tested on a test board that contains the same processor used later in the target device. This level can be used to make black box tests.
- *Hardware-in-the-loop (HIL)*
In the last level the compiled code is uploaded to the target device under test. The environmental simulation, used in this level, has to meet real time requirements to allow conclusions about the reliability of the controller software. These methods can also be used in production to assure quality.

The transition between these development stages can be simplified by model-driven architecture methods (MDA, see [12]). Using these methods, the platform independent model of the MIL test can be transformed to the code tested by SIL. This code can then be cross-compiled for the target device and tested by PIL and HIL test methods.

One example for a simulation tool that supports MDA is MATLAB/Simulink [11] that allows to transform to code for embedded systems with Real-Time Workshop Embedded Coder [10] or TargetLink [7]. MATLAB/Simulink uses a function block oriented graphical meta language like many standard simulation tools as illustrated in Figures 10 and 11. The simulation is designed abstractly by connecting function blocks to a data flow graph. These signal oriented models are then compiled by the software to an efficient simulation.

In-the-loop methods are often used in the automotive domain. In [13] for example, HIL is used to test controllers in their reaction to injected faults and [9] gives an example of testing an antilock brake system with HIL.

2.2 Generative Programming

Generative programming (GP) is a software engineering paradigm to create software products from a software family automatically by a generator. The generation process is intended to create highly customized and optimized products based on elementary components according to a specification. These products can be software components or whole applications. Further information about GP can be found in [3].

GP is often used to create code in high level languages, like C# or Java [6, 8], or to generate graphical user interfaces (GUI), which represent a higher level compared to simple code generation [15]. Among other application areas, like business process software, GP is used for embedded systems, too. For instance, Czarnecki et al. [2] describe the experiences with GP in embedded domains, like automotive, space and aerospace. They generate code directly for electronic control units or other embedded systems.

Weiland and Richter [18] use software product lines to configure Simulink models. But they do not generate Simulink models directly. Instead they create MATLAB scripts, which *patch* a given reference Simulink model. That means that only a few components are removed or added. Their goal was not to create optimal models, i. e. models without unused parts, or to solve the library management, but to simplify the configuration process.

In previous work [14], Ryssel et al. presented an approach, where function block based models are generated directly by

generative programming. The paper additionally introduced an active library concept [4] that is extended by interoperability definitions. This approach is continued in this paper.

3. EXAMPLE DOMAIN

For a better understanding this section introduces a brake system model from our industrial partner working in the automotive domain. They use HIL tests for quality improvement and validation of the functions of electronic control units.

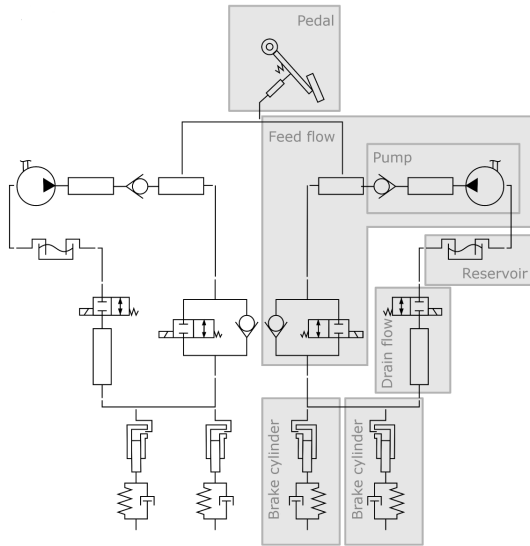


Figure 2: Schema of a brake system

Figure 2 shows the hydraulic schema of the modeled example brake system. The model varies in the input signal simulating the *brake pedal* behavior, in the number of *brake circuits* (one or two) and brake cylinders (four, six or eight), and in the distribution of the cylinders to the circuits.

MATLAB/Simulink is chosen as an example modeling language for this approach due to its function block based graphical language that is common in the automotive domain to model the environment for HIL tests.

4. COMPONENT MODEL AND EXTENDED PORTS

Before any models can be generated with Generative Programming, the components have to be described formally. Our approach uses a function block based component model. According to IEC 61499 and IEC 61804, a function block is an encapsulated algorithm with input and output data points complemented by parameters, which can alter both the function and the structure [5]. This corresponds to the component model shown in Figure 3. In this model the less abstract term *port* is used instead of *data point* for the interfaces. Input and output ports have predefined data types like *integer*, *float* or any structured type based on numeric types. As restriction for compatibility only ports, which have the same data type, can be connected.

The blocks used in MATLAB/Simulink can be represented by such a component model. However, Simulink blocks have

only syntactical data types, which make it difficult to automatically evaluate the compatibility between ports. Especially if components are connected that were created by different design groups, often compatibility problems will occur. For example, if one group designs the model for the brake system and another group for the tire, it will happen that the brake system model outputs the *braking force*, while the tire model needs a negative *acceleration* as input. Simulink permits to connect these models without noticing the incompatibility, because both use the *double* type to code the values.

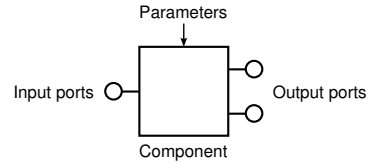


Figure 3: Structure of a basic component

Hence, it is necessary to evaluate the compatibility of models. But this requires semantical information that is usually only available informally in the function block documentation. To achieve both compatibility checks and the generation of components, they are decomposed in *subparts* and *extended ports*. A *subpart* is an atomic implementation of a simple functionality like signal sampling, a moving-average filter or a PID controller algorithm. The *extended port* is an interface concept to encapsulate single input and output ports with advanced signal processing functionality like alarm-thresholds or signal limiters. This decomposition of components is introduced in [14].

In the example mentioned above, the brake system decomposes to the subparts *pedal*, *feed flow*, *drain flow*, *reservoir* and the *brake cylinders*. The braking force output is defined as extended port, while the tire model has an acceleration input as extended port. To assure a compatible connection two participating extended ports have to be predefined as compatible, the so-called *complementary port pair*. In this case the tire model and the brake system model are incompatible (see Figure 4). First, the tire model has to be completed to a wheel model with a braking force input, to get a compatible connection (Figure 5).

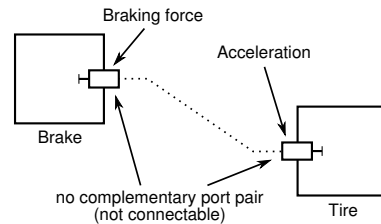


Figure 4: Example of a non-complementary port pair

The concept of complementary port pairs is not limited to the same physical value type, like force or pressure. The semantical information can further include ranges of values and extended functionality like alarms, too. As an example a pair can describe the transmission of a braking force signal between 30 and 3000 N, sampled with 100 Hz and creating an alarm above 2800 N. If these extended ports are reused

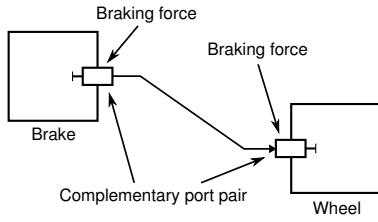


Figure 5: Example of a complementary port pair

in other components, the compatibility definition covers also these implemented and even new developed components.

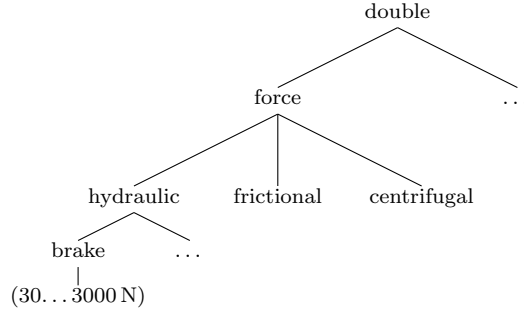


Figure 6: Hierarchical order of port types

Because complementary port pairs can be detailed and extended in their definition, they can be hierarchically ordered to control their high number. Figure 6 shows a part of the hierarchy used in the example domain. The root is the numerical data type (*double*), which branches to physical value types and then to semantical types. Each level adds semantical information. The deepest leaf of this example means: a *hydraulic braking force* between 30 and 3000 N coded as *double* value.

5. EXTENDED ACTIVE LIBRARY

5.1 Active Elements

As mentioned in the introduction, a simple component library cannot solve the software reuse problems. The *active library* of generative programming is suitable for software reuse, but lacks the compatibility evaluation. Therefore the active library is extended by complementary port pairs to an *extended active library*. This library contains all elements described in the component model: The subparts, the extended ports and the structure of components and subsystems. These elements are specified in an implementation component configuration language (ICCL, see [3]), which is in our case expressed in XML [19].

Because components and subsystems are composed of subparts and other components, both structural elements are abstracted to so-called *active elements* in the library. These active elements define the construction plans for the items and can contain other active elements and subparts. Hence, they represent the hierarchy of the domain: Active elements are *systems* at the highest level and detail to *subsystems* and *components* on the basic level. Also *extended ports* and *complementary port pairs* are treated as special active elements. An example will illustrate the different types:

Figure 7 shows the active element **Brakesystem** representing a *system*. This is a simplified version of the brake system model introduced in Section 3. An active element of the type *system* is defined as a closed group of other active elements, which are connected to implement certain functions. If the group offers an external interface, it will be called *subsystem*, like the active element **Distribution**. The content of this element is visible in this view and it contains other active elements and two subparts (the sum blocks). The other active elements **Pedal**, **Feed flow** and **Drain Flow** are *components*. They are the basic active elements and contain no other active elements, except for the special *extended ports*. Extended ports can contain subparts to implement certain interface functionality, like signal limitation, and are therefore with active elements. Extended ports are often described as a complementary port pair, which is a pair of by definition compatible ports. The example uses two complementary port pair types: **p** for a hydraulic pressure and **Q** for a flow. The types are indicated on the connection.

The second brake circuit in the example is optional. Each break circuit consist of one *Feed flow* and *Drain flow* component and the shared *Distribution* subsystem, which contains the brake cylinders (the four active elements inside the subsystem). These variants are expressed in ICCL as introduced in the next section.

5.2 The ICCL

Our ICCL uses configuration parameters to differentiate the variants of a system. There are two types of configuration parameters: *Structure parameters* change the structure of the active element, which can affect the inner composition and interfaces. By contrast, *behavior parameters* change only the function, like filter parameters or other constants. Including these parameters the ICCL has to contain further information to express active elements:

- *Parameters*
Structure and behavior parameters for the configuration of the active elements.
- *Interfaces*
The input and output ports of the active element.
- *Parts*
A list of the contained subparts and active elements with their parameter settings.
- *Connections*
The connections between the parts.

These information are independent of the target system, the active elements are created for. Hence, with a new generator and some additional information, like library paths of the subparts, the described active elements can be instantiated in every function block based modeling language.

The instantiation of interfaces, parts and connections can be varied by the structure parameters. Listing 1 shows the sample code of the ICCL definition for the active element **Brakesystem**. In the *structparameterlist* section the structure parameter **c** is declared, which defines the number of brake circuits (one or two). This structure parameter changes the appearance of the system as defined in the specific elements. The **interface** section is empty because a system has no interface.

The subparts and active elements contained in the specified element are defined in the **parts** section. The above

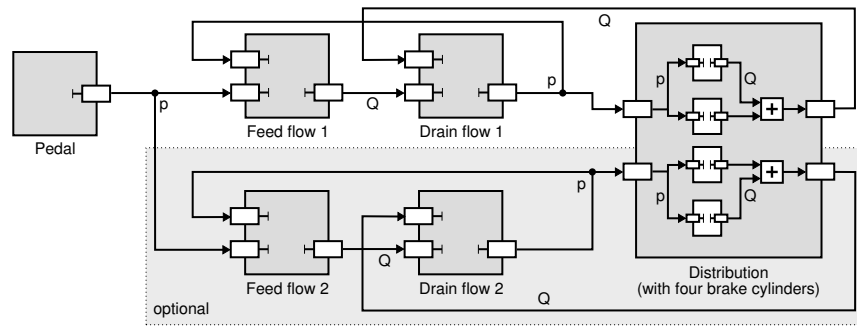


Figure 7: The active element Brakesystem

```

<defineactiveelement type="System" name="Brakesystem">
  <structparameterlist>
    <parameter name="c" default="2" min="1" max="2"/>
  </structparameterlist>

  <interface/>

  <parts>
    <activeelement type="Pedal" name="Pedal"/>
    <activeelement type="FeedFlow" name="FeedFlow1"/>
    <activeelement type="FeedFlow" name="FeedFlow2"
      cond="c==2"/>
    ...
    <activeelement type="Distribution" name="Distribution">
      <setparameter name="bc" value="c"/>
    </activeelement>
  </parts>

  <connections>
    <connect out="Pedal.Out" in="FeedFlow1.In2"/>
    <connect out="Pedal.Out" in="FeedFlow2.In2"
      cond="c==2"/>
    ...
  </connections>

  <dsl>
    <edit description="Number of brake circuits"
      parameter="c" default="2"/>
  </dsl>
</defineactiveelement>

```

Listing 1: Example of an active element specification

declared parameter `c` conditions the active element `FeedFlow2` and the not enlisted `DrainFlow2`. These elements are instantiated only if the condition is fulfilled. This is the case if the system is configured for two brake circuits ($c = 2$). The active element `Distribution` will be created in any case, but it has to be configured to consider the number of brake circuits. Hence, its parameter `bc` is set to `c`, which means they have always the same value. In this way, subordinated parts are automatically configured by the system.

The `connections` section contains all connections between the defined parts. In the example there are two connections between the `Pedal` element and the two `FeedFlow` elements, whereas the second connection is conditioned.

The last section contains information for the dynamic creation of the DSL GUI, which is presented in the next section. The formal description of the active elements and the used subparts form the *extended active library*.

5.3 The DSL

A *domain-specific language* (DSL) is a programming language to describe a specific task in a certain domain [17].

They are used in generative programming to specify the requirements of the system to generate [3]. The language can be a script language, a graphical modeling language or simply a GUI.

Our approach uses a GUI as DSL for an easy access by the user. Like in previous work [14] the GUI is dynamically created from additional information in the ICCL. Therefore, the active element definition can contain a `dsl` section that specifies the GUI elements to set the parameters. The example in Listing 1 would create for instance a text edit field with the description *Number of brake circuits* and a start value of two.

The dynamic creation of the DSL GUI minimizes the effort to create the DSL. It requires neither a separated complex modeling nor a text-based specification language and therefore with no expert knowledge of the system engineer. However, this approach is only possible as long as the generation targets a single, specified model language, which is Simulink in this case.

There is no explicit dependency management across the active elements, but the hierarchical structure of the active elements defines implicit dependencies. As the system engineer later will work with the created models, i. e. recreate parts of it and adapt it manually to his needs, an explicit dependency management would reach its limit.

Figure 8 shows a screenshot of the DSL GUI during configuration of a more complex brake system model than the model in Figure 7.

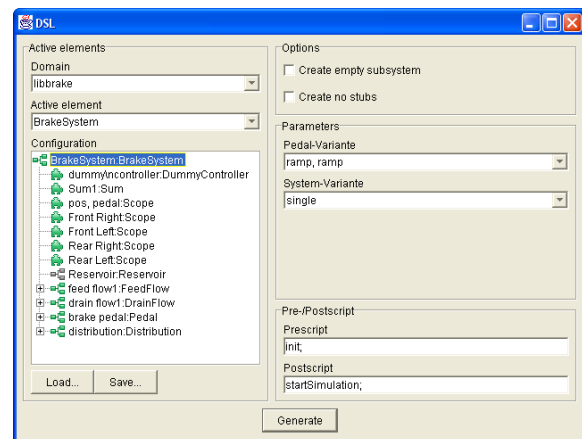


Figure 8: The DSL GUI

5.4 The Model Generator

The generator instantiates the configured active element by assembling the contained active elements and subparts according to the construction plan specified in ICCL (Section 5.1). The contained active elements are created recursively, i.e. the generator recalls itself to generate the contained elements. These recalls are done until the lowest level is reached, where only subparts have to be copied from a subpart library. After all contained active elements were created, the generator connects them. The result of the generation process is an executable simulation model, based on the requirements specified in the DSL GUI.

The generator and the DSL GUI are implemented in the MATLAB script language. This language provides special functions to create models in Simulink and to integrate Java classes that implement the GUI.

6. COMPONENT MIGRATION

6.1 The Necessity of Migration

The introduction of software product lines is mostly an organizational process. There are many case studies that deal with the process of creating a new software product line and migrating existing knowledge. An example can be found in [16], where a more hardware-centric product line is migrated to a new software product line.

In the previous sections the application of generative programming for HIL models were explained. This approach can be applied only if the components are newly developed. But in real world many components already exist and it is not reasonable to discard them for a new engineering paradigm. The solution is to integrate existing components and their variants. Doing this manually is very time-consuming and compensates the time saving benefits of generative programming.

To accelerate the migration process, existing software components need to be automatically integrated. For example Yoshimura et al. [20] merge existing code of embedded systems automatically to get reusable components. Their merging process works on code level and is therefore not applicable to simulation models. But their basic approach, to merge existing models to new components, is transferable.

Figure 9 illustrates the inclusion of the automatic migration in the whole design process. Existing models are taken from the old component library (a simple set of component and model blocks) which are analyzed and migrated by a model analyzer. The migration is a transformation of mod-

els to active elements and can therefore also be used to simplify the design of new components. The domain engineer can adhere to his common software tool like Simulink to design the model and then he can convert this model to an active element without the need to describe it manually.

The created active elements are added to the active library and then can be used by the generator. Because the conversion between models and active elements is possible in both directions with the model analyzer and the generator, the active library can be actually hidden from the system and application engineers.

6.2 The Model Analyzer

To automate the integration of existing components the *model analyzer* has to accomplish two consecutive duties:

The first step is to *identify potential component variants*. That means that all existing models have to be searched and compared to find similar subsystems, which are candidates for active elements.

And in the second step the found component variants have to be *described automatically in the ICCL*. So the commonalities and differences have to be identified and combined to one active element.

6.3 Identification of Potential Component Variants

The identification of potential component variants is a complex task, because existing models have to be scanned for structural similarities. The simplest approach is to compare two components and count the number of equal parts. However, each part gets its functional meaning by the context of the connected parts and it would be more promising to look for equal component pairs and larger groups. But also this approach is limited to find only syntactic similarity. Many complex functions can be designed in different ways and still have the same behavior. A simple example is a series of additions, which are commutative and exchangeable in sequence. These sequences are semantically identical but syntactically different. To find such semantical similarities more semantical information is necessary like the commutativity rule. However, due to these problems all scan algorithms can only identify candidates for an active element and need to be reviewed by the domain engineer.

In the current approach the identification is still done manually by the domain engineer. He selects a set of existing components and passes it to the conversion algorithm. An automated identification is in research.

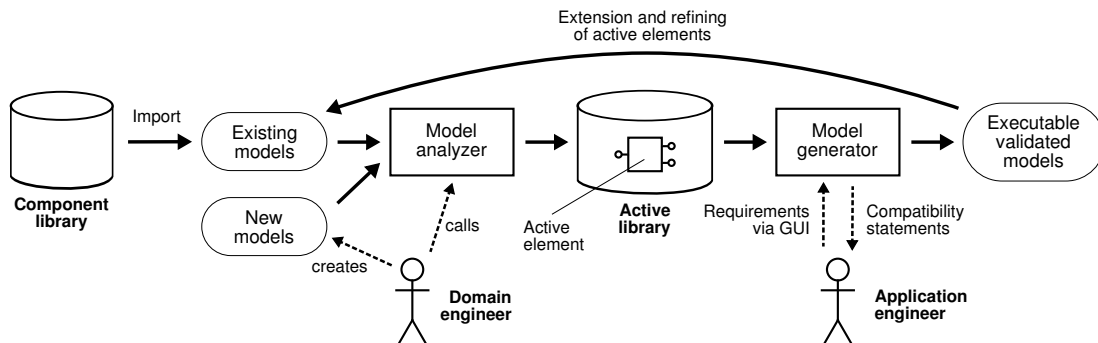


Figure 9: Managing of models using model analyzer and generator

6.4 Converting the Variants to ICCL

An ICCL description of an active element contains all information needed to generate their variants. Section 5.1 has explained that the generated variants depend on conditions containing structure parameters. The converter has to gather all possible parts and connections, create the structure parameters and formulate the appropriate conditions.

In the simplest case only one structure parameter exists that also enumerates the variants. But in many cases the component has more than one point of variation and structure parameter. Figures 10 and 11 introduce the variation points of a tube model, a part of the example domain of this paper. Equal in all variants is the block **V1** that represents the volume of the tube. The input of this block is a volume flow **QA**, which is summed of up to four input flows (**QA1**, ..., **QA4**). This is the first variation point as shown in Figure 10. The second variation point is an optional pressure input port **p1** that is added to the result as done in Figure 11. This can be for example a back pressure. The varying subparts of the example are the sum blocks in front of and behind **V1** that are omitted in the simplest case.

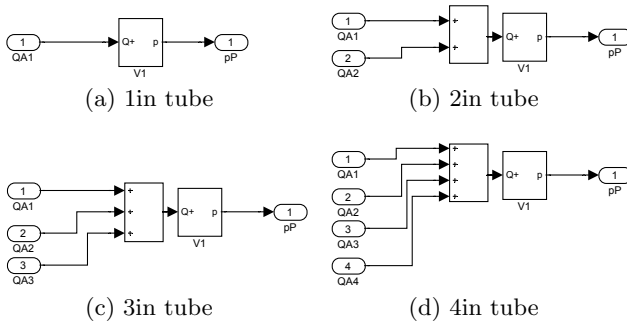


Figure 10: Variants of the variation point ‘Number of flow inputs’

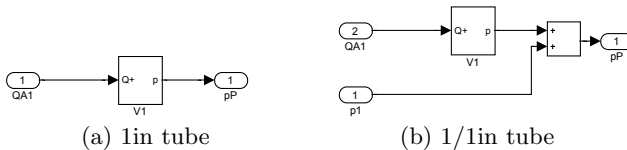


Figure 11: Variants of the variation point ‘Existence of additional pressure input’

The conversion is done in two steps:

- In the first step active elements are created for each variation point. In the example that means that two active elements are created: one that varies the number of flow inputs by a parameter **nQ** and one that varies the additional pressure input by a parameter **p**.
- In the second step, these two active elements are merged to one active element, which contains two structural parameters that can be set independently.

6.5 Creating a Single Active Element

To convert a variation point to an active element, the single components are converted to lists of interfaces, parts and

connections as they are used in the active element description (see Section 5.1). The parts correspond to Simulink blocks, which are treated as **subparts**. Then these lists are compared pairwise to the lists of the other variants. In the part list, entries are equal when they have the same Simulink block type and name. In the example the block **V1** of the type **Volume-Block** appears in all variants and is equal according to the comparison rule. Interface entries are equal, if the ports have identical names, like **QA1**. Connection list entries are equal, if the same parts are connected.

The usage of the name as comparison criterion is not optimal. Two blocks with slightly different names like **V1** and **V_1** would not be set equal. Reducing the identification to the block type is more complex, because the connections have to be involved in the comparison of models. This addresses the semantical problem discussed in Section 6.3 and will be probably extended with the solutions found there. Until then, this simple, syntactical solution works for all examples, but potentially creates more entries than needed.

The resulting list consists of the merged entries of the source lists extended by automatically created condition expressions. For example **QA1** is common to all variants and is unconditioned. Instead, **QA4** exists only in the fourth variant and therefore the condition **nQ==4** is formulated.

Beside the blocks itself, their configurations are compared, too. The left sum block exist in the variants 2 to 4 and the number of input ports needs to be adjusted with **nQ**. This results in the following listing:

```
<subpart type="Sum" name="Sum1" cond="nQ>=2">
  <setparameter name="j" value="++" cond="nQ==2"/>
  <setparameter name="j" value="+++" cond="nQ==3"/>
  <setparameter name="i" value="++++" cond="nQ==4"/>
</subpart>
```

Listing 2: Example of an automatically created subpart configuration

6.6 Merging Active Elements

In the second step, the set of active elements is merged to one active element with a set of independent variation points. The algorithm compares the lists of elements in the same way as during the creation of the single-variant active elements. Only the recombination rules of the conditions are different.

However, in some cases the merging is not possible: If the volume block **V1** of the example depends on both **nQ** and **p**, the merging would fail due to a decision problem occurring later in the generator. If the condition of the first active element with **nQ** evaluates to true, but the condition of the second active element with **p** decides contrary, the generator cannot decide, whether the component should be created or not. It lacks semantical information to combine both conditions (by AND, OR, etc.). This information has to be included in the condition by the model analyzer, but it lacks the same knowledge. However, this case cannot occur as long as the variation points are independent. That is why, the current implementation requires independent variation points or manual merging by the domain engineer.

6.7 Revising of the Active Elements

As a last step the domain engineer is instructed to review the created active element and add additional information

not contained in the existing models, such as extended port types or DSL descriptions. In the current implementation a set of MATLAB functions is developed to adapt the created active elements accordingly.

7. CONCLUSION

The paper has introduced a generative programming approach to manage the diversity of simulation models based on function block like components. Therefore, it reduces the number of models by combining variants in a generative model, called active element. Further, it solves the compatibility problem by defining complementary port pairs. The model is stored in an extended active library, which structure was introduced.

In our implementation, the user can access the library and specify the DSL in his used way via the Matlab/Simulink interface. Furthermore, the newly introduced migration approach permits the user to import existing models and to create new generative models in the common way with Simulink. In this sector, we are still researching the automatic identification of model variants for migration in an existing library. The common graphical interface and the migration approach simplify the paradigm change to a generative programming approach and make its usage convenient.

The selected example domain of simulation models for in-the-loop tests of automotive embedded systems is particularly suitable for generative programming as it is component oriented, has a high model diversity, and requires optimized model realizations. Therefore, the developed solution is already in use by the industry. However, the introduced concept is applicable to any domain with comparable attributes, e. g. the device and system design in building automation.

8. REFERENCES

- [1] Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the Third International Conference of Software Reuse*, pages 102–109. IEEE, 1994.
- [2] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 156–172. Springer-Verlag, 2002.
- [3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
- [4] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag.
- [5] Christian Diedrich, Terry Blevins, Ludwig Winkel, and Francesco Russo. Function block applications in control systems based on IEC 61804. *Annual Conference and Exhibition, ISA's Houston (TX)*, 2001.
- [6] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Generative programming for C#. *SIGPLAN Not.*, 40(8):29–33, 2005.
- [7] dSPACE GmbH. *TargetLink*, visited 2007.
- [8] Manuel Fährdrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, New York, NY, USA, 2006. ACM Press.
- [9] Ki-Chang Lee, Jeong-Woo Jeon, Don-Ha Hwang, Se-Han Lee, and Yong-Joo Kim. Development of antilock braking controller using hardware in-the-loop simulation and field test. In *Proceedings of the 30th Annual Conference of IEEE (IECON)*, volume 3, pages 2137–2141, 2004.
- [10] The MathWorks, Inc. *Real-Time Workshop Embedded Coder*, visited 2007.
<http://www.mathworks.com/products/rtwembedded/>.
- [11] The MathWorks, Inc. *Simulink — Simulation and Model-Based Design*, visited 2007.
<http://www.mathworks.com/products/simulink/>.
- [12] Object Management Group (OMG). *Model Driven Architecture*, visited 2007.
<http://www.omg.org/mda/>.
- [13] M. Sonza Reorda and M. Violante. Hardware-in-the-loop-based dependability analysis of automotive systems. In *IOLTS '06: Proceedings of the 12th IEEE International Symposium on On-Line Testing*, pages 229–234, Washington, DC, USA, 2006. IEEE Computer Society.
- [14] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Generative function block design and composition. In *Proceedings of the 6th IEEE Workshop on Factory Communication Systems (WFCS), Torino, Italy*, pages 253–262, 2006.
- [15] Max Schlee and Jean Vanderdonckt. Generative programming of graphical user interfaces. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 403–406, New York, NY, USA, 2004. ACM Press.
- [16] Mikael Svahnberg and Michael Mattsson. Conditions and restrictions for product line generation migration. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 143–154, London, UK, 2002. Springer-Verlag.
- [17] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.
- [18] Jens Weiland and Ernst Richter. Konfigurationsmanagement variantenreicher Simulink-Modelle. In *Informatik 2005 — Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e. V., Band 2*, pages 176–180, 2005.
- [19] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*, visited 2007.
<http://www.w3.org/XML/>.
- [20] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Defining a strategy to introduce a software product line using existing embedded systems. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 63–72, New York, NY, USA, 2006. ACM Press.