Early Fault Detection Using Design Models for Collision Prevention in Medical Equipment*

Arjan J. Mooij[1], Jozef Hooman[1,2], and Rob Albers[3]

[1] Embedded Systems Innovation (ESI) by TNO, P.O. Box 513, 5600 MB Eindhoven,
The Netherlands
{arjan.mooij,jozef.hooman}@tno.nl
[2] Computing Science Department, Radboud University Nijmegen, The Netherlands
[3] Philips Healthcare, Best, The Netherlands
r.albers@philips.com

Abstract. In the medical domain there is a tension between the requested speed of innovation and the time needed to deliver a certifiable system. To ensure the required safety, usually a long test and integration phase is needed. To shorten this phase and to avoid late bug fixing, the aim is to detect faults (if any) much earlier in the development process. This can be achieved by combining a number of model-based techniques such as (1) architecture validation by simulating executable models, (2) development of a Domain-Specific Language (DSL) to combine precision with higher levels of abstraction, and (3) transformations from DSLs to analysis models for performance evaluation and formal verification. We illustrate such techniques using an industrial study project on a new architecture for movement control including collision prevention.

1 Introduction

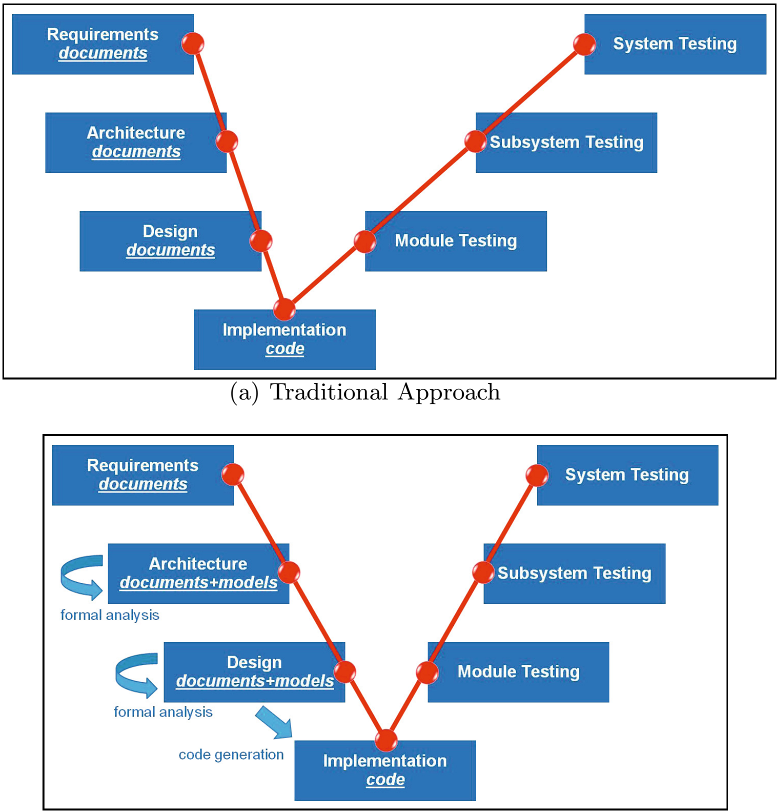
In the medical domain there is a tension between the requested speed of innovation and the time needed to deliver a certifiable system. To ensure the required safety, usually a long test and integration phase is needed. The problem is that often faults are found in this late phase of the development process.

In industrial development processes, the first formal artefacts are usually at the level of implementation code (or close to it), whereas the architecture and design phases are based on informal documents; see Fig. 1(a). Faults are often detected during or after developing such formal artefacts. However, the detected faults do not only include implementation faults, but also faults that were introduced in the requirements, architecture and design phases [19]. For example, during test and integration it may turn out that the requirements are incomplete, or that a design cannot fulfil the requirements. Such faults are often costly to repair in such a late phase.

The aim of our work is to detect such faults (if any) much earlier in the development process, thus reducing late bug fixing in the test and integration phase,

* This research was supported by the Dutch national program COMMIT and carried out as part of the Allegio project.

J. Gibbons and W. MacCaull (Eds.): FHIES 2013, LNCS 8315, pp. 170–187, 2014.



(a) Traditional Approach

(b) Model-based Approach

Fig. 1. Development Processes and Phases

and hence increasing the rate of innovation. To this end, we combine a number of model-based techniques, which introduce formality in earlier development phases; see Fig. 1(b). We emphasize high-level design models and their potential for early analysis.

We have applied these techniques while participating in an industrial study project at Philips Healthcare, in the context of interventional X-ray systems; see Fig. 2. Such systems are used for minimally-invasive surgery, where X-ray images guide the surgeon during an operation. These systems consist of one or two so-called C-arms, each carrying an X-ray generator and a detector. During the treatment, the C-arms, the detectors, and the patient table may move to obtain optimal projections for the images.

Physical safety of these systems includes avoiding any collisions between these heavy moving objects and the humans, such as patient and medical staff. The study project has focused on redesigning the software components for movement control, which includes functionality for collision detection and prevention.

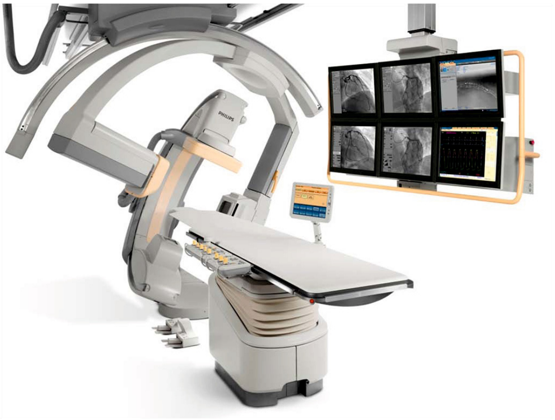


Fig. 2. Interventional X-Ray system

We have first developed a reference architecture for movement control. Architectures are usually described using informal drawings. However, it is difficult to use such drawings to decide whether the architecture will really work in practice. To obtain more confidence in the feasibility of the architecture, we have used high-level formal models, which can be analysed using simulation and domain visualization. By limiting the amount of detail in such models, the required effort remains acceptable. In our experience, such models trigger more detailed discussions about the architectural concepts and their interactions.

Afterwards we have designed the component that is responsible for collision prevention. The main challenges were to identify the main reasoning concepts, and to represent the rules for collision prevention in a concise, precise and readable way, such that they can be inspected and analysed easily. To stay close to the requirements formulation, we have developed a Domain-Specific Language (DSL) that is targeted at the type of rules we want to express. We have generated code from these DSL models, and evaluated it on the physical hardware. In addition we have analysed these models by defining transformations to and from several analysis tools. As the collision prevention component is a safety-critical real-time component, we have evaluated the required execution time and we have verified some formal correctness properties.

Overview In Sect. 2 we address the description and analysis of the proposed reference architecture. In Sect. 3 we focus on the design and analysis of the collision prevention component. In Sect. 4 we discuss related work. Finally, in Sect. 5 we draw some conclusions and sketch further work.

2 Reference Architecture for Movement Control

In this section we focus on the architecture phase from Fig. 1(b), where the aim was to develop a reference architecture for movement control. An architecture provides a high-level view of a specific system, whereas a reference architecture

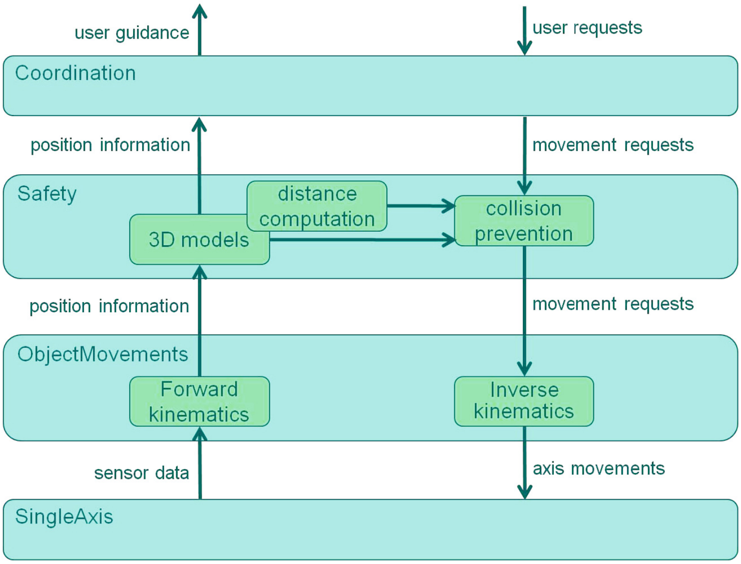


Fig. 3. Reference Architecture for Movement Control

provides a high-level view of a family of systems (or a product line). Architectures are developed in an early development phase, where the goal is to decompose the functionality in layers and components, and to identify the interfaces.

Traditionally (reference) architectures are described using various informal drawings that are easy to make and modify. However, once the developers start to reach an agreement on such informal drawings, it is still very difficult to decide whether the architecture will really work in practice. Often the only way to decide this is to start implementing it.

To gain more confidence in the feasibility of the architecture, we investigate the use of formal modelling and analysis. In this development phase, where there are many uncertainties and fast experimentation with variations is needed, it is inappropriate to make a detailed formal model of the complete system. It would be too time consuming, both to design and to modify the model, and it would require too many details. In a later phase, it may be useful to model and analyse some safety-critical components in detail. In this section we discuss the potential for high-level formal modelling and analysis in the architecture phase.

2.1 Reference Architecture for Movement Control

Fig. 3 contains a simplified description of the reference architecture for movement control that was proposed in this study project and that uses a layered approach [8]. It consists of four layers:

- *Coordination*: interaction with the user about movements and procedures;
- *Safety*: restriction of movement requests in order to prevent collisions;

- *ObjectMovements*: translation of complex movements to individual axes;
- *SingleAxis*: interaction with the physical motors and sensors.

There are two main flows of information. At the left side, sensor readings are propagated through the layers and finally displayed to the user as user guidance. At the right side, movement requests from the user are propagated to motors.

The interfaces of the layers are such that the safety layer can be removed; so the safety layer acts as a kind of filter on the movement requests. The safety layer stores the sensor information in 3D geometric models. Such a model contains a representation of the physical objects (patient table, C-arm, detector, etc.) and their 3D position. The safety layer contains, among others, a model for the current position of the physical objects and a look-ahead model for their expected position at a future point in time.

The safety layer is executed in a real-time loop with a certain frequency. In this loop the distances between some of the objects in the various 3D models are computed. Based on these distances, the collision prevention component determines whether a given movement request is safe. If the request might lead to collisions, the speed of the request is reduced or even set to zero. The restrictions on the movements depend on many aspects such as the objects involved, the accelerations and positions of the objects, the reaction time needed to influence movements and the required brake distances. Look-ahead models are needed to take response times and brake distances into account.

2.2 Required Functionality for Movement Control

The architecture phase has also been used to increase the insight in the functionality that needs to be redesigned. In this type of systems there is always a tension between safety and usability. On the one hand no dangerous situation may be introduced, but on the other hand the physician wants to have maximum freedom in controlling the system to obtain optimal projections for the images. Moreover, users expect a very predictable system.

The existing system has evolved over many years, starting as a simple X-ray system for diagnosis, and later on extended with functionality for interventional X-ray, 3D scans, and hybrid operating rooms. In particular the latter leads to new requirements and interfaces for additional external equipment. There have also been several modifications and extensions on request of medical users. To achieve a consistent user experience within the product family, both regular design decisions and late modifications may become requirements that restrict the design of future systems. These trends have introduced a number of incidental complexities and intricate dependencies between pieces of functionality.

To be able to develop a new reference architecture where components have clear responsibilities and clean interfaces, it was essential to challenge the need for complex functionality and to identify the real underlying requirements. By discussions with domain specialists, the requirements have been simplified significantly, eliminating some notorious complexities and disentangling pieces of functionality. We expect that these simplifications will reduce the potential for faults in further development phases.

2.3 High-level, Formal Modelling

Before starting the implementation, we would like to validate the architecture in order to reduce risks. For instance, to gain confidence that several typical scenarios can really be implemented effectively, and to investigate how the architectural choices impact the system as the user would experience it. To keep the workload manageable, it is crucial to focus on critical scenarios that have a high impact on the system. In this application, we have considered some manual joystick movements and a few medical procedures that involve moving the C-arm according to a special trajectory.

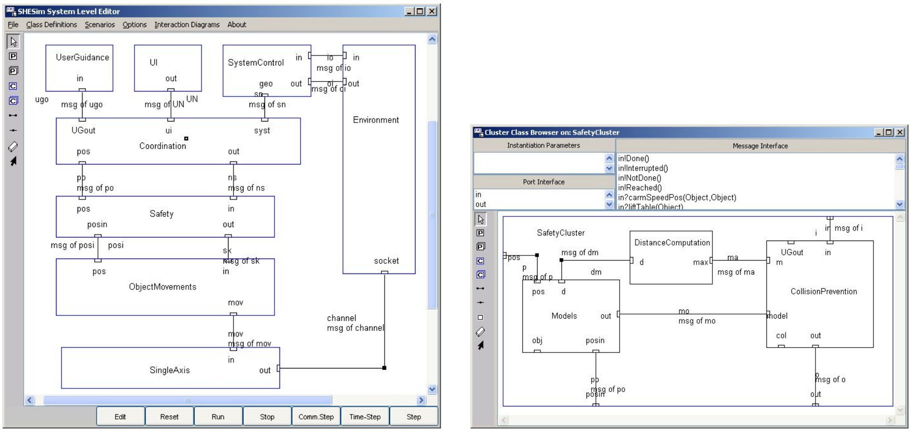
To perform such validation, we have added some formality to the informal drawings such that we obtain executable models. In this development phase we only make high-level models that abstract from many of the details. We have used very simplistic models of the hardware (motors and sensors); detailed models with continuous behaviour and differential equations are far outside our scope. Making such models triggers many questions to the developers about their exact ideas. By clarifying these issues in an early development phase, costly misunderstandings and repairs later on can be avoided.

To model the relevant scenarios, we have used a language called POOSL (Parallel Object-Oriented Specification Language, [31]), which has a formal semantics defined in terms of a timed probabilistic labelled transition systems. POOSL uses two types of building blocks: cluster (with a blue border) and process (with a black border). Clusters can contain again clusters and processes, and thus they can be used to model hierarchical system structures. Processes focus on individual behaviours and are specified using a textual object-oriented process algebra. Each block has an external interface consisting of ports (drawn near its border) that can be used for synchronous one-to-one message communication; that is, a message can be communicated when a pair of a sender and a receiver are both ready for communication.

Fig. 4(a) shows the graphical representation of our POOSL model in the SHESim tool [14]. All the blocks in this diagram are clusters. In particular in the bottom left corner there are four clusters for the architectural layers from Fig. 3. The other blocks are clusters that expose the external interfaces of the architecture to other validation tools as we need in the next section. In turn, Fig. 4(b) shows the graphical representation of one of the clusters. All the blocks in this diagram are processes. Processes themselves are specified textually. Most of our processes follow the structure of a state machine.

Note that in this development phase we do not aim at formal verification. The emphasis is on rapid prototyping of architectural concepts in a powerful modelling language. Formal verification would impose additional restrictions on the models and would require further abstractions from many relevant data aspects, such as 3D position information and movement requests.

It is not easy to show that later implementations of the architecture are a formal refinement of our architectural model. Our high-level models do not cover all scenarios, they typically ignore detailed timing issues, and they can use simplified external interfaces. Hence, at best we can expect this refinement only



(a) Top-Level Specification

(b) Specification of a Cluster

Fig. 4. Executable Model in POOSL

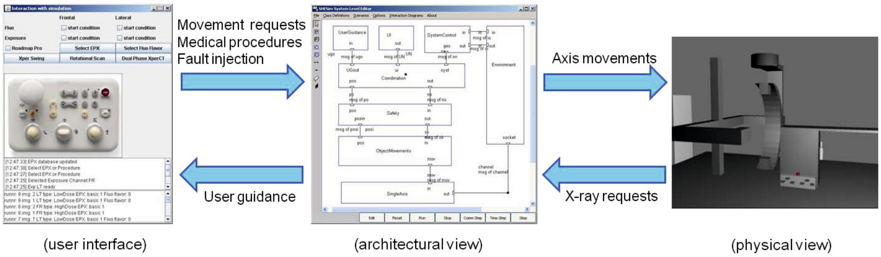
for restricted scenarios, and with additional conversions to make the interfaces homogeneous; see also [19]. In this study project we have not tried to show any formal refinement between different models. Still the analysis of individual design models gives useful results.

2.4 Interactive Simulation and Domain Visualization

To validate the modelled behaviour, we use interactive simulation. In our experience, it is important to relate the architectural model to the user-perceived system behaviour. That is, not only consider internal software aspects, but also include their impact on the full system. This includes user interactions and external system behaviour such as physical movements and X-ray. To achieve this, we combine simulation with domain visualizations [24,19].

Fig. 5 shows the combination of simulation and two types of domain visualization. In the middle there is a simulator for the architectural model, which focuses on the system's functionality. At the left side there is a Java GUI that simulates user interfaces such as joysticks, that displays user guidance messages, and that can inject special simulated events such as faults or collisions. At the right side there is a 3D model of the physical hardware that is modelled using the 3D modelling and game engine Blender [7]. It shows the physical state of the system, and it can be used to trigger X-ray requests via the pedals.

For example, by clicking on the picture of the joysticks, the user can trigger events that start a movement or select a certain medical procedure that involves movements. These events are sent to the simulator of the architecture. The architectural model then determines whether certain user guidance messages have to be generated; in this case an event is send to the Java GUI. The architectural model might also trigger certain movements; in this case events are send to Blender. The communication between these tools uses sockets (see also [24,19]).



[12:47:33] EPX database updated
[12:47:30] Select EPX or Procedure
[12:47:27] Select EPX or Procedure
[12:47:25] Selected Exposure Channel:FR
[12:47:25] Exp LT ready

runnr: 9 img: 2 LT type: LowDose EPX: basic 1 Fluo flavor: 0
runnr: 9 img: 1 LT type: LowDose EPX: basic 1 Fluo flavor: 0
runnr: 8 img: 2 FR type: HighDose EPX: basic 1
runnr: 8 img: 1 FR type: HighDose EPX: basic 1
runnr: 7 img: 7 LT type: LowDose EPX: basic 1 Fluo flavor: 0

Fig. 5. Interactive Simulation and Domain Visualization

The combination of interactive simulation and domain visualization helps to see the impact of design decisions on the system as a whole. Moreover, it turns out to trigger again a lot of discussion, although there was common agreement on the earlier informal drawings. Certain implications, omissions or faults can be seen more quickly using an interactive simulation, in particular in combination with a realistic domain visualization. This applies to the validation of both the architectural concepts and the component interfaces.

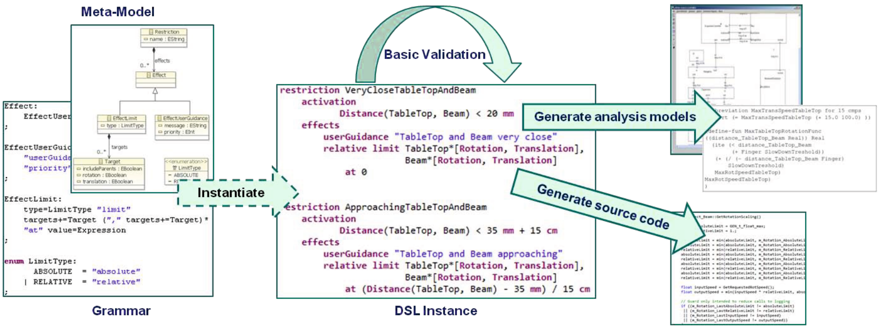
Validation includes checking the completeness of the interfaces, the information that is available in each component, and whether the components can together collect enough information for making the right decisions. This concerns, for instance, sensor data and decisions where to store movement trajectories. By formally modelling different choices, the developers can really experience the consequences of different architectural decisions.

The analysis is particularly interesting when it comes to feature interaction. Because we have a single executable model for multiple scenarios, interferences can be identified early. For instance, in our model we had to make very explicit which manual movements are allowed during certain medical procedures. Moreover, the possibility to inject faults makes it easy to experiment with, for instance, graceful degradation strategies.

For the simulation we have used the SHESim tool [14] that includes an interactive simulator for POOSL. It can show interaction diagrams with the flow of messages between parts of the model. Moreover, during simulation the internal state of the model can be inspected. In our experience these facilities give insight in the model, and also help in debugging the models, in a way that is more convenient than debugging prototype implementations in some traditional programming language.

2.5 Results

The applied modelling and simulation approach focuses on rapid prototyping of the proposed reference architecture. In particular we have connected the reference architecture with the user-perceived system behaviour. This approach has triggered more discussions among the developers during the architecture phase, in which it is relatively easy to experiment with different alternatives. Thus we have gained confidence in the feasibility of the proposed architecture.



```
Effect:
    EffectUser
;

EffectUserGuic
    "userGuide
    "priority"
;

EffectLimit:
    type=LimitType "limit"
    targets+=Target ("," targets+=Target)*
    "at" value=Expression
;

enum LimitType:
     ABSOLUTE  = "absolute"
   | RELATIVE  = "relative"
;
```

```
restriction VeryCloseTableTopAndBeam
    activation
          Distance(TableTop, Beam) < 20 mm
    effects
        userGuidance "TableTop and Beam very close"
        relative limit TableTop*[Rotation, Translation],
                       Beam*[Rotation, Translation]
          at 0

estriction ApproachingTableTopAndBeam
    activation
          Distance(TableTop, Beam) < 35 mm + 15 cm
    effects
        userGuidance "TableTop and Beam approaching"
        relative limit TableTop*[Rotation, Translation],
                       Beam*[Rotation, Translation]
          at (Distance(TableTop, Beam) - 35 mm) / 15 cm
```

```
 breviation MaxTransSpeedTableTop for 15 cmps
 rt (= MaxTransSpeedTableTop (* 15.0 100.0) ))

define-fun MaxTableTopRotationFunc
((distance_TableTop_Beam Real)) Real
  (ite (< distance_TableTop_Beam
        (+ Finger SlowDownTreshold))
   (* (/ (- distance_TableTop_Beam Finger)
       SlowDownTreshold)
    MaxRotSpeedTableTop)
MaxRotSpeedTableTop)
)
```

```
ct_Beam::GetRotationScaling()

 oluteLimit = GEN_t_float_max;
 tiveLimit = 1.;

 eLimit = min(absoluteLimit, m_Rotation_AbsoluteLi
 luteLimit = min(absoluteLimit, m_Rotation_AbsoluteLi
relativeLimit = min(relativeLimit, m_Rotation_RelativeLi
absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLi
relativeLimit = min(relativeLimit, m_Rotation_RelativeLi
absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLi
relativeLimit = min(relativeLimit, m_Rotation_RelativeLi
absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLi
relativeLimit = min(relativeLimit, m_Rotation_RelativeLi

float inputSpeed = GetRequestedRotSpeed();
float outputSpeed = min(inputSpeed * relativeLimit, abso

// Guard only intended to reduce calls to logging
if ((m_Rotation_LastAbsoluteLimit != absoluteLimit)
 || (m_Rotation_LastRelativeLimit != relativeLimit)
 || (m_Rotation_LastInputSpeed != inputSpeed)
 || (m_Rotation_LastOutputSpeed != outputSpeed))
```

Fig. 6. Domain-Specific Language and Transformations

The proposed architecture has finally been implemented as a prototype, which has been evaluated on the physical hardware.

3 Design for Collision Prevention

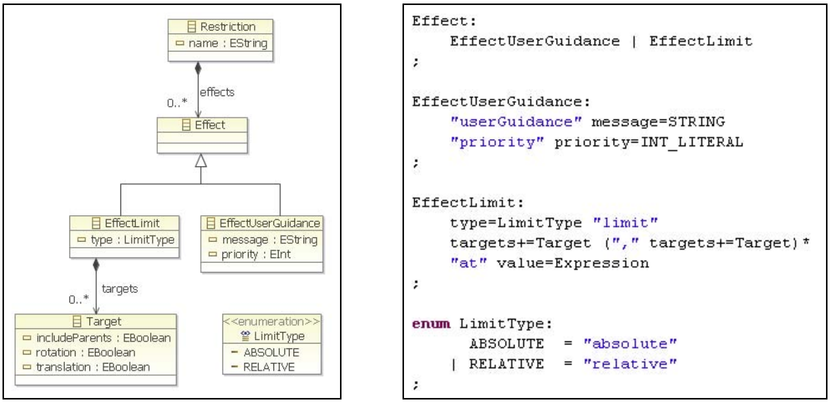
In this section we focus on the design phase from Fig. 1(b), and consider the collision prevention component from Fig. 3. Collision prevention is based on rules that describe in which situations the movement requests have to be restricted. Although there are basic rules about required minimal distances between objects (for example, between C-arm and table), there are many exceptions for special procedures or situations. In addition there are detailed slowdown patterns that describe how to restrict the movement requests in a safe and comfortable way. Moreover, the set of rules varies depending on the product configuration.

An important goal of the redesign is to simplify the collision prevention component, and to support further modifications of the collision prevention rules. To represent the rules in a concise, precise and readable way, we aim to capture the rules at a high level of abstraction, close to the requirements formulation. To this end, we have developed a Domain-Specific Language (DSL).

Fig. 6 summarizes our work in relation to this DSL. At the left side, there is a fragment of the developed meta-model and grammar for this DSL. These are not specific for a single system, but they can be used for a family of similar systems (for example, a product line). In the middle of Fig. 6, there is a fragment of a DSL instance, which is specific for a single system. Based on such a DSL instance we generate source code for an implementation, and we generate analysis models for early validation of the rules.

3.1 Domain-Specific Language for Collision Prevention

The development of a Domain-Specific Language (DSL, [13,33]) includes a meta-model and a grammar; see Fig. 6. The meta-model (abstract syntax) identifies the essential concepts that we need for modelling the collision prevention rules, and the relations between these concepts; see the snapshot in Figure 7(a). The



```
Effect:
    EffectUserGuidance | EffectLimit
;

EffectUserGuidance:
    "userGuidance" message=STRING
    "priority" priority=INT_LITERAL
;

EffectLimit:
    type=LimitType "limit"
    targets+=Target ("," targets+=Target)*
    "at" value=Expression
;

enum LimitType:
      ABSOLUTE  = "absolute"
    | RELATIVE  = "relative"
;
```

(a) Meta-Model (b) Grammar

Fig. 7. DSL Snapshots

grammar (concrete syntax) defines the textual language used for describing instances of the language; see the snapshot in Figure 7(b).

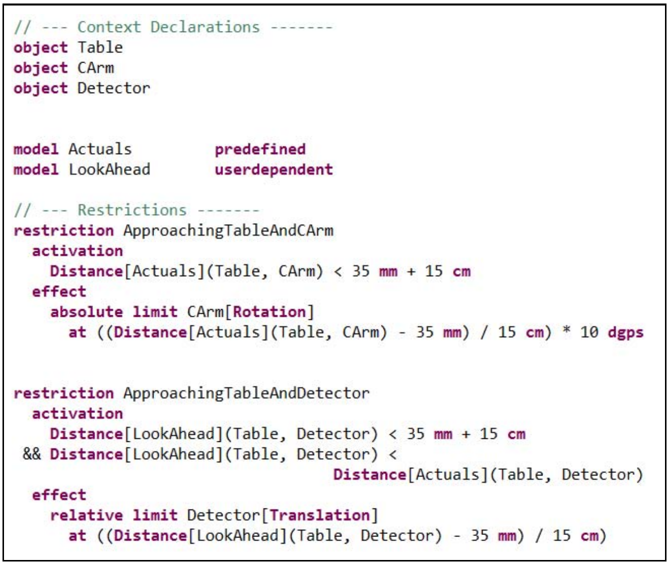
Fig. 8 contains a snapshot of an instance of the language. It starts with a specification of the available 3D geometric models (as mentioned in Section 2.1), including the geometric objects they contain. The collision prevention rules themselves are specified in terms of restrictions. For each restriction there are criteria that specify when the restriction is active. The effects of an active restriction can include user guidance and speed limits on the movements. Moreover, there are several mechanisms to specify how to deal with hysteresis effects of the sensors and processing latencies in the system, but these are not shown in this snapshot.

As mentioned in [25], DSLs trade generality for expressiveness in a limited domain, leading to improved ease of use compared to general-purpose languages. We aim that also non-programmers are able to use our language. As an example, we have used measurement units for numbers instead of data types; confusion about measurement units is a known source of faults.

The development of this DSL has been an iterative process. We have started with a small set of language concepts, which were clearly essential for basic collision prevention rules. Afterwards we have analysed more challenging rules, thus gradually identifying some extra language concepts, which we have added to the DSL. More details of this process are described in [26]; special emphasis is given to gaining industrial confidence for the use of DSLs.

We have defined the DSL using Xtext [2], which is based on the Eclipse Modelling Framework (EMF, [30]). Based on the grammar, an Eclipse-based editor, parser and meta-model are generated automatically. Also convenient starting points are provided for defining validation and code generation.

Clearly the definition of a DSL for modelling collision prevention rules requires some additional effort. To make sure that these efforts pay off, DSLs should mainly be considered when several instances (or modifications over time) are to be expected, which is the case for the collision prevention rules.



```
// --- Context Declarations -------
object Table
object CArm
object Detector


model Actuals          predefined
model LookAhead        userdependent

// --- Restrictions -------
restriction ApproachingTableAndCArm
  activation
    Distance[Actuals](Table, CArm) < 35 mm + 15 cm
  effect
    absolute limit CArm[Rotation]
      at ((Distance[Actuals](Table, CArm) - 35 mm) / 15 cm) * 10 dgps


restriction ApproachingTableAndDetector
  activation
    Distance[LookAhead](Table, Detector) < 35 mm + 15 cm
 && Distance[LookAhead](Table, Detector) <
                                 Distance[Actuals](Table, Detector)
  effect
    relative limit Detector[Translation]
      at ((Distance[LookAhead](Table, Detector) - 35 mm) / 15 cm)
```

Fig. 8. Snapshot of a DSL Instance

Moreover, it is important to have a clear understanding of the semantics of the modelling language. In the context of DSLs, the code generator often defines the semantics. In [21] we give a mathematical semantics of some key concepts in our DSL. Furthermore we have chosen the syntax of the DSL in such a way that most of the language elements are practically self-explanatory; there are only a few features whose semantics requires a more detailed explanation.

3.2 Generation of Source Code

For the generation of source code in Fig. 6, we have developed a custom code generator (using Xtend [1]) that transforms the high-level concepts from our DSL into executable code. By means of some glue code, we have integrated the generated code with existing systems.

The generated code can be used in various ways. First of all, the code generation gives semantics to the concepts in the meta-model and grammar of the DSL. When developing the DSL, we use the generated code to evaluate whether we have correctly captured all essential concepts. Next, we use the code generation for testing whether a specific set of rules has been modelled correctly, by running the generated code on the physical hardware. Finally, the code generation can be used for generating production code and then it adds immediate value to the modelling efforts [16].

3.3 Basic Validation and Generation of Analysis Models

A high-level description in terms of a DSL has a lot of potential for analysis, even before generating any source code for implementation in a general-purpose language. When applied to implementation code, many analysis techniques would require the prior use of abstraction techniques. DSLs facilitate analysis by providing a domain-specific abstraction that naturally fits the problem domain.

Fig. 6 shows two types of analysis. We have started with some basic validation on the DSL instance. This includes, for instance, type checking and checks that certain relations are acyclic. Clearly such checks are limited and there is a need for more analysis before code generation and time-consuming system tests. As the collision prevention component is a safety-critical real-time component, we have considered two types of analysis: performance evaluation of the required execution times and formal verification of some correctness properties.

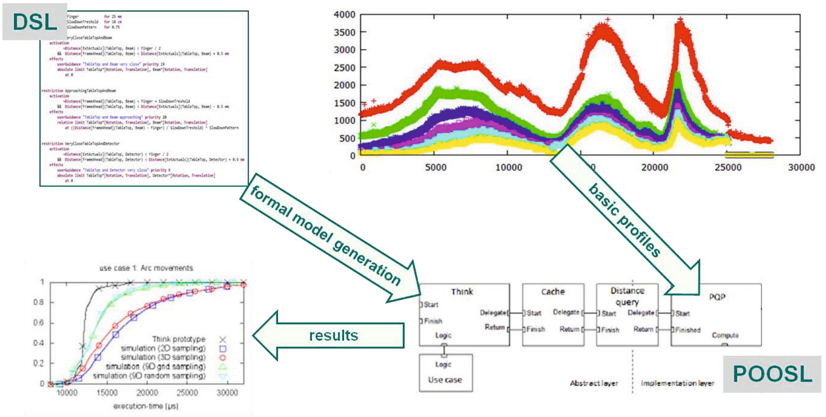
In selecting analysis tools, we have used two criteria [21]. The first one is that no user interaction should be required, as the idea is to hide the analysis tools from the user of the DSL. This means that the analysis models have to be generated from the DSL instance, and the analysis results have to be translated back to the DSL level. Ideally, the results are showed as warnings and errors in the DSL editor.

The second criterion is that analysis results should be available in a short amount of time, such that the analysis can be run after any modifications of the DSL instance, such as changing or adding rules. Basic validation of DSL instances is performed whilst typing. The other types of analysis are more time consuming, and hence we do not intend to perform them whilst typing. We aim for an analysis time of at most a few minutes (say a short coffee break).

Performance Analysis. The collision prevention component is part of a real-time control loop that executes with a certain frequency; see Sect. 2.1. Hence it is important that the collision prevention component can execute within the period of the real-time loop. The performance analysis aims to predict quickly how much execution time is needed based on the description in the DSL instance; any other scheduling issues are not considered.

Fig. 9 gives an overview of the approach as described in [6]. In the top-left corner we start with a DSL instance. To meet our criteria for the analysis tools (fully automated, and time efficient), we create a POOSL model and analyse it using the high-speed simulation tool Rotalumis [14]. The POOSL model is generated using a model transformation (in Xtend [1]) from the DSL instance. In addition, in the top-right corner, we collect performance profiles of basic operations. These are added to the POOSL model. Finally, in the bottom-left corner the statistical results are depicted, consisting of expected execution times and their likelihood.

As the collision prevention acts as a safety layer, one may expect that we are only interested in worst-case scenarios. However, we have performed a statistical analysis instead, as a worst-case analysis would assume the extremely unlikely case that all functionality under-performs at the same time. Focusing on the worst-case only may thus result in serious over-dimensioning of the system, which, in turn, increases its costs. Moreover, the safety layer is not the only means to ensure safety; for example, the lower layers in the architecture from Fig. 3 contain various collision detection mechanisms.



```
    Finger              for 25 mm
    SlowDownTreshold    for 10 cm
    SlowDownPattern     for 0.75

    eryCloseTableTopAndBeam
  activation
      ¬Distance[ExtActuals](TableTop, Beam) < Finger / 2
    && Distance[FrameAhead](TableTop, Beam) < Distance[ExtActuals](TableTop, Beam) + 0.5 mm
  effects
    userGuidance "TableTop and Beam very close" priority 19
    absolute limit TableTop*[Rotation, Translation], Beam*[Rotation, Translation]
      at 0


restriction ApproachingTableTopAndBeam
  activation
      ¬Distance[FrameAhead](TableTop, Beam) < Finger + SlowDownTreshold
    && Distance[FrameAhead](TableTop, Beam) < Distance[ExtActuals](TableTop, Beam) - 0.5 mm
  effects
    userGuidance "TableTop and Beam approaching" priority 20
    relative limit TableTop*[Rotation, Translation], Beam*[Rotation, Translation]
      at ((Distance[FrameAhead](TableTop, Beam) - Finger) / SlowDownTreshold) ^ SlowDownPattern


restriction VeryCloseTableTopAndDetector
  activation
      ¬Distance[ExtActuals](TableTop, Detector) < Finger / 2
    && Distance[FrameAhead](TableTop, Detector) < Distance[ExtActuals](TableTop, Detector) + 0.5 mm
  effects
    userGuidance "TableTop and Detector very close" priority 9
    absolute limit TableTop*[Rotation, Translation], Detector*[Rotation, Translation]
      at 0
```

Fig. 9. Performance Analysis using the DSL

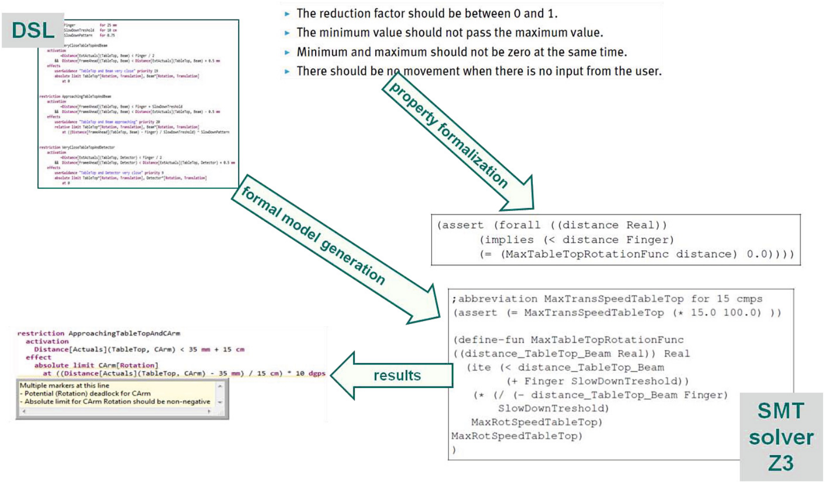
The most time-consuming basic operations are the distance computations between objects in the geometric models. The required *amount* of distance computations is basically determined by the instance of the DSL, which specifies which distances are relevant for the specified set of rules. However, the generated implementation code uses conditional and lazy evaluation. That is, depending on the positions of objects, some distances may not need to be computed.

The required *time* for a single distance computation depends on the used package for distance computations. In [6] we have considered distance computations on the basis of the Proximity Query Package (PQP, [23]), which is used in the context of robotics [9]. The execution times in PQP vary depending on the shapes of the objects and their relative geometric position. For each pair of objects, we have profiled the execution time for distance computations.

In the POOSL performance model we abstract from the geometric positions of the objects, and perform random sampling from the basic performance profiles. In addition we use probabilities to model the conditional and lazy evaluation. By simulating the performance model, we obtain a statistical indication of the expected execution times. In [6] these times are compared with measurements based on the generated implementation code.

Formal Verification. The collision prevention component is part of the safety layer; see Fig. 3. The formal verification aims to check quickly whether the following types of correctness properties hold for the DSL instance:

- well-defined expressions: checks whether there cannot be any "division by zero" or "exponentiation resulting in a complex number".
- ranges: checks whether the specified speed limits come from a proper range of values; for example, whether they are non-negative in all situations.
- safety: checks various properties such as that if two objects are close to each other and still approaching, then their speeds are restricted.
- deadlock: checks whether there is no position of the objects such that no further movements are possible.



```
Finger             for 25 mm
SlowDownTreshold   for 10 cm
SlowDownPattern    for 0.75

eryCloseTableTopAndBeam
  activation
      ¬Distance[ExtActuals](TableTop, Beam) < Finger / 2
    && Distance[FrameAhead](TableTop, Beam) < Distance[ExtActuals](TableTop, Beam) + 0.5 mm
  effects
    userGuidance "TableTop and Beam very close" priority 19
    absolute limit TableTop*[Rotation, Translation], Beam*[Rotation, Translation]
      at 0

restriction ApproachingTableTopAndBeam
  activation
      ¬Distance[FrameAhead](TableTop, Beam) < Finger + SlowDownTreshold
    && Distance[FrameAhead](TableTop, Beam) < Distance[ExtActuals](TableTop, Beam) - 0.5 mm
  effects
    userGuidance "TableTop and Beam approaching" priority 20
    relative limit TableTop*[Rotation, Translation], Beam*[Rotation, Translation]
      at ((Distance[FrameAhead](TableTop, Beam) - Finger) / SlowDownTreshold) ^ SlowDownPattern

restriction VeryCloseTableTopAndDetector
  activation
      ¬Distance[ExtActuals](TableTop, Detector) < Finger / 2
    && Distance[FrameAhead](TableTop, Detector) < Distance[ExtActuals](TableTop, Detector) + 0.5 mm
  effects
    userGuidance "TableTop and Detector very close" priority 9
    absolute limit TableTop*[Rotation, Translation], Detector*[Rotation, Translation]
      at 0
```

- The reduction factor should be between 0 and 1.
- The minimum value should not pass the maximum value.
- Minimum and maximum should not be zero at the same time.
- There should be no movement when there is no input from the user.

```
(assert (forall ((distance Real))
      (implies (< distance Finger)
      (= (MaxTableTopRotationFunc distance) 0.0))))
```

```
;abbreviation MaxTransSpeedTableTop for 15 cmps
(assert (= MaxTransSpeedTableTop (* 15.0 100.0) ))

(define-fun MaxTableTopRotationFunc
((distance_TableTop_Beam Real)) Real
  (ite (< distance_TableTop_Beam
         (+ Finger SlowDownTreshold))
    (* (/ (- distance_TableTop_Beam Finger)
         SlowDownTreshold)
     MaxRotSpeedTableTop)
MaxRotSpeedTableTop)
)
```

```
restriction ApproachingTableTopAndCArm
  activation
    Distance[Actuals](TableTop, CArm) < 35 mm + 15 cm
  effect
    absolute limit CArm[Rotation]
      at ((Distance[Actuals](TableTop, CArm) - 35 mm) / 15 cm) * 10 dgps
```

Multiple markers at this line
- Potential (Rotation) deadlock for CArm
- Absolute limit for CArm Rotation should be non-negative

Fig. 10. Safety Analysis using the DSL

Fig. 10 gives an overview of this approach as described in [21]. In the top-left corner we start with a DSL instance. To meet our criteria for the analysis tools (fully automated, and time efficient), we create a Satisfiability Modulo Theories (SMT, [3]) model and analyse it using the SMT-solver Z3 [12]. The SMT model is generated using a model transformation (in Xtend [1]) from the DSL instance. In addition, in the top-right corner, we formulate the properties, and also formalize them in an SMT model. The combination of these models is analysed by the SMT-solver. For any failed property, an automated debugging procedure identifies a place in the DSL instance that contributes to the failure. At this place the results are displayed in the Eclipse-based editor for the DSL.

To make the formal verification feasible, we have applied several abstractions. First of all, we abstract from the acceleration characteristics of the physical objects. Similarly, we abstract from timing aspects of the system, such as the latency between sensing and acting in the real-time control loop of the safety layer. Moreover, we do not consider the physical shapes of the objects; basically we consider completely independent object positions and distances between objects.

Using these abstractions, the experiments in [21] show that fast analysis and user feedback is feasible for realistic instances of the DSL. For the correctness properties mentioned above, the applied abstractions may result in false positives. Moreover, for the deadlock check it may also result in false negatives. Nevertheless, also the deadlock check is useful as it can detect certain typical mistakes in the collision prevention rules.

3.4 Results

The development of this DSL has triggered many discussions about the collision prevention rules. By making the rules more explicit, we continuously had to decide what is really essential, and what is just an implementation detail.

The initial expectation was that there would be a lot of variability in the rules. However, the combined study of the movement control architecture and the collision prevention design has shown that a lot of the variability can be isolated. In further work we will consider whether DSLs can play a useful role in the design of these isolated parts.

Using language development frameworks like Xtext/Xtend, DSL instances can easily be transformed to various types of analysis models. When applied to implementation code, many analysis techniques would require the prior use of abstraction techniques. Instead, DSLs facilitate analysis by providing a domain-specific abstraction that naturally fits the problem domain.

4 Related Work

To model architectures, we have used the POOSL language and tooling, but the general approach is not restricted to POOSL. An overview of many other formal languages for describing architectures is provided by [17]. Our focus is on how to apply such languages and methods effectively in industrial practice. A particular challenge identified in [17] is scalability, which we address by making high-level models that omit many details and focus on parts of the functionality.

For example, we could also consider architecture description languages such as the SAE Architecture Analysis and Design Language (AADL, [29]) standard. However, at the moment that we decided to use POOSL, the AADL tools were still too much under development without convenient simulation possibilities. We could also have used MatLab, but we prefer a light-weight tool specializing in discrete event systems. In particular we focus on specification and not on implementation or verification. The goal here is to keep the models simple, in order to facilitate rapid experimentation.

In the context of the architectural validation, the goal is not to generate code out of the POOSL models, in contrast to formal methods such as VDMTools [11], Atelier B [10], SCADE Suite [15], and ASD:Suite [32]. The last tool has also been used at Philips Healthcare [20,19,27]. The goal of our architectural validation is rapid prototyping of the architectural concepts, in such a way that alternatives can easily be explored, and changes can be made quickly.

It is interesting to compare such formal methods approaches with the DSL approach. Both approaches aim for models at a higher abstraction level than implementation code, and aim to generate implementation code from these models. The code generators for formal methods approaches are usually generic and predefined (based on a formal semantics), and hence the modelling cannot abstract a lot from the implementation level. Typically the formal methods models are a kind of state machines. As the code generators for DSLs are custom, higher-level and domain-specific abstractions are feasible. Finally, the formal methods approaches focus on formal verification using sophisticated model checkers. To bring similar benefits to the DSL approach, we have introduced the transformations to various types of analysis models.

The architecture models from [22] are similar to our POOSL models. Their analysis, however, focuses on validating the requirements by observing external behaviour, whereas in our analysis we focus equally on how this behaviour is established internally.

The BIP (Behavior, Interaction, Priority; [5]) framework as described in [28] uses a similar structure as our DSL approach. However, we use a DSL whereas BIP is more a general-purpose language somewhat comparable to POOSL.

In earlier work [18] we have translated UML models into the formats of several existing validation tools like model checkers and theorem provers. Also the work on design space exploration from [4] has a similar flavour as our transformations to analysis models. Also there the goal is to create a specification at a convenient abstraction level, and then provide transformations to and from various analysis tools. In both cases the transformations hide all the low-level encodings.

5 Conclusions and Further Work

We have participated in an industrial study project for redesigning the collision prevention components of interventional X-ray systems. In this context, we have demonstrated various model-based techniques that allow for fault detection in early development phases. Thus we have gained confidence in the feasibility of the proposed redesign. Finally a prototype implementation has been delivered, which has been evaluated on the physical hardware.

The architecture analysis has led to more discussions before going into the design and implementation phases. The DSL approach has led to simple descriptions of collision prevention rules, from which a prototype implementation is generated. In our experience, the Xtext/Xtend tools enable the quick development of languages and their transformations. Our analysis techniques for DSL instances show potential, but they became available to late during the study project to have a significant impact.

An important challenge for all the analysis techniques described here is to find a good balance between the level of detail and the potential for analysis. In general, modelling in more detail is more time consuming, but it can enable more extensive analysis. Note that more detail can also significantly increase the time needed for the analysis. This is a serious issue, as we aim for quick feedback in early development phases (when many details are still unknown).

The composition of the study team has also contributed to its success. First of all it contains young people with fresh ideas. Secondly, it contains people with a lot of experience and domain expertise, which was important to avoid too many iterations of erroneous attempts. Finally it contains developers that made prototype implementations of new ideas using the existing code base; this includes both the new product features and the use of new development techniques like DSLs.

Further Work. Architectural POOSL models are currently used to validate proposed architectural changes of other parts of the system. It requires more experimentation to define clear guidelines on how to handle with the tension between level of detail and analysis power. In particular we would like to support a notion of refinement on a series of models from requirements via architecture to design.

To make sure that analysis results on the DSL are relevant for the generated code, we need to establish that the code generators and model transformations are consistent. We are currently investigating an approach where we first define a formal semantics for the DSL, and then analyse which properties of the DSL are maintained by the code generators and model transformations. A particular challenge is to do this in such a way that it can be applied in industrial practice in a cost effective way.

Finally we are improving the usability aspects of the POOSL tools. This includes the use of DSL technology for developing an Eclipse-based editor. In particular we are adding all kinds of validation checks while editing the POOSL models, in order to support early fault detection.

Acknowledgements. The authors like to thank Freek van den Berg and Sarmen Keshishzadeh for their work on analysis models for our DSL, and for providing relevant screenshots. The authors also like to thank Hans Driessen and Jan Stevens for their active participation in this study project.

References

1. Xtend. version 2.3 (2012), `http://www.eclipse.org/xtend/`
2. Xtext. version 2.3 (2012), `http://www.eclipse.org/Xtext/`
3. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability Modulo Theories. Handbook of Satisfiability 185, 825–885 (2009)
4. Basten, T., Hendriks, M., Trcka, N., Somers, L., Geilen, M., Yang, Y., Igna, G., de Smet, S., Voorhoeve, M., van der Aalst, W., Corporaal, H., Vaandrager, F.: Model-driven design-space exploration for software-intensive embedded systems. In: Model-Based Design of Adaptive Embedded Systems. Springer (2013)
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proceedings of SEFM 2006, pp. 3–12. IEEE Computer Society (2006)
6. van den Berg, F., Remke, A., Mooij, A., Haverkort, B.: Performance evaluation for collision prevention based on a domain specific language. In: Balsamo, M.S., Knottenbelt, W.J., Marin, A. (eds.) EPEW 2013. LNCS, vol. 8168, pp. 276–287. Springer, Heidelberg (2013)
7. Blender, `http://www.blender.org/`
8. Brooks, R.: A robust layered control system for a mobile robot. IEEE J. Robot. Autom. 2(1), 14–23 (1986)
9. Carpin, S., Mirolo, C., Pagello, E.: A performance comparison of three algorithms for proximity queries relative to convex polyhedra. In: Proceedings of ICRA 2006, pp. 3023–3028 (2006)
10. ClearSy: Atelier B, `http://www.atelierb.eu/en/`
11. CSK Systems Corporation: VDMTools, `http://www.vdmtools.jp/en/`
12. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
13. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
14. Eindhoven University of Technology: Software/Hardware Engineering (SHE) - Parallel Object-Oriented Specification Language (POOSL), `http://www.es.ele.tue.nl/poosl/`
15. Esterel Technologies: SCADE Suite, `http://www.esterel-technologies.com/products/scade-suite/`

16. Fitzgerald, J.S., Larsen, P.G.: Balancing insight and effort: The industrial uptake of formal methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 237–254. Springer, Heidelberg (2007)
17. Garlan, D.: Formal modeling and analysis of software architecture: Components, connectors, and events. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 1–24. Springer, Heidelberg (2003)
18. Graf, S., Hooman, J.: Correct development of embedded systems. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 241–249. Springer, Heidelberg (2004)
19. Hooman, J., Mooij, A.J., van Wezep, H.: Early fault detection in industry using models at various abstraction levels. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 268–282. Springer, Heidelberg (2012)
20. Hooman, J., Huis in 't Veld, R., Schuts, M.: Experiences with a compositional model checker in the healthcare domain. In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 93–110. Springer, Heidelberg (2012)
21. Keshishzadeh, S., Mooij, A.J., Mousavi, M.R.: Early fault detection in DSLs using SMT solving and automated debugging. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 182–196. Springer, Heidelberg (2013)
22. Kramer, J., Magee, J., Uchitel, S.: Software architecture modeling & analysis: A rigorous approach. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 44–51. Springer, Heidelberg (2003)
23. Larsen, E., Gottschalk, S., Lin, M., Manocha, D.: Fast distance queries with rectangular swept sphere volumes. In: Proceedings of ICRA 2000, vol. 4, pp. 3719–3726 (2000)
24. Li, L., Hooman, J., Voeten, J.: Connecting technical and non-technical views of system architectures. In: Proceedings of CPSCom 2010, pp. 592–599 (December 2010)
25. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
26. Mooij, A.J., Hooman, J., Albers, R.: Gaining industrial confidence for the introduction of domain-specific languages. In: Proceedings of COMPSAC workshops, IEESD 2013, pp. 662–667. IEEE (2013)
27. Osaiweran, A., Schuts, M., Hooman, J., Wesselius, J.H.: Incorporating formal techniques into industrial practice: an experience report. In: Proceedings of FESCA 2013. ENTCS, vol. 295 (2013)
28. Poulhiès, M., Pulou, J., Rippert, C., Sifakis, J.: A methodology and supporting tools for the development of component-based embedded systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop 2006. LNCS, vol. 4888, pp. 75–96. Springer, Heidelberg (2007)
29. SAE International: Architecture Analysis & Design Language (AADL). SAE Standard AS5506B (September 2012)
30. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. Pearson Education (2008)
31. Theelen, B.D., Florescu, O., Geilen, M., Huang, J., van der Putten, P.H.A., Voeten, J.: Software/hardware engineering with the Parallel Object-Oriented Specification Language. In: Proceedings of MEMOCODE 2007, pp. 139–148. IEEE (2007)
32. Verum Software Technologies: ASD:Suite, `http://www.verum.com/`
33. Voelter, M.: DSL Engineering, Version 1.0 (2013), `http://dslbook.org`