Impact Analysis of Changes in Functional Requirements in the Behavioral View of Software Architectures

---

Master thesis
Looman, S.A.M.
August 17, 2009

Committee
A. Goknil, Msc.
dr. I. Kurtev
dr.ir. K.G. van den Berg
prof.dr.ir. M. Aksit

Research group
University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Software Engineering

Abstract

In software development, customer requirements are the driver for the produced system. The requirements are ultimately translated by software engineers into a solution. An architecture is created based on the requirements. The architecture describes the system at a high level of abstraction using one or more views. During and after the process of creating the solution, the customer needs often evolve resulting in changed requirements. The architecture has to be changed accordingly to satisfy the changed requirements. Change impact analysis will help changing the architecture to support the satisfaction of the changed requirements. Currently, performing change impact analysis is hard, because both the requirements and the architecture are not formalized.

We propose an approach on how to perform change impact analysis in software architectures. This approach is based on the validation of functional requirements. The functional requirements are transformed into a formal behavior description. This formal behavior description states the behavior which must be present, or absent, in the behavior of the architecture. If one or more of the formal behavior descriptions fail, we can identify which requirements are not satisfied by the architecture. Based on traces between the requirements and architecture, and the requirements which are not satisfied by the architecture, we can identify the components of the architecture which are not satisfying the requirements.

In order to validate requirements, we need to derive the behavior of the architecture. The behavior of the architecture is derived by building a simulatable model based on the architecture. AADL is used to record the architecture itself. The architecture is annotated with additional properties to specify the behavior of individual components. The simulation of the architecture is performed using the modeling language Alloy. Using the simulation in Alloy, we derive the state space which represents the behavior of the architecture. After simulation of the architecture, we can assert whether the architecture satisfies the requirements.

Given the derived state space through simulation, satisfaction of functional requirements, and traces from functional requirements to architecture components, we perform change impact analysis. We provide several guidelines for different activities of this process. To identify strengths and weaknesses of the approach, the apporoach is evaluated by performing five change scenarios in an existing project. The results of the change scenarios are discussed afterwards.

From this research we can conclude that the proposed approach is promising. However, Alloy imposes limitation on the size of the architecture and length of simulation. Also, Alloy is slow when simulating architectures. Other directions are given for future research.

Acknowledgments

I would like to thank the people who have helped me during this research project.

Arda Göknil, my first supervisor, helped me by guiding me during this work and often giving constructive feedback which helped me greatly. Ivan Kurtev, my second supervisor, for providing many directions on the research. Klaas van den Berg, my third supervisor, for very helpful providing comments on the thesis. Also, Klaas and Ivan, for providing the Advanced Design in Software Architectures course, which made me enthusiastic about model driven engineering.

My parents and siblings for being supportive in many ways. My parents for making it possible to study at Twente University in many ways. Joris for showing interest in my research and providing helpful comments.

My fellow students from the room 5066 (formerly 5070), who provided a pleasant working place and atmosphere. We had numerous enjoyable discussions which gave further insight into problems and possibilities.

My fellow colleagues at Atos Origin – Technical Automation, specifically Herbert Reesink and Marcel Winkelhorst. Without them, I would not have an example architecture, which made this research further possible.

Steven Looman, August 2009, Enschede

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In software development, customer requirements are the driver for the produced system. The requirements are ultimately translated by software engineers into a solution which fulfills the customers' needs and wishes. An architecture is created based on the requirements. This architecture describes the to-be-system software at a high level of abstraction using one or more views. During the process, however, the customer needs often evolve resulting in changes to the requirements. Because the architecture is driven by the requirements, the architecture has to be changed accordingly to fulfill the evolved requirements.

Known as the ripple effect, a single change to a part of the system can result in changes to other parts of the system. Change impact analysis has to be performed to investigate which parts of the system are affected by the initial change and which parts of the system need maintenance.

Also, given that the architecture is a high-level abstraction of the whole system, verifying that the updated architecture conforms to the changed requirements in an early phase decreases the cost of the maintenance. If the changes first would have to be fully implemented and tested, danger exists that the updated system does not satisfy the updated requirements and the system has to be modified again. As such, testing if the architecture conforms to the new requirements, defects of the updated system can be identified early and maintenance cost can be saved.

This research is conducted in the line of the QuadREAD project. The QuadREAD tries to improve the alignment of the earlier phases of system development, requirements engineering and architectural design, as shown by the circle in figure 1.1. As the later phases of software development are dependent on the earlier phases, improving the quality of the earlier phases improves the quality of the later phases. The project tries to bridge the gap between requirements engineering and architectural design.

The QuadREAD project tries to fulfill this goal by researching traceability between requirements and architectural design choices. Relations are created between different requirements and architectural elements. Given that the relations between requirements and architecture are valid, related architecture elements can be identified when one or more requirements are changed. This

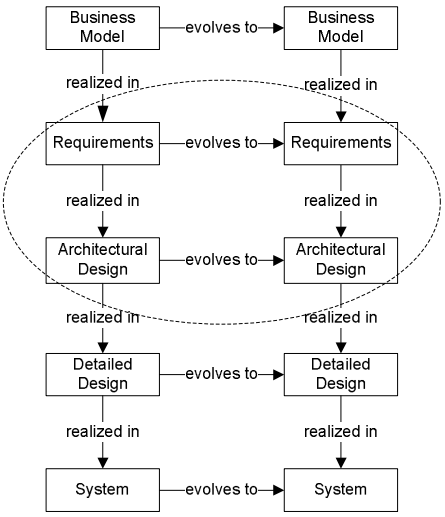


Figure 1.1: Traceability in system development

research focuses on the evolution of architectural designs, driven by the evolution of requirements.

1.2 Problem statement

The architecture should fulfill the requirements. When either the requirements or the architecture itself evolves, the (evolved) architecture should be validated against the (evolved) requirements to verify that the architecture still fulfills its purpose. The requirements engineer and the architect should be able to validate if the architecture satisfies the requirements in order to determine design defects in the early stages of the software development process.

In the current state of the practice, software architectures are often informal descriptions and/or figures consisting of boxes and arrows without clear semantics. Validating the architecture against the requirements becomes a manual, time consuming, and error-prone task which cannot be automated or even aided by a tool. Instead, the system is validated against the requirements after the system itself has been built. As a result, performing change impact analysis based on the architecture hardly possible, if possible at all.

To be able to improve change impact analysis, the architecture needs clear semantics. To provide clear semantics of the architecture, the architecture has to be formalized. Change to one part of the system can result in a ripple effect throughout the entire system. This can be due to introduced incompatibility of communication protocols, which can be detected by static analysis. However, a change can also introduce new behavior in the architecture. The change in behavior is only detectable through architecture behavior analysis. For example, the result of a change in behavior can cause a possible dead lock situation within the architecture. Static and behavior semantics of the architecture must be taken into account when performing change impact analysis.

The main research question of this study is as follows:

How can the validation of requirements in software architectures and performing change impact analysis in software architectures be improved?

The main research question is divided into a number of sub questions:

- What kind of system properties can be checked in the architectural design level?
- How can we reformulate requirements in terms of solution domain (architecture) in order to validate requirements in the architecture?
- How can we use the validation of requirements in architecture to perform change impact analysis?
- What tool(s)/method(s) can be used, if there are any at all, to support change impact analysis?
- Which parts of requirements validation and change impact analysis can be automated?
- Does the requirements validation and change impact analysis result in a simple yes/no analysis? If not, what kind of additional information can we provide as a result of this analysis?

1.3 Approach

The approach taken for this project is as follows. At first, we conducted literature research to investigate the current state of the art. Different materials related to software architectures, such as standards (IEEE-1471[IEE00]), architecture description languages (Acme[GMW97, Mon01], xADL2.0[DvdHT01a], and AADL[Fei]), and other related material (the 4+1 view model[Kru95] and classification framework for ADLs[MT00]) were investigated. From this material, static and behavioral properties were extracted.

Secondly, an architecture description language is chosen to record the architectures. Several criteria were chosen which the architecture description language has to support. For example, the architecture description language has to be actively being used, provide a basic toolset, and promises to be powerful enough to express the static and behavioral properties which are required in our context. Chapter 4 introduces the chosen architecture description language, AADL.

Thirdly, we formalize the selected static and behavioral semantics. To the largest extent possible, the semantics are derived from the literature. If the literature was lacking, we introduced semantics ourselves. These semantics are used for the simulation of architectures through which we derive the architecture behavior. Chapter 5 describes the simulation of AADL architectures.

A method is proposed to validate the architecture against the functional requirements and perform change impact analysis based on the simulation of the architecture. The method to perform change impact analysis is based on the satisfaction of requirements by the software architecture. Chapter 6 describes the validation of functional requirements, and change impact analysis.

Finally, a case study is carried out to identify the strengths and weaknesses of the proposed method. An example project is provided by Atos Origin - Technical Automation. We use the requirements and architecture from this project. The architecture itself is re-specified in the chosen architecture description language. Based on the requirements and architecture, we define change scenarios which we use to investigate the proposed method. The results of the case study can be found in chapter 7.

A more elaborate description of the approach is given in chapter 3.

1.4 Contributions

The contributions from this research are as follows. First, we have formalized the semantics of AADL. The formalization of semantics allows us formally specify the behavior of the architecture. As a result, tools can be developed which use the behavior of the architecture and reason about it.

Secondly, we show how an architecture is simulated using Alloy. Simulation of an architecture gives early insights in the behavior of an architecture, instead of having this ability only in a later phase of software development, such as the implementation phase. This provides a means to predict properties of the system under design based on the architecture.

Thirdly, based on the simulation of the architecture, we show how requirements are validated in the architecture. The simulation is used to derive the behavior of the architecture. As the simulation gives insight in the behavior of the architecture, functional requirements can be validated against the architecture.

Finally, we proposed a method to perform change impact analysis. By providing details of this method, we show how to perform change impact analysis in an architecture based on its behavior. This method can be used as a basis for future work. A case study is performed to identify the strengths and weaknesses of this method.

1.5 Outline of this thesis

The outline of this thesis is given in figure 1.2. It gives the chapters in this thesis and the relations between the chapters.

Chapter 1 is the introduction of the thesis.

Chapter 2 introduces the basic concepts which are essential for understanding this work. The concepts which are explained in this chapter are: Model driven engineering, software architectures, and change impact analysis.

Chapter 3 describes the approach taken for this research. A method to validate functional requirements in software architectures is given. Based on the validation of functional requirements, a process to perform change impact analysis is proposed. A short description of the supporting framework to perform change impact analysis is given.

Chapter 4 gives an introduction to AADL and Alloy. A description of the concepts of AADL is given. The rationale behind the choice for AADL is

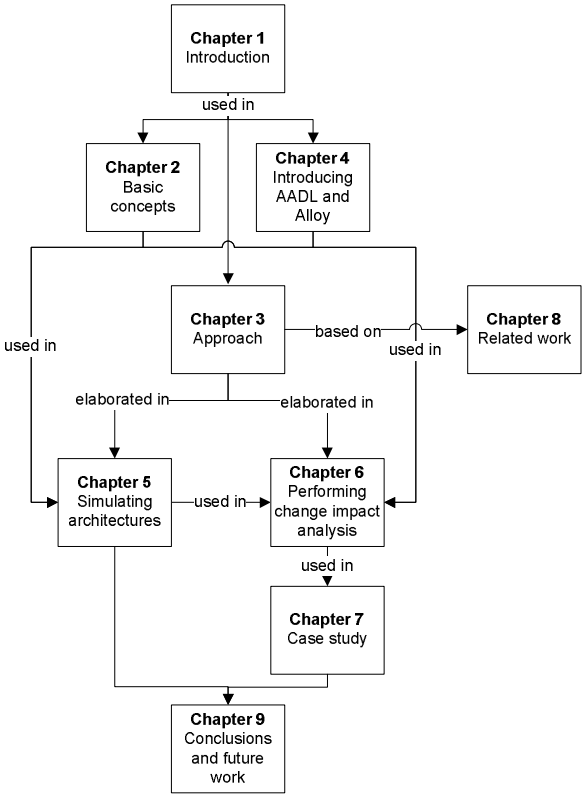


Figure 1.2: Thesis outline

explained. Furthermore, an example architecture is given which is used throughout the thesis. Also, the concepts of Alloy are given, together with the rationale behind the choice for Alloy.

Chapter 5 describes the simulation of the architecture specified in AADL using Alloy. The concepts of the simulation are explained, together with examples. A different approach, which allows for parallel simulation, is also given. From the basic simulation and parallel simulation, benchmarks are derived to test the implementation of the simulation. Also, limitations of the implementation are given.

Chapter 6 explains how, given the requirements and the architecture, change impact analysis is performed. Different parts of the approach given in chapter 3 are elaborated. An example supports the explanation of the approach. Also, a method to partially synthesize an architecture which fulfills its requirements is described.

Chapter 7 evaluates five change scenarios. Each change scenario is performed using the proposed method. After each change scenario, a discussion on the usability of the proposed method is given.

Chapter 8 describes the related research for change impact analysis in software architectures. The related work is split into: behavior analysis of software architectures, requirements validation in software architectures, and change impact analysis in software architectures.

Chapter 9 concludes the thesis and reflects on the research. The research questions given in this chapter are answered. Furthermore, the chapter proposes directions for future work.

Appendix A gives the Alloy module used by the simulation. The Alloy module is derived from the AADL standard and meta-model.

Appendix B describes the RPM architecture used by the example in chapter 4, and the case study in chapter 7. The appendix includes: informal requirements, architecture in AADL, formalized requirements, and traces from requirements to architecture.

Appendix C contains a part of the RPM architecture in Alloy.

Chapter 2

Basic concepts

2.1 Introduction

This chapter provides the basic concepts needed to understand this research. The chapter elaborates on model driven engineering in section 2.2. Section 2.4 introduces the concepts relevant for software architectures. In section 2.5 the concepts of change impact analysis are provided. Finally, section 2.6 concludes this chapter.

2.2 Model driven engineering

The main focus of model driven engineering is the creation of models of the problem domain, in which the problem has to be solved by a given system under development. Rather than focusing on the computing domain, the focus is on the problem domain.

Model driven engineering increases productivity in various ways. For example, it simplifies the design as it is close to the problem domain. It also maximizes compatibility between systems, due to the abstraction of the (possibly different) solution domains.

[Ken02] introduces the notion of a software development process and modeling space for the organization of models. Three proposed dimensions for the organization of models are: degree of abstractness or concreteness of models, the subject area the models belong to, and organizational issues such as authorship, versioning, and location over models.

2.2.1 Models and meta-models

Multiple definitions of the term model exist. For example, the MDA guide[Obj03] defines a model as: “A formal specification of the function, structure and/or behavior of an application or system.". Kleppe et al.[KWB03] defines a model as: “a description of (a part of) a system in a well-defined language". Another definition of model, by Kurtev[Kur05], is: “A model represents a part of the reality called the object system and is expressed in a modeling language. It provides knowledge for a certain purpose that can be interpreted in terms of the object system.”

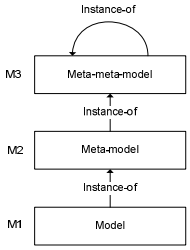


Figure 2.1: Generic meta-modeling architecture

To be successful, a model should provide a number of properties to be successful[Sel03]. A model should provide an *abstraction* of the real world. The model captures the core of the problem and abstracts away from irrelevant details. The model should be *understandable* by humans. The model should *accurately* represent the real world. Furthermore, if the model is accurate, the model can be used to reason about, or *predict*, properties of the system. Finally, analysis of a model should cost less resources than the analysis of a real system, *low cost*.

Models are instances of their meta-models. For meta-models, Kurtev[Kur05] provides the following definition: "a model of a modeling language". A meta-model gives an abstract syntax, or structure, to which its models must conform. If a model conforms to the allowed structure by the meta-model, a model is said to be an instance of the meta-model.

2.2.2 Model driven architecture

In 2001, the Object Management Group (OMG) launched the Model Driven Architecture (MDA) framework. The MDA guide[Obj03] describes an overview of the framework and gives definitions for the terms used in MDA. The MDA framework tries to address two main problems found in software engineering: *Portability*, and *interoperability*.

The first problem, *portability*, results from the fact that each time a new and 'hot' technology appears, companies are forced to port their applications to this technology. For example, a shift is observed in the use of middle-ware technology. CORBA is partially replaced by the newer Web Services. Many existing systems need to be ported to make use of this new technology, while their functionality remains unmodified. The resources required for porting an existing system to a new technology can result in a large effort.

Secondly, MDA tries to address the *interoperability* problem. Larger systems are built in a modular fashion, where the whole system is divided into several smaller modules. These smaller modules are often implemented using the technology which suits the problem best. Given the use of different technologies, interoperability between these modules is hard to maintain.

MDA tries to cope with the first problem by introducing modeling and models, and by the separation of specification of a system and the details of the implementation of a system. The specification of a system is more abstract, while the details of the implementation of a system are more concrete. The concrete details of the implementation, for each technology, are derived from

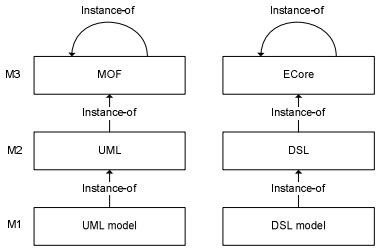


Figure 2.2: Examples of meta-modeling architectures

the abstract specification, which is defined once.

The main concepts MDA introduces are:

Platform Independent Model A platform independent model is a view of a system from the platform independent viewpoint.

Platform Specific Model A platform specific model is a view of a system from the platform specific viewpoint.

The abstract specification of a system is recorded in a Platform Independent Model (PIM). The Platform Specific Model (PSM), containing the concrete details of the implementation, is based on the platform independent model. Model transformations are used to convert a PIM to a PSM. MDA aims at automating this process, but the transformation can also be done manually.

2.2.3 Meta-modeling architectures

Several meta-modeling architectures exist. The foundation for these meta-modeling architectures is given in figure 2.1. In this figure, a meta-model itself is considered to be a model, which conforms to a given meta-meta-model. The meta-meta-model is, in practice, expressed in itself, instead of using higher level models. The meta-meta-model is said to be self-reflective.

Several meta-modeling architectures exist, as shown in figure 2.2. The OMG provides the Meta Object Facility (MOF). It uses it to express UML meta-models and UML models, shown on the left. Shown on the right in the figure, the Eclipse Modeling Foundation is another example of a meta-modeling architecture.

2.3 Software requirements

In [KS98], the following definition for software requirements is given:

> Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem.

The SWEBOK[AMBD04] gives the following definition of a software requirement:

> At its most basic, a software requirement is a property which must be exhibited in order to solve some problem in the real world.

A software requirements is a property that describes constraints on the software that will be used to solve a problem. A problem can be the automation of a task, or to support a business process.

An important property of software requirements is that these must be verifiable. Other important properties are ones such as: maintainability, verifiability, completeness, and correctness. [MK95] provides a technique to assist in properly constructing requirements. The paper specifies that requirements have to be SMART. SMART requirements are: Specific, Measurable, Attainable, Realizable, and Time bounded.

Software requirements can impose constraints on both the *product* (the software to be developed), and the *process* which is used to develop the software. An example of a product requirement is that the software has to be verify that a student meets all prerequisites before he/she can register for a course. An example of a process requirement is that the product has to be written in C++.

Software requirements can generally be divided into two categories: *functional* and *non-functional* requirements.

2.3.1 Functional requirements

Functional requirement specify required functionality, or behavior, of the system. These can be calculations, technical details, data manipulation and processing. For example, a functional requirement may state that the system must be able to modulate a signal. A synonym for functional requirements is *capabilities*.

A use case can be generated from a set of functional requirements. The use case specifies how the system shall act on a certain input, as described by the chosen set of functional requirements.

2.3.2 Non-functional requirements

Non-functional requirements are requirements which constrain the system under development. A synonym for non-functional requirements is *constraints* or *quality requirements*. For example, a non-functional requirement can state that the software built for a call-center must be able to handle 100 concurrent calls.

[AMBD04] specifies that non-functional requirements can be further classified into groups, such as: performance requirements, maintainability requirements, safety requirements, and reliability requirements.

2.4 Software architectures

The IEEE 1471-2000 standard [IEE00] gives the following definitions of terms applicable to software architectures:

Acquirer An organization that procures a system, software product, or software service from a supplier. (The acquirer could be a buyer, customer, owner, user, or purchaser.)

Architect The person, team, or organization responsible for systems architecture.

Architecting The activities of defining, documenting, maintaining, improving, and certifying proper implementation of an architecture.

Architectural description A collection of products to document an architecture.

Architecture The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

Life cycle model A framework containing the processes, activities, and tasks involved in the development, operation, and maintenance of a software product, which spans the life of the system from the definition of its requirements to the termination of its use.

System A collection of components organized to accomplish a specific function or set of functions.

System stakeholder An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.

View A representation of a whole system from the perspective of a related set of concerns.

Viewpoint A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and techniques for its creation and analysis.

Figure 2.3 gives the conceptual model as it is defined by the IEEE 1471-2000 standard.

A system can be seen as an individual application, a system with multiple subsystems, a product line, or even a whole enterprise. A system is inhabited in an environment, or context, which influences the system. Influences come, either directly or indirectly, from other systems which interact with the system under development. The boundaries of the system under development are determined by the environment.

Each system has one or more stakeholders, which all have a concern in the system. An example of a concern is a consideration such as performance or reliability. The system has one or more missions which the system is intended to fulfill for one or more stakeholders.

Every system has an architecture which is described by an architectural description. An architectural description contains one or more views, which describe the concerns of one or more stakeholders of the system. A viewpoint specifies the template for a view to which it must conform. A view consists of at least on architectural model.

Stakeholders include clients, users, architects, developers, testers and the like. Each stakeholder has at least one role in the creation and use of a software architecture description. The key roles for stakeholders are the client and the architect.

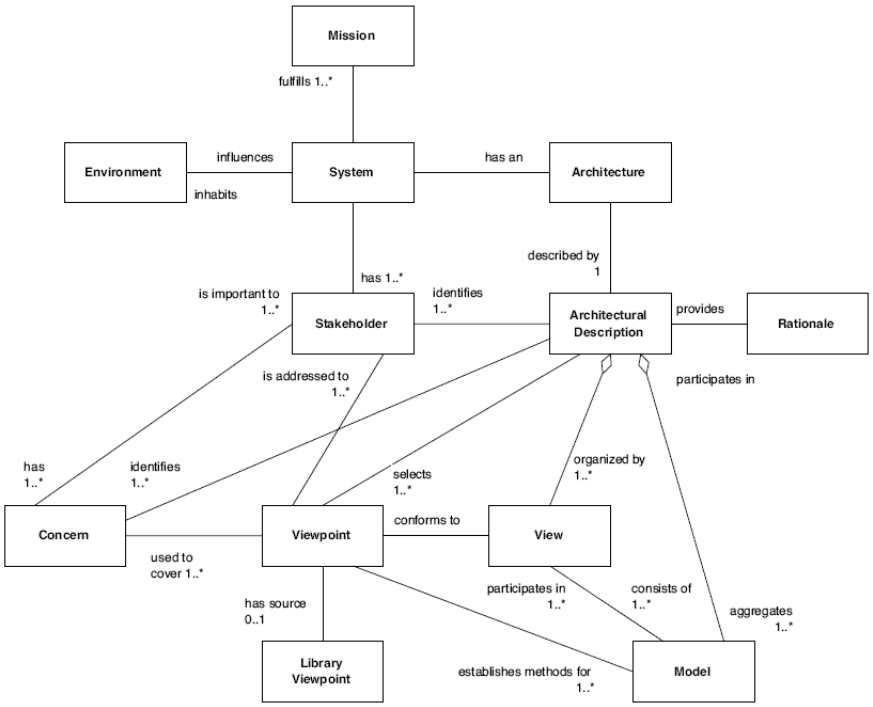


Figure 2.3: IEEE 1471-2000 conceptual model for software architectures[IEE00]

2.4.1 Architectural views

Different stakeholders have different concerns. If a single diagram for the whole the system would be given, the information would be overwhelming and incomprehensible to the stakeholders. Therefore, an architecture is usually composed of multiple views, or a representation of (a part of) the system from a single perspective. A view consists of a model. The viewpoint gives the meta-model for the view. While several predefined viewpoints exist, an architect is by no means restricted to those viewpoints. New viewpoints can be created when appropriate.

The book Software Architecture in Practice - Second Edition[CK03] specifies three common categories for views/viewpoints:

- Module structures;
- Component and connector structures;
- Allocation structures.

The Modules structures category describes different modules of the system. This category represents a static structure in a code-based way of the system; the modules of a system are defined as units of implementation.

The component and connector structures category focuses on the run-time components of a system. The elements in this category are components and connectors.

Finally, the allocation structures category describes the relationships between the elements of the system and the environment of the system.

IEEE-1471 example views

[IEE00] provides examples of viewpoints and examples of views for the viewpoints. One of the example viewpoints is called the structural viewpoint. It provides a language with the entities components (individual units of computation), components have ports (interfaces), connectors (interconnection between components), and connectors have roles (attachments to components). This viewpoint addresses the concerns related to the structure of the system. It can be used to analyze attachments of connectors and type consistencies.

A second example of a viewpoint is the behavioral viewpoint. Elements of the language provided by this viewpoint are events, processes, states, and operations on the given entities. The concerns it addresses are related to the behavior of the system. Some of the analytic methods which can be applied are Communicating Sequential Processes [C.A78], and the pi-calculus [Mil99].

A third example viewpoint, physical interconnect viewpoint, addresses the concerns related to the physical communication within the system. The language vocabulary consists of physically identifiable node, point-to-point link, and a shared link. This viewpoint focuses on the structure of a system, but unlike the structural viewpoint it focuses on the physical implementation of the system.

4+1 view model

Kruchten gives in the paper [Kru95] a framework consisting of different views. The "4" in "4+1" represents four views. The logical view represents the functional requirements of the system, divided into several key abstractions taken from the problem domain. Secondly, the process view is used to represent the non-functional requirements such as performance and availability. This view shows the communicating programs in the system, possibly described at several levels of abstractions. The development view is used to cut the software into small chucks and represents internal requirements. This view can again be layered across several levels of abstractions. Finally, the physical view represents non-functional requirements with regard to availability and reliability. It gives the network of nodes on which the system executes. All of the above views are supported by their own notation, often based on or derived from existing notations.

The "+1" in "4+1" stands for the scenarios view and is a view which is mostly redundant with the other views. This view is used to discover architectural elements. The discovery of elements is supported by a proposed scenario-driven approach. This approach consists of several steps to create an initial architecture from a small number of key-scenarios and use further iterations to extend the architecture by introducing more scenarios. Secondly, the scenario view is also used to illustrate and validate the architecture after the architecture design is complete.

The given views are overlapping as elements in one view and another view are connected with each other. Several heuristics are given to map elements from one view to another view.

Component and Connector view

The *component and connector* view consists of three main concepts: components, connectors, and configuration. The following definitions for the three elements come from [MT00]:

Component A component in an architecture is a unit of computation or a data store. Therefore, components are loci of computation and state.

Connector Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions.

Configuration Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure.

In [CBB+02], components and connectors at least have two properties: name, type. The name is used for identification of a component or connector. It should also suggest the nature of the functionality or interaction, for a component or connector, respectively. Dependent on the need to model other properties, the components and connectors can be annotated with domain specific properties, such as performance and reliability values.

In a configuration, the component and connector types are instantiated. The component and connector instances have a runtime manifestation, are consuming execution resources, and contributing to the execution behavior of that system[CBB+02]. A configuration, together with its components and connectors, conforms to a specific style.

2.4.2 Architectural styles/patterns

A style specifies rules to which an architecture must conform. An architectural style specifies the vocabulary that can be used to build an architecture, but also specifies constraints on how the elements of the vocabulary may be interconnected (for example, no cycles), or constraints on the execution semantics of the elements found in the vocabulary. A synonym for architectural style is architectural pattern.

A style dictates a form for an architecture, restricting many of the possible design choices and preventing possible errors in the design. Also, it serves as a vehicle for communication, because when an architect talks about a style, another stakeholder who knows the style quickly knows what the architect is talking about. Furthermore, tools can aid the architect when building an architecture in a specific style as the tool imposes constraints provided by the viewpoint.

A commonly seen example of a style is layered style[GS92]. The style requires that a system has two or more layers stacked upon each other. A layer n is only allowed to communicate with the layer n-1 or the layer n+1, in other words, with the layers it has direct contact with. A layer provides services to the layer above it and is a client to the layer below it. Figure 2.4 provides a graphical representation of the layered style.

Another example of an architectural style is the C2 style[MORT96]. The C2 style vocabulary contains components and connectors. Each component has two ports, an upper port and a lower port. Components can only send requests through the upper port and send notification through the lower port.

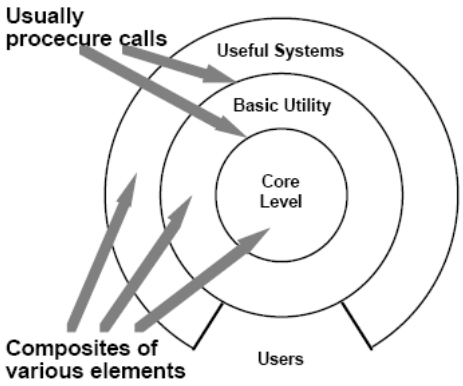


Figure 2.4: Layered architectural style[GS92]

A connector connects two or more components, or directly connects to another connector. An example of the C2 style can be seen in the figure 2.5.

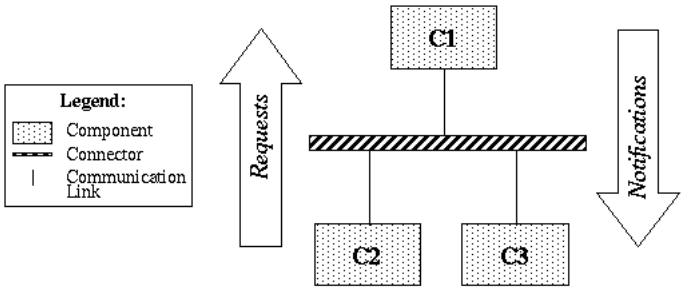


Figure 2.5: C2 architectural style[UCI]

2.4.3 Modeling software architectures

Software architectures are recorded by the use of architecture description languages (ADL). An ADL defines a set of types in which the architecture can be expressed (meta-model), and a way to represent the architecture. The architecture can be recorded through the use of a textual or graphical specification. Roughly two generations of ADLs can be defined.

First generation architecture description languages

The first generation languages all have a focus on a specific domain. Because of this focus, one ADL might fit the needs for one project, but not the needs for another project. Examples of first generation languages and their scopes are as follows [MT00]:

Aesop Specification of architectures in specific styles

C2 Architectures of highly-distributed, evolvable, and dynamic system

Darwin Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings

MetaH Architectures in the guidance, navigation and control (GN&C) domain

Rapide Modeling and simulation of the dynamic behavior described by an architecture

SADL Formal refinement of architectures across levels of detail

UniCon Glue code generation for interconnecting existing components using common interaction protocol

Weaves Data-flow architectures, characterized by high-volume of data and real-time requirements on its processing

Wright Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems

[MT00] provides a classification framework for these ADLs consisting of three elements: components, connectors, and configuration. The research shows that each of the ADLs roughly conforms to the classification.

Second generation architecture description languages

The second generation ADLs focus more on the general component and connector style. They also focus on extendability of the language itself. This can be either through the annotation of the architecture with key-value-pairs, or by extending/altering the meta-model. Examples of second generation languages are: Acme, xADL2.0, and AADL.

Acme/Armani[GMW97, Mon01] is designed to be an interchange language for several existing ADLs. Using Acme, one should be able to model an architecture in one language and use the tools of another language to analyze the model. Apart from this main goal, Acme was designed to provide an easy way for development of new tools for analysis and visualization, provide a foundation for new (domain specific) ADLs, define a standard for architectural information and be human readable and writable. The elements found in the Acme language can be annotated by key-value-pairs.

xADL 2.0[DvdHT01a] is built upon XML and schemas. The default schema provides the concepts of component, connector, interfaces, and configuration. The language can be easily extended by extending the existing schema. Several schemas are already provided by the xADL2.0 team.

AADL[FGH06] is a language developed by CMU, and is standardized by the SAE[SAE04, SAE09]. AADL stands for Architecture Analysis and Description Language. The language was originally developed as the Avionics and Aviation Description Language, and thus has a focus on embedded systems. Yet, it provides component types relevant for general software development such as process, and thread. The language can be extended through the use of key-value-pairs and annexes. An annex is a complete sub-language which can be embedded in the AADL itself.

2.4.4 Software architecture analysis techniques

Different kind of software architecture analysis techniques exist[DN02]. Three types of techniques are given in this subsection: *inspection and reviews*, *model-based analysis*, and *simulation-based analysis*.

Inspection and reviews

Inspection and reviews of software architectures are a manual technique. A person, or a group of persons, inspect and review the architecture. Examples of different methods which have been proposed are *SAAM* and *ATAM*. A short overview of both SAAM and ATAM are given below.

Scenario-based Architecture Analysis Method (SAAM) is a scenario-based analysis technique to evaluate software architectures. The SAAM consists of five stages. A short description of each stage is as follows:

Describe candidate architecture The architecture is recorded by the use of an ADL. It should clearly identify the components and connectors.

Develop scenarios Scenarios are developed to show what kind of activities the architecture must support. These scenarios should capture all important uses of the system. This means that scenarios should include tasks relevant to all different holders, such as end users, system administrators, and developers. Scenarios can thus include testing functionality of the system, but also modifying the system itself.

Perform scenario evaluations Each of the change-scenarios, modifications to the architecture are recorded, together with the estimated cost of the modification.

Reveal scenario interaction If two scenarios require the modification of the same component(s) or connector(s), these scenarios interact.

Overall evaluation Each scenario is given a weight, and an overall ranking for each scenario is determined, determined by its interactions and relative importance.

In the end, SAAM provides a collection of small metrics. Given the metrics, competing architectures can be evaluated on a per-scenario basis and tradeoffs can be made.

The Architecture Tradeoff Analysis Method[KKC00, CK03] (ATAM) is a method to evaluate software architectures. ATAM is based on SAAM. The method is aimed at evaluating quality goals, and the tradeoffs between different quality goals within the architecture.

The ATAM requires participation of three groups. The first group, *the evaluation team*, is external to the project and consists of competent and unbiased outsides. The second group, *the project decision makers*, are the people who have the authority to make change the direction of the project. Finally, *the architecture stakeholders*, are the stakeholders of the architecture, such as developers, testers, and users of the system.

Two phases are defined in the ATAM. The first phase is used to present the ATAM, present the business objectives of the architecture, present the architecture, identify architectural approaches, generate quality attribute utility

tree, and analyze the architectural approaches. The second phase continues the evaluation of the architecture. The second phase consists of brainstorming and prioritization of scenarios, analyzing architectural approaches, and presenting the results.

The whole process is manual and aimed at evaluating quality attributes of the architecture. ATAM is not meant for evaluation of the requirements, neither is it meant for code evaluation or actual system testing. Also, ATAM identifies possible areas of risk in the software architecture, and is not a precise instrument.

Model-based analysis

Model-based analysis of software architectures uses the model as defined by the software architecture to perform analysis[EKHL03]. This type of analysis is usually automatic. Model-based analysis can be used to analyze a number of properties of the system to be built, such as structural properties, behavioral properties, and non-functional properties.

An example of structural analysis is the analysis of compatibility of interfaces between components.

The behavior of an architecture is often specified by state chart for the components contained by the architecture. These state charts are used to analyze the architecture for different behavioral properties, such as deadlock freedom. [EKHL03] provides analysis by transforming state charts, a behavior specification, into CSP, a process algebra, and exhaustively check this model for a number of properties.

Finally, an example of a non-functional property is the time required to perform a certain action after an event in the environment has been detected, by analyzing the time required by the individual components in the architecture involved in performing the action.

Simulation-based analysis

Simulation-based analysis of software architectures is used to analyze the software architecture through simulation of the architecture. The results of the simulation can be manually inspected by a person or automatically by a program. The simulation gives information about behavioral properties, and non-functional properties of the architecture.

Manual inspection of the simulation can be used in several ways. One example is to get familiar with the system. Another example is the manual validation of behavior of the architecture.

Automatic inspection can be performed if a number of rules have been specified to which the simulation must conform. For example, in [LHS08], traces are tested against a number of defined specification behavior patterns. Types of proposed behavior patterns are: *occurrence*, *order*, and *compound*. These pattern types are divided into several more specific patterns. For example, from the occurrence type, the patterns *absence*, and *existence* are derived. These patterns require that behavior is absent or must exist in the simulation, respectively.

Changes during requirements	**Changes during design**	**Changes during implementation**	**Changes during test**
Enhanced functionality requirements	Design trade-offs	Design error	Design changes
Deleting obsolete requirements	Timing issues	Algorithm coding adjustments	Redefining test cases

Table 2.1: Example causes of software change[Boh02]

2.5 Change impact analysis

When a single part of software is changed, other parts of the software are possibly subject to change as well. To investigate which parts of the software are to be changed, change impact analysis is performed. Bohner defines change impact analysis as: "The determination of potential effects to a subject system resulting from a proposed software change"[Boh02].

When no change impact analysis is performed, the ripple-effect can cause unforeseen consequences. These unforeseen consequences can cost a large amount of resources. Software systems are more and more developed in a distributed-platform base. Problems of interoperability between elements of distributed software systems arise. The benefit of performing change impact analysis is that one can predict the cost of performing a change to a software system.

Bohner identifies three basic software change activities: *understanding software with respect to the change*, *implementing the change*, and *retesting the newly modified system*. The first activity is important to find the impacted parts, and find other possible impacted parts. The second activity requires us to be aware of the riple-effect. Finally, the third activity, requires us to do regression testing of the system, and possibly creating new tests based on new requirements.

2.5.1 Causes of software change

Changes to software can be introduced during the whole software engineering process. Bohner provides four basic categories: *changes during requirements*, *changes during design*, *changes during implementation*, and *changes during test*. Examples of causes of software change are given in table 2.1.

2.5.2 Semantics in Change Impact Analysis

In the same paper, Bohner states the need for the use of semantics when performing change impact analysis. An example is given when only dependency relations are used. A graph is used to represent the structure of a software system. Nodes represent parts of the software system, directly edges are dependency relations between the nodes. Using the structural information, a reachability graph can be built and used for change impact analysis. This reachability graph can be built using Warshall's algorithm[War62]. However, when no ad-

ditional semantics are used, the number of impacted parts explodes, as every part of the system will be impacted by the other parts, and, recursively, will be impacted again.

Bohner states that there are three challenges in change impact analysis: *information source volume*, *change semantics*, and *analysis methods*. The first challenge can be addressed by the use of automated assistance to lower the time needed for analysis and improve the error rate. The last challenge can be address by change impact analysis methods which can be classified in the groups: *semantically guided*, *heuristically guided*, *stochastically guided*, *hybrid guidance*, and *unguided/exhaustive*.

In [GKvdB08], static semantics are used to perform change impact analysis in requirements models. Four formalized types of relations between requirements are given: *requires relation*, *refines relation*, *contains relation*, and *conflicts relation*. Based on these formalized relations, impact rules are defined to identify actual impacted requirements, and candidate impacted requirements. The paper classifies two types of changes in the requirements model: *changes to the requirement entities*, and *changes in the requirements relations*. Based on this classification, several rules are given to identify the actual and candidate impacted requirements.

Change impact analysis can be performed at the different artifacts involved with software engineering, such as requirements and architectures. For example, in [FM06] a formalism is given to describe components, their interfaces, and the relationships among the components and interfaces. Based on the composition of the components, a model is derived which describes the dynamic interaction among the components. A taxonomy for changes is introduced. Roughly, this taxonomy discriminates between atomic and composite changes as well as changes between and within components. Atomic changes consist of changes such as adding or removing interfaces to a component and adding/removing methods to existing interfaces. The two proposed concepts are put together to be able to perform dynamic change impact analysis.

In [RST+04], a formalism is given in which atomic changes to Java source code are defined, such as *added class* (AC), *deleted class* (DC), *added method* (AM) and the like. Adding a method might affect the dynamic dispatch in a program. As such, to be able to reason about changes which affect dynamic dispatch (LC), functions are defined which are used to look up the dynamic dispatches. Using these atomic changes, syntactic dependency graphs for changes in source code can be calculated. The dependency graphs can, in turn, be used to identify changes to retain a syntactically correct program.

Impacted tests are identified by use of a call graph for each test, for the original and the changed program. The call graphs can be built statically, through source code analysis, or dynamically, through runtime analysis. These call graphs are compared to identify tests which are impacted by the changes made in the program.

2.6 Conclusion

In this chapter we have given the basic concepts needed to understand the content of the thesis. In section 2.2, Model Driven Engineering and the Model Driven Architecture are introduced. It aims at improving the software develop-

ment process by introducing the modeling concept. Modeling aims at creating models which abstract away unneeded details, such that one can better reason about the details which do matter. Meta-modeling frameworks are introduced to support the modeling process.

In section 2.4, the definition of a software architectures is given, as specified in the IEEE-1471 standard. Software architectures are described by multiple views. Using a single view to describe the whole software architecture would make the software architecture incomprehensible, given the amount of detail available. Examples of different views are given. Each view of the architecture conforms to a specific viewpoint. A viewpoint contains its own vocabulary (meta-model) to describe each view, and a set of rules (style) to which the architecture must conform. Different Architecture Description Languages (ADLs) exist to record the software architecture. Only a handful of the older ADLs have evolved and are still in use. Most of the current ADLs aim at being extensible, such that the architect can introduce his own semantics.

Finally, section 2.5 introduces the concepts of change impact analysis. Change impact analysis is the process of finding impacted elements, based on an initial change. Change impact analysis can be performed at different artifacts, such as a requirements model, or source code. There is a clear need for semantics in change impact analysis. Without semantics, it is hard to determine the actual impacted set of elements. Only an overestimation of impacted elements can be safely made when the semantics are lacking.

Chapter 3

Approach

3.1 Introduction

This chapter elaborates the approach found in section 1.3. Section 3.2 we elaborate on the validation of behavior in architectures. In section 3.3 we give the process to perform change impact analysis. Section 3.4 gives a short overview of the case study. In section 3.5 we elaborate on the supporting framework to perform change impact analysis. Finally, this chapter is concluded in section 3.6.

3.2 Validation of requirements

Validating the functional requirements of the architecture is one key activity in our approach to perform change impact analysis. The functional requirements represent behavior which the architecture must satisfy. The informal functional requirements are reformulated as formal behavior descriptions. The behavior of the architecture is derived from the architecture by simulating the architecture. The formal behavior descriptions are tested against the derived behavior, in order to validate the presence (or absence) of behavior. A graphical overview of this is given in figure 3.1. In this figure, boxes represent artifacts, arrows represent general relations between the artifacts.

We define the *problem domain* as the domain in which the problem lives

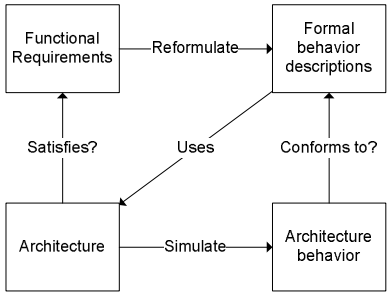


Figure 3.1: General approach for requirements validation

which needs to be solved. The *solution domain* is the system (architecture), which is created to solve the problem as defined in the problem domain. The IEEE Standard Glossary of Software Engineering Terminology[IEE90] defines "validation" as:

> The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

We define the validation of a software architecture as:

> Testing whether the software architecture satisfies its functional requirements.

The following list gives a description of the artifacts found in the figure:

Functional requirements The functional requirements of the system to be built. This is a collection of functional requirements specified in a natural language. Given that the requirements are given in natural language, the requirements cannot be interpreted directly by a program/algorithm. The requirements are specified in the problem domain.

Formal behavior description The formal description of the required behavior of the architecture. Given that the informal requirements are not interpretable by a program, the informal requirements need to be formalized such that these become interpretable. The formal behavior description is in the solution domain, and based on the informal functional requirements.

Architecture The architecture of the system to be built. This is an architecture specified in an architecture description language, possibly annotated with additional properties to increase the level of detail. The architecture is in the solution domain, and driven by the requirements.

Architecture behavior The behavior of the architecture. The architecture itself incorporates behavior. This behavior can be expressed in multiple ways. This artifact represents the behavior of the architecture. The architecture behavior is in the solution domain.

The artifacts found in figure 3.1 are related to each other. The relations are given by the arrows between the artifacts. The description of the relations is as follows:

Reformulate The informal requirements are reformulated into formal behavior descriptions. This also includes a translation from the problem domain to the solution domain. Given that the formal behavior description is based on the architecture, the architecture itself is used.

Satisfies? The satisfies relation is the validation of the functional requirements against the architecture. I.e., if the architecture supports the behavior specified by the functional requirements. Given that the requirements are informal descriptions, we have to manually validate the architecture against the requirements. Also, the requirements are in the problem domain, while the architecture is in the solution domain.

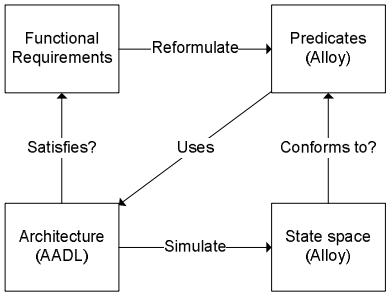


Figure 3.2: AADL/Alloy specific approach

Conforms to? The conforms-to relation represents the validation of the architecture behavior against the formal behavior descriptions. I.e., the architecture behavior has to conform to the behavior described by the formal behavior descriptions. Both the formal behavior description and the architecture behavior are in the solution domain.

Uses The formal behavior description states facts about the behavior of the architecture. The architecture behavior states facts over the components and other elements found in the architecture. The architecture behavior itself is derived from the architecture. Therefore, the formal behavior description is specific for the architecture itself.

Simulate To derive the architecture behavior from the architecture itself, the architecture is simulated. The architecture behavior, derived through simulation of the architecture, is stored in the architecture behavior artifact.

We use AADL to record architectures. AADL is an architecture description language. It provides a toolset, OSATE, which we use to convert a textual specification into an architecture model. Additional properties which describe the behavior of the individual components, and invariants are added to the architecture. More information about AADL is given in section 4.2.

To simulate the architecture, we use Alloy. The simulation is a discrete event simulation. This form of simulation introduces a state space. This state space is explored using Alloy. An overview of Alloy is given in section 4.5. Also, given that the behavior of the architecture is expressed as an Alloy model, Alloy predicates are used to represent the formal behavior descriptions. Figure 3.2 introduces the uses of AADL and Alloy.

Three key activities are shown in figure 3.1. The key activities are: *simulating the architecture*, *formalization of informal functional requirements*, and *validating functional requirements*. The following subsections elaborate on each individual key activity.

3.2.1 Simulation of the architecture

The simulation of the architecture is a discrete event simulation. Discrete event simulation introduces the notion of events, states, and state space. Events are actions which occur within the architecture. A state describes the loci of data values within the architecture. Two states are possibly connected by a transition.

All states are captured by the state space. The behavior of the architecture is expressed as a sequence of events. Alloy is used to simulate the architecture. The simulation is fully automatic and does not need any interaction during the simulation itself. A more elaborate description of the simulation of the architecture is given in chapter 5.

As Alloy is a declarative language, only pre- and post-conditions are required to describe the behavior of the components. The Alloy Analyzer tries to find the state space where the pre- and post-conditions are satisfied for each subprogram. An example of the behavior a component can contain is that it performs a computation based on a received data value, and sends the results to another component. The architecture is also subject to invariants. For example, an invariant might state that a data instance can only hold data values of a specific type. These invariants are expressed as Alloy predicates.

We introduce scenarios to capture the detection of the occurrence of an observed event in either the environment or architecture. A scenario describes the first state after the detection of an environment or architecture event. From the first state, the simulator derives the other states of the state space. An example of an environment event is that a button is pushed. A sensor in the architecture is triggered and an event is generated. Likewise, an example of an architecture event is the expiration of a timer. In this case, the timer generates an event. Guidelines to create scenarios are given in chapter 5.

3.2.2 Formalization of informal functional requirements

The informal functional requirements are not interpretable by a program. To be able to automatically validate the behavior of the architecture against the requirements, we need to reformulate the informal functional requirements such that these are interpretable by a program. We use Alloy to find the state space which represents the behavior of the architecture. Also, we use Alloy to validate the behavior of the architecture against the formalized requirements. Therefore, we need to express the informal functional requirements as Alloy predicates. The predicates are used to test the state space for the presence, or absence, of required behavior.

Also, the informal functional requirements are in the problem domain. The predicates are in the solution domain. This translation from problem domain to solution domain is manual. The predicates representing the informal functional requirements are closely related to the architecture. For example, a predicate can explicitly state the presence of a data value at a port found in the architecture. As such, the architecture is used when specifying the predicates.

Chapter 6 elaborates on the formalization of informal functional requirements. Guidelines are given for the translation of informal functional requirements to Alloy predicates. Also, the method to test requirements against the state space is shown.

3.2.3 Validation of functional requirements

The validation of the functional requirements is represented by the *conforms to?*-relation in figure 3.3. The state space describes the behavior of architecture. Predicates are used to describe the required functionality of the architecture, expressed as formulas over the state space. Scenarios are extended to test a

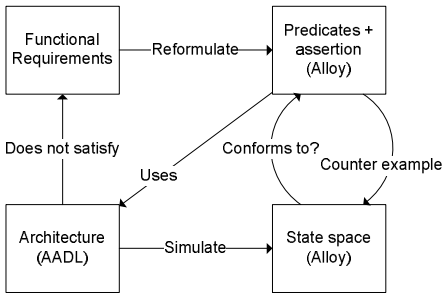


Figure 3.3: AADL/Alloy specific approach with counter example

number of requirements. These requirements are bound to the scenario. I.e., the requirements are bound to the detection of an event in the environment or architecture.

We use an assertion to test that, after the detection of an environment or architecture event, a number of requirement predicates hold. If one or more of the predicates do not hold for the simulation, the state space is returned as a counter example by the Alloy Analyzer. This is described by the *Counter example*-relation in figure 3.3, from the *Assertion*-artifact to the *State space*-artifact. To find the unsatisfied requirement, we use the Alloy Analyzer to test individual requirement predicates against the counter example.

Architecture invariants are tested using the same construct. All architecture invariants need to hold in each scenario at all times. I.e., for all the scenarios, each invariant must hold for every state of the state space for that scenario. If an architecture invariant does not hold, the state space is returned as a counter example by the Alloy Analyzer. To find the violated invariant, we use the Alloy Analyzer to test individual invariant predicates against the counter example.

Chapter 6 elaborates on the validation of functional requirements and invariants in software architectures.

3.3 Performing change impact analysis

When performing change impact analysis, either the original requirements artifact is changed, or the original architecture is changed. This is shown in figure 3.4 and figure 3.5, respectively.

In figure 3.4, the requirements are changed. The requirements specify functionality, which is either added, updated, or removed. To be able to check the new functionality, the changed requirements have to be (re)formulated as assertions.

These changed requirements are validated against the generated state space. If one or more of the changed requirements fail, a counter example is given. This shows us that the old architecture does not satisfy the evolved requirements and the architecture has to be changed in order to satisfy the new requirements. As the requirements are formulated as Alloy predicates, we use Alloy to query individual requirements against the counter example and find the requirements which are not satisfied. Since functional requirements specify functionality, we

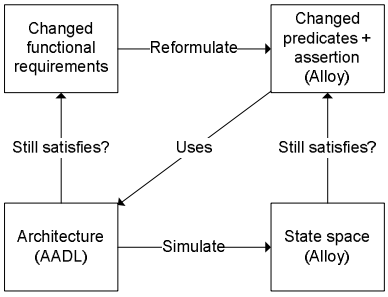


Figure 3.4: Change impact analysis after evolved requirements

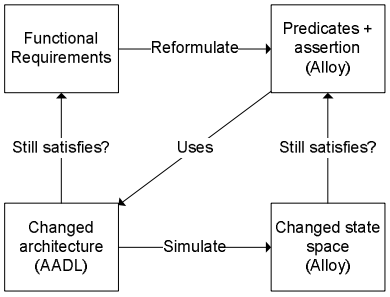


Figure 3.5: Change impact analysis after evolved architecture

find which part of the required functionality is not (correctly) implemented in the architecture.

Figure 3.5 describes the situation when the architecture itself is changed. To restrict the scope of this thesis, we do not consider this case.

The process of performing change impact analysis is shown in figure 3.6. The activity diagram shows the activities for performing change impact analysis in architectures. In this figure, change is defined as the addition, update, or deletion of an entity. The description of the activities shown in the activity diagrams is as follows:

Change requirements The informal functional requirements are changed. The new/updated informal requirements describe the new/update required behavior of the architecture.

Change architecture The architecture is changed by the architect. The architect modifies a part of the architecture. Examples of changes are adding new components, altering existing components, or deleting existing components.

Change requirement predicates The changed informal functional requirements result in changed requirement predicates. Dependent on the changes to the architecture, existing requirement predicates might need to be updated to reflect the changes in the architecture. New functional requirements result in adding requirement predicates. Deleted requirements results in removing requirement predicates.

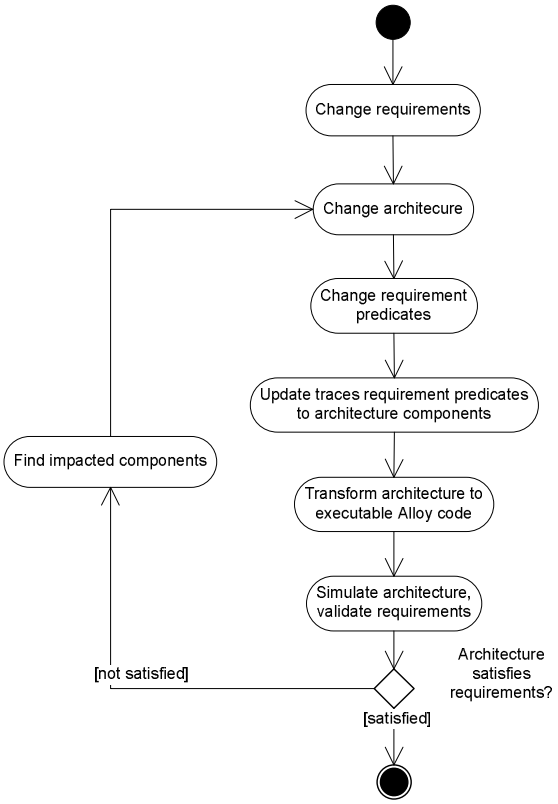


Figure 3.6: Activity diagram for performing change impact analysis

Update traces requirement predicates to architecture components The traces from the changed requirement predicates to architecture components are updated. Updated requirement predicates might impact other components, after the changes.

Transform architecture to executable Alloy code The architecture is transformed to executable Alloy code. This Alloy code is used to simulate the architecture and validate the requirements. The transformation from AADL architectures to Alloy code is automatic. The supporting framework for the transformation is given in section 3.5.

Simulate architecture, validate requirement predicates The architecture is simulated to derive the behavior of the architecture. This simulation is fully automatic; no interaction during the simulation itself is required. Chapter 5 elaborates on the simulation of the architectures. Behavior of the architecture is validated against the requirement predicates. Chapter 6 elaborates on the validation of requirements.

Find impacted components In case one or more formalized functional requirements are not satisfied by the architecture, we try to find the impacted components by use of traces from requirement predicates to components in the architecture. Chapter 6 elaborates on finding impacted components, based on the counter example.

The process is iterative. It is possible that a single change to the architecture does not result in satisfaction of its requirements. Therefore, the process might have to be performed multiple times until the architecture satisfies its requirements again. This is shown by the control flow from the last branch back to the *change architecture* activity.

3.4 Evaluation of approach

We evaluated the proposed approach by performing a case study. This case study applies the process found in section 3.3 to the Remote Patient Monitoring project. It was a project at Atos Origin Technical Automation. The requirements and architecture used in this thesis are a subset of the real Remote Patient Monitoring system. The case study evaluates five change scenarios. The results of the case study are found in chapter 7.

The stakeholders of the system are the following: patients, doctors, system administrator. The patients are the stakeholders who are being monitored. The doctors are the stakeholders who keep track of the patients' situation. Finally, the system administrator is the stakeholder who administers the system.

The Remote Patient Monitoring system has several main goals. The first goal is the ability to monitor patients' temperature. Periodically, the system has to perform a temperature measurement at a patient. The patient carries a device which does this. Each temperature measurement is transferred to a central system which stores the temperature measurement. The requirements R1 to R3 found in table B.1 of chapter B describe this.

The doctors are able to review the temperature measurements performed by the system. They can access the temperature measurements through a web-interface. The web-interface provides graphs and tables with the temperature

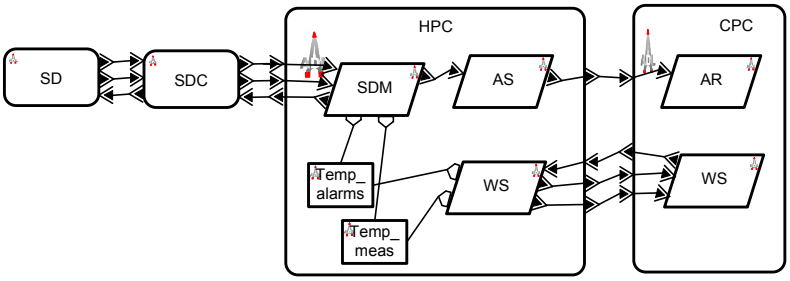


Figure 3.7: Overview of the RPM architecture

measurements from a patient. This is described by the requirements R15 and R16 in table B.1 of chapter B.

Also, the doctors are able to set a temperature threshold for a patient. If this threshold is exceeded, the system triggers an alarm which is received by the doctor at his alarm receiving device (a program running at his computer). The ability to set a temperature threshold for a patient is specified by requirements R9 to R11, found in table B.1 of chapter B. Requirements R4 to R8 found in table B.1 of chapter B specify the ability to generate alarms and send them to the doctors' computer. The doctors are also able to review the temperature alarms of a patient.

The system uses ZigBee for wireless communication between the device carried by the patient and the central system. ZigBee is a wireless protocol meant for devices which must be able to run on a single battery for a long time span. It is often used in sensor networks with multiple nodes and the required maintenance to those nodes has to be as low as possible. Requirements R12 and R13, table B.1 in chapter B, specify the use of ZigBee and a dedicated central node which performs the task of network coordination.

The architecture derived from the requirements is given in figure 3.7. A full specification in AADL of the architecture is given in listing B.1 of chapter B. For brevity and readability, the figure does not give the full refinement of all components. The architecture consists of several components.

The *SD* component is the device which is carried by the patient. It performs temperature measurements at a regular interval. If required, it sends the temperature measurement to the *HPC* component through the *SDC* component. If a temperature threshold is exceeded, the *SD* components sends an alarm to the *SDC* component, which forwards it to the *HPC* component.

The SDC component is the ZigBee network coordinator. It forwards the temperature measurements and temperature alarms from the *SD* component to the *HPC* component. Furthermore, it forwards temperature thresholds from the *HPC* component to the *SD* component. The details of the real coordinating task are omitted in the architecture description.

The *HPC* (Host PC) component consists of several subcomponents. It contains the *SDM* (Sensor Device Manager) components which interfaces with the *SDC* component. If the *SDM* component received a temperature measurement it stores it in the *temperature_ measurements* database. If it receives an alarm, the alarm is stored in the *temperature_ alarms* database, and forwarded to the *AS* component (Alarm Service). The HPC component also contains the WS

(Web Server) component, which serves the web-interfaces for the doctors. The *AS* component, contained by the *HPC* component, forwards the alarms it receives to the *CPC* component.

The *CPC* (Client PC) component is the system which the doctors use to monitor the patients. The CPC contains two subcomponents. The first subcomponent is the *WC* (Web Client). This is a normal web browser and is used to access the web-interface provided by the *WS* component, found in the *HPC* component. The second subcomponent is the *AR* (Alarm Receiver). As a web-interface does not provide a means for instant data updates pushed by the web server, a separate program is required. The *AR* component is a stand-alone program which receives the requirements and notifies the doctors.

From this case study, we identify the strengths and weaknesses of the approach. After each change scenario, an evaluation is given and results are discussed. Chapter 9 gives the final conclusion based on the case study as well as the other chapters.

3.5 Supporting framework

In order to automate the simulation of the architecture and validation of the requirements, we build a supporting framework. This framework consists of a transformation from an AADL instance model to a simulatable model of the architecture in Alloy. The activity diagram of the creation of the support framework is given in figure 3.8. It consists of the following activities:

Define structural semantics in Alloy The AADL meta-model expressed in ECore[EMF] and the AADL standard[SAE09] are used for this activity. The AADL meta-model describes the allowed structure of the architectures. Where the AADL meta-model is lacking, we use the AADL standard to further formalize the architecture. The result of this activity is an Alloy module which defines the AADL meta-model. The simulatable models of the architecture are based on this Alloy module. Section 5.2 gives the results of this activity.

Define behavioral semantics in Alloy The simulation is based on the behavioral semantics of AADL. The AADL standard[SAE09] serves as a basis for the definition of the behavioral semantics. Where the AADL standard is lacking, we introduce self-defined semantics. The result of this activity is the basis for discrete event simulation of AADL models in Alloy. Section 5.3 gives the results of this activity.

Build transformation from AADL to Alloy We build a model-to-text transformation from AADL instance models to Alloy code. The structural and behavioral semantics are used as a basis for the transformation. The output of this activity is a transformation which transforms AADL instance models into simulatable Alloy models. We use XPand2 as a transformation engine, which is part of the openArchitectureWare[ope] platform.

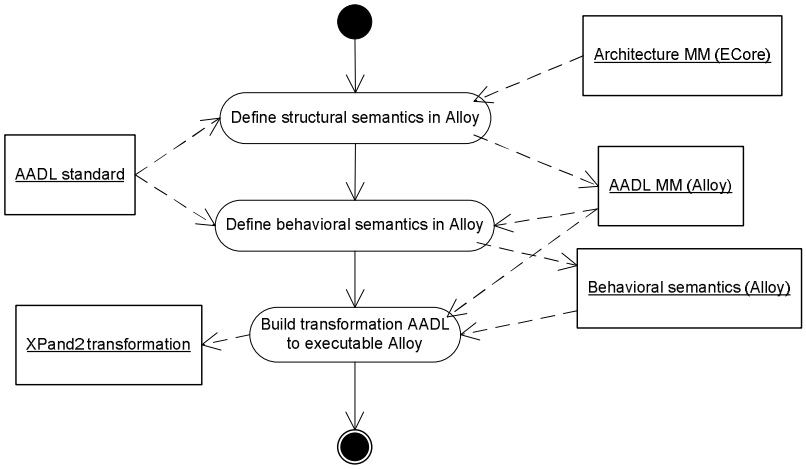


Figure 3.8: Activity diagram for building supporting framework

3.6 Conclusion

In this chapter we give an overview of the approach for this research project. Section 3.2 describes the validation of requirements against the architecture. To be able to do this, three key activities have been identified. The first key activity is formalizing the requirements into formal behavior descriptions. The second key activity is the simulation of the architecture in order to derive its behavior, which results in a state space. The third key activity is the validation of the presence (or absence) of the formal behavior descriptions (requirements) in the state space.

In section 3.3 we described the process of performing change impact analysis, through an activity diagram. The activities include changing the requirements, changing the architecture, changing the formal behavior descriptions, simulating the architecture, validating the formal behavior descriptions, and, in case of a counter example, finding the impacted elements by use of the counter example.

Section 3.5 describes the supporting framework which we use to simulate the architecture. The supporting framework includes the definition of the structural and behavioral semantics of AADL in Alloy, and a model-to-text transformation to automatically convert AADL instance models to an Alloy model. Using the supporting framework, we can easily and quickly take an AADL specification and automatically convert it into a simulatable model.

Chapter 4

Introducing AADL and Alloy

4.1 Introduction

This chapter will introduce the architecture description language AADL[Fei] and the model finder/checker Alloy Analyzer[MIT]. AADL and Alloy are introduced to provide a better understanding the concepts found in the later chapter.

Section 4.2 will give an introduction to the basic concepts AADL. The rationale for the choice of AADL is given section 4.3. An example architecture, which is used throughout the thesis, is given in section 4.4. In section 4.5, we give a short overview of the basic concepts of Alloy and the Alloy Analyzer. The rationale for the choice of Alloy is given in section 4.6. Finally, this chapter is concluded by section 4.7.

4.2 AADL

AADL is an architecture description language developed by the Software Engineering Institute (SEI) at the Carnegie Mellon University (CMU). The Society of Automotive Engineers (SAE) released the aerospace standard AS5506[SAE09] in November 2004, which describes the first version of AADL. In January 2009, the SAE released the standard AS5506A[SAE09], which is the second version of the AADL. The document [FGH06] provides an introduction into AADL, based on the first standard.

4.2.1 Language abstractions

Figure 4.1 shows the core language concepts of AADL. Components are defined by component types and component implementation declarations. The externally visible interface and observable attributes are defined by the component type. The internal structure of the components is defined by the component implementation. The internal structure is composed of sub components, connection, call sequences, modes, flows, and properties.

Three categories exist for components: Software, execution platform, and composite components. The first group is composed of the components: Data, subprogram, thread, thread group, and process. The execution platform group

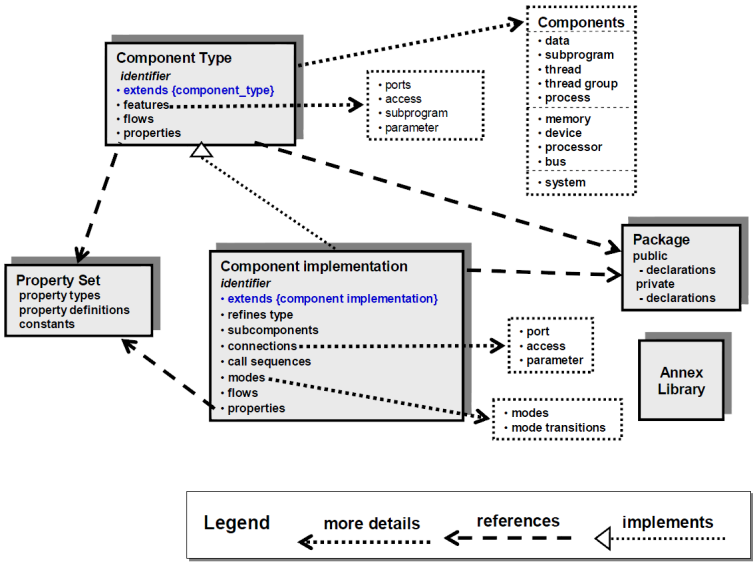


Figure 4.1: AADL core language concepts[FGH06]

consists of the components: Memory, device, processor, and bus. Finally, the composite components group is defined by the component System.

Property Sets are used to annotate the system with additional data through one or more properties (key-value pairs). An Annex Library can be used when the property sets are insufficient. Sublanguages can be introduced through an Annex Library.

AADL provides both a textual and a graphical notation for the specification of architectures. An overview of the components of the graphical notation is given in figure 4.2. An example architecture specified using the textual notation can be found in appendix B.

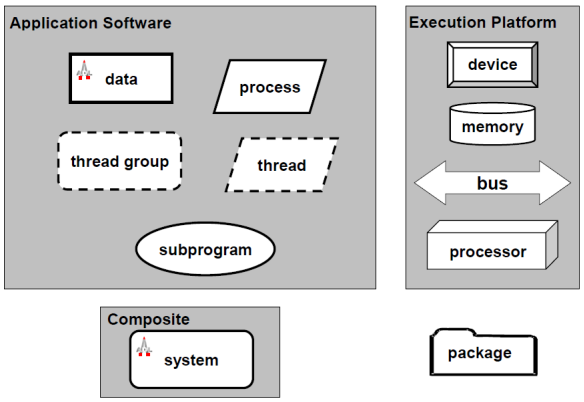


Figure 4.2: AADL graphical notation of components[FGH06]

4.2.2 Software components

AADL provides the following software components abstractions:

Process Processes are used to represent a protected address space. This process space contains: Executable binary image; Executable binary images from its subcomponents; Server subprograms and data which are referenced by external components.

Thread The thread component is used to represent a concurrent schedulable unit of sequential execution. Multiple threads can be used to represent concurrent execution within the architecture. Threads can be dispatched periodically, or when events are received at event, or event data ports. A thread implementation can contain data, and subprogram subcomponents.

Thread group Thread groups are used to logically organize threads, data, and thread groups within a process. The thread groups are used to provide a means for the separation of concerns. Thread group implementations are allowed to contain data, thread, and thread group subcomponents.

Data The data component is used in multiple ways. Firstly, it is used to define application data types. Secondly, data components are used to define the substructure of data types via data implementation subcomponents. Finally, a data component can be used to represent a data instance within the architecture. A data component implementation is only allowed to have data subcomponents. A data type is used to define the data associated with ports and parameters. Strong typing of elements allows for verification of the architecture. Data subcomponents can be used to define the substructure of a data type. A data instance is represented by the use of a data subcomponent within another component implementation.

Subprogram Finally, the subprogram component is used to represent sequentially executable source text. Subprograms are used to model: A method call for operation on data; Basic programs calls and call sequences; Remote service/procedure calls.

4.2.3 Execution platform components

The following execution platform components are defined in AADL: Processor, Memory, Bus, and Device. These components are used to represent computational and interfacing resources within a system. Software components are mapped onto execution platform components within a system. Using this mapping, the locations of the execution of code and the storage of data are specified in the system.

Processor The processor component is used to represent an abstraction of hardware and software responsible for the scheduling and executing software components. An example of a processor can be a server which runs Linux as its operating system. A processor component implementation can have memory subcomponents.

Memory A memory component represents memory such as RAM, ROM, but also storage such as hard disks.

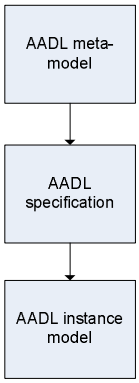


Figure 4.3: Conceptual AADL hierarchy

Bus Busses are used to represent hardware and an associated communication protocol. A bus enables interaction between different execution platform components. For example, two threads each bound to a different processor required to communicate with each other, a bus is used to provide the means for communication between the threads. A bus cannot have any subcomponents.

Device A device is used to represent entities that interface with the external environment, such as sensors and actuators. Devices are often complex elements, but do not need to be modeled as such in an architecture. Device component implementations cannot have any subcomponents.

4.2.4 System structure and instantiation

The system component abstraction is used to represent a composite of the software, execution platform components, or system components. Systems can be used to represent a complex system of systems, or the integrated software and hardware components of a dedicated application system. A complete system specification includes the binding of software components to execution platform components. For example, a process component must be bound to a memory component and a thread component must be bound to a processor component.

From the architecture specification, a system instance is generated. The architecture specification conforms to the AADL meta-model. This system instance represents the run-time architecture of a physical system. The system instance is created by instantiating the top-level system instance, and then recursively instantiates the contained subcomponents, and their subcomponents, and so on. The architecture instance model conforms to the architecture specification. A graphical overview of this hierarchy is given in figure 4.3.

4.2.5 Component interaction

A port connection represents an explicit relationship declared between two ports, or two port groups which allows for directional exchange of data and events between components. The directions of ports can be either incoming, outgoing, or both incoming and outgoing.

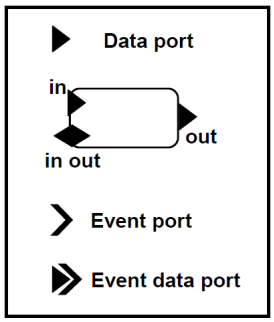


Figure 4.4: AADL graphical notation of ports[FGH06]

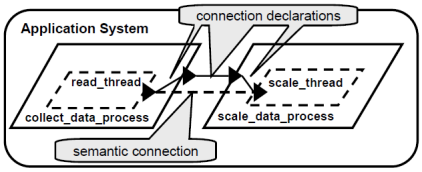


Figure 4.5: Semantic connection between thread instances[FGH06]

Three different types of ports are specified by the AADL. The different ports are shown in figure 4.4. The following port types are available:

Data port The data port is used for (typed) data transmission, and does not contain a queue.

Event port An event port is a port which is used for the communication of events. Events can be raised by subprograms, thread, processors, or devices. An event port can provide a queue which can store multiple events. An example of an event is an alarm which is triggered by a device.

Event data port The event data port is used for transmission of events with data, also called **messages**. Event data ports can be queues. An example where an event data port can be used is when sensors are used to trigger an alarm, annotated with a concrete value when a threshold is violated.

In case of a fully specified system instance models, semantic port connections represent direct communication channels between two thread instances, a thread instance and a processor instance, or a thread instance and device instance. The port connections declarations are abstracted in favor of the semantic port connections. Figure 4.5 shows several port connection declarations and the semantic port connection between two threads.

The subcomponents data and bus are made accessible through the use of component access features. Data subcomponents can be accessed by other components through the use of a data access feature. For example, a subprogram can be given access to a data instance contained within another component through data access features and data access connections. The access direction of a component access feature is either provides or requires. Figure 4.6 gives a graphical example where a data instance is used by several threads. The data access connections are annotated with properties.

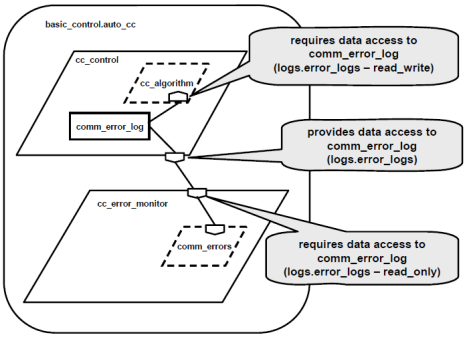


Figure 4.6: Data access example[FGH06]

Busses can be made accessible through a similar construct, called bus access features and bus access connections. The access direction of a component access feature is either provides or requires.

Subprograms are called by the use of a call sequences within a thread or subprogram implementation. A call sequence calls one or more subprograms.

Subprograms can exchange with other components through parameters, and data access. Parameters represent the call and return data from a subprogram call. A subprogram can have one or multiple parameters. Parameters are connected to other parameters or ports through the use of parameter connections. The parameter connections are similar to the port connections.

Subprograms can also have data access features which are used to specify access to data instances.

4.2.6 Modes

In AADL, modes are used to represent different run time configurations for components. In a cruise control system, for example, the modes can be initializing, inactive, and active. Each of these modes specifies a different set of active threads and connections among those threads. Mode switches are driven by events received at event ports.

A state within a state machine is used to represent a mode. The states in the state machine represent the different configurations of the mode-specification. A model-specification must at least include two states, where one state is declared as the initial state.

4.2.7 Flows

Flow specifications in AADL give a detailed description of the data flow path through a component. A flow starts at a source and ends at a sink, flowing through subcomponents.

A port or parameter can be declared as a flow source. Similarly, a port or parameter can be declared as a flow sink. A flow path is used to represent a data flow from one feature of a component to another feature of the same component. The specification of data flows can be used for analysis of the architecture.

4.2.8 Properties and annexes

Properties sets are used to annotate the architecture with project-specific information. A property set consists of one or more properties, which are key-value pairs. Elements annotated with properties can be used by tools for analysis and verification, as well as additional specification of elements.

An annex is a more elaborate version of the property. An annex is used to introduce a sublanguage into AADL. For example, a formal language can be used to describe behavior of a component. Tools can use this behavior description for deeper analysis to verify correctness of the architecture. Examples of three different annexes that are provided by the SEI are the behavioral annex, the error model annex, and the programming language API annex. The behavioral annex allows for a more elaborate description of the behavior of components found in an architecture. The error model annex is used to declare error models for components. The programming language API annex defines language-specific rules for source texts to be compliant with the architecture.

4.3 Rationale for AADL

This section gives the rationale for the choice of AADL as the language to record architectures. Several criteria were specified in order to find the ADL best suited to our needs:

- The architecture description language has to supply behavioral semantics of the architecture elements;
- The architecture description language has to support annotation of the architecture elements;
- The architecture description language must have an actively supported toolset.

During the literature research, the architecture description languages AADL[Fei], xADL2.0[DvdHT01a, DvdHT01b], and ACME/Armani[GMW97, Mon01] were investigated. Older languages, such as C2SADL, Wright, and Darwin, are investigated, but not thoroughly as these seem to be no longer in use and do not have an actively supported toolset.

From the literature research, we chose AADL as it provides the largest set of pre-defined behavioral semantics. In contrast to the other languages, specialized components types are given, such as Process, Thread, and Subprogram. This differentiation gives us a richer set of semantics compared to the other languages. In other aspects, it is also more elaborate, such as the different port and connection types. Also, it is standardized by the SEI, as AS5506A/1[SAE09]. The architecture in AADL can be annotated by properties. When properties are lacking, AADL can be even further annotated/extended through the uses of annexes.

The language xADL2.0 is a language which only provides the basic concepts of Component, Connector, Interfaces, and Configuration. The language itself does not specify any semantics for these elements, but rather lets the architect specify these. The language is meant for extension by altering/adding XML schemas. This means that new elements can be added, all with their own

properties as described in the XML schemas. In our case, given that it does not provide any behavioral semantics, it is not very usable for us. The language itself is very extendable, however.

ACME/Armani is also a language which primarily provides the basic concepts found in architectures, based on the component & connector view: Components, Connectors, and Systems/Configuration. As with xADL2.0, it is meant to be extended by the architect using custom properties. But in contract to xADL2.0, where entirely new elements can be added, the existing elements can be annotated with key-value pairs. ACME/Armani does provide a (limited) semantic framework, but this framework can only be used to reason about structure.

The older architectural languages, such as Wright[AG96] and Darwin[KM85], also provide only the basic concepts of architecture: Components, Connectors, and Configurations. Although some of the languages are very formal, these languages tend to focus on one domain. A comparison of these languages is given in [MT00].

The UniCon architecture description language[Zel96], a first generation language, provides specialized components. Specialized components include Computation, Filter, and Process. These specialized components have more precise semantics, compared to the abstract components. However, this language appears to be no longer in use and the supported toolset is no longer available.

4.3.1 Supported AADL subset

To limit the scope of the research, only a subset of all the AADL elements are supported. In addition, certain concepts are simplified or omitted for the same reason. For example, AADL supports two generic types of components, the execution platform (hardware) components and software components. The execution platform components and its related features are not considered, except for the system component. Software components are the main target in this prototype. Elements which are not considered as core concepts, but provide grouping mechanisms are also removed. Furthermore, some features such as event ports are omitted.

The AADL provides a divergence between component types and component implementations. A component type specifies its interface and attributes which are externally visible, while a component implementation defines the components inner structure - similar to the black box/white box approach. These two concepts have been merged into a single concept, as this separation does not provide any gain for this prototype. The same divergence also comes forward in the flow concept, and is merged as well.

The supported elements can be found in table 4.1.

Certain limitations are also imposed on the AADL models under consideration. The constraints on the source models are as follows:

- All components must be refined to the level of subprograms;
- All ports and data accesses are typed;
- Only the elements found in 4.1 are supported;
- Modes are not supported.

Category	Elements
Components	System, data instance, process, thread, subprogram
Features	Data port, event port, event data port, parameter, data access
Connections	Data port connection, event data port connection, parameter connection, data access connection
Call sequence	Subprogram call

Table 4.1: Supported AADL concepts

The components must be refined to the level of subprograms. In our approach, the subprograms are the components for which we define the behavior. The system, process, and thread components are not annotated with additional behavioral properties. We do not support modes because we do not cover dynamic configuration of software architectures in this thesis.

4.4 Remote patient monitoring

Throughout the following chapters, an example architecture is used to clarify different concepts. This section gives the example architecture. The example architecture is a small part of the architecture given in appendix B. The example architecture implements the first three requirements, also found in appendix B.

Figure 4.7 gives the graphical representation of the architecture. The example architecture consists of three subsystems called *SD*, *SDC*, and *HPC*. The *SD* represents a device which is carried by a patient. This device performs a temperature measurement on regular intervals and sends the temperature measurement to the *SDC*. The *SDC* is a device which supports all the communication between the *SD* and *HPC*. Although not visible in the architecture presented here, it performs some network supporting tasks. The temperature measurement received by the *SDC* from the *SD* is forwarded to the *HPC*. The *HPC* system represents a central system which serves as the central system which performs several tasks, such as storing the measurements. The *HPC* system contains a process called *SDM*. This process receives the temperature measurement, and stores the temperature measurement in the data instance which it has access to, as shown in the figure. All the processes in the architecture are refined to the level of subprogram. Unless otherwise noted, the default name for a process, thread, and subprogram are *proc1*, *thread1*, *c1*, respectively.

A naming scheme is used to identify the different components in the architecture. For example, the device *Sensor1*, as shown in figure 4.7, is referred to as: *SD.Sensor1*. *SD* is the parent of the device, and *Sensor1* is the name of the device itself. Another example is *hpc.sdm.thread1.c1*, which is the full name for the subprogram (indirectly) contained by the *HPC* system. Features, such as event data ports, are also identified by this naming scheme. For example, the thread *sdc.proc1.thread1* contains the two event data ports called *sdc.proc1.thread1.measurement_in* and *sdc.proc1.thread1.measurement_out*.

The systems *SD*, *SDC*, and *HPC*, are also contained by a system, called

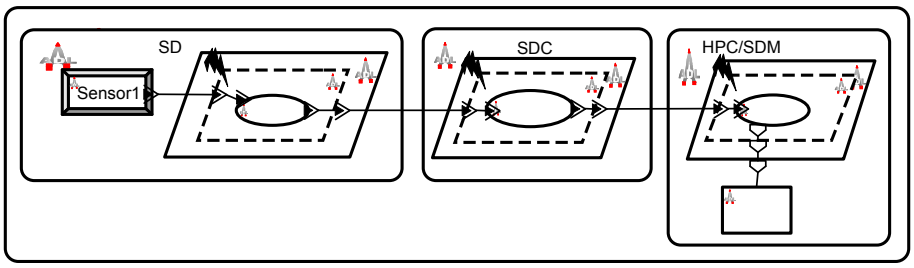


Figure 4.7: Example architecture in AADL

RPM, but the *RPM* part of the identifiers is omitted here.

4.5 Alloy

Alloy is a modeling language which can be used to model one or more aspects of the system. Alloy is based on first order logic and is a declarative language. This means that instead of specifying behavior using operational or imperative semantics (how to accomplish a task), pre- and post conditions are specified to detect a task has happened. Being a declarative language, it allows for smaller and more concise models. Alloy is comparable to Z[Spi87], OCL[OCL06], and VDM[Jon90].

Alloy is both a model finder and a model checker. Given a specification and constraints, it finds models. In turn, the found models can then be checked against formulas. If the model does not conform to the given formulas, a counter example is returned. The examples presented below are based on [MIT, All09].

For this section, a separation between modeling structure, modeling behavior, and validating behavior is made. Subsections 4.5.1 and 4.5.2 are generally used to model the structure of a model. Subsection 4.5.3 and 4.5.4 are used to model the behavior of the model. Sections 4.5.4, 4.5.5, and 4.5.6 are used to validate the behavior of a model.

4.5.1 Signatures and fields

Throughout this section, we will use an example Alloy model which is also used in the Tutorial for Alloy Analyzer 4.0[All09]. The model describes a file system, which consists of files, directories, and one root directory. The Alloy model of the file system is given in listing 4.1.

An Alloy model is described using paragraphs. Each paragraph describes a part of the model. A *signature* paragraph describes a basic type, together with a number of *fields* for that signature. The identifier of the signature represents a set of atoms which can be accessed by the identifier. The example model gives 4 signatures: *FSObject*, *File*, *Dir*, *FileSystem*.

A field of a signature represents a relation from an atom of the signature to another atom. The FileSystem signature has four fields: root, live, contents, parent. The *root* field represents the relation from *FileSystem*→*Dir*. Relations with an arity higher than two are described by the -> symbol. The parent field

Listing 4.1: Alloy file system example[All09]

```
abstract sig FSObject { }

sig File, Dir extends FSObject { }

sig FileSystem {
  root: Dir,
  live: set FSObject,
  contents: Dir lone-> FSObject,
  parent: FSObject ->lone Dir
} {
  no root.parent
  live = root.*contents
  contents in live->live
  parent = ~contents
}

pred example {}
run example for exactly 1 FileSystem, 4 FSObject

pred move [fs, fs': FileSystem, x: FSObject, d: Dir] {
  (x + d) in fs.live
  fs'.parent = fs.parent - x->(x.(fs.parent)) + x->d
}
run move for 2 FileSystem, 4 FSObject

pred remove [fs, fs': FileSystem, x: FSObject] {
  x in (fs.live - fs.root)
  fs'.parent = fs.parent - x->(x.(fs.parent))
}
assert removeOkay {
  all fs, fs': FileSystem, x: FSObject |
    remove[fs, fs', x] => fs'.live = fs.live - x
}
check removeOkay for 2
```

Listing 4.2: Alloy fact example[All09]

```
fact ProperFileSystems {
  all fs: FileSystem |
    no (fs.root).(fs.parent) and
    fs.live = (fs.root).*(fs.contents) and
    fs.contents in fs.live->fs.live and
    fs.parent = ~(fs.contents)
}
```

of the FileSystem signature describes the relation *FileSystem→Dir→FSObject*. An atom is a unary-relation.

Signatures can extend other signatures, much like the inheritance of objects in the object oriented-paradigm. In the example model, the *File* and *Dir* signature extend the abstract *FSObject* signature. The File and Dir signature both represents a subset of the FSObject signature. If two or more signatures extend a signature, these sets are disjoint from each other. Following the example model, the sets of atoms of the File and Dir signatures are disjoint. Abstract signatures are not instantiated to atoms.

Quantifiers can be added to restrict the number of relations for a field. The quantifiers are: *lone*, *one*, *some*, *set*. These quantifiers limit the models to: less than or one, exactly one, one or more, zero or more, respectively. In no quantifier for a field is given, the default of exactly one is used.

The model consists of four signatures. The FSObject signature is abstract and never used directly. The File and Directory signatures represent the files and directories of the file system, respectively. Finally, the FileSystem signature represents the container for the file system.

The fields of the *FileSystem* are: *root*, *live*, *contents*, and *parent*. The *root* field represents the directory which is the root of the file system. The *live* field contains all the *FSObjects* (Files and Directories) present in the file system. The *contents* field is used to specify the structure of the FSObjects held by each Directory. Finally, each item in the FileSystem has no or one parent, which is described by the *parent* field.

4.5.2 Facts

Facts are used to constrain the models. A fact paragraph is an assumption over the model which always holds. A fact paragraph consists of expressions which always hold for the model.

Facts over models can be given in two manners. The first manner is to describe a fact by using the fact keyword. An example is given in listing 4.2. The second manner is to append the fact to a signature. This fact then describes constraints of the model over this signature. An example of this is given in listing 4.1, appended to the FileSystem signature. Both the facts given in listing 4.2 and in listing 4.1 represent the same constraints. Note that the fact appended after the FileSystem signature constraints the model with regard to the FileSystem signature only, and does not explicitly have to state the use of the FileSystem signature.

Keyword	Description
||, &&, !	Disjunction, conjunction, negation
+, -, &	Union, difference, intersection
=>, <=>	Implication, bi-implication
++	Override
.	Composition
in, =	Subset, equality
no, lone, one, some, set	No, zero or one, exactly one, one or more, zero or more
~, *, ^	Relational transpose, reflexive transitive closure, transitive closure.
all, some, one, no	universal, existential, singleton, none

Table 4.2: Alloy operators

Expressions

The fact appended to the FileSystem signature in listing 4.1 has four expressions. The first expression specifies that the root directory does not have a parent. The no-keyword is a quantifier and, in this case, is used to restrict the number of relations for an atom.

The .-keyword is the relational join operator. $p.q$ represents the relational join of p to q. The result is the combination of every element in p and every element in q, if any exists. p and q may not both be unary relations (atoms). Given that Alloy is typed, the right type of p must match the left type of q. In this case, the relation *Dir* is combined with the relation *FSObject→Dir*. Given that *Dir* extends from *FSObject*, the join is valid. The join yields the relational image of root in parent.

The second expression states that all the FSObjects contained by the FileSystem signature is equal to the contents of the root directory, and its sub directories. The =-operator specified the equality of two relations. The *-operator denotes the reflexive transitive closure. The ^-operator denotes the transitive closure. * is equivalent to (iden + ^contents) where iden is the keyword to represent the 'current' atom.

Thirdly, the expression states that the structure (contents) of the file system must be a subset of the relations between the product of all the elements in the file system. The in-operator specifies the left operand of the in-operator is a subset of the right operand. The →-operator denotes the product operation of the concatenation of the combination of the left-operand with the right-operand. In this case, all the contents-relations must be contained in the product of the live-relation with itself.

Finally, the last expression states that the parent-relation is the transpose of the contents relation. The ~-operator denotes the transpose of a relation.

A summary of all the operators in Alloy with a short description is given in table 4.2.

Listing 4.3: Example Alloy function

```
fun between [lower:Int, upper:Int]: Int {
    { answer: Int | lower < answer && answer < upper }
}
```

4.5.3 Functions and predicates

A function paragraph, indicated by the *fun*-keyword, is a formula which can be invoked from somewhere else. A function takes a zero or more arguments. Quantifiers can be added to the arguments, but are not enforced by the Alloy Analyzer. A function returns a number relations conformant to the given return-type. The body of the function is an expression which must result in set of relations. Note that a relation can also be a unary relation. Functions are typically useful to define commonly used code and structure the code.

An example of a function is given in listing 4.3. This function takes two arguments, lower and upper, and returns a set of Int. The body of the function states that the result of the function are the Int atoms which are bigger than the lower-argument, and smaller than the upper-argument.

A predicate paragraph, indicated by the *pred*-keyword, is a function which evaluates to true or false. An example of a predicate is given in listing 4.1, named *move*. The *move* predicate takes two FileSystem atoms (fs, fs'), a FSObject atom (x), and a Dir atom (d), and describes the moving of the FSObject atom to the Dir atom. The parameter fs represents the filesystem before the move, fs' represents the filesystem after the model.

4.5.4 Scope

The Alloy Analyzer can be used to automatically analyze the model within a finite scope. The scope restricts the number of atoms for each signature, and is user-defined. The Alloy Analyzer instantiates up to a number of atoms for a signature, or exactly a number of atoms for a signature. The command example, in listing 4.1, defines a scope of exactly one FileSystem atom, and up to four FSObject atoms.

The analysis performed by the Alloy Analyzer is sound and incomplete, since it only checks for within a given scope. However, no guarantees can be made for a larger scope, without actually analyzing the larger scope. The idea behind this approach is the "small scope hypothesis". This hypothesis states that if a failure will happen, it will most likely happen within a small scope.

Figure 4.8 gives the found model when running the *example*-predicate in the given example model. It shows a file system which contains two directories and one file. A second file is in the model, but not contained by the file system.

Figure 4.9 gives an instance of a model for running the *move*-command. It shows two FileSystem atoms, one Dir atom, and one File atom. The first FileSystem atom (FileSystem1) contains the File atom, the second FileSystem atom (FileSystem0) does not. The File is removed from FileSystem1. In this case, FileSystem1 and FileSystem2 represent two different states of the file system.

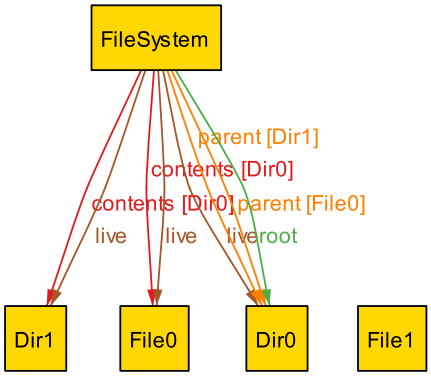


Figure 4.8: Instance of FileSystem model (based on [All09])

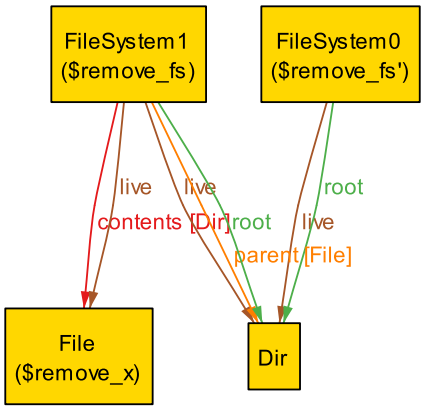


Figure 4.9: Resulting instance of *move*-predicate (based on [All09])

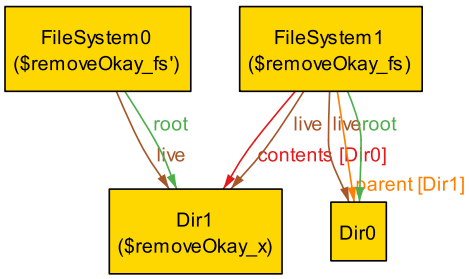


Figure 4.10: Counter example found by the *removeOkay*-assertion (based on [All09])

4.5.5 Assertions

In contrast to a fact, which forces all models to conform to that fact, an assertion is used to claim something that must already be true for the model. If an assertion is violated over a model, the Alloy Analyzer will return the model as a counter example. An example of an assertion is given in listing 4.1, called *removeOkay*. The assertion states that for all two FileSystems, and one FSObject, the remove predicate implies the actual removal of the object from the file system.

Checking the *removeOkay* assertion results in a counter example. The output from the Alloy Analyzer is given in figure 4.10. This counter example shows that in the pre-state (FileSystem1), Dir0 is the root of the filesystem, Dir1 is the directory which is to be removed. In the post state (FileSystem0), Dir0 is removed, but Dir1 is chosen as the new root, which is incorrect.

Counter examples helps us to identify models which are invalid and tells us why these models are invalid. A model can be under constrained, meaning that it is not constrained enough by facts. A model can also be over constrained, meaning that it is too constrained by facts. Assertions are used to check whether the model is correct given a number of formulas.

The use of assertions allows us to incrementally build the specification. We start with a minimal model, and perform checks over the model using assertions. If the model is under constrained or over constrained, a counter example will be given. From the counter example, we can correct the model and re-run the verification. If all assertions hold, we can further extend the model.

The evaluator of the Alloy Analyzer is used to query the counter examples. The evaluator allows us to evaluate formulas over the model which has been found.

4.5.6 State space exploration using Alloy

Alloy can be used to explore the state space of a discrete event based simulation. Basically, two approaches are possible. The first approach is generating the whole state space at once. Each state could have zero or more subsequent states. Only a single model is returned by the Alloy Analyzer.

The second approach generates traces from the whole state space. The Alloy Analyzer returns multiple models, which together represent the whole state space. Each state now has zero or one subsequent state, instead of multiple

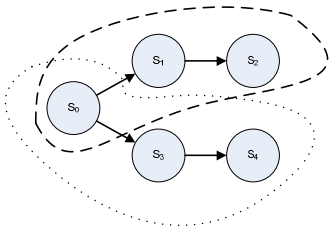


Figure 4.11: Approaches in state space exploration using Alloy

states. The ordering-module is used to impose an ordering over the states. This allows for optimizations by the Alloy Analyzer, resulting in faster state space exploration. If the state space is tested against predicates, the Alloy Analyzer tests the predicates against each of the found predicates.

Figure 4.11 gives a graphical overview of the two approaches. The whole state space consists of the five states S_0, S_1, S_2, S_3, and S_4. S_0 is considered to be the initial state. Using the first approach, the Alloy Analyzer would return the complete state space as a single model. Using the second approach, the Alloy Analyzer returns two models representing two traces. The first trace consists of the states S_0, S_1, and S_2; the states captured by the dashed line. The second trace consists of the states S_0, S_3, and S_4; captured by the dotted line.

4.6 Rationale for Alloy

To be able to reason about the architecture a semantic framework is needed. In our case, the reasoning about the architectures consists of generating a state space and validating the existence or lack of existence of behavior. We have defined several criteria to find the semantic framework best suited for our purposes. The criteria are as follows:

- The framework must return a counter example if an assertion is violated;
- The framework must support state space exploration;
- The framework must support fully automatic analysis.

Given these criteria, Alloy was chosen as it suites our needs. The Alloy Analyzer returns a counter example, in case an assertion is violated. We use this counter example to find the predicates which are not satisfied by the architecture. Alloy also provides a way of formally specifying requirements. These requirements state behavior which is to be found in the architecture. Also, the Alloy Analyzer promises to be very fast at doing its job.

Prolog[Wie09] can be used as an alternative, but was hardly investigated. It also provides a first order logic language, and can be used as a model checker/finder. To us, however, the AlloyAnalyzer appears to be more suited to our needs. Other alternatives, such as Z, OCL, and VDM are not usable. For example, Z is not automatically analyzable. There are no tools for VDM which support model checking as Alloy does. For OCL, no counter examples are generated, and also seems less suited to our needs.

4.7 Conclusion

In this chapter, a description of AADL is given. AADL is used in the automotive, avionics and aerospace industries to specify architectures of complex systems. Companies which use AADL include Ford, Toyota, Boeing, and ESA. A number of tools for AADL are available. These tools are used to reason different properties about the architecture, such as real-time schedule analysis and code generation. Examples of tools include OSATE, Ocarina, and PolyORB-HI-C.

Section 4.2 introduced the concepts of AADL. In section 4.3 we gave the rationale behind the choice for AADL. Section 4.4 describes an example architecture, which is used throughout the thesis to explain different concepts.

Alloy is used to perform complex structural and behavioral analysis over models. The language is based on declarative first-order logic. Being declarative, we can write statements to specify that an event has occurred, instead of specifying how an event occurs. Alloy also allows for automatic analysis, in contrast to Z and OCL. Alloy has been used for several case studies, including the analysis of the Firewire specification, analysis of a flash filesystem, and validating compliance with privacy legislation.

In section 4.5 we have introduced Alloy. The main concepts of Alloy are given, and we have shown how to use these concepts. Section 4.6 gives the rationale for the choice behind Alloy.

Chapter 5

Simulating architectures

5.1 Introduction

This chapter introduces the simulation of AADL architectures using Alloy. The representation of the static structure of the architecture in Alloy is given. A discrete event simulation is introduced to derive the behavior of architectures.

Section 5.2 gives the representation of the static structure of AADL architectures in Alloy. A discrete event simulation of architectures in AADL is introduced in section 5.3. In section 5.4 we give an extended version of the simulator. Results of benchmarks for both the simple as well as the parallel version of the simulator are given in section 5.5. Section 5.6 gives a discussion about the limitations of the simulation. Finally, section 5.7 concludes this chapter.

5.2 Architecture structure

This section describes the *Define structural semantics in Alloy* activity, found in figure 3.8 of section 3.5. The structure of the architecture is based on the component and connector view. An overview of the component and connector view is given in section 2.4.1.

Alloy is used to simulate AADL architectures. To simulate the behavior of the architecture, we need to represent the structure of the architecture in Alloy. For each supported element in the AADL meta-model, we have created an abstract signature in Alloy. The structure of the Alloy module is as close as possible to the AADL meta-model, although simplifications have been applied. This allows for a transparent mapping from the AADL instance models to Alloy models.

The full Alloy module is given in appendix A. Throughout this section, excerpts are given of the Alloy module. Figure 5.1 gives a graphical overview of the module. An example of a part of the RPM architecture, found appendix B, in Alloy is given in appendix C.

The transformation of the architecture structure from AADL instance models to Alloy code is fully automatic. This transformation is created in the *Build transformation from AADL to Alloy* activity, found in section 3.5. The transformation consists of an XPand2 model-2-text transformation. However, for brevity, we omit description of the transformation itself.

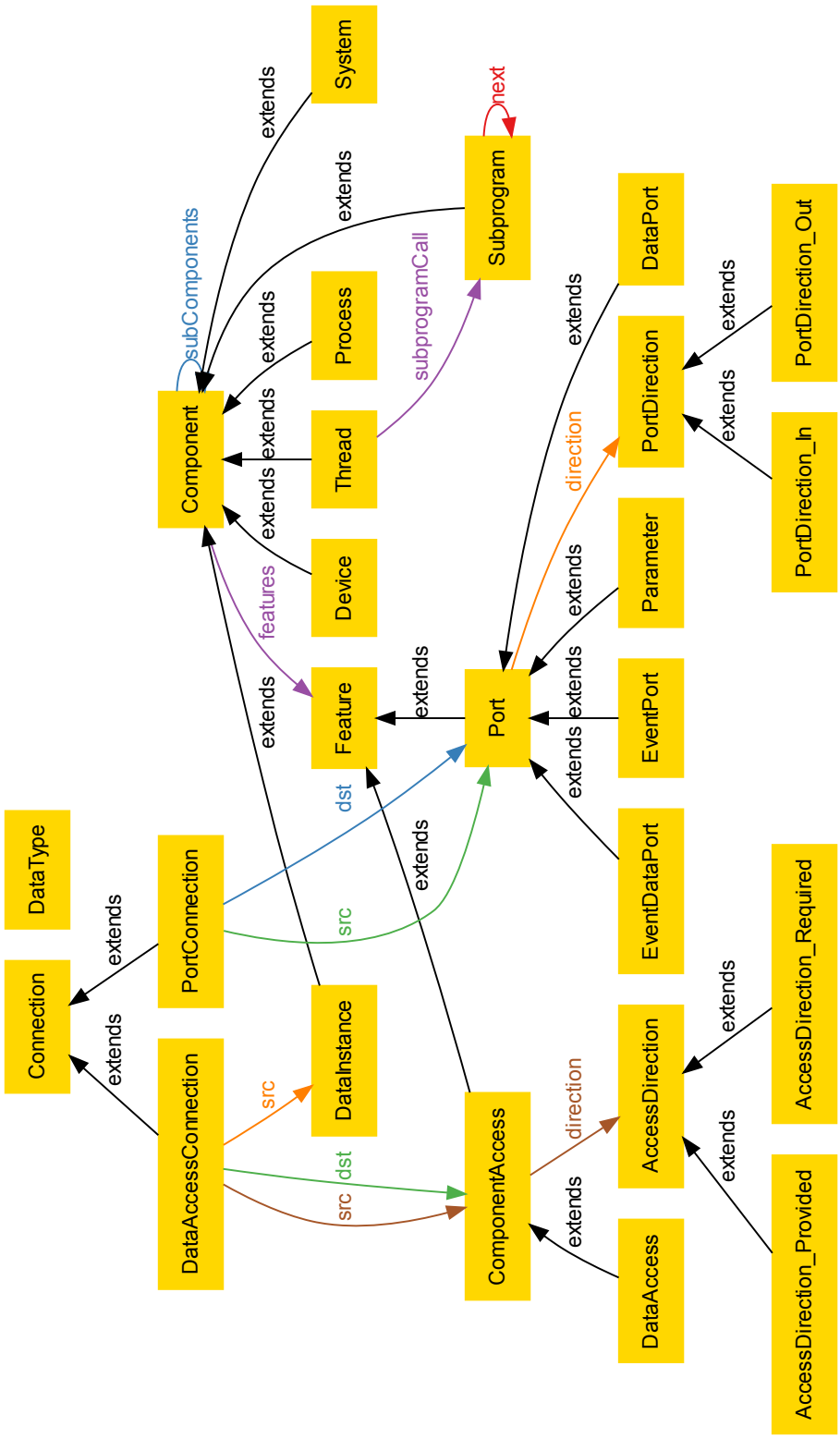


Figure 5.1: Graphical representation of AADL module in Alloy

Listing 5.1: Components in Alloy

```
abstract sig Component {
  features: set Feature,
  subComponents: set Component,
}

abstract sig Process extends Component {
} {
  subComponents in DataInstance + Thread
  features in DataPort + EventPort + EventDataPort +
      DataAccess
}
```

5.2.1 Components

A single abstract signature, called *Component*, is used as a basis for the *Process*, *Thread*, *Subprogram*, *DataInstance*, and *System* component signatures. The signature is given in listing 5.1. This signature has two fields: *features* and *subComponents*. The *features*-field records the features of the component. The *subComponent*-field records the subcomponents for the component. As a component is not required to have any features or subcomponents, we use the *set*-quantifier for both fields.

The abstract Process signature is given as an example in listing 5.1. The signature extends the Component signature, and thus has the *features* and *subComponents*-fields. Facts are added to the *Process* signature to enforce correctness of the architecture structure. The fact attached to the Process signature describes two constraints. The first constraint is that a process can only have data instance and thread subcomponents. The second constraint describes that a process can only have data port, event port, event data port, and data access features.

The signatures representing other component types are shown in listing A.1.

The naming scheme proposed in section 4.4 is used, but altered, to identify the different component instances. Because Alloy does not allow dots (.) to be used in the signature names, we replace the dots by an underscore (_). Thus, the name of subprogram *sd.proc1.thread1.c1*, found in the example architecture from section 4.4, becomes *sd_proc1_thread1_c1*. As an example, the signature for the thread *hpc.sdm.proc1.thread1* is given in listing 5.2. The thread instance contains one subcomponent, two features and one subprogram call. It has to be noted that the OSATE toolset prefixes the names with a name, based on the model. In the example, this prefix is *RPM_RPM_i_Instance_*.

5.2.2 Features

Similar to components, a single abstract feature which is extended by the other feature types. The supported features are: data port, event port, event data port, parameter, and data access. An excerpt of the Alloy module regarding features is given in listing 5.3. The abstract *Feature* signature does not contain any fields. The data port, event port, event data port, and parameter features

Listing 5.2: Code for *hpc.sdm.proc1.thread1* in Alloy

```
one sig RPM_RPM_i_Instance_hpc_sdm_thread1 extends Thread {
} {
  subComponents = none
      + RPM_RPM_i_Instance_hpc_sdm_thread1_c1
  features = none
      + RPM_RPM_i_Instance_hpc_sdm_thread1_measurement_in
      + RPM_RPM_i_Instance_hpc_sdm_thread1_temperature_measurements
  subprogramCall = none
      + RPM_RPM_i_Instance_hpc_sdm_thread1_c1
}
```

Listing 5.3: Features in Alloy

```
abstract sig Feature {
}

enum PortDirection {
  PortDirection_In,
  PortDirection_Out
}

abstract sig Port extends Feature {
  direction: one PortDirection,
}

abstract sig DataPort extends Port {
}
```

all extend from the abstract Port signature. This abstract signature contains one field called direction. It represents the direction of the port which is either incoming or outgoing. The enumerator *PortDirection* is used to specify the direction of the port. The *DataPort* signature is given as an example in listing 5.3. The signature extends the Port signature. No constraints are added to the signature.

Again, the naming scheme proposed in section 4.4 is used. Again, dots are replaced by an underscore. Listing 5.4 shows the Alloy code for the *hpc.sdm.proc1.thread1.measurement_in* features, from the example architecture given in section 4.4. The direction of the event data port is incoming.

Listing 5.4: Code for *hpc.sdm.proc1.thread1.measurement_in* in Alloy

```
one sig RPM_RPM_i_Instance_hpc_sdm_thread1_measurement_in extends
    EventDataPort {
} {
  direction = PortDirection_In
  all s: State | this.(s.eventDataPortValues) in Measurement_i
}
```

Listing 5.5: Connections in Alloy

```
fun portConnections[] : Port -> Port {
  DataPort1 -> DataPort2
}

fun dataAccessConnections[] : (DataInstance + DataAccess) ->
    DataAccess {
  DataInstance1 -> DataAccess1 +
  DataAccess1 -> DataAccess2
}

abstract sig Connection {
}

abstract sig PortConnection extends Connection {
  src: one Port,
  dst: one Port,
}

abstract sig DataAccessConnection extends Connection {
  src: one DataInstance + DataAccess,
  dst: one DataAccess,
}
```

5.2.3 Connections

Connections in architectures can be represented by two different methods. The first method is by a function, as shown by the first two functions on the top in listing 5.5. The first function, *portConnections[]*, returns a set of relations of the type *Port→Port*. This relation represents a port connection from a source port to a target port. The *dataAccessConnections[]* function is used to represent the data access connections in the architecture. The function returns a set of relations of the type *(DataInstance+DataAccess)→DataAccess*.

The second method is to extend the abstract *PortConnection* and *DataAccessConnection* signatures, for port connections and data access connections respectively. Similar to the *portConnections[]* and *dataAccessConnections[]* functions, these signatures capture the source port and destination port, or source data instance/data access and target data access for the port or data access connection, respectively. The field *src* is used to represent the source feature of the port and data access connection. The *dst* field is used to represent the destination feature of the port and data access connection. The rationale for the use of the two different methods is to increase the speed of the simulation, and the need to be able to induce connections. The induction of connections is found in section 6.4.

Using the example architecture as given in 4.4, listing 5.6 gives an excerpt of the port connection function.

5.2.4 Data types

For each data type defined in the architecture, a signature is generated. The generated data type signatures extend from the abstract *DataType* signature, given in listing 5.7. The data types in the architecture can be annotated with additional fields, through the use of the relations-property. For example, in

Listing 5.6: Port connections function for example architecture in Alloy

```
fun portConnections[] : Port -> Port {
  none -> none
    + (RPM_RPM_i_Instance_sd_sensor1_measurement ->
        RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in)
    + (RPM_RPM_i_Instance_sd_proc1_thread1_measurement_out ->
        RPM_RPM_i_Instance_sdc_proc1_thread1_measurement_in)
    + (RPM_RPM_i_Instance_sd_proc1_thread1_alarm_out ->
        RPM_RPM_i_Instance_sdc_proc1_thread1_alarm_in)
}
```

Listing 5.7: Data types in Alloy

```
abstract sig DataType {
}
```

appendix B in listing B.1, the Measurement.i data type contains additional fields.

5.3 Discrete event simulation

In section 3.5, we gave the different activities of building the supporting framework. This section describes the *Defining behavioral semantics in Alloy* activity.

To simulate architectures, several approaches can be taken. Our simulation is a discrete event simulation[BC84]. The simulation of the architecture is driven by events that occur within the architecture. The state of the architecture is recorded between these events. The state space of a simulation is defined as all the states discovered during the simulation of the architecture, together with transitions between these states.

The same transformation from AADL instance models to Alloy code is used to create (parts of) the simulator. We have created this transformation in the *Build transformation from AADL to Alloy* activity, found in section 3.5. The transformation consists of an XPand2 model-2-text transformation. However, for brevity, we omit description of the transformation itself.

5.3.1 States

In section 3.2, we stated that the architecture behavior is captured in a state space. The *state space* contains all the states in which an architecture can be during the simulation of the architecture. A *state* records the data values, events, and messages bound to data ports, event ports, event data ports, or data instances in the architecture, and the subprograms which are queued for dispatch, at a given time during the simulation.

Given that we are using Alloy for the simulation of the architecture, we define a *State* signature which represents a state in Alloy. The *State* signature is given in listing 5.8. The State signature has the following fields:

dataPortValues The *dataPortValues*-field gives the data values which are bound to data ports in the architecture. A data port can be bound to

no or one data value.

eventPortValues The *eventPortValues*-field records the events which are bound to event port in the architecture. An event port can be bound to no or one data value. The AADL standard specifies that event ports can be queues. We limit the queue size of an event port to at most one, to improve the speed of the simulation.

eventDataPortValues The *eventDataPortValues*-field gives the messages (a data value associated with an event) which are bound to event data ports in the architecture. An event data port can be bound to no or one data value. Note that events are omitted in this field. In the AADL standard, event data ports are queues. For simplicity, we limit the queue size of an event data port to at most one, to improve the speed of the simulation. Note that we make no explicit differentiation between data values and messages in the simulation.

parameterValues The *parameterValues*-field records the data values which are bound to parameters in the architecture. A parameter can be bound to no or one data value.

dataInstanceValues The *dataInstanceValues*-field gives the data values which are bound to data instances. Data instances can be bound to a set of data values.

toDispatch The *toDispatch*-field records the set of subprograms which are scheduled for dispatch.

A signature is created for each of the data types found in the architecture. These signatures extend the abstract *DataType* signature, found in listing 5.8. The data type signatures are used to represent zero or more data values in the architecture. The *Event* signature is a special signature which extends the *DataType* signature. It is used to represent an event. It is instantiated once. The *Event* signature is used by the *eventPortValues*-field of the *State* signature.

The Alloy Analyzer provides the ordering-module which imposes a linear ordering over a specific signature. The ordering-module adds a next-relation to the signature it is imposed on. This next-relation is used to describe the relation to the next state, from the current state. When using the ordering-module on

Listing 5.8: State signature in Alloy

```
abstract sig DataType {}

one sig Event extends DataType {}

sig State {
  dataPortValues: DataPort -> lone DataType,
  eventPortValues: EventPort -> lone Event,
  eventDataPortValues: EventDataPort -> lone DataType,
  parameterValues: Parameter -> lone DataType,
  dataInstanceValues: DataInstance -> DataType,
  toDispatch: set Subprogram,
}
```

the state signature, the scope of the state signature becomes strict. This means that the Alloy Analyzer can no longer check up to a number of states, but only check for a specific number of states. As a result, if a simulation consists of 10 states, we have to specify this number before the simulation. When the exact number of states is not known beforehand, a trial-and-error approach has to be taken to find the exact number of states.

As a remedy to the strict scope with regard to the number of states, we can introduce stuttering states[RR98]. A stuttering is a transition between two states which has no observable effect. The result of a stuttering transition is that the two states connected by the stuttering transition are equivalent. If we only allow stuttering to happen at the tailing states of the state space, we ensure different simulations (with a different number of states) result in the same behavior, as long as the number of states is overestimated. Consequently, we can over-estimate the number of states and the scope is no longer strict.

As an example, assume that the full simulation of an arbitrary architecture would consist of exactly 10 states. If we introduce the stuttering transition to the simulation and overestimate the number of states to 15, the last 5 states will be stuttering states and equivalent to the 10th state. When we overestimate the number of states to 20, the last 10 states will be stuttering states. In both examples, the first 10 states, which describe the behavior of the architecture, are present in the state space.

5.3.2 Transition function

The transition function describes how a state is altered from the current state to the next state. A transition represents the happening of one or more events during the simulation. An *event* is a notable occurrence within the architecture during simulation of the architecture.

Two different kinds of events are defined: *events introduced by the AADL standard*, and *events introduced by the architect*. These two kinds are described in this subsection.

Events introduced by AADL

Different types of events can occur within the architecture, as defined by the AADL standard[SAE09]. The events given here are all based on the transfer of data values, events, or messages from one port to another port through a port connection, or the scheduling of dispatch of a subprogram. As the behavioral properties of subprograms are specified by the architect, no events can be defined for these.

All the events introduced by AADL are extracted from the AADL standard. For each of the events listed below, a reference to the related section in the AADL standard is given.

- Transfer of a data value from a data port to another, through a data port connection: section 9.2;
- Transfer of an event from an event port to another event port, through an event port connection: section 9.2;
- Transfer of a message from an event data port to another event data port, through an event data port connection: section 9.2;

- Transfer of a data value from a data port to a parameter, through a parameter connection: section 9.3;
- Transfer of a message from an event data port to a parameter, through a parameter connection: section 9.3;
- Transfer of a data value from a parameter to another parameter, through a parameter connection: section 9.3;
- Transfer of a data value from a parameter to a data port, through a parameter connection: section 9.3;
- Transfer of a data value from a parameter to an event data port, through a parameter connection: section 9.3;
- Enqueue dispatch of a subprogram: section 5.2, section 5.4.

Listing 5.9: Transition function in Alloy

```
pred transition[s, s': State] {
  s'.dataPortValues =
    dp2dp[s.dataPortValues] + p2dp[s.parameterValues]
  s'.eventPortValues =
    e2e[s.eventPortValues] + producedEps[s]
  s'.eventDataPortValues =
    edp2edp[s.eventDataPortValues] + p2edp[s.parameterValues] +
        producedEdps[s]
  s'.parameterValues =
    p2p[s.parameterValues] + edp2p[s.eventDataPortValues] +
        producedParams[s]
  s'.dataInstanceValues =
    s.dataInstanceValues + producedDataInstanceValues[s] -
        removedDataInstanceValues[s]
  s'.toDispatch =
    enableDispatch[s] + { s.toDispatch.next }
}
```

In the simulation, we use a single transition function. This transition function is given in listing 5.9. This transition function captures all the events described in this section. The transition predicate describes the previous state (parameter *s*) and the next state (parameter *s'*) which are connected by the transition. The post-condition of the transition (parameter *s'*) is described by several functions. Each of these functions represent the events given before, with the exception of the *producedEps*, *producedEdps*, *producedParams*, *producedDataInstanceValues*, and *removedDataInstanceValues* functions.

An excerpt of the functions is given in listing 5.10. The first function, *dp2dp*, returns the resulting set of data values bound to data ports, from a given state. Its parameter, (*r*), is a set of relations from the previous state. The function returns a set of relations *DataPort→DataType* which describe the set of data values bound to data ports in the next state. The result is computed by finding all combinations of data port (*dp*) and data values (*dt*) for which there is a relation in the set of bindings from data port to data value in the original state (*r*), and the port holding the data value is connected to the new port (*dp*) by a port connection (*dp.~portConnections[]*).

Listing 5.10: Events introduced by AADL in Alloy

```
fun dp2dp[r: DataPort -> DataType] : DataPort -> DataType {
  { dp: DataPort, dt: DataType |
    one (dp.~portConnections[] -> dt & r) }
}

fun p2dp[r: Parameter -> DataType] : DataPort -> DataType {
  { dp: DataPort, dt: DataType |
    one (dp.~portConnections[] -> dt & r) }
}

fun ep2ep[r: EventDataPort -> DataType] : EventPort -> Event {
  { ep: EventPort, e: Event |
    one (ep.~portConnections[] -> e & r) }
}

fun enableDispatch[s: State] : set Subprogram {
  { sp: Subprogram | one t: Thread, edp: EventDataPort |
    sp = t.subprogramCall and edp in t.features and edp.direction =
         PortDirection_In and edp in univ.~(s.eventDataPortValues)
       }
}
```

The other functions are similar to the *dp2dp* function. The *p2dp* differs from the *p2dp* function in that it transfers the data value from a parameter to a data port. The parameter r takes a set of relations from parameters to data values, while the result is a set of relations from data ports to data values. The *ep2ep* function transfers the special event type from one event port to another event port.

Finally, the *enableDispatch* function schedules subprograms to be dispatched. If a thread receives an event or message at an incoming event or event data port, and the thread contains a subprogram, the subprogram is scheduled for dispatch in the next state.

Events introduced by architect

Subprograms in the architecture are considered the components that perform actual manipulation of data. To specify this computation, subprograms need to be annotated with pre- and post-conditions. The pre-conditions state the conditions of a state before the subprogram can be dispatched. The post-conditions of a subprogram describe how a state is altered when the subprogram is dispatched. For example, if a data value bound to a parameter and the subprogram is dispatched, the subprogram can produce a new message at one of its outgoing event data ports. Another example is that the subprogram writes the received parameter to a data instance. A subprogram can alter a state in the following way, upon dispatch:

- A subprogram can output an event through an event port;
- A subprogram can output an message through an event data port;
- A subprogram can output a data value through a parameter;
- A subprogram can store one or more data values at a data instance;

- A subprogram can remove one or more data values from a data instance.

The AADL standard does not give a way to model behavior. In order to define the behavior of a subprogram into the architectures, we use a property set. An excerpt of this AADL property set is given in listing 5.11. For each of the effects given before, a property is introduced. The *post_ep* property is used to describe the effects of the subprogram dispatch on the *eventPort* field of a state. These properties correspond with the *producedEps*, *producedEdps*, *producedParams*, *producedDataInstanceValues*, and *removedDataInstanceValues* functions, used in the transition function in listing 5.9.

Listing 5.11: Property set to specify subprogram behavior in AADL

```
property set Alloy is
  ...
  post_ep: aadlstring  applies to (subprogram);
  post_edp: aadlstring  applies to (subprogram);
  post_param: aadlstring  applies to (subprogram);
  post_diProduced: aadlstring  applies to (subprogram);
  post_diRemoved: aadlstring  applies to (subprogram);
  ...
end Alloy;
```

The subprogram effect on a state are introduced into the architecture by annotating the subprograms in the AADL specification with properties. These annotation are used by the simulator to describe the behavior of the subprograms. An example subprogram is given in listing 5.12. This is transformed to a single function in the Alloy model. The function *subp1::post_edps* describes the results of dispatching the subprogram, in the Alloy model. The resulting function is given in listing 5.13. In this example, if a data value is received at the parameter *subp1_sensor1_in*, the data value is forwarded to the event data port *subp1_measurement_out*.

Listing 5.12: Subprogram annotated with behavioral properties in AADL

```
subprogram subp1
  features
    sensor1_in: in parameter Measurement.i;
    measurement_out: out event data port Measurement.i;
end subp1;

subprogram implementation subp1.i
  properties
    Alloy::post_edp => "
      one subp1_sensor1_in.(s.parameterValues) =>
        { subp1_measurement_out -> subp1_sensor1_in.(s.
            parameterValues) }
      else
        none -> none
    ";
    ...
end subp1.i;
```

The effects of a subprogram on a state are taken into account by the *producedEps*, *producedEdps*, *producedParams*, *producedDataInstanceValues*, and *re-*

Listing 5.13: Predicate defining subprogram behavior in Alloy

```
fun subp1_i::post_edps[s: State] : EventDataPort -> DataType {
  one subp1_sensor1_in.(s.parameterValues) =>
    { subp1_measurement_out -> subp1_sensor1_in.(s.parameterValues)
        }
  else
    none -> none
}
```

Listing 5.14: Grouping of subprogram effects in Alloy

```
fun producedEdps[s: State] : EventDataPort -> DataType {
  none -> none
    + (subp1_i in s.toDispatch => subp1_i.post_edps[s] else none ->
        none)
    + (subp2_i in s.toDispatch => subp2_i.post_edps[s] else none ->
        none)
}
```

movedDataInstanceValues functions. These functions group the effects of the subprograms in the architecture. An example of the *producedEdps* function is given in listing 5.14. This grouping function collects the results with regard to produced message for the event data ports it contains of the two subprograms *subp1* and *subp2*.

5.3.3 Architecture invariants

The architecture can be annotated with architecture invariants. An *architecture invariant* is a formula over a subprogram or data instance which asserts correct behavior. Thus, architecture invariants are used to validate the behavior of individual components within the architecture. An example of an architecture invariant is that a given data instance can hold up to four data values. If this property is violated, the architecture is invalid.

Subprogram and data instance invariants are recorded through properties in the AADL models. Listing 5.15 gives an excerpt of the property set used to annotate the invariants of data instance and subprograms. The listing also gives an example of an annotated data instance in AADL. In this example, the data instance can hold at most four data values.

The invariant-property is used to annotate subprograms and data instances with Alloy code. For each data instance, this property is transformed into a predicate. This predicate must hold for over all the states of the state space. Based on listing 5.15, the generated Alloy code from the invariant-property is shown in listing 5.16. This predicate gives the property that there can be up to four data values in the data instance *di1*, for the given state *s*. The predicate is tested against all the states of the state space. If one or more invariants are violated, we can use the Alloy Analyzer to return the state space as a counter example.

Listing 5.15: Property set to annotate elements with invariants in AADL

```
property set Alloy is
  ...
  invariant: aadlstring applies to (subprogram, data);
  ...
end Alloy;

system implementation S.i
  subcomponents
    di1: data DataType.i {
      Alloy::invariant => #this.(s.dataInstanceValues) <= 4";
    };
end S.i;
```

Listing 5.16: Invariant for data instance in Alloy

```
pred di1::invariant[s: State] {
  #this.(s.dataInstanceValues) <= 4
}
```

5.3.4 Simulation constraints

To ensure correct simulation, further constraints are imposed on the architectures. The ports in the architecture are typed. Given the way the Alloy Analyzer works, it can generate a state where a port would be bound to a data value from a different data type. This is invalid. A fact is attached to each port signature, as found in listing 5.17.

This fact describes that for each state, the port can only be bound to data values of type Measurement. Measurement is a signature which extends the *DataType* signature. A similar fact is added to data instance signatures, as shown in figure 5.18.

Again, this fact describes that each data instance can only be bound to data values of the type Measurement.

5.3.5 Executable scenarios

Section 3.2 introduced the notion of a scenario. This section elaborates on the creation and use of scenarios. Scenarios will be further extended in chapter 6. For now, a *scenario* contains the initial state of the architecture after a stimulus in the environment has been detected. Furthermore, scenarios are used to test the architecture invariants during the simulation.

Using the simulation implemented in Alloy, we need to define an initial state to the simulator. This initial state describes the state of the architecture and

Listing 5.17: Fact attached to a port signature

```
one sig port1 extends EventDataPort {
} {
  all s: State | this.(s.eventDataPortValues) in Measurement
}
```

Listing 5.18: Fact attached to a data instance signature

```
one sig di1 extends DataInstance {
} {
  all s: State | this.(s.dataInstanceValues) in Measurement
}
```

data values, events, and messages within the architecture at the beginning of the simulation. From this initial state, the simulation is continued and subsequent states are generated by the simulator.

Two main **guidelines** can be given for the scenarios. The first **guideline** is that a scenario should be used to describe the occurrence of an event in the environment. Events in the environment are observed by architecture components. This can be a device representing a sensor, or a subprogram which models a program which is used by an end-user. The initial state of the scenario describes that there is an event at an outgoing event port. This event represents the observation of the event.

The second **guideline** is that a scenario should be used to describe the occurrence of an event in the architecture. For example, a timer can expire. The initial state of the change scenario describes that there is an event at an outgoing event port of a component representing a timer.

We give another **guideline** to find the number of data values for a specific data type used in the simulation. If the data type is not used during the simulation as described by the scenario, the scope is to be restricted to 0 for that data type. If the data type is used, we start at 1. Following the executing of the data flow, we end up with a set of subprograms which are executed. If a subprogram alters the data value, we increase the scope for that data type by 1. If the subprogram only forwards or stores the data value, we do not increase the scope.

Note that the number of data values for a data type can be safely overestimated. The result of the overestimation is a slower simulation, while the behavior of the architecture will remain the same. Underestimation of the number of data values for a data type will result in a incomplete simulation, as the resulting state space will not have enough data values. After all the data values have been used, the simulation will stop.

5.3.6 State space

We use the Alloy Analyzer to generate the state space which represents the behavior of the architecture. As we use to ordering module of the Alloy Analyzer, we generate individual traces which, together, form the complete state space for a given scenario. This concept is given in section 4.5.6.

If the behavior of the architecture is deterministic, only one trace is generated by the Alloy Analyzer. This trace is the complete state space for the given scenario, as no choices are to be made during the simulation of the architecture. No choices during the simulation means that there is no state which has to two direct subsequent states.

If the architecture is non-deterministic, the Alloy Analyzer will generate multiple traces. During the simulation, one or more choices are to be made.

Listing 5.19: Scenario for example architecture in Alloy

```
pred scenario1_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.dataInstanceValues
  no s.toDispatch

  one s.eventDataPortValues
}

Run_scenario1 : run {
  StateTransitions and scenario1_initial[so/first]
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  exactly 25 State,
  exactly 1 Measurement_i
```

These choices result in one or more states which have multiple direct subsequent states.

5.3.7 Example simulation

For the example simulation, the example architecture given in section 4.4 is used. We create a scenario which describes the initial state of the simulation. In this scenario, we simulate the occurrence of the event that a sensor has measured temperature at a patient. The initial state describes that there is a data values of the *Measurement.i* at the outgoing event data part of the device *Sensor1*. This data value represents the measurement which the sensor has measured. The scenario, together with the Alloy command to execute the simulation, is given in listing 5.19.

The predicate *scenario1_initial* describes the initial state. The event data port *RPM_RPM_i_Instance_sd_sensor1_measurement* is bound to a data value. This data value is of type *Measurement.i*, as constrained by the facts given in subsection 5.3.4. The Alloy command *Run_scenario1* runs the simulation. The Alloy Analyzer tries to find a model for which the predicates *scenario1_initial* (initial state) and *StateTransitions* (correct simulation) both hold.

The result of the simulation is given in listing 5.20. The listing gives the trace, describe by all the states in the state spaces, resulting from the simulation of the scenario. For example, State$0 describes the initial state, the outgoing event data port of *sensor1* holds a data value. In State$1, the measurement is transferred from the device to the thread in the SD. In State$2, the subprogram of the SD is scheduled for dispatch, while the parameter of the subprogram of the SD is bound to the measurement. In State$3, the subprogram of the SD is dispatched and sent the measurement to its outgoing event data port. A graphical representation of the trace is given in figure 5.2.

The stuttering of states is observable in State$11 and State$12. Both State$11

Listing 5.20: Textual representation of trace from scenario in example architecture

```
State$0:
  Event data ports:
    RPM_RPM_i_Instance_sd_sensor1_measurement$0 : Measurement_i$0
State$1:
  Event data ports:
    RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in$0 :
        Measurement_i$0
State$2:
  Parameters:
    RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in$0 :
        Measurement_i$0
  To dispatch:
    RPM_RPM_i_Instance_sd_proc1_thread1_c1$0
State$3:
  Event data ports:
    RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out$0 :
        Measurement_i$0
State$4:
  Event data ports:
    RPM_RPM_i_Instance_sd_proc1_thread1_measurement_out$0 :
        Measurement_i$0
State$5:
  Event data ports:
    RPM_RPM_i_Instance_sdc_proc1_thread1_measurement_in$0 :
        Measurement_i$0
State$6:
  Parameters:
    RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in$0 :
        Measurement_i$0
  To dispatch:
    RPM_RPM_i_Instance_sdc_proc1_thread1_c1$0
State$7:
  Event data ports:
    RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_out$0 :
        Measurement_i$0
State$8:
  Event data ports:
    RPM_RPM_i_Instance_sdc_proc1_thread1_measurement_out$0 :
        Measurement_i$0
State$9:
  Event data ports:
    RPM_RPM_i_Instance_hpc_sdm_thread1_measurement_in$0 :
        Measurement_i$-
State$10:
  Parameters:
    RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in$0 :
        Measurement_i$0
  To dispatch:
    RPM_RPM_i_Instance_hpc_sdm_thread1_c1$0
State$11:
  Data instances:
    RPM_RPM_i_Instance_hpc_temperature_measurements$0 :
        Measurement_i$0
State$12:
  Data instances:
    RPM_RPM_i_Instance_hpc_temperature_measurements$0 :
        Measurement_i$0
```

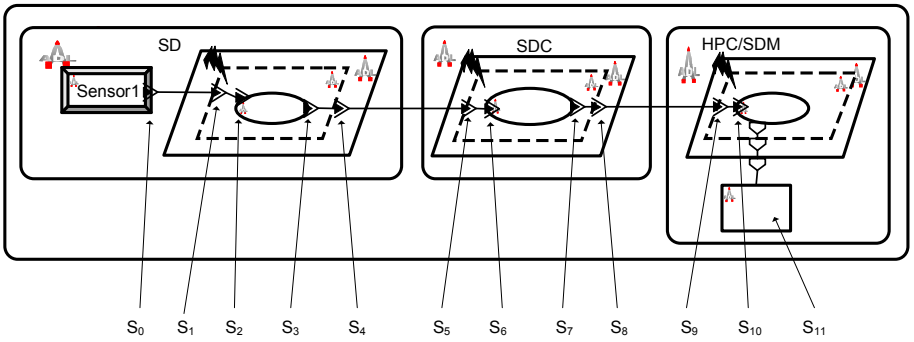


Figure 5.2: Graphical representation of trace from scenario in example architecture

Listing 5.21: Validating architecture invariants for example architecture in Alloy

```
pred scenario1_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.dataInstanceValues
  no s.toDispatch

  one s.eventDataPortValues
}

Check_scenario1 : check {
  (StateTransitions and scenario1_initial[so/first]) =>
  ArchitectureInvariants
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  exactly 25 State,
  exactly 1 Measurement_i
```

and State$12 describe that there is a data value in the same data instance, and thus are equal to each other. No observable event has occurred between these states. We have omitted State$13 to State$24, as these states are all stuttering states and equal to State$11.

To validate that the architecture invariants are not violated during the simulation, we need to adapt the *Run_scenario1* command. The adapted version is given listing 5.21. The command is now a check-command, which Alloy uses to validate assertions over a model. The new command consists of an implication. The left part of the implication is the old command, simulate the architecture from the given initial state. The right part of the implication states that all architecture invariants must hold. If the implication fails, a counter example is given by the Alloy Analyzer. In this case, the architecture satisfies all the invariants and thus no counter example is returned.

Listing 5.22: Main transition function for parallel simulator in Alloy

```
pred StateTransitions {
  all s: State - so/last |
    dp2dp[s, s.next] or dp2p[s, s.next] or
    ep2ep[s, s.next] or edp2edp[s, s.next] or
    p2dp[s, s.next] or p2edp[s, s.next] or
    prepareDispatchByEventPort[s, s.next] or
        prepareDispatchByEventDataPort[s, s.next] or
    -- custom subprogram transitions
    subp1.transition[s, s.next] or
    subp2.transition[s, s.next] or
    stutter[s, s.next]

  all s: State - so/last | stutter[s, s.next] =>
    all s': s.^next | stutter[s, s']
}
```

5.4 Parallel simulation

We name the previously introduced simulation the *basic simulation*. In the basic version of the simulation, multiple events can occur during the same transition from one state to the next state. For example, if two distinct event ports hold an event in a state and are both connected to another event port through a port connection, the events will both be transferred to the connected event ports in the next state. Using this approach, the simulator cannot find all the possible interleavings. To be able to find all interleavings, the parallel version of the simulation is introduced. This version is not explained in detail in this thesis, for brevity.

To find all the interleavings, we need to modify the transition function. The updated transition function allows only one event to occur at the same time. Listing 5.22 describes the updated transition function. The updated transition function specifies that one of the transitions is taken. The semantics derived from the AADL standard found in subsection 5.3.2 are in the updated transition function. The predicates *dp2dp*, *dp2p*, *ep2ep*, *edp2edp*, *p2dp*, *p2edp*, *prepareDispatchByEventPort*, and *prepareDispatchByEventDataPort* describe this. The stuttering predicate is also present, named *stutter*. For each subprogram, the dispatch of that subprogram is given its own transition. This transition function is described in the AADL model by the architect. In listing 5.22, the transitions for the dispatch of two subprograms are included, shown by the *subp1.transition* and *subp2.transition* predicates. Before one of the subprogram transitions can be taken, the subprogram itself has to be scheduled for dispatch.

Given the atomic events, the simulation finds all interleavings. As such, we can simulate queues for event and event data ports. The *State* signature is modified, such that the *seq* quantifier is used for the *eventPortValues* and *eventDataPortValues* field of the *State* signature. The updated part of the *State* signature is shown in listing 5.23.

As an example, the *dp2dp* predicate is given in listing 5.24. This predicate describes that all the fields of the *State* signature, except for the *dataPortValues* field, remain the same. Only the *dataPortValues* field is updated, such that one data value is transferred from one data port to another data port, which are connected by a port connection.

Listing 5.23: Updated state signature in Alloy

```
sig State {
        ...
        eventPortValues: EventPort -> (seq Event),
        eventDataPortValues: EventDataPort -> (seq DataType),
        ...
}
```

Listing 5.24: Transition function for data port to data port transfer in Alloy

```
pred dp2dp[s, s': State] {
  s'.eventPortValues = s.eventPortValues
  s'.eventDataPortValues = s.eventDataPortValues
  s'.parameterValues = s.parameterValues
  s'.dataInstanceValues = s.dataInstanceValues
  s'.toDispatch = s.toDispatch
  one dp: DataPort |
    some dp.portConnections[] and
    one dp.(s.dataPortValues) and
    s'.dataPortValues = s.dataPortValues
      - dp -> dp.(s.dataPortValues)
      - dp.portConnections[] -> DataType
      + dp.portConnections[] -> dp.(s.dataPortValues)
}
```

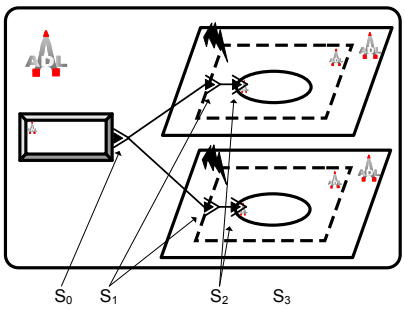
Compared to the basic simulator, the *dp2dp*-transition predicate is different in the sense that only a single data value is transferred from one data port to another data port. In the basic simulator, the function takes all the bindings from data values to data ports, and returns the resulting bindings from data values to data ports, where the data value is transferred from one data port to another data port, when the two data ports are connected through a port connection.

A constraint is added such that only the tailing states of the state space are allowed to stutter, as shown by the last two lines of the transition predicate in listing 5.22. No stuttering states can be introduced during the simulation itself. If we do not use this constraint, the *stutter*-transition can be taken while other transition can also be taken. We only allow the *stutter*-transition to be taken, if all the following transitions are *stutter*-transitions.

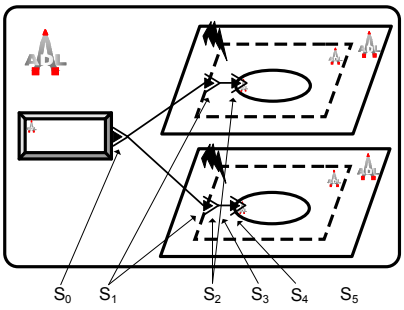
For example, the simulation of an arbitrary architecture, consisting of five states, could result in a state space for which the transitions taken are as follows: *dp2dp*, *dp2dp, dp2dp,* and *stutter*. We do not want stuttering when other transitions are also possible, as it might result in an incomplete simulation. For example, without the constraint, we could end up with the taken transitions during the simulation of the same architecture: *dp2dp*, *stutter*, *stutter*, *dp2dp*. One *dp2dp*-transition is replaced by a stutter transition. This results in an incomplete simulation.

5.4.1 Example simulation

To give a clearer overview of the parallel simulation, we give a comparison of the basic simulation versus the parallel simulation in figure 5.3. The architecture



(a) Resulting trace from basic simulation



(b) Resulting trace from parallel simulation

Figure 5.3: Simulation results of basic simulation and parallel simulation

in the figure consists of one device, and two processes which are refined to the subprogram level. The outgoing event data port of the device is connected to both the incoming event data ports of the processes.

Traces found by both the simulations are given in the figures. As shown in figure 5.3.a, the trace consists of four distinguishable states. In the initial state, S_0, a message is bound to the outgoing event data port of the device. In state S_1, the message is transferred to the incoming event data ports of both threads. Then, in S_3, the message is transferred to the incoming parameters of both subprograms. Both subprograms are also scheduled for dispatch, although this is omitted in the figure. Finally, in $S_4$the subprograms have been dispatched and the message are consumed by the subprograms. The simulator only returns this trace, no different interleavings are found. This is due to the fact that all events happen at the same time in the architecture.

Figure 5.3.b gives one of the six traces found by the parallel simulation. Again, there is an message bound to the outgoing event data port of the device. Equal to the basic simulation, the message is transferred from the device to the incoming event data ports of the threads, in state S_1. However, in states S_2 to S_3, the upper message is transferred to the upper subprogram and the upper subprogram is dispatched, while the lower message remains at its position. From state S_4 to S_5, the lower message is transferred to the lower subprogram and the lower subprogram is dispatched.

5.5 Benchmarks

Two benchmarks are performed to investigate the properties of the basic and parallel versions of the simulators. Two variables are tested in two benchmarks. The first variable the number of states specified in the simulation. We define the **number of states** in the simulation as follows: the number of states the simulation is enforced to explore. This can include stuttering states.

The second variable is the size of the architecture. We define the size of the architecture as follows: number of component instances + number of feature instances + number of port connections. The number of component instance, feature instances, and port connections are extracted from the AADL instance models.

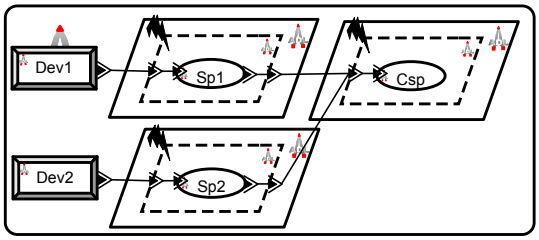


Figure 5.4: Architecture used for benchmarking

The benchmark tests the time in milliseconds needed for the simulation, dependent on the variables. Memory consumption is not measured. We run each benchmark for both the basic version and the parallel version. The runs for each benchmarks are executed three times. The average is taken from the three runs.

The benchmarks are done on a Intel Core 2 Duo, Q9400, running at 2.66GHz, and 4GB of memory, running Windows XP Professional, service pack 2. We use version 4.1.10 of the Alloy Analyzer, together with the Sun Java Runtime Environment 1.6.0_13. The memory limit for the Alloy Analyzer is set to 1024M. We use the MiniSat solver.

The benchmarks are based on a simple architecture, which is given in listing 5.4. The architecture consists of two data flows, starting at the devices *Dev1* and *Dev2*, respectively, and ending at subprogram *Csp*. The devices are connected to the subprogram of two processes (*Sp1* and *Sp2*, respectively), by an event data port connection. These subprograms are connected to the central subprogram *Csp* through an event data port connection. The subprograms *Sp1* and *Sp2* simply forward received message from the device to the *Csp* subprogram. The subprogram *Csp* consumes the messages it receives.

The two variables seem closely related to each other. If the architecture size grows, it is likely that the number of states required for complete simulation also grows. However, this does not always have to be the case. For example, if we introduce new functionality to the architecture, not related to the existing functionality, we do not have to increase the number of states for the existing scenarios. Given the initial state of the existing scenarios, and the unaltered relevant part of the architecture for those scenarios, the simulation remains unaltered. Only if existing functionality is extended, we need to increase the number of states for the simulation.

5.5.1 Number of states

The length of the simulation is defined by the number of states the simulator is forced to explore. In this benchmark, we use the proposed architecture, and execute the scenario. For each subsequent run, we increase the number of states by 10. The first run simulates 10 states; the last run simulates 100 states. We expect that when the number of states is increased linearly, the time taken to complete the simulation increases exponentially.

The results of the benchmark are shown in figure 5.5. The figure shows the expected results. The time taken for the parallel simulator grows faster than the time taken for the basic simulator, if the number of states is increased

Number of states	Simulation time (ms)
10	240
20	391
30	578
40	802
50	1078
60	1349
70	1651
80	2015
90	2375
100	2781

(a) Basic simulation, architecture size = 38

Number of states	Simulation time (ms)
10	5364
20	14213
30	29910
40	53713
50	82718
60	105833
70	143729
80	180360
90	229198
100	300922

(b) Parallel simulation, architecture size = 38

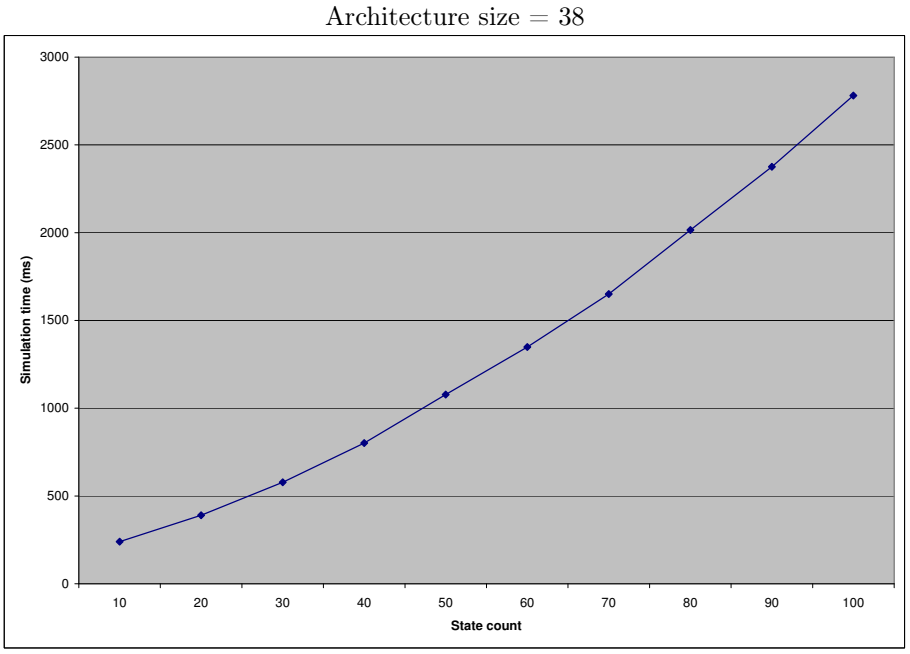
Table 5.1: Benchmark 1 results

linearly. This can be explained by the fact that the parallel simulator has multiple transition functions versus the single transition function of the basic simulator. For each subsequent state generated by the parallel simulation, the Alloy Analyzer has to test if either of the transition functions match. For the basic simulator, the Alloy Analyzer only has to test one transition function.

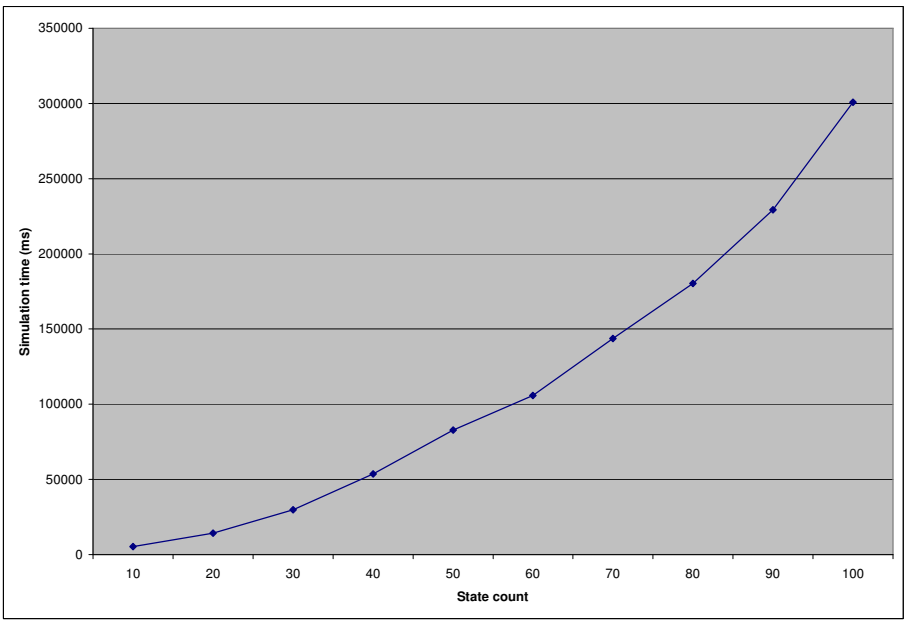
5.5.2 Size of architecture

Each state can hold several bindings. An example is the binding of a data value to a data port. The more architecture elements there are, the more potential bindings exist. This benchmark tests the size of the architecture versus the simulation time. The number of states for each run is equal. The earlier introduced architecture (figure 5.4) is used. We start with an architecture which consists of only one data flow. This means that the components *Dev2*, and *Sp2* are removed for the first run. We increase the size of the architecture by adding a device, a process (refined to the level of subprogram), and port connections to the architecture. For each run of the architecture, we add three devices, three processes, and connect the introduced devices to the introduced processes, and the introduced processes to the existing central process.

For the basic simulation, the transition function is extended. For the par-



(a) Basic simulation



(b) Parallel simulation

Figure 5.5: Benchmark 1 results

Architecture size	Simulation time (ms)
23	255
68	2646
113	11427
158	33406

(a) Basic simulation, states = 30

Architecture size	Simulation time (ms)
23	17708
68	55312
113	131505
158	264626

(b) Parallel simulation, states = 30

Table 5.2: Benchmark 2 results

allel version of the simulator, the extra subprogram results in the addition of a transition function. Given that we increase the number of potential transition functions, we expect to see a steep growth in simulation time for the parallel simulator. For the basic simulator, we expect an exponential growth as well, but less steep.

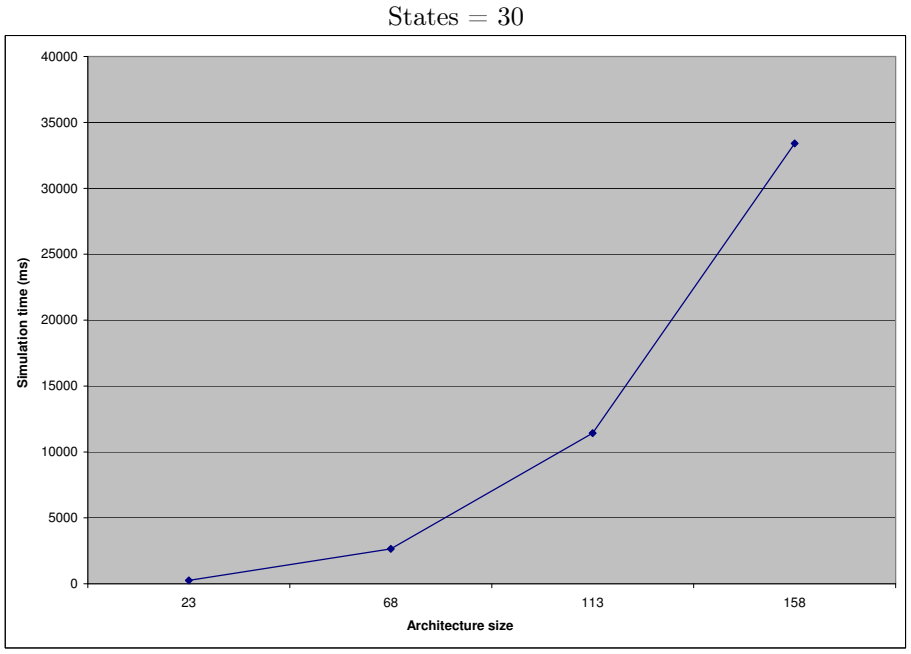
The results of the benchmark are shown in figure 5.6. As expected, the time taken by the parallel simulator grows faster than the time taken by the basic simulator. This is due to increased complexity of the simulation, by the addition of a transition function. Given the increased number of elements in the architecture, the number of signatures in the Alloy model is increased as well. This results in more potential bindings per state, as explained before. As such, there are more potential states to be examined by the Alloy Analyzer.

5.6 Limitations of implementation

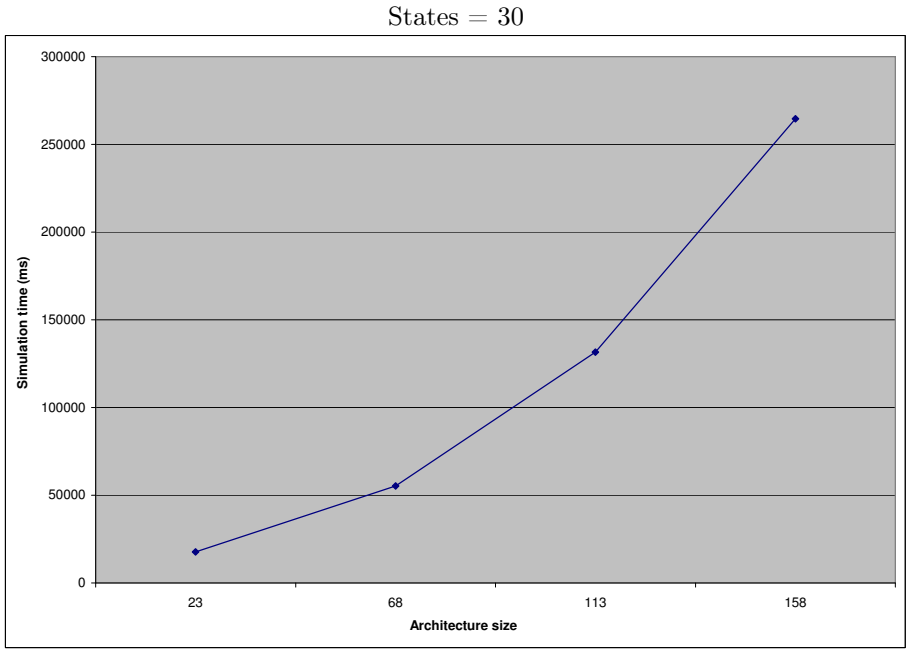
The implementation of the simulator in Alloy has shortcomings. The shortcomings are discussed in this section.

Event and event data ports cannot hold multiple values in the basic version of the simulator. In the AADL, an event port and event data port are queues which can hold multiple values. A queue is an ordered collection. Given that multiple events can occur at the same time (for example, an event data port receives two values at the same time), no order of arrival can be specified and therefore we cannot make any statements about this order. The parallel version of the simulator supports queues.

If an event can happen in the architecture at a state, it will have happened after the next transition. This means that two data values can be transferred at the same time during the simulation. No different interleavings are found through the basic simulation. As such, it is harder - if not impossible - to test any properties related to concurrency and synchronization. The parallel version of the simulator does find all interleavings and thus makes it possible to reason about properties related to concurrency and synchronization.



(a) Basic simulation



(b) Parallel simulation

Figure 5.6: Benchmark 2 results

The detection of the occurrence of environment or architecture events during the simulation is not supported. The effects of environment or architecture events are described by the initial state of a scenario. We are not able to introduce the observation of an event during the simulation. This means we are not able to simulate two events being observed at different times. We can, however, simulate the behavior of the architecture after the observation of two events at the same time.

The process of checking/finding a model using the Alloy Analyzer consists of two steps. The first step is generating the CNF formula. CNF stands for conjunctive normal form, and is a standard notation for SAT-solvers. The second step is the actual solving. For the RPM architecture, found in appendix B, generating the CNF formula takes by far the largest amount time. As an example, for generating a state space consisting of 25 states, the CNF generation step takes 25 seconds on the benchmark machine, while the solving step takes less than one second. Time improvements can be gained by replacing fields from signatures by functions. The portConnections-function found in section 5.2 is an example of this. When using the function instead of a PortConnection signature, CNF generation time is reduced dramatically.

Before we can run a simulation, we need to estimate the number of states which will be required for the full simulation. If the estimated number of states is too low, the simulation will be incomplete. If the estimated number of states is too high, running the simulation/finding the state space will take longer than necessary. Also, we need to give a strict scope for the data values used. Exactly specifying this can be a hard task, if the exact behavior of the subprograms is unknown.

5.6.1 Size of architecture and state space

The Alloy Analyzer imposes limits on the number of atoms and the arity of the largest relation in the model. In the parallel simulation, the State signature contains the field *eventDataPortValues*. This field results in a relation with the arity of 4 (*State*→*EventDataPort*→*Int*→*DataType*). As a result, the Alloy Analyzer calculates the following Cartesian product: $atoms \times atoms \times atoms \times atoms$, where *atoms* is the set of all the atoms in the model. The size of the resulting set has to be smaller than 2^{31}, or the Alloy Analyzer will return an error. As a result, the number of atoms for the parallel simulator can at most be 215 ($215^4 \leq 2^{31}$). Note that each state also results in a single atom. Also, the Alloy Analyzer introduces 16 atoms through the Int signature. In the basic simulator, the largest relation has an arity of 3. These relations result from the fields, expect for *toDispatch*, from the *State* signature. By applying the same reasoning, we find the limit of 1290 atoms ($1290^3 \leq 2^{31}$).

Each feature and component in the architecture results in a single atom. Also, each state introduces an atom. This allows us to identify an upper limit of the architecture and the state space. Using the basic simulator, given the largest relation with an arity of 3, the maximum number of atoms is $1290 - 16$. The 16 atoms are introduced by the Alloy Analyzer through the Int signature. This gives us the following limit for the basic simulator:

$$architecture\,size + number\,of\,states \leq 1290 - 16$$

For the parallel simulator, given the largest relation with an arity of 4, the maximum number of atoms is 215. This gives us the following limit for the parallel simulator:

$$architecture\,size + number\,of\,states \leq 215 - 16$$

5.7 Conclusion

This chapter describes the simulation of AADL architectures. In section 5.2, the representation of the architecture structure in Alloy is given. Signatures are introduced which represent the AADL components, features, connections, and data types. Facts are added to constrain the Alloy models, such that only valid architectures can be modeled.

Section 5.3 introduces a discrete event simulation for the simulation of AADL architectures. Each state describes the loci of the data values in the architecture. A single transition function is given which describes the transition from one state to another state. This transition function captures all of the events which can occur in the architecture. Different events are given which describe the behavioral semantics of the architecture. One part of the behavioral semantics is extracted from the AADL standard. The behavioral semantics are formalized using several Alloy functions. The second part of the behavioral semantics of the architecture is introduced by the subprograms. Architecture invariants are introduced to validate the correctness of the behavior of subprograms and data contained by data instances. Scenarios are used to describe the initial state after the system observes an event in either the environment or in the architecture itself.

In section 5.4, we describe an extension of the discrete event simulation. The extended version allows for a simulation which performs all events atomically. Different interleavings now come forward in the state space. Based on the different interleavings, properties related to concurrency can now be verified in a more elaborate manner.

Section 5.5 benchmarks the two different simulation types. Limitations of the simulations are identified and discussed in section 5.6. The benchmarks show that simulation of the architecture becomes exponentially harder when the size of the architecture, or the number of states for the simulation is increased linearly. Also, a hard upper limit is identified with regard to the size of the architecture and the number of states used for simulation. This raises the question whether Alloy is suitable for simulation of architectures.

Chapter 6

Performing change impact analysis

6.1 Introduction

This chapter elaborates on performing change impact analysis in software architecture. Functional requirements are validated against the behavior of the architecture. Using this approach, we can find impacted components.

Section 6.2 elaborates on the formalization and validation of functional requirements. In section 6.3 we use the evaluation of functional requirements to perform change impact analysis. The induction of elements in architectures to satisfy requirements is given in section 6.4. Finally, section 6.5 concludes this chapter.

6.2 Validation of functional requirements

This section elaborates on the approach given in section 3.2.

For an architecture to be valid, the architecture must satisfy its requirements. A graphical overview is given in figure 3.1. In this research, we only consider functional requirements. Functional requirements specify required behavior in the architecture. An example of a functional requirement, found in appendix B in table B.1, R2, is that when a measurement is performed at the patient, the measurement must be transferred to a central system. A functional requirement can also restrict behavior. The restricted behavior is not allowed to occur in the architecture. If such behavior does occur in the architecture, the architecture does not satisfy the requirement.

The behavior of the architecture is recorded by the state space, as specified in figure 3.2. The state space is derived from the architecture by the use of simulation, found in chapter 5. Before the state space can be validated against the requirements, we need to re-formulate the requirements into predicates.

6.2.1 Formalization of functional requirements

This subsection elaborates on the formalization of requirements, as given in subsection 3.2.2.

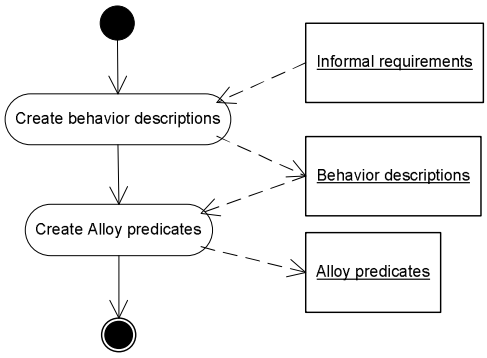


Figure 6.1: Activity diagram for formalization of functional requirements

The functional requirements found in the requirements document are often textual specifications. These textual specifications are not formalized and not interpretable by a program. Also, these requirements are in the problem domain. To be able to verify these informal functional requirements against the architecture, two steps need to be performed. The first step is the formalization of the functional requirements such that requirements can be validated against the state space. This step is represented by the activity *Create behavior descriptions* in figure 6.1. The activity takes the *informal requirements* and produces *behavior descriptions*.

Using Alloy as a formalism, we translate informal requirements into Alloy predicates which asserts allowed or disallowed behavior of the architecture. This step is represented by activity *Create Alloy predicates* in figure 6.1. The activity takes the *behavior descriptions* and produces *Alloy predicates*. Using the simulator, we can test the predicates against the generated state space. Figure 6.1 gives the complete activity diagram for the formalization of functional requirements.

Behavioral patterns

We define a number of patterns which describes properties of behavior over an architecture through behavior descriptions. The first pattern describes the properties that must hold for a single state. An example is that at one state, there must be a data value at a specific data port. There are variations of this pattern: the property must hold for no state, some states, and all states.

The second pattern describes properties which must hold for a sequence of states. Usually, the sequence only consists of two ore more directly connected states. A variant of this pattern is that the states do not have to be connected directly, but are connected indirectly.

To capture these behavior descriptions, we define the following format:

- Data type;
- From port (feature instance in model);
- Next or subsequent state ($\rightarrow$ or $\Rightarrow$, respectively);

- To port (feature instance in model).

The next or subsequent state specifies whether the data value must be at the to-port in directly the next state, or must be at the to-port in a subsequent state. If the next or subsequent state and to-port are not present, it means that the binding of a data value to the from-port must be satisfied for only one state.

These patterns can be regarded as a form of leightweight LTL[Muk97]. The only operators we are using are the next operator, and the future operator. The patterns can be extended if needed.

From problem domain to solution domain

The translation from the problem domain to the solution domain (architecture) is performed by hand. The architect knows both the problem and solution domain and thus can translate between these two domains. When building the architecture, the architecture records the rationale for all the elements in the architecture. The rationale for each element is (at least) backed up by one or more requirements. These are also the traces from requirements to architecture, used in subsection 6.2.3. In appendix B, the informal functional requirements are given in table B.1. From these informal functional requirements, we extract informal, but structured, behavior descriptions, conformant to the previously defined patterns. These behavior descriptions live in the solution domain, and thus describe properties over the architecture elements. Table 6.1 gives the behavior description(s) for requirement R1, based on the example architecture found in section 4.4. The behavior description states that in one state, there must be a temperature measurement at the port *sd.sensor1.measurement*, and in s subsequent state, the temperature measurement must be at the *sd.proc1.thread1.c1.measurement_out* port. Table 6.1 also gives the refinement of the requirement R1. The behavior descriptions for all the requirements can be found in table B.2.

Req.	**Ref. req.**	**Description**
R1		Temperature measurement: sd.sensor1.measurement ⇒ sd.proc1.thread1.c1.measurement_out
	R1_a	Temperature measurement: sd.sensor1.measurement → sd.proc1.thread1.c1.sensor1_in
	R1_b	Temperature measurement: sd.proc1.thread1.c1.sensor1_in → sd.proc1.thread1.c1.measurement_out

Table 6.1: Behavior description for R1

From behavioral descriptions to Alloy predicates

The two proposed patterns can easily be translated to Alloy predicates. The first pattern describes a property over no, one or more states. The Alloy predicate constrains the state space such that there must be exactly one state for which the formula holds. The resulting predicate constraining the state space is as follows:

```
pred R {
        one s: State | formula-over-s
}
```

The variants of the first pattern can be created by replacing the one-quantifier in the predicate to *no*, *one*, *some*, or *all*.

The second pattern describes that a property at one state (s) a formula must hold, and in the next state ($s.next$) another property must hold. The predicate constraining the state space is as follows:

```
pred R {
        one s: State | formula-over-s and formula-over-s.next
}
```

The variant of this pattern which specifies that the states do not have to be directly connected. The difference is in the number of allowed transitions between the two states. The variation allows for multiple transitions to occur between the current and subsequent states. Alloy allows for this construct using the non-reflexive transitive closure operator (^). The future state is specified as ($s.\hat{}next$). The predicate is as follows:

```
pred R {
        one s: State | formula-over-s and formula-over-s.^next
}
```

Adding traces

Given the translation from the requirements (problem domain) to the architecture components (solution domain), we define traces between the requirements and the predicates, and the predicates and the architecture. These traces are used when interpreting the counter example and discovering the impacted components, as shown in section 6.2.3.

The translation from the problem domain to the solution domain is already performed when describing the formal behavior descriptions, based on the functional requirements. The formal behavior description describes the ports, which are involved in the behavior. These ports are connected to a specific component. The components that contain the port are the components that the formal behavior description (and the requirement) traces to.

A single requirement can result in multiple formal behavior descriptions. Thus, a single requirement can trace to multiple formal behavior descriptions. Each formal behavior description is transformed into a single predicate. Figure 6.2 gives the relations between the informal functional requirements, formal behavior descriptions and predicates.

It has to be emphasized that defining the traces is an important step. The coarseness of the traces from requirement to architecture components is important with regard to the accuracy of the change impact analysis results. If a requirement traces to many components, the not-satisfaction of that requirement by the architecture will result in a large number of impacted components and thus the results are likely to be inaccurate.

Example

Listing 6.1 shows the requirement predicate for requirement R1. This requirement is based on the formal behavior description given in table 6.1, and uses

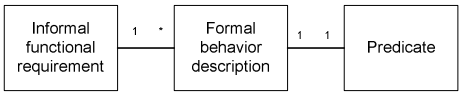


Figure 6.2: Trace-relations

Listing 6.1: Requirement predicate R1

```
pred R1 {
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.^
        next.eventDataPortValues)
}
```

the variation of the second behavioral pattern. The predicate states that there must be a state for which there is a data value bound to the event data port *sd.sensor1.measurement*. In a future state (*s.^next*), the value must be bound to the event data port *sd.proc1.thread1.c1.measurement_ out*.

In listing 5.20 an example trace is shown. The trace shows the behavior of the example architecture, introduced in section 4.4. The behavior required for requirement R1, as found in table B.1 of appendix B, is visible in first four states of the trace. The example trace can be used to create the behavior descriptions and Alloy predicates. However, it should be noted that using the existing behavior can introduce a bias towards the existing behavior, compared to the wanted behavior as expressed by the functional requirements.

6.2.2 Validation of formalized requirements

This subsection elaborates on subsection 3.2.3. In that subsection we gave a definition for the term scenario. We extend this definition in this subsection.

Using the formalized requirements, we can validate the behavior of the architecture against the predicates (formalized requirements). Alloy can be used to find models which violate an assertion. Using the simulator, if an assertion does not hold, Alloy will give a counter example. If an assertion is not violated, no counter example is given.

We extend of scenarios, as described in section 5.3.5, to validate the requirement predicates. Like the validation of architecture invariants, we can use scenarios to validate the behavior of an architecture. A scenario describes the initial state which is used by the simulation. We extend the scenario by validating that one or more requirements must hold over the derived state space. We formulate this as an Alloy assertion. The assertion states that if the initial state holds, the requirements must also hold. If one or more of the of the requirement predicates do not hold, a counter example is given by the Alloy Analyzer. The new definition for scenario is as follows: A *scenario* contains the initial state of the architecture after a stimulus in the environment has been detected, and one or more requirements which must hold during the simulation of the architecture. Furthermore, scenarios are used to test the architecture invariants during the simulation.

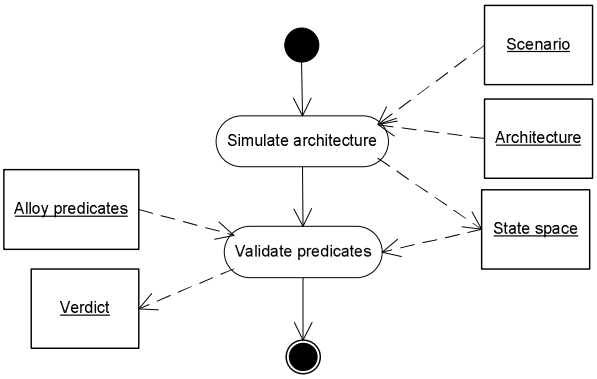


Figure 6.3: Activity diagram for validation of requirements

Figure 6.3 gives the activities for the validation of the requirements encoded as predicates. The activity diagram consists of two activities: *Simulate architecture*, and *Validate predicates*. The *simulate architecture* activity consists of simulating the architecture by use of the Alloy Analyzer. This activity takes the *architecture* and one *scenario* as input. The output of the activity is the *state space*. Because we use the Alloy Analyzer, the state space consists of multiple traces. These traces are returned by the Alloy Analyzer as individual traces, and not as a complete state space.

The second activity, *validate predicates*, takes the *state space* and the *Alloy predicates* (formalized requirements) as an input. These output of the activity is a *verdict*. This verdict state whether all the *Alloy predicates* are satisfied by the *state space*. If one or more predicates do not hold, the verdict is negative. In this case, the *verdict* consists of the state space for which one or more predicates are not satisfied. If all predicates are satisfied, the verdict is that all predicates are satisfied and no state space is returned.

An example of a scenario and validating requirements is given in listing 6.2. This scenario is used to test the requirement R1 the example architecture found in section 4.4. The example scenario consists the predicates *R*, *scenario1_ initial*, and the assertion *Check_ scenario1*. The predicate *R1* is a partial formalization of requirement R1, found in table B.1. The predicate describes that at one state, a measurement is received by the subprogram of the *SD* system. And at a future state, the subprogram of the *SD* must forward the measurement through its outgoing *measurement_ out* event data port. The *scenario1_ initial* predicate specifies the initial state. Finally, the assertion *Check_ scenario1* provides the check-command for the Alloy Analyzer. The check-command validates that the predicates *R1* and *ArchitectureInvariants* hold, over the state space. If either of those predicates does not hold, a counter example is returned by the Alloy Analyzer.

6.2.3 The use of counter examples

In this subsection, we elaborate the use of counter examples. The activity diagram found in figure 6.4 continues after the activity diagram presented in figure 6.4. If the verdict of activity diagram in figure 6.4 is that one or more predicates

Listing 6.2: Scenario to validate a requirement

```
pred R1 {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.^
        next.eventDataPortValues)
}
pred scenario1_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch

  one s.eventDataPortValues
  one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
      eventDataPortValues).zaM
  one m: Measurement_i | m.zaM = ZigBeeAddress_i
}
Check_scenario1 : check {
  (StateTransitions and scenario1_initial[so/first]) =>
  (ArchitectureInvariants and R1)
} for
  0 DataType, 0 Component, 0 Feature, 0 Connection,
  exactly 25 State,
  exactly 1 Measurement_i, exactly 1 ZigBeeAddress_i
```

Listing 6.3: Querying a counter example

```
R1
  false
```

are not satisfied, this activity diagram is executed to find the components which are malfunctioning. If the verdict of the activity diagram in figure 6.4 is that all predicates are satisfied, no further action is taken.

The activity diagram for using counter examples is given in figure 6.4. Two activities are given in the activity diagram: *Find violated predicates*, and *Trace predicates to components*. The first activity, *Find violated predicates*, takes the *counter example*, and *predicates* as input. The *counter example* consists of the state space for which one or more predicates were not satisfied. The *predicates* are the predicates which represent the formalized requirements. The output of the activity is the set of *violated predicates,* i.e. the predicates which are not satisfied by the *counter example*.

We use the evaluator of the Alloy Analyzer, given in section 4.5.5, to find the predicates which are not satisfied by the counter example. An example based on the example architecture, found in section 4.4, of testing requirement predicates is given in listing 6.3. In this example, we have removed the connection between the *sd.sensor1.measurement* and *sd.proc1.thread1.sensor1_in* ports, as shown in figure 6.6. As a result, we expect that requirement predicate *R1* will fail. The listing shows that this is the case, after querying the predicate *R1*.

The second activity, *trace predicates to components*, takes the set of *violated predicates* and *traces from predicates to components* as input. The activity

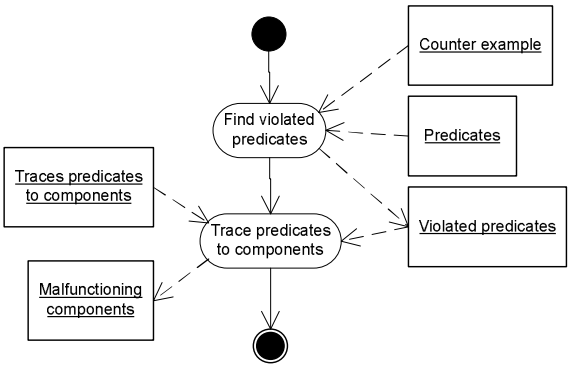


Figure 6.4: Activity diagram for the use of counter examples

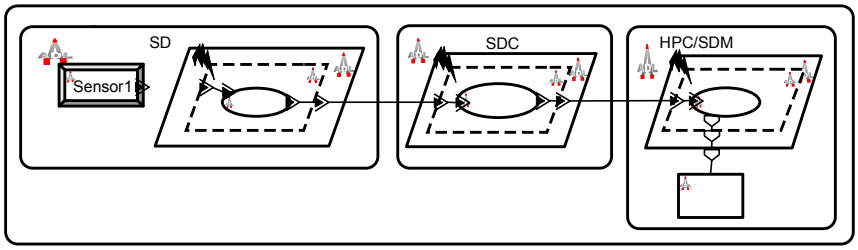


Figure 6.5: Example architecture with removed port connection

traces from the requirements to the components which are not satisfying the requirements. The result of the activity is the set of *malfunctioning components*.

The statement "garbage in = garbage out" is important when defining and using the traces from requirements to architecture. If the requirements trace to a large number of the architecture components, it is hard to identify the component which is causing the invalid behavior.

The counter example consists of a trace describing the behavior of the architecture. The names of the Alloy signatures can easily be mapped back to their respective AADL component instance names. As given in section 5.2, the naming scheme consists of replace dots with underscores. Reversing this replacement results in the AADL component instance names, which are used to identify the components in the AADL instance model.

For example, requirement *R2* of the RPM project, found in table B.1, states that temperature measurements must be transferred from the patient to the central system. The relevant part of the architecture is given in figure 4.7. A coarse grained version of the formalized requirement, given in listing 6.4, specifies that at a state (s) there must be a value at the *measurement_ out* port of the *SD* system, and in a future state ($s.\hat{}next$) there must be a value at the *measurement_ in* port of the *SDM* subprogram. If we inspect the architecture, we see that the *SDC* system is also involved in the measurement data flow. As a result, the *SDC* component has to be included in the traces from requirement *R2* to the architecture components. When the *SDC* system is malfunctioning and the measurement data value is never forwarded, we can only state that the *SD* and *SDC* systems and the *HPC.SDM* process are impacted. It is not

Listing 6.4: Coarse grained version of requirement R2

```
pred R2 {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(s.^
        next.parameterValues)
}
```

possible to pinpoint the *SDC* directly.

To overcome the lack of preciseness, we have to decrease the granularity of the trace from predicate to architecture component. This can be done by splitting the requirement into multiple predicates. To improve the previous example, we define predicates which test if the measurement is transmitted from the *SD* to the *SDC*, handled by the *SDC*, and transmitted from the *SDC* to the *HPC/SDM*. A more elaborate version formalized requirement is provided in listing B.2. If we get a counter example, we test the predicates *R2_ a*, *R2_ b*, or *R2_ c*. If *R2_ a* holds, but *R2_ b* does not hold, we know the *SDC* is not handling the measurement correctly.

A note with regard to dependencies between the formalized requirements has to be made. In case of *R2*, the *SDC* can only handle the measurement if it actually receives it. So *R2_ b* is dependent on *R2_ a*. These dependencies between requirements are not considered in this research.

6.3 Performing change impact analysis

This section elaborates on the process given in section 3.3. Using the validation of functional requirement over an architecture, as given in section 6.2, we can perform change impact analysis in architecture.

Change impact analysis has to be performed in two cases. In the first case, the requirements evolve and the architecture remains the same. This case is given in figure 3.4. As the architecture is unchanged, the behavior of the architecture, and thus the state space, also remains the same. The evolved requirements need to be re-formalized and tested against the state space to find the state space which violates one or more requirements, if there are any.

In the second case, the architecture evolves while the requirements remain unchanged. This case is given in figure 3.5. As the architecture is changed, its behavior is also likely to be changed. As a result, the generated state space based on the architecture is also changed. The (unchanged) formalized requirements have to be validated against the new state space. Re-running the simulator will give state space as a counter example if any of the existing requirement predicates (formalized requirements) fail.

Note that the traces from the requirements to the architecture also need to be checked and, where required, updated, in either case. Evolved requirements may be traced to additional or other architecture components. For example, functionality may be moved from one component to another component in the architecture.

6.3.1 Iterative process

The process of performing change impact analysis is an iterative process. If a component is changed, the data flow through this component may be changed as well. For example, it can send data to another output port. As a result, the architecture has to be re-simulated to generate the new state space. The activity diagram given in figure 3.6.

The process starts by applying a change to the architecture. Along with changing the architecture, the formalized requirements might need to be updated as well, along with the traces from the requirements to architecture.

The (new) architecture can now be validated against the (new) formalized requirements by running the simulator again. The simulator will generate the (new) state space, and test the (new) formalized requirements against it. If all formalized requirements are satisfied by the state space, the simulator will not return a counter example. If the state space does not satisfy the formalized requirements, the simulator returns a counter example. Using the counter example, the formalized requirements, and the traces from formalized requirements to architecture, we can identify the components which do not satisfy the requirements. Individual formalized requirements can be queried against the counter example to identify the failing formalized requirements. When one (or more) failing formalized requirements are identified, the components are identified by using the traces between the requirements and architecture elements.

If a failing component has been identified, the component is suspect to change. It is possible that the identified component has to be updated. When the component is updated, the whole process has to be performed again, as the behavior of the architecture might have changed. If the last change results in a satisfying architecture, the process is complete.

6.3.2 Example

In chapter 7, we perform a case study. This subsection will give a short example of performing change impact analysis, using one of the change scenarios from that chapter.

In this example, we introduce three new requirements. The requirements R1 to R3, found in table B.1 of appendix B, state that the system must be able to measure temperature and store those measurements. We copy requirements R1 to R3, and replace 'temperature measurement' by 'blood pressure measurement'.

The iterative process starts by changing the example architecture, as found in section 4.4. We add a blood pressure sensor to the *SD* system, similar to the temperature sensor. The added blood pressure sensor is connected to the subprogram of the *SD* system. The resulting architecture is given in figure 7.2.

We formalize the new requirements. The new formalized requirements are similar to the formalized requirement for the temperature-related requirements, but constrain the state space with regard to the behavior of the new blood pressure sensor and its measurements. The formalized requirements are given in listing 7.1. The formalized requirement *R1'_a* traces to the components *sd.sensor2* and *sd.proc1.thread1.c1*, given in table B.4. The requirement predicate *R1'_b* traces to the *sd.proc1.thread1.c1* subprogram.

Simulating the changed example architecture, and validating the new requirements, the Alloy Analyzer returns a counter example. The requirement

predicate *R1'_ b* is not satisfied by the architecture. Using the traces, we find the *sd.proc1.thread1.c1* subprogram, and mark it as impacted. The subprogram is investigated in the next iteration of the process.

This process is repeated until the requirements are satisfied by the architecture.

6.4 Induction of elements in architectures

Alloy models are defined by signatures, facts, and a scope. Alloy tries to find instances of the model which conform to the restrictions found in the model. A scope limits the number of instances generated from a signature.

For the simulation of the architecture, the structure of the architecture is static. The scope of the static structure is strict in the sense that no additional components can be introduced by Alloy. The dynamicity of the architecture in the simulator is in the generated state space. For each state, for example, different relations from ports to data values are tested such that these conform to the defined models.

We can use Alloy to try to find new components, for example. If we increase the scope and specify that all the found models must conform to a given scenario, we let Alloy find models (architectures) which conform to the scenario. If a model is found, we can extract the new induced elements from the model, which might provide a possible fix. We define **induction** as trying to find an architecture which satisfies its requirements by increasing the scope with regard to a certain element type.

6.4.1 Induction of port connections

In chapter 5 we have given the formalized architecture. Port connections are recorded through the use of a function *portConnections[]*. This function provides a set of binary relations between two ports, which represent the port connections in the architecture. The relations returned by the function are entirely static. Thus no new relations can be induced by Alloy, as Alloy can only increase the scope for signatures, and not relations returned by a function. As such, we need an abstract *PortConnection* signature which provides a means to record the port connections in the architecture. This abstract signature was already given in listing 5.5, but is given again in listing 6.5. The source port of the port connections is recorded by the *src* field, the target port of the port connection is recorded by the *dst* field.

Given the new PortConnection signature, we need to update the existing behavioral semantics to include the new *PortConnection* signature. In our case, we need to update the transitions which transfer data from one port to another port, over a port connection. The updated semantics for the transfer of a data value from one event data port to another event data port are given in the second part of listing 6.5. The new *edp2edp* function takes the original *portConnections[]* function as well as the new PortConnection signature into account.

If we set the scope for the *PortConnection* signature to zero or one, Alloy will try to find all models which satisfy the scenario predicates and architecture invariants. The scope may be increased to induce more port connections. If

Listing 6.5: PortConnection signature and updated semantics

```
abstract sig Connection { }

abstract sig PortConnection extends Connection {
  src: one Port,
  dst: one Port,
}

fun edp2edp[r: EventDataPort -> DataType] : EventDataPort ->
    DataType {
  { edp: EventDataPort, dt: DataType |
    one (edp.~portConnections[] -> dt & r) } +
  { edp: EventDataPort, dt: DataType | one pc: PortConnection |
    edp = pc.dst and one (pc.src -> dt & r) }
}
```

Listing 6.6: Changed scenario predicate

```
Run_scenario1 : run {
  (StateTransitions and scenario1_initial[so/first])
  (ArchitectureInvariants and
  R1 and R2 and R3)
} for
  ...,
  exactly 1 PortConnection
```

Alloy finds a satisfying model, it will be returned and allows us to query the new induced PortConnection signature, together with its relations, the source and target port.

It should be noted that we can only infer new port connections, and not alter existing or removing port connections all together. The *portConnections[]* function itself is static and cannot be altered during the solving process. Also, only new PortConnection signature instances can be inferred.

As an example, we take the architecture given in section 4.4 and remove the port connection between *sd.proc1.thread1.measurement_out* and *sdc.proc1.thread1.measurement_in* event data ports, by removing the appropriate line from the *portConnections[]* function. The architecture is given in figure 6.6. We extend original scenario 1 (found in listing B.2). We replace the implication by a formula which states that the predicates *StateTransitions*, *scenario1_initial*, *ArchitectureInvariants*, and the requirements must hold. Furthermore, we increase the scope by adding a port connection signature. The resulting command is given in listing 6.6.

The Alloy Analyzer will try to find an architecture and state space for which the architecture invariants and requirements R1, R2, and R3 hold. The Alloy Analyzer is allowed to introduce a port connection to the architecture. Running the *Run_scenario1* command, the Alloy Analyzer finds a model for which the architecture invariants and requirements are satisfied. We can query this model using the evaluator of the Alloy Analyzer, as shown in listing 6.7. The *src* field of the *PortConnection$0* signature atom gives the port *sd.proc1.thread1.measurement_out*. The *dst* field of the *PortConnection$0* signature atom gives the port

Listing 6.7: Querying the Alloy architecture model for the induced port connection

```
PortConnection
  aadl_meta/PortConnection$0
aadl_meta/PortConnection$0.src
  {RPM_RPM_i_Instance_sd_proc1_thread1_measurement_out}
aadl_meta/PortConnection$0.dst
  {RPM_RPM_i_Instance_sdc_proc1_thread1_measurement_in}
```

Listing 6.8: DataAccessConnection signature and getDataInstance function

```
abstract sig Connection { }

abstract sig DataAccessConnection extends Connection {
  src: one DataInstance + DataAccess,
  dst: one DataAccess,
}

fun getDataInstance[da: one DataAccess] : DataInstance {
  da.~^dataAccessConnection & DataInstance +
  { di: DataInstance | one dac: DataAccessConnection | dac.dst = da
      and dac.src = di }
}
```

sdc.proc1.thread1.measurement_ in. This is the port connection which we have removed earlier.

6.4.2 Induction of data access connections

The same approach taken for inducing port connections can be taken to induce data access connections. The same function-construct is used to record data access connections. This function provides a set of relations from a data instance or data access, to another data access. The signature, already given in listing 5.5, is listed again in listing 6.8 is used to record data access connections.

As with the port connection induction, the semantics need to be updated. In this case, the updated semantics are entirely structural. The *getDataInstance* function is used to get a data instance component which is (indirectly) from a data access feature. The second line in the function has been added to include the *DataAccessConnection* signatures. Again, we need to increase the scope such that one or more additional data access connections are induced, such that Alloy will try to find models which satisfy all the predicates. If Alloy finds a satisfying instance, it will return the instance and the newly induced data access connection can be extracted from it.

Only new data access connection can be induced. Existing data access connections cannot be altered or removed. The reason is the same as for the port connections, the use of the static dataAccessConnection function.

As an example of inducing a data access connection, we take the example architecture and remove the data access connection between the *hpc.temperature_ measurements* data instance and the *hpc.sdm.proc1.temperature_ measurements* data access. The resulting architec-

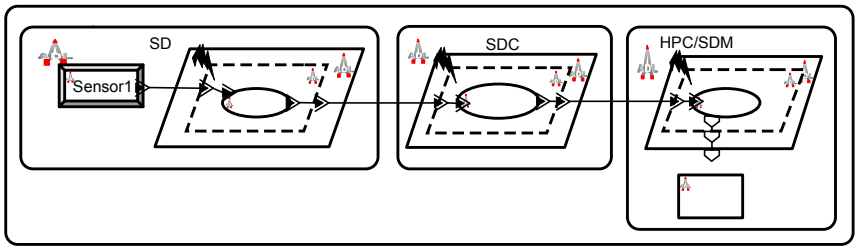


Figure 6.6: Example architecture with removed data access connection

Listing 6.9: Querying the Alloy architecture model for the induced data access-connection

```
DataAccessConnection
  aadl_meta/DataAccessConnection$0
aadl_meta/DataAccessConnection$0.src
  {RPM_RPM_i_Instance_hpc_temperature_measurements}
aadl_meta/DataAccessConnection$0.dst
  {RPM_RPM_i_Instance_hpc_proc1_temperature_measurements}
```

ture is given in figure 6.6. We use the a scenario similar to the one found in listing 6.6, but change the scope such that instead of a PortConnection atom, it will add a DataAccessConnection atom.

After executing the scenario, we query the found model and find the data access connection which was originally removed, as found in listing 6.9.

6.4.3 Induction of other elements

In the current form of the architecture and simulation, it is not possible to induce other elements than port connections and data access connections. Inducing other elements, such as features or even whole components does not seem to be useful. Inducing of features or components will not introduce new behavior, as the behavior of the subprograms remains unaltered. As a result, the complete behavior of the architecture remains unaltered.

Altering the behavior of the architecture requires introducing, updating, or deleting subprograms and their behavior. The behavior of subprograms, however, is specified by the architect as pre/post-conditions.

Using Alloy to induce pre/post-conditions is not possible for (new) subprograms. Another, more limited, approach is possible however. We can specify a number of standard subprograms with predefined behavior, such as store the incoming data value into a data instance, or translate the incoming value by the use of a data instance and forward it. If we have a number of default subprograms, we can use Alloy to link these together and possibly find an architecture which satisfies the requirements. However, it should be noted that the pre-defined subprograms can only provide a limited set of functionality. Often the subprograms, which are specified by the architect, perform more complex tasks than any combination of the pre-defined subprograms can provide.

6.5 Conclusion

In this chapter we describe the process of performing change impact analysis in AADL architecture models. In section 6.2, we describe the process of validation of functional requirements in a software architecture, mentioned in section 3.2. This consists of two activities: formalization of functional requirements, and the validation of the formalized functional requirements based on the simulation of the architecture. The formalized requirements can be verified by the simulator, given in chapter 5. Whenever a formalized requirement is violated, the Alloy Analyzer will return the state space which violates one or more requirements as a counter example.

Section 6.3 elaborates the process of performing change impact analysis, mentioned in section 3.3. Individual formalized requirements can be tested against the counter example. By using traces from the formalized requirements to the architecture components, we can identify which component is violating a requirement and thus likely to be impacted. The whole process of performing change impact analysis is an iterative process. If the behavior of one component is modified, the simulation has to be run again to find the new state space. From this new state space, other impacted components can be found and possibly updated.

Finally, section 6.4 introduces the concept of inducing port connections and data access connections by the Alloy Analyzer. We can increase the scope of the model such that one (or more) extra port connection(s) is added to the simulated model. If the Alloy Analyzer then finds an model for which all the requirements and architecture invariants holds, we can extract the new port or data access connection from that model. This allows us to use the Alloy Analyzer to automatically synthesize a satisfying architecture for the given requirements. Although the current approach is limited to port connections and data access connections, other options might be possible to further synthesize an architecture.

Chapter 7

Evaluation of change impact analysis process

7.1 Introduction

Based on the approach to perform change impact analysis found in chapter 6, this chapter evaluates for the proposed approach.

Several change scenarios for the RPM architecture are defined and performed by the use of the proposed approach. The end-result of each change scenario is evaluated to analyze strengths and defects of the approach.

Sections 7.2 to 7.6 describe five change scenarios, the iterative process to perform change impact analysis, and a evaluation of the result. Section 7.7 concludes the chapter.

For each of the change scenarios, we describe the following:

- Changes to requirements;
- Following the iterative process, until the architecture satisfies all requirements:
  - Changes to the architecture;
  - Changes to the requirement predicates and traces from requirement predicates to architecture components;
  - Validation of the requirements in the architecture;
  - In case of a counter example: finding the impacted components by using the traces from requirement predicates to architecture components.
- Evaluation of the process for this change scenario.

The evaluation of the process for each scenario discusses the proposed process. Also, the results of the process are compared to the results of a data flow based approach. The data flow based approach uses the data flows which flow through a changed component, and mark all the involved components as impacted.

ID	Requirement
R1'	The system must measure blood pressure from a patient
R2'	The system must transfer the blood pressure measurements from the patient to the central system
R3'	The system must store blood pressure measurements for a patient in a central database

Table 7.1: Change scenario 1 - added requirements

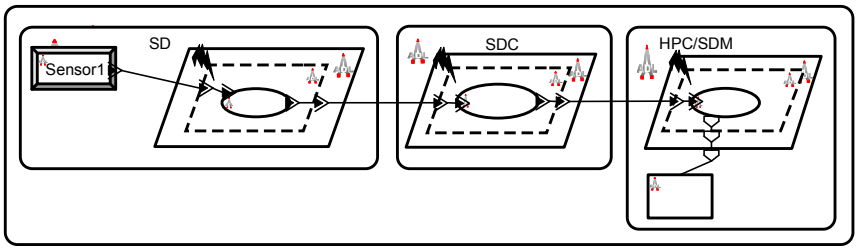


Figure 7.1: Change scenario 1 - iteration 0

7.2 Change scenario 1

This change scenario describes the additional for support for measuring blood pressure, next to temperature. Only support for blood pressure measurements is added. Adding support for blood pressure alarms and blood pressure threshold is similar to adding support for blood pressure measurements and is omitted for brevity.

This change scenario is used to evaluate the addition of requirements to the system.

7.2.1 Changes to requirements

The requirements R1, R2, and R3, found in appendix B, are copied and 'temperature' is replaced by 'blood pressure', as shown in table 7.1. These requirements are added and do not replace the original temperature related requirements, new functionality is added to the system.

7.2.2 Iterative process

The relevant part of the base architecture for this change scenario is given in figure 7.1. The data flow from the *sd.sensor1* device (temperature) to the *hpc.sdm.thread1.c1* subprogram is shown. Details of the *sd* system and *sdc* system, such as the data instances, are omitted as these are not relevant for this change scenario.

Iteration 1

In the first iteration, we add the blood pressure sensor to the sd system in the architecture. An additional parameter is added to the *sd.proc1.thread1.c1* subprogram, and a connection is created between the new *sd.sensor2* and

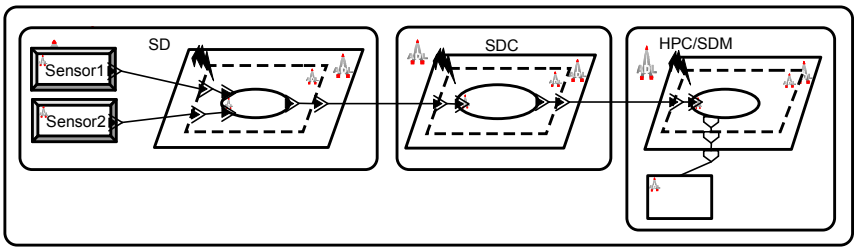


Figure 7.2: Change scenario 1 - iteration 1

Listing 7.1: Change scenario 1 - formalized requirements

```
pred R1' {
  R1'_a and R1'_b
}
pred R1'_a {
  some s: State |
    one CS1_iteration0_RPM_i_Instance_sd_sensor2_measurement.(s.
        eventDataPortValues) and
    one CS1_iteration0_RPM_i_Instance_sd_proc1_thread1_c1_
      sensor2_in.(s.^next.parameterValues)
}
pred R1'_b {
  some s: State |
    one
        CS1_iteration0_RPM_i_Instance_sd_proc1_thread1_c1_sensor2_in
        .(s.parameterValues) and
    one CS1_iteration0_RPM_i_Instance_sd_proc1_thread1_c1_
      measurement_out.(s.^next.eventDataPortValues)
}
```

sd.proc1.thread1.c1 subprogram. The result is shown in figure 7.2. We do not add any event data ports and parameter, and create connections between the ports/parameters, as the *sdc.proc1.thread1.c1* and *hpc.sdm.thread1.c1* components is supposed to abstract from the measurement type. The system was built with this kind of extension in mind.

At the same time, we formalize the new requirements. Given that the system abstracts from the measurement type 'after' the *sd.proc1.thread1.c1* subprogram, we re-use the original formalization of the requirements R2 and R3. The formalization of the requirement R1' is given in listing 7.1. The formalization states that the measurement must be transferred from the *sd.sensor2* device to the *sd.proc1.thread1.c1* subprogram (R1'_a), from *sd.proc1.thread1* to *sd.proc1.thread1.c1* (R1'_b), and that the subprogram *sd.proc1.thread1.c1* must process the received measurement (R1'_c).

The traces from requirements to architecture components for the requirement predicates R2' and R3' are copied from the requirements R2 and R3. The traces for requirement R1' are similar to the traces for requirement R1: R1'_a traces to the components *sd.sensor2* and *sd.proc1.thread1.c1*, R1'_b traces to the components *sd.proc1.thread1* and *sd.proc1.thread1.c1*, and R1'_b traces to the component *sd.proc1.thread1.c1*.

Verification of the requirements R1, R2, and R3 can be done by running the Alloy command Check_scenario1. Running the simulation does not give a

counter example, and thus the requirements R1, R2, and R3 are still satisfied by the architecture.

To verify the new requirements R1', R2', and R3', we create a new scenario. The new scenario has a different initial state compared to the original scenario; a measurement is at *sd.sensor2.measurement*, instead of *sd.sensor1.measurement*. Also, the validation of requirements R1, R2, and R3 has been replaced by R1', R2', and R3'.

Running the command Check_scenario1a results in a counter example. We query the counter example to identify the requirements which are failing. This is easily done by bringing up the counter example in Alloy, open the evaluator, and test the predicates R1', R2', and R3'. All of the predicates evaluate to false, which means that none of the requirements are satisfied by the architecture. However, a dependency relation can be identified between R1' and R2', and R2' and R3'. Before R2' can be satisfied, R1' needs to be satisfied. The blood pressure measurement cannot be transported from the *SD* to the *SDM* (R2'), before the measurement is created (R1'). As no semantics are added to the requirements themselves, we are not able to express this relation, and have to keep track of it manually. As R3' depends on R2', and R2' depends on R1', using reflexivity, we can say R3' depends also on R1'.

Given that R1' is the first requirement which fails, we use the traces from R1' to the architecture to identify the failing components. The predicate R1' consists of three sub-predicates: R1'_a, R1'_b and R1'_c. Using Alloy to evaluate these two predicates against the counter example, we see that R1'_a and R1'_b evaluates to true, but R1'_c evaluates to false. The predicate R1'_c traces to the *sd.proc1.thread1.c1* subprogram, and is the next point of investigation, in the next iteration.

Iteration 2

When inspecting the SD subprogram, we see that the default implementation does not handle values received at the new parameter *sd.proc1.thread1.c1.sensor2_in*. To make the *sd.proc1.thread1.c1* subprogram handle the new parameter, we need to update the *Alloy::post_edps* property of the *sd.proc1.thread1.c1* subprogram, as shown in listing 7.2.

The formalized requirements and traces from the formalized requirements to the architecture remain unaltered.

We again verify the architecture using the Alloy command Check_scenario1a. A counter example is returned by Alloy. The requirements R1', R2', and R3' are all satisfied by the architecture. Further investigation of the counter example shows that the ArchitectureInvariants predicate evaluates to false, specifically invariant of the data instance *hpc.temperature_measurements* data instance is violated. The invariant for this data instance states that all contained measurements should be produced by the device *sd.sensor1*. However, the stored blood pressure measurement is produced by the device *sd.sensor2*.

The *hpc.sdm.thread1.c1* subprogram stores the value in the data instance *hpc.temperature_measurements*. The *hpc.sdm.thread1.c1* subprogram and the *hpc.temperature_measurements* data instance are the next points of investigation.

Listing 7.2: Change scenario 1 - iteration 2 - update to SD subprogram

```
subprogram implementation SD_Subp.i
  properties
    Alloy::post_edp => "
      one
          CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
          .(s.parameterValues) and no GenerateAlarm_i =>
        {
            CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
            ->
            CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
            .(s.parameterValues) }
      else one
          CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor2_in
          .(s.parameterValues) and no GenerateAlarm_i =>
        {
            CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
            ->
            CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor2_in
            .(s.parameterValues) }
      else one
          CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
          .(s.parameterValues) and one GenerateAlarm_i =>
        {
            CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out
            -> { a: Alarm_i | a.sensorA =
            CS1_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
            .(s.parameterValues).sensorM and one a.zaA } }
      else
        none -> none
    ";
  ...
end SD_Subp.i;
```

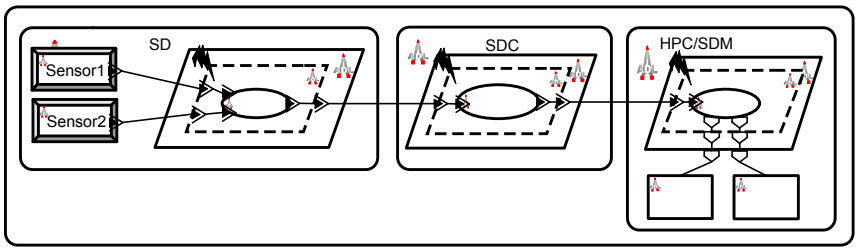


Figure 7.3: Change scenario 1 - iteration 3

Iteration 3

Two options exist to repair the architecture. The first option is to add a new data instance *hpc.bloodpressure_measurements* to the architecture, dedicated to blood pressure measurements. If the new data instance is added, the *hpc.sdm.thread1.c1* subprogram needs to be given access to the data instance by data access features and connections. Also, the semantics for the *hpc.sdm.thread.c1* subprogram needs to be updated to write the blood pressure measurements to the new data instance. The second option is to make the *hpc.temperature_measurements* data instance less strict such that it also supports blood pressure measurements. To make this happen, we need to update the invariant for *hpc.temperature_measurements* data instance, and - preferably - rename the data instance to give it a better name. We choose the first option and add the new *hpc.bloodpressure_measurements* data instance to the architecture, connect the data instance to the *hpc.sdm.thread1.c1* subprogram, and update the semantics for the *hpc.sdm.thread1.c1* subprogram. The structure of the new architecture is shown in figure 7.3.

Also, we update the formalized requirement R3' to state that the blood pressure measurements need to be stored in the *hpc.bloodpressure_measurements* data instance, instead of the old *hpc.temperature_measurements* data instance.

Running the simulation again does not give a counter example. This means that the new requirements R1', R2', and R3' are satisfied, as well as all the architecture invariants. The architecture is now able to support blood pressure measurements.

7.2.3 Evaluation

In iteration 1, the *sd.proc1.thread1.c1* subprogram is identified as an impacted component. Iteration 2 gave us the components *hpc.sdm.thread1.c1*, and *hpc.temperature_measurements*. During this change scenario, we have actually changed the following components: *sd.proc1.thread1.c1*, and *hpc.sdm.thread1.c1*. Also, we have added a new data instance to the hpc system. This shows that the candidate impacted set over-estimated, as the actual impacted set is a subset of the candidate impacted set.

In iteration 3 all the requirements are satisfied, but an architecture invariant is violated. The counter example is used to identify the violating component, much like the formalized requirements and traces from formalized requirements to architecture are used to identify components.

Had we used the data flow of requirements R1, R2, and R3, we had marked

ID	Requirement
R12'	The system must use WiFi for the wireless connection between the patient device and the central system

Table 7.2: Change scenario 2 - updated requirements

the components *sd.sensor1*, *sd.proc1.thread1.c1*, *sdc.proc1.thread1.c1*, *hpc.sdm.thread1.c1*, and *hpc.temperature_measurements* as impacted components. This set is larger than the actual impacted set of components.

7.3 Change scenario 2

ZigBee is used for the communication between the SDs and central system. ZigBee requires that there is a coordinator node (the SDC in the RPM architecture) which coordinates the ZigBee network. Another technique for communication between devices is 802.11b/g, or better known as WiFi. When using WiFi, a similar coordinator node is used (called an access point), but this node does not impose any functionality relevant for the architecture and is abstracted from.

This change scenario is used to test the approach when deleting requirements, and updating requirements.

7.3.1 Changes to requirements

The requirement R13 is removed, as there is no need for an explicit coordinator role anymore, and requirement R12 is updated to state the use of WiFi. The updated version of requirement R12 (R12') is given is table 7.2. Given that requirement R12 is a non-functional requirement, the requirement is not formalized. Also, no traces are created from the formalized requirement to the architecture.

7.3.2 Iterative process

Iteration 1

In the first iteration, we remove the complete sdc system, and connect the ports related to measurements, alarms, and thresholds from the sd system directly to the hpc system. An excerpt of the updated architecture, showing the data flow from *sd.sensor1* to *hpc.sdm.thread1.c1* for temperature measurements, is shown in figure 7.4.

The formalized versions of the requirements R2, R5, and R10 need to be updated. The formalizations of these requirements referred to the *sdc.proc1.thread1.c1* subprogram, which no longer exists. The formalizations of the requirements and the traces from the formalized requirements to architecture are updated. The updated traces are given in table 7.3.

To validate that the architecture satisfies the requirements, we run scenario 1, 2, and 3. Running scenario 1, results in a counter example. Querying the counter example shows that requirements R1 and R2 are satisfied by the architecture, but R3 is not. Using table B.4, we see that R3 traces to the components

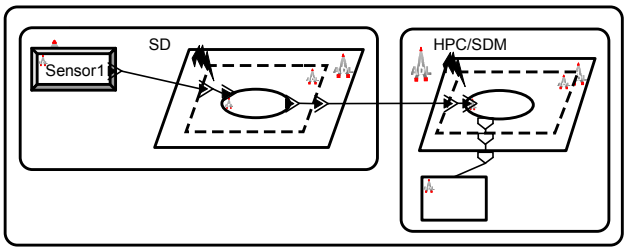


Figure 7.4: Change scenario 2 - iteration 1

Req.	Pred.	Component
R2	R2	sd.proc1.thread1.c1, hpc.sdm.thread1.c1 (pc)
R5	R5	sd.proc1.thread1.c1, hpc.proc1.thread1.c1 (pc)
R10	R10_a	cpc.wc.thread1.c1, hpc.ws.thread1.c1 (pc)
	R10_b	hpc.ws.thread1.c1
	R10_c	hpc.ws.thread1.c1, hpc.sdm.thread1.c1 (pc)
	R10_d	hpc.sdm.thread1.c1
	R10_e	hpc.sdm.thread1.c1, sdc.proc1.thread1.c1 (pc)

Table 7.3: Change scenario 2 - updated traces

hpc.sdm.thread1.c1 and *hpc.temperature_measurements*. Also, the invariant for the subprogram *hpc.sdm.thread1.c1* is violated.

Running scenario 2 does not result in a counter example. This is unexpected, as scenario 2 is similar to scenario 1, but tests if the alarm-related requirements are satisfied by the architecture. For now, we assume the requirements R4-R8 are satisfied by the architecture.

Scenario 3 also gives a counter example. The predicates for the requirements R9, R10, and R10 all evaluate to true. Further inspection of the counter example shows the architecture invariant for the *sd.proc1.thread1.c1* subprogram is violated. From this scenario, we mark the *sd.proc1.thread1.c1* as impacted.

The iteration gives us the components *hpc.sdm.thread1.c1*, *hpc.temperature_measurements*, and *sd.proc1.thread1.c1* as impacted components. However, it is unclear in what way these components are failing. Further investigation of the counter examples from the previous iteration is needed to discover why the subprogram invariants are violated.

In the counter example of first scenario, the *sd.proc1.thread1.c1* subprogram sends a temperature measurement. The measurement is the identified by a ZigBee-id. The *hpc.sdm.thread.c1* subprogram, however, expects to receive a measurement identified by a device-id. As a result, the invariant for the *hpc.sdm.thread1.c1* subprogram is violated. In the original architecture, the *sdc.proc1.thread.c1* subprogram would translate the ZigBee-id to a device-id. As the *sdc.proc1.thread1.c1* subprogram is no longer involved in the measurements data flow, this task is no longer performed.

Inspection of the counter example of the third scenario shows a similar result. The *sd.proc1.thread1.c1* subprogram receives a threshold message, for which a device-id is set. However, the *sd.proc1.thread1.c1* subprogram expects

to receive a message for which a ZigBee-id is set. As a result, the invariant for the *sd.proc1.thread1.c1* subprogram is violated. Again, in the original architecture, the *sdc.proc1.thread1.c1* subprogram would translate the ZigBee-id to a device-id. But the *sdc.proc1.thread1.c1* subprogram is no longer involved in the threshold data flow.

From this iteration, we mark the *sd.proc1.thread1.c1* and *hpc.sdm.proc1.thread1.c1* subprograms, and the *hpc.temperature_measurements* data instance as impacted elements.

Iteration 2

Following the previous iteration, we choose to alter either or both the *sd.proc1.thread1.c1* or/and *hpc.sdm.proc1.thread1.c1* subprogram. The *hpc.temperature_measurements* data instance is ignored for now. We have identified the missing task of translating between ZigBee-id and device-id, and vice versa. We choose to alter the *hpc.sdm.thread1.c1* subprogram, such that it does not translate between the patient-id and device-id, but patient-id and ZigBee-id.

The behavior of the *hpc.sdm.thread1.c1* subprogram is updated. Also, the data types related to device-id are removed from the architecture, and a data type used to couple patient-id to ZigBee-id is introduced. The data instance *hpc.dp_coupling* is renamed to *hpc.zp_coupling* and its data type is updated to ZigBeePatient.i.

Running the three scenarios now results in no counter examples. The architecture satisfies all the formalized requirements.

7.3.3 Evaluation

The device-id/ZigBee-id mismatch could also have been identified by enforcing more strict typing and static analysis. In the architecture, all ports related to measurements have the data type *Measurement.i*. If the *Measurement.i* data type had been split up into sub-types, one for each domain, static analysis would already give a type-mismatch when connecting the measurement ports of the *sd.proc1.thread1.c1* to the *hpc.sdm.thread1.c1* subprograms. However, in this case a generic type was used, with added behavioral semantics. Change impact analysis was only possible because the architecture itself was simulated and requirements and invariants were checked.

Scenario 2 did not give any counter example. This is unexpected, as the alarms data flow is similar to the measurements data flow. Translation between the ZigBee-id and device-id is also required for alarms. When inspecting the scenario and the simulator itself, we see what happened. The semantics of the *hpc.sdm.thread1.c1* subprogram appear to be invalid. In the simulation, the *hpc.sdm.thread1.c1* subprogram receives an alarm message at the *hpc.sdm.thread1.c1.alarm_in* parameter, for which the ZigBee-id is set. Upon execution of the *hpc.sdm.thread1.c1* subprogram, it output to data values to the *hpc.sdm.thread1.c1.alarm_out* event data port, as shown in listing 7.3.

The Alloy State signature is restricted in the sense that it can have at most one DataType instance per EventDataPort ('lone' quantifier.) The *hpc.sdm.thread1.c1* subprogram, however, gives two DataType instances. The restriction on the State signature is violated, and thus Alloy discards the generated model/trace. This shows the simulator can easily be broken, which is

Listing 7.3: Change scenario 2 - breaking the simulator

```
let s = State$6 |
  { CS2_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_out -> {
       a: Alarm_i | a.patA =
      CS2_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
      parameterValues).devA.(getDataInstance[
      CS2_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_dp_coupling
      ].(s.dataInstanceValues).deviceToPatient) } }

{CS2_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_out$0 ->
    Alarm_i$0,
    CS2_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_out$0 ->
     Alarm_i$2}
```

clearly a problem.

We do not support the hardware components of AADL. Although for simulating behavior, in our case, this does not seem to be important, we were unable to specify whether ZigBee or WiFi hardware is used to communicate. It is hard - if possible at all - to identify this from the architecture.

If we had used the data flows which all flow through the sdc system, and marked the components involved in those data flows as impacted, we would have marked the whole architecture as impacted. Thus, this greatly overestimates the impacted set of components.

7.4 Change scenario 3

The ZigBee standard specifies the need for a separate node to coordinate the ZigBee network. During the design of the architecture, it was decided to separate the coordinator functionality from the central hpc system by the use of an extra system, the sdc system. The coordination task can also be done by the hpc system itself, if the hpc is given a ZigBee radio and additional functionality.

This change scenario tests how maintenance from the architecture itself can be handled, no requirements are changed.

7.4.1 Changes to requirements

The requirements R12 and R13, found in table B.1, are relevant for this change scenario. Because these requirements are considered non-functional requirements, the requirements are not formalized. Also, no traces are defined from the requirements to the architecture.

7.4.2 Iterative process

Iteration 1

The first iteration is identical to the first iteration of change scenario 2. Needless to say, the results are the same as the iteration 1 of change scenario 2. The candidate impacted elements are considered to be the *sd.proc1.thread1.c1* and *hpc.sdm.proc1.thread1.c1* subprograms, and the *hpc.temperature_measurements* data instance.

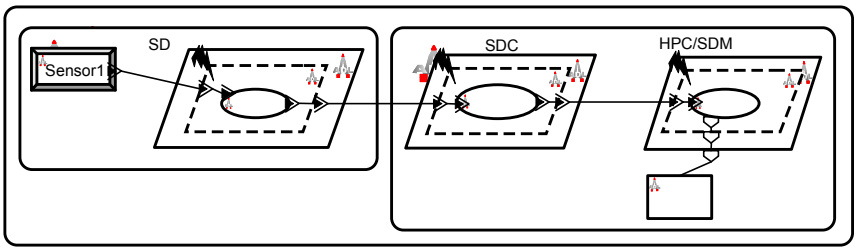


Figure 7.5: Change scenario 3 - iteration 2

Iteration 2

From the previous iteration we know the translation step is no longer performed. Also, given that the hpc system is given a ZigBee radio, it needs a process to drive this radio. As a solution, we move the *sdc.proc1* process to the hpc system. Also, the data instance used to store the translation table (between ZigBee-id and device-id), is migrated to the hpc system. The updated architecture is given in figure 7.5.

The formalized requirements need to be re-updated, as the *sdc.proc1.thread1.c1* subprogram now exists again. Thus, the formalized versions of requirements R2, R5, and R10 can be used from the original architecture, except that the *sdc.proc1.thread1.c1* subprogram is now identified by *hpc.sdc.thread1.c1*.

When we run the scenarios 1, 2, and 3, we see that the architecture satisfies the requirements and the process ends.

7.4.3 Evaluation

As in change scenario 2, we are not able to model any hardware in the architecture. It is unclear that hpc system is given a ZigBee radio, and thus the need for a driver of the ZigBee radio in the hpc system is not clear.

Again, if we would have used the data flows to find the impacted set of components, the whole architecture would have been marked as impacted.

7.5 Change scenario 4

Change scenario 4 adds additional functionality to the system. The alarms data flow is elaborated such that alarms are sent to both the existing alarm receiver subprogram (*cpc.ar.proc1.thread1.c1*) and to the newly added cell phone device.

7.5.1 Changes to requirements

Two requirements are added, derived from the R7 and R8. The added requirements are shown in table 7.4.

ID	Requirement
R7'	The system must transfer the temperature alarm (R4) form the patient to the doctors cell phone
R8'	The system must show the doctor the temperature alarm (R4) at the doctors cell phone

Table 7.4: Change scenario 4 - added requirements

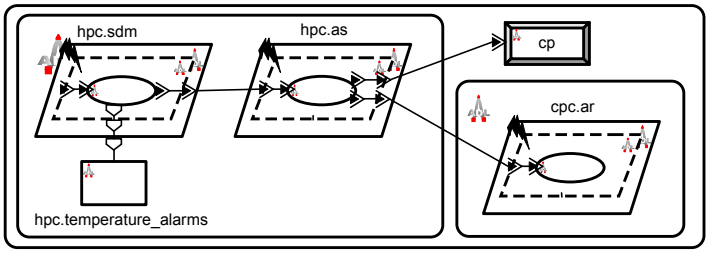


Figure 7.6: Change scenario 4 - iteration 1

7.5.2 Iterative process

Iteration 1

In the first iteration we add a cell phone (*cp*) device to the architecture, and create a connection from the *hpc.as.thread1.c1* subprogram to the new *cp* device. The cell phone is regarded a device, as we do not consider the cell phones internals. We make the assumption that when the cell phone receives an alarm, it will show the alarm. The updated architecture is shown in figure 7.6.

After we have updated the architecture, we formalize the requirements R7' and R8'. These requirements are similar to R7 and R8, and as such, so are their formalized counterparts. The new formalized requirements are shown in listing 7.4.

Additional traces from the formalized requirements to the architecture are created. Table 7.5 gives the traces for R7' and R8' to the architecture.

The Alloy command Check_scenario2 is updated to verify R7' and R8', next to the original R4, R5, R6, R7, and R8 predicates. When we check the

Listing 7.4: Change scenario 4 - formalized requirements

```
pred R7' {
  some s: State |
    one CS4_iteration1_RPM_i_Instance_sd_proc1_thread1_alarm_out.(s
        .eventDataPortValues) and
    one CS4_iteration1_RPM_i_Instance_cp_alarm_in.(s.^next.
        eventDataPortValues)
}
pred R8' {
  some s: State |
    one CS4_iteration1_RPM_i_Instance_cp_alarm_in.(s.
        eventDataPortValues)
}
```

Req.	Pred.	Component
R7'	R7'	sd.proc1.thread1.c1, sdc.proc1.thread1.c1, hpc.sdm.thread1.c1, hpc.as.thread1.c1, cp (pc, comp)
R8'	R8'	cp

Table 7.5: Change scenario 4 - iteration 1 - new traces

architecture by running the Check_scenario2 command, a counter example is returned. Checking the requirements against the counter example shows that the original R4-R8 predicates are satisfied, but the new requirements R7' and R8' are not.

Using the traces from table 7.5 for requirement R7', we get the following impacted elements: *sd.proc1.thread1.c1*, *sdc.proc1.thread1.c1*, *hpc.sdm.thread1.c1*, *hpc.as.thread1.c1*, and *cp*. Requirement R8' traces to the *cp* device, which was already found through requirement R7'.

The impacted set of elements is large, compared to the architecture. The requirement R7', which was not further split up like R2, R5, and R10, is traced to a large number of elements.

Iteration 2

The impacted elements found through the traces from R7' is a large set and makes it hard to find the real impacted element(s). To overcome this problem, the requirement R7' is further split up into several sub-requirements, as done with R2, R5, and R10. Using the same scenario, but with the finer grained version of R7', we can now identify the *hpc.as.thread1.c1* subprogram as the failing component. When the subprogram receives an alarm, it does not send a value to the *hpc.as.thread1.c1.alarm_out_cp* event data port.

We mark the subprogram *hpc.as.thread1.c1* as the impacted component.

Iteration 3

Using the results from iteration 2, we investigate the *hpc.as.thread1.c1* subprogram. When checking the semantics for this subprogram, it shows that the subprogram lacks functionality. As already identified in iteration 2, the *hpc.as.thread1.c1* subprogram does not send an alarm to *hpc.as.thread1.c1.alarm_out_cp* event data port. The behavioral semantics for this subprogram are updated, as shown in listing 7.5.

Running scenario 2 again gives no counter example and the architecture satisfies the old and new requirements.

7.5.3 Evaluation

The lesson learned from this change scenario is that the coarseness of requirement perdicates and the traces from the requirement predicates to the architecture are important. If a single formalized requirement (predicate) traces to a large number of components, it is hard to find the actual impacted component. Using the finer grained predicates for requirement R7', we were able to more accurately identify the component which was actually impacted.

Listing 7.5: Change scenario 4 - iteration 3 - updated semantics for *hpc.as.thread1.c1*

```
subprogram implementation AS_Subp.i
  properties
    Alloy::post_edp => "
      one CS4_iteration3_RPM_i_Instance_hpc_as_thread1_c1_alarm_in
          .(s.parameterValues) =>
        { CS4_iteration3_RPM_i_Instance_hpc_as_thread1_c1_alarm_out
            ->
          CS4_iteration3_RPM_i_Instance_hpc_as_thread1_c1_alarm_in
          .(s.parameterValues) } +
        {
          CS4_iteration3_RPM_i_Instance_hpc_as_thread1_c1_alarm_out_cp
            ->
          CS4_iteration3_RPM_i_Instance_hpc_as_thread1_c1_alarm_in
          .(s.parameterValues) }
      else
        none -> none
    ";
  ...
end AS_Subp.i;
```

Using the data flow-based approach, we would have identified the *sd.proc1.thread1.c1*, *sdc.proc1.thread1.c1*, *hpc.sdm.thread1.c1*, *hpc.as.thread1.c1*, and
cpc.ar.thread1.c1 components as impacted.

7.6 Change scenario 5

In this change scenario, the system is changed such that there does not need to exist a permanent link between the patient and the central system. The patient device will temporarily store the temperature measurements, and transmits them at a later point to the central system.

7.6.1 Changes to requirements

The requirements are updated to support the non-persistent connection between the patient device and the central system. Requirement R2 is replaced by the requirements from table 7.6. The updated requirements introduce differentiation of the existence or non-existence of the port connection used for measurements between the patient device and the central system.

7.6.2 Iterative process

The requirements given in table 7.6 are not all implemented directly. A staged approach is taken to implement the requirements in the architecture. First, the two requirements R2a' and R2b' will be implemented in the architecture. Then, requirement R2c' will be implemented.

This change scenario introduces dynamicity in the architecture. In AADL, dynamicity of the architecture is modeled through the use of modes. Modes, however, are not supported by our simulator. As a result, we need to either

ID	Requirement
R2a'	If there is a connection between the patient device and the central system, the system must transfer temperature measurements (R1) from the patient to the central system
R2b'	If there is no connection between the patient device and the central system, the system must store the temperature measurements (R1) at the patient
R2c'	If there is a connection between the patient device and the central system, and the central system requests the stored measurements from the patient device (R1), the patient device must transfer the stored measurements to the central system

Table 7.6: Change scenario 5 - updated requirements

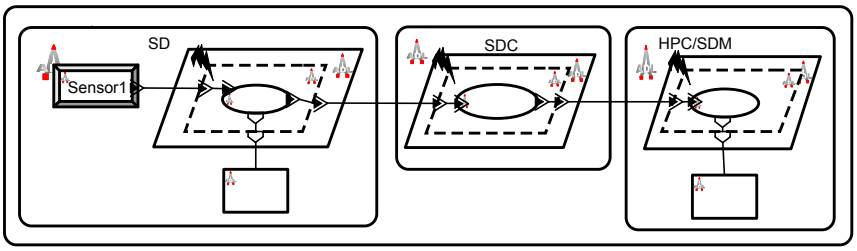


Figure 7.7: Change scenario 5 - iteration 1

manually alter the generated Alloy file, or keep different versions of the same model. We have chosen the first approach.

Iteration 1

In the first iteration, we add a data instance *sd.temperature_measurements* to the sd system to support the storage of temperature measurements, and connect it to the *sd.proc1.thread1.c1* subprogram. The result is shown in figure 7.7.

Given the updated behavior, the formalization of requirements R1 needs to be updated. R1 states that a measurement must be created. The old formalization check if the measurement was transports from *sd.sensor1.measurement* to *sd.proc1.thread1.c1.sensor1_in*, and if the subprogram *sd.proc1.thread1.c1* produced a value at the *sd.proc1.thread1.c1.measurement_out* event data port. In the new architecture, the measurement is either produced at *sd.proc1.thread1.c1.measurement_out*, or stored at the data instance *sd.temperature_measurements*. Listing 7.6 gives the new formalization of requirement R1.

The formalization of requirements R2a' and R2b' are shown in listing 7.7. The original formalization of R2 can be re-used for the requirement R2a'. However, the formalization of requirement R2 cannot be used for the formalization of requirement R2b'. Requirement R2 states that is a measurement is received by the *sd.proc1.thread.c1* subprogram, it will be sent to event data port *sd.proc1.thread1.c1.measurement_out*. This is not the case for R2b', as instead, it will be stored in the data instance *sd.temperature_measurements*,

Listing 7.6: Change scenario 5 - iteration 1 - formalized requirements R1

```
pred R1 {
  R1_a
  one
      CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_measurement_out
      .portConnections[] =>
    R1_b
  else
    R1_c
}
pred R1_a {
  some s: State |
    one CS5_iteration1_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one
        CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
        .(s.^next.parameterValues)
}
pred R1_b {
  some s: State |
    one
        CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
        .(s.parameterValues) and
    one
        CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
        .(s.^next.eventDataPortValues)
}
pred R1_c {
  some s: State |
    one
        CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
        .(s.parameterValues) and
    one CS5_iteration1_RPM_i_Instance_sd_temperature_measurements.(
        s.dataInstanceValues)
}
```

and not sent to event data port *sd.proc1.thread1.c1.measurement_out*.

Requirement R3 is also updated, such that it only needs to hold whenever there is a connection between the *sd.proc1.thread1.c1.measurements_out* and *sdc.proc1.thread1.c1.measurements_in*. The updated formalization is given in listing 7.7.

After the formalization of the new requirements, we update the traces from the formalized requirements to the architecture. The traces are given in table 7.7.

The first scenario is updated, such that it checks the new and updated requirements. We now execute the scenario twice. The first time we execute the command, we make sure the *sd.proc1.thread1.c1.measurements_out* and *sdc.proc1.thread1.c1.measurements_in* ports are connected by a port connection. No counter example is given by the simulator. This means that the architecture satisfies the requirements.

After breaking the port connection (by removing the line which connects the two ports from the portConnection function), we get a counter example. Investigation of the counter example shows that requirements R1, R2b', and R3 are not satisfied by the architecture. We know that before R3 can be satisfied, R1 has to

Listing 7.7: Change scenario 5 - iteration 1 - formalized requirements R2 and R3

```
pred R2a' {
  one
      CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_measurement_out
      .portConnections[] => R2a'_a and R2a'_b and R2a'_c
}
pred R2a'_a {
  some s: State |
    one
        CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
        .(s.eventDataPortValues) and
    one
        CS5_iteration1_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in
        .(s.^next.parameterValues)
}
pred R2a'_b {
  some s: State |
    one
        CS5_iteration1_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in
        .(s.parameterValues) and
    one
        CS5_iteration1_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_out
        .(s.^next.eventDataPortValues)
}
pred R2a'_c {
  some s: State |
    one
        CS5_iteration1_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_out
        .(s.eventDataPortValues) and
    one
        CS5_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in
        .(s.^next.parameterValues)
}
pred R2b' {
  no CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_measurement_out
      .portConnections[] =>
    some s: State |
      one
          CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
          .(s.parameterValues) and
      one CS5_iteration1_RPM_i_Instance_sd_temperature_measurements
          .(s.dataInstanceValues)
}
pred R3 {
  one
      CS5_iteration1_RPM_i_Instance_sd_proc1_thread1_measurement_out
      .portConnections[] =>
    some s: State |
      one
          CS5_iteration1_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in
          .(s.parameterValues) and
      one
          CS5_iteration1_RPM_i_Instance_hpc_temperature_measurements
          .(s.^next.dataInstanceValues)
}
```

Req.	Pred.	Component
R1	R1_a	sd.sensor1, sd.proc1.thread1.c1 (pc)
	R1_b	sd.proc1.thread1.c1
	R1_c	sd.proc1.thread1.c1, sd.temperature_measurements
R2a'	R2a'_a	sd.proc1.thread1.c1, sdc.proc1.thread1.c1 (pc)
	R2a'_b	sdc.proc1.thread1.c1
	R2a'_c	sdc.proc1.thread1.c1, hpc.sdm.thread1.c1 (pc)
R2b'	R2b'	sd.proc1.thread1.c1, sd.temperature_measurements
R3	R3	hpc.sdm.thread1.c1, hpc.temperature_measurements

Table 7.7: Change scenario 5 - updated traces

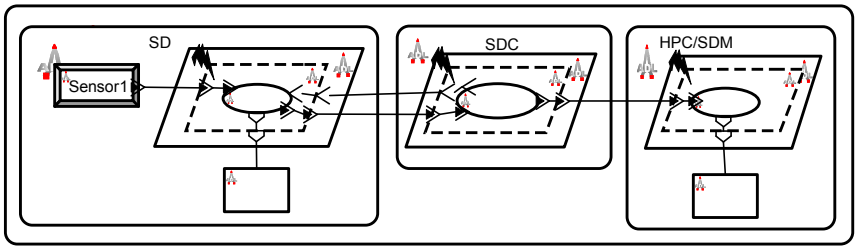


Figure 7.8: Change scenario 5 - iteration 3

be satisfied first, is R3 is ignored for now. Further investigation of R1 shows that predicate R1_a and R1_b evaluate to true, but R1_c evaluates to false. Using the traces from table 7.7, we can identify the components *sd.proc1.thread1.c1*, and *sd.temperature_measurements* as impacted components.

Iteration 2

In iteration 1, the subprogram *sd.proc1.thread1.c1* and the data instance *sd.temperature_measurements* are identified as impacted components. At first, we inspect the semantics of the subprogram *sd.proc1.thread1.c1*. We see the subprogram is lacking semantics. After we have updated the semantics of the subprogram, as shown in listing 7.8, we run the verification of the architecture again twice. One time with the existence of a port connection between the *sd.proc1.thread1.c1.measurement_out* and *sdc.proc1.thread1.c1.measurement_in* features, and one time without a port connection between the features.

Running the simulator does not give a counter example. Thus, the requirements R1, R2a', R2b', and R3 are satisfied by the architecture.

Iteration 3

Requirement R2c' states that if there is a connection between the sd and sdc systems, and the sdc system requests the temperature measurements from the sd system, the sd system must send the stored temperature measurements to the sdc system. To implement this requirement, we add an event port to the *sdc.proc1.thread1.c1* and *sd.proc1.thread1.c1* subprograms and connect these new event ports. The result is shown in figure 7.8.

Listing 7.8: Change scenario 5 - iteration 2 - updated semantics for *sd.proc1.thread1.c1*

```
subprogram implementation SD_Subp.i
  properties
    Alloy::post_diProduced => "
      one
          CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in
          .(s.parameterValues) =>
        { getDataInstance[
            CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_temperature_threshold
            ] ->
            CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in
            .(s.parameterValues) }
      else one
          CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
          .(s.parameterValues) and no
          CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_measurement_out
          .portConnections[] and no GenerateAlarm_i =>
        { getDataInstance[
            CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_temperature_measurements
            ] ->
            CS5_iteration2_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
            .(s.parameterValues) }
    else
      none -> none
    ";
  ...
end SD_Subp.i;
```

Lessons learned from change scenario 4, the requirements R2c' is formalized in a fine grained manner, as shown in listing 7.9. The predicate R2c'_a traces to the *sdc.proc1.thread1.c1* and *sd.proc1.thread1.c1* subprograms. The predicate R2c'_b traces to the *sd.proc1.thread1.c1* subprogram. A new scenario is created to verify the new behavior. The initial state for the new scenario states that the *sdc.proc1.thread1.c1.measurements_request_out* event port holds an event. Furthermore, the *sd.temperature_measurements* contains one temperature measurement. The requirements R2a', R2c', and R3 are verified by this scenario.

Running the scenario results in a counter example. Testing the predicates, R2c'_a evaluates to true, but R2c'_b evaluates to false. Using the traces, we identify the *sd.proc1.thread1.c1* subprogram as an impacted component.

Iteration 4

After inspection of the *sd.proc1.thread1.c1* subprogram, we conclude the subprogram lacks the proper behavioral semantics. After implementing the behavior for the subprogram, we run the scenario again. Needless to say, the formalized requirements are unaltered.

The simulator does not give any counter example, which means the architecture now satisfies the requirements.

Listing 7.9: Change scenario 5 - iteration 3 - updated formalized requirement R2

```
pred R2c' {
  R2c'_a and R2c'_b
}
pred R2c'_a {
  some s: State |
    one
        CS5_iteration3_RPM_i_Instance_sdc_proc1_thread1_measurements_request_out
        .(s.eventPortValues) and
    one
        CS5_iteration3_RPM_i_Instance_sd_proc1_thread1_measurements_request_in
        .(s.^next.eventPortValues)
}
pred R2c'_b {
  some s: State |
    one
        CS5_iteration3_RPM_i_Instance_sd_proc1_thread1_measurements_request_in
        .(s.eventPortValues) and
    one
        CS5_iteration3_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
        .(s.^next.eventDataPortValues)
}
```

7.6.3 Evaluation

The lack of modes in the simulator proved to be a small barrier here. Fortunately, we could easily omit this by breaking the connection manually. However, in a more elaborate architecture, it might not be satisfactory to simply break a connection and re-run the simulation. Often, connections are broken or created during the simulation itself.

The candidate impact set found by the process are the *sd.proc1.thread1.c1*, and *sd.temperature_measurements* components. The actual impacted set is *sd.proc1.thread1.c1*. It shows that the candidate impact set overestimates the actual impact set.

Using the data flow based approach, we would have marked the *sd.proc1.thread1.c1*, *sdc.proc1.thread1.c1*, *hpc.sdm.thread1.c1*, *hpc.temperature_measurements* components as impacted.

7.7 Conclusion

This chapter describes the case study which is performed to test the effectiveness of the approach. Five change scenarios are given and performed to identify strengths and weaknesses of the approach. From this case study, a number of important aspects come forward. The first is that the coarseness of the formalized requirements and the traces from the formalized requirements to the architecture is important with regard to the accuracy of the resulting set of impacted components.

Secondly, data instances are often identified as candidate impacted components, but are often not actually impacted components. In other words, impacts of data instances are often overestimated. Finally, architecture invariants are, next to requirements, an important tool to identify impacted components. Of-

ten architecture invariants fail, while requirements do hold in the simulation. The failures of the invariants is often a result of invalid data values received by subprograms, or stored in data instances.

The proposed approach proves to be more accurate than simply taking all the components involved in the data flow defined by the changed requirements. Using the data flow to find impacted components usually over-estimates the set of impacted components.

Chapter 8

Related work

8.1 Introduction

This chapter investigates the works of others related to our work. Briefly, the related work is investigated with regard to several defined aspects. After that, for each work, a comparison is given to our work.

In section 8.2 related works with regard to behavioral analysis of architectures is given and compared. Section 8.3 elaborates on the validation of requirements within an architecture. Change impact analysis methods are compared in section 8.4. Finally, this chapter is concluded in section 8.5.

8.2 Behavior analysis

This section investigates the related work with regard to behavior analysis of software architectures. The following aspects will be investigated in the related work, and compared to our work:

- Notations used to specify the architecture and behavior;
- Technology used for analysis of the behavior of the architecture;
- Depth of analysis/analysis possibilities;
- Whether the analysis is architecture based or implementation based.

8.2.1 State charts

In [DV00, VDR00], the C2 architecture description language is used to record the architecture. To specify the behavior of the components found in architectures, the components are augmented with state charts. The (un)desired behavior properties are expressed in LTL.

A tool, called Argus-I, is used to perform analysis on the architecture. On the (individual) component level, the tool allows for structural analysis, static behavioral analysis (model checking), and simulation. Model checking is achieved by converting the state chart of a component to a Promela model. The SPIN model checker[Hol97] is used to validate properties over the behavior of the architecture. The properties are expressed in LTL.

At the architecture level, dependency analysis, interface mismatch, behavioral analysis (model checking), and simulation is performed. To perform model checking of the whole architecture, all the state charts of the components are converted to concurrent processes in a Promela model. This model is then tested against given LTL properties by the SPIN model checker. Simulation is used to explore and visualize the behavior of the architecture.

Possibilities of behavioral analysis include checking properties related to nondeterminism of the state chart, deadlock analysis in the architecture, and reachability analysis.

Furthermore, the actual component implementation, in Java, is tested and validated against the state chart-based specification of the component. The complete architecture implementation is validated through the observation of events between components by use of a debugger and monitor. The discovered traces are visualized in the architecture model. Finally, validation of the conformance of architecture implementation against architecture specification is made possible by use of the monitor tools, and parallel simulation of the components state charts.

Discussion

A state chart uses events as its driver. The events do not carry any data. As a result, no deep inspection is possible with regard to the messages/events exchanged between the components. In our work, different types of messages are used such as data value, event, and event data. As such, we can perform a deeper type of analysis based on the data carried by the messages.

On the other hand, the components are annotated with state charts. A single state chart can model complex behavior. Deadlocks can occur when two systems are running in parallel. Given the ability to model this complex behavior, and the analysis tools, dead locks can be identified. In our work, subprograms - the components which perform actual computation in the simulation - are annotated with pre- and post-conditions. The post-condition is executed as an atomic action. Not only is it harder to model complex behavior using pre- and post-conditions, deadlock detection also becomes harder.

In the analysis of the architecture, only the processes are taken into account. The structure of the architecture, such as port connections and their properties, are ignored when analyzing the architecture. In our approach, the simulator takes the delay of data value propagation through the architecture - introduced by port connections - into account.

8.2.2 Control flow graphs

[WSMD08] provides a complete framework to specify and analyze architectures. It uses its own meta-model for the structure of the architecture, but is based on the main architectural elements: component, connector, port. Component contracts are used to specify structural, behavioral, and data flow properties. The behavior of each component is specified by its Control Flow Graph (CFG). A CFG specifies the order of all messages sent and received by a component. An edge in a CFG represents the sending or receiving of a message. Nodes represent operators. Complex behavior is modeled by use of operators such as *sequence, alternative*, *fork*, and *merge*. Given that each node in the CFG contains the

exchanged-message history, the CFGs are context-sensitive. Data flow properties are used to specify the allowed ranges of incoming and outgoing values. Assumptions and guarantees are used to specify these properties, respectively.

The CALICO framework is introduced in order to analyze the behavior of the architecture. The framework introduces its own type of analysis. From the individual CFGs, one large CFG of the whole architecture is computed by the CALICO framework. As the component CFGs are annotated with guarantees and assumptions, it is possible to reason about the guarantees and assumptions in the architecture. In the CFG, guarantees are propagated forward, assumptions are propagated backward. Based on the forward/backward propagation, the assumptions and guarantees are validated at other nodes in the CFG.

At the platform level (implementation level), CALICO is used to analyze the messages sent in the system. The messages sent between the components are intercepted by special components called interceptors, and validated for conformance with the architecture. This approach thus allows for analysis at both the architecture level and the platform level.

Discussion

This paper proposes a way to model the behavior of the components with regard to the messages sent and received. Data flows within the architecture are extracted from the configuration of the architecture and the components behavior. The difference with our work is that the architecture itself is not simulated, but analyzed by the use of model composition and reasoning. In our work, we have the ability to specify pre-conditions for sub-programs which are checked by the simulator. Pre-conditions are comparable to the assumption properties in the CFGs. Post-conditions of subprograms are comparable to the guarantee properties in the CFGs.

Like in the previous approach, the structure of the architecture itself is not taken into account. Port connections are abstracted away and thus properties of these port connections are not taken into account during the verification. At the platform level, of course, the connections are taken into account.

8.2.3 Labeled Transition Systems

In [MKG99], architectures are recorded through the Darwin architecture description language. The behavior of individual components is specified by the textual notation Finite State Processes (FSP). A FSP is a simple process algebra which includes guarded choices, local processes and conditional processes. The FSP of a component is translated to a Labeled Transition System (LTS). The structure of the architecture is used to combine the processes of all the components. Combining all the behavior specifications of the components results in one large concurrent FSP.

The resulting FSP is converted to a LTS. A tool called Labeled Transition System Analyzer (LTSA) is used to simulate the architecture based on the LTS. Also, LTSA is used to validate behavioral properties over the whole system. The tool can also be used as a simulator to animate the architecture and explore different scenarios manually. Behavioral properties, such as reachability analysis, checking of safety properties, and checking of liveness properties, are validated by model checking based on the LTS.

Discussion

Similar to the state chart approach, the behavior analysis is based on events. No deep analysis of the data flow, like in the second approach, is possible with this method. Similar to the state chart approach, this approach provides a more elaborate way to specify complex behavior. Our approach uses pre- and post-conditions which are considered atomic actions in the simulator.

8.3 Requirements validation

In this section, we investigate the related work with regard to requirements validation in software architectures. The following criteria are investigated, and compared to our work:

- Notations used to specify the requirements;
- Notations used to specify the architecture and behavior, if any;
- Technology/techniques used to analyze the behavior of the architecture and validate the requirements;
- Whether the analysis is architecture based or implementation based.

8.3.1 Goal monitoring system

In [Rob08], Goal-oriented requirements engineering is used to model goals of an architecture. Goal modeling is used to describe behavior of the system. The behavior is described through the specification of input and outputs. Inputs are complex objects, exchanged by components. The approach works at the level of implementation. The architecture of the system is not recorded through any notation.

The requirements on the stream of input are specified using OCL_{TM}, an extended version of OCL 2.0. TM stands for Temporal Message logic. Extensions are in the form of specification over received messages, linear temporal logic operators such as *next state*, *prior state*, *always in the future*, and improved syntax. Additionally, operators specifying timeouts are added to the language. Using the monitors and OCL_{TM} expressions, the requirements satisfaction of a system can be determined based on the satisfaction of the OCL_{TM} expressions.

A monitoring system is proposed which monitors the behavior of the implementation. The monitors determine the state of the requirement satisfaction of a system from the stream of inputs and the OCL_{TM} specification.

Discussion

Requirements are expressed in the OCL_{TM} language. The approach allows for much more elaborate specification of requirements, compared to our approach. For example, based on an incoming event, a time limit is specified in which the system must react. Our simulation of architecture does not take time into account. Also, the OCL language provides a rich set of LTL operators. Our approach does not support any LTL related-operators, but requires the specification of temporal properties using Alloy predicates. These Alloy predicates

consist of statements over individual states and their (indirect) predecessor and successor states.

The implementation of the architecture itself is monitored; no model of the architecture is simulated/verified. This imposes the need for a (partially) implemented architecture. No predictions are made based on the models, only validation.

8.3.2 Requirements sequence diagrams

[Fle99] proposes a way to formalize the requirements, which can then be check against the architecture. The requirements are recorded using a *structured use case*. A template is given for the structured use cases. The template records the following information: goal, pre-condition, post-condition, actions, main scenario, optional alternative scenarios, variations, exceptions, and notes. The architecture is specified using the RAPIDE and ROOM. The architecture is specified through three views: the structure model, the communication model, and the execution model.

Based on the structured use cases, requirements sequence diagrams are derived. Requirements sequence diagrams describe the causal ordering of messages between components, and timing requirements for the causal ordering. The transformation from structured use cases to requirements sequence diagrams is manual. Using the behavioral specification of individual components, the component model itself is simulated. The simulation returns a sequence diagram.

The requirements sequence diagram and the sequence diagram derived through simulation of the architecture are then compared to each other. Differences are returned. A difference means that the behavior of the architecture and the required behavior to not comply.

Discussion

The approach taken in this paper is similar to our approach. At first the requirements are formalized in a machine-understandable way. Then, a simulation is performed to derive the actual interaction among the components in the architecture. Finally, the formalized requirements are tested against the derived interaction. The formalized requirements of both approaches are in the solution domain.

The approach differs in the sense that here a clear method is given for the recording of the scenarios for each requirement. Exchanges messages are made explicit by the structured use cases, while in our approach the developer is free in which events he wants to monitor/verify. Also, sequence diagrams also seem to be a more natural way of specifying behavior, compared to specifying first order logic predicates.

8.3.3 Behavior trees

[Dro03] takes an entirely different approach. The article proposes a process to derive a component based system from a set of functional requirements, based on behavior trees. A behavior tree is a formal, tree-like, specification of behavior of individual or networks of entities. A node in the structure represents a state

of the system. Different semantics for the nodes are given, such as if-state, when-state, and data-out state. Functional requirements are expressed using the behavior tree notation, called requirements behavior trees (RBTs).

A software engineering process, called the genetic software engineering method, is introduced. This process is used to derive a system specification from a set of functional requirements, by integrating RBTs into a design behavior trees (DBTs). The process of integrating two behavior trees is given by a number of axioms.

From the final DBT, which lives in the problem domain, a software architecture is derived. The DBT contains information about the components found in the architecture (structure), as well as the behavior of the individual components. Given that both the structure of the architecture, and behavior of individual components and interaction among components are derived, interfaces are specified automatically. The architecture is specified using a component based notation. Implementation is not considered by this approach.

Discussion

Given that the architecture is directly derived from the requirements, the architecture always satisfies the requirements. As such, the architecture does not have to be validated against the requirements. However, faults in the formalization of requirements will result in an architecture which is broken. In our work, the formalization of requirements and the design of architecture are two different processes. Given that there are two manual processes, faults may creep in the formalized requirements and/or architecture.

8.4 Change impact analysis

Finally, in this section we investigate the related work with regard to change impact analysis in software architectures. The following points are investigated and compared to our approach:

- Notations used to specify the architecture and behavior;
- Technology/algorithms used to perform change impact analysis;
- Preciseness of analysis.

8.4.1 Architectural slicing and chopping

In [ZYXX02] change impact analysis is performed on a Wright architecture. Wright uses CSP to model the behavior of the components.

Two techniques, called slicing and chopping, are applied to perform change impact analysis. Backward slicing is used to find the subset of the architecture that might directly or indirectly affect the behavior of a component. Forward slicing is used to find the subset of the architecture that might be directly or indirectly affected by the behavior of a component. Finally, chopping is used to find the subset of the architecture involved in the communication between two components.

To apply these techniques, an Architectural Flow Graph (AFG) is derived from the architecture. In an AFG, ports and roles of components and connectors, respectively, are represented by nodes. Edges represent interaction among the components through the ports and roles. Algorithms are given to find the slices and chops over the architecture. From the given set of nodes and edges, the subset of the architecture can be computed.

The analysis results in a subset of the architecture. This subset includes the components, connectors, ports, roles, and connections which are relevant for the behavior. The analysis is precise in the sense that all the elements of the architecture involved in the behavior are returned.

Discussion

Using this technique, it is easy to find the relevant parts of the architecture which provides a subset of the complete functionality. The technique only uses the architecture itself, and does not need other artifacts such as formalized requirements, or traces from requirements to architecture.

However, this technique can only be used to identify a subset of the architecture involved in some functionality. The architecture is not verified against the requirements, and as such, other possible impacted elements will not be found.

8.4.2 Dependency matrix

[MZL07] uses a Component Dependence Graph (CDG) to specify the weight of interaction among different components. Interaction weight is calculated based on scenarios, the probability of the scenario being executed, and the number of times interaction occurs between two components based on those scenarios. From the CDG, a Component Dependence Matrix (CDM) can be constructed. The CDM is a representation of the CDG in matrix form.

Based on the CDM, the probability of impact is calculated for components. Different algorithms are given to calculate the probability, based on whether the initial change occurs in a single component, or multiple components. As a result, the approach is not precise, but based on probability theory.

Discussion

The approach presented in this paper does not directly rely on a scenario to perform change impact analysis, like in our work. Instead, it does use scenarios, but only to calculate the likeliness of interaction between two components. If the likeliness of interaction between two components is high, the second component is likely to be impacted when the first component is changed. However, it does not have to be impacted for certain, it is a clue that the component is likely to be impacted. Likewise, a component might actually be impacted, while the approach might mark the possibility of impact low.

Our approach performs change impact analysis by direct execution of the architecture, based on one or more scenarios. This does seem to give a more reliable result. Although, as noted in chapter 7, the quality of the result does depend on the requirements.

8.4.3 Extended use of slices

In [FM06], architectures are specified consisting of components, provides or requires interfaces, and methods. A component contains one or more interfaces, and an interface contains one or more methods. Rules are given for the composition components of the architecture. Component Interaction Traces (CITs) are used to specify the interaction among the components in the architecture. CITs are similar to (message) sequence diagrams.

To perform change impact analysis in the architecture, a taxonomy of atomic changes to the architecture is defined. Examples of atomic changes include adding a method to an interface, and deleting an empty interface from a component. Based on the atomic changes, a number of impact rules are given.

To perform dynamic change impact analysis, a slicing technique is applied to identify the impacted elements. The slicing algorithms use the static structure of the architecture and the CIT. The slicing technique is similar to the one proposed in [ZYXX02], but takes individual methods of interfaces into account as well.

Discussion

Compared to [ZYXX02], the architecture is more detailed in [FM06]. The interfaces of the components (ports/roles) contain one or more methods. The slicing technique proposed in this paper is more elaborate than the one proposed in [ZYXX02]. However, this technique still only uses the architecture itself. Formalized requirements are not verified.

8.5 Conclusion

In this chapter, Parts of our approach are compared to other approaches. Three different aspects are compared: behavioral analysis of architectures, testing requirements against architectures, and change impact analysis in architectures.

The first aspect, behavioral analysis, is given in section 8.2. Two of the approaches use a process algebra/formalism to describe the behavior of the architecture: state charts and labeled transition systems. These approaches allow for deep analysis of the behavior, such as dead lock analysis. However, these approaches do not take properties of the data flow into account. The third approach uses control flow graphs, annotated with assumptions and guarantees, to verify the correctness of the data flow.

The second aspect, validating requirements in architectures is given in section 8.3. The first approach tests requirements, expressed as OCL_{TM} statements, over the architecture. The statements describe properties of the data received and sent from a component. OCL_{TM} is an extended version of OCL, with additional temporal properties. In the second approach message sequence charts are used to describe the functional requirements of the architecture. Using a simulator, the actual behavior is extracted. The actual behavior and wanted behavior are then compared. Finally, the last approach is a total different approach. Requirements are expressed as behavior trees. A behavior tree describes the behavior of the architecture. From these trees, an architecture is generated which conforms to the behavior trees.

The last aspect, performing change impact analysis, is given in section 8.4. The first approach uses architectural slicing and chopping to get a subset of the architecture involved in some behavior. This approach identifies the involved components in a data flow, but does not verify the correctness of the components themselves. The second approach uses a dependency matrix to identify impacted components. Weights between components are calculated by the use of scenarios and the likelihood of interaction between two components.

Chapter 9

Conclusion and future work

9.1 Summary

In software development, customer requirements are the main driver for the produced product. The requirements are provided by the customer and are ultimately translated by software engineers into a solution. During and after the process of requirements analysis, an architecture is created. This architecture describes the system at a high level of abstraction in one or more ways. During the process however, the customer needs often evolve resulting in changes to the requirements. Because the architecture is driven by the requirements, the architecture has to be changed accordingly to fulfill the evolved requirements. Change impact analysis will help changing the architecture.

We propose an approach to perform change impact analysis in software architectures in chapter 3. This approach is based on the validation of functional requirements in the software architecture. In order to validate the functional requirement, we need to derive the behavior of the architecture. Also, we need to reformulate the (informal) functional requirement to a formal behavior description. These behavior descriptions can then be tested against the derived behavior of the architecture. Using the results of the behavior descriptions, and traces from requirement to architecture components, we can identify individual components which do not satisfy the requirements.

In order to derive the behavior of the architecture, we simulate the architecture. AADL is used to record the architecture. Structural and behavioral semantics are derived from the AADL standard. Alloy is used for the simulation of the architecture. The simulation of the architecture results in a state space, which represents the behavior of the architecture. Benchmarks are performed to test the feasibility of simulating architectures using Alloy. Alongside, an upper limit is identified, making the simulation of larger architectures questionable.

To be able to validate functional requirements over the state space, the requirements need to be reformulated as Alloy predicates. These predicates assert the presence or absence of behavior in the derived state space. Guidelines are given for the reformulation of functional requirements to Alloy predicates. If one or more requirements are not satisfied by the architecture, a counter example is returned by the Alloy Analyzer. This counter example includes the state space resulting from the simulation. We query the counter example to identify which

requirements are not satisfied. Using the results of the requirements and traces from requirements to architecture, we are able to identify failing components. The failing components are marked as impacted. An iterative process is used to identify all the impacted components.

A case study is performed to identify the strengths and weaknesses of the approach. From the case study, we have learned the lesson that the coarseness of the requirements is an important aspect with regard to the preciseness of the approach. Also, the approach sometimes overestimates the impacted component set.

The proposed approach is compared to other works on three different levels: behavioral analysis, requirements validation, and change impact analysis. Aspects of other approaches are comparable to our approach. Some approaches provide a more elaborate behavioral analysis, compared to our approach. For requirements validation, other approaches use a more user-friendly notation for the specification of functional requirements, such as sequence charts. Finally, for change impact analysis, our approach seems to allow for more elaborate analysis.

9.2 Answering the research question

The main research question of this study is as follows:

How can validation of requirements in software architectures and performing change impact analysis in software architectures be improved?

We answered this question by proposing a process to perform change impact analysis, as given in chapter 3 and elaborated upon in chapters 5 and 6. Functional requirements are reformulated in as Alloy predicates, such that these can be automatically validated against the behavior of the architecture. We simulate the architecture in Alloy to derive the behavior of the architecture, based the on the static and behavioral semantics of AADL. The derived behavior is expressed as a state space. Using the derived behavior and the formalized requirements, we come to a verdict whether the architecture satisfies the functional requirements. Using the verdicts and traces from formalized requirements to architecture, we identify the impacted components of the architecture.

The main research question is split into sub questions. Each sub question is answered below.

"What kind of system properties can be checked in the architectural design level?"

In chapter 5, we have shown how to simulate the behavior of the architecture. By using this simulation, we are able to validate a number of behavioral properties. Structural properties, not involving any behavior, are not considered in this research. The formalized requirements state properties on the behavior of the architecture. For example, a functional requirement can state causality: if event A happens, event B must happen afterward. Using the simulator, we can validate whether this behavioral property is present in the behavior of the architecture.

The correctness of data flows through the architecture can also be validated, as shown in chapter 7. Subprograms and data instances can be annotated with invariants. These invariants state properties over the components which must always hold. For example, an invariant can restrict the allowed inputs of a subprogram, comparable to a pre-condition of a function. Using the simulated architecture, we can validate whether the data value does or does not violate the invariant of the subprogram.

Also, architecture invariants can be used to state restrictions on the data values a data instance can hold. As shown in section 7.2, the simulator was used to check whether all the data values for a data instance were valid. A counter example was given, as the architecture invariant was violated.

"How can we use the validation of requirements in architecture to perform change impact analysis?"

Using the simulator, as presented in chapter 5, we derive the behavior of the architecture. This behavior is expressed as a state space. Requirements are used to find invalid or missing behavior of the architecture, as shown in chapter 6. When the simulator finds a part of the state space for which one or more requirements do not hold, a counter example is returned. This counter example represents the invalid behavior. Individual requirements can be tested against the counter example. By doing this, we can identify exactly which requirement is not satisfied by the architecture.

In chapter 6 we use traces from the individual formalized requirements to components of the architecture. By using the unsatisfied requirements and the traces and, we can identify which components are responsible for the invalid or missing behavior.

It has to be stressed that in order to get accurate results for change impact analysis, the traces have to be accurate. If the traces are not accurate enough, the resulting set of impacted components found through change impact analysis will be over-estimated.

"What tool(s)/method(s) can be used, if there are any at all, to support change impact analysis?"

OSATE is used to record the architecture. OSATE provides a parser which converts the textual specification of an AADL architecture into an EMF model. This EMF model conforms to the given AADL meta-model, also provided by OSATE. Furthermore, OSATE is used to create an AADL instance model from an AADL specification. openArchictectureWare/XPand2 is used to transform the AADL instance model to a simulatable architecture. The simulatable architecture is expressed in Alloy, and is executable by the Alloy Analyzer, as shown in chapter 5.

"Which parts of requirements validation and change impact analysis can be automated?"

A number of steps of the process described in chapter 3 have been automated. First, simulation of the architecture, chapter 5, is fully automatic. The Alloy Analyzer is used to derive the behavior of the architecture. This behavior is expressed in a state space. Formalized requirements can be automatically tested against the state space, also performed by the Alloy Analyzer.

In the implementation of this research, a number of steps are not automated and have to be performed manually. An example of a manual process is finding the components by use of the traces from formalized requirements to architecture components. This step can be automated by writing a program which traces formalized requirements to architecture components.

"Does the requirements validation and change impact analysis result in a simple yes/no analysis? If not, what kind of additional information can we provide as a result of this analysis?"

The validation of requirements results in a simple yes/no analysis, as shown in chapter 6. However, we can derive components from failing formalized requirements by using traces from formalized requirements to architecture components. This process is given in chapter 6, subsection 6.2.3.

9.3 Future work

This section provides the ideas for future work. Several different directions are given which can be further investigated.

9.3.1 Extending the simulation

The simulation can be extended in several ways.

Supporting more AADL semantics

The simulation presented in this thesis only supports a subset of AADL. For example, only the software components are supported, hardware components are not supported. Also, only subprograms directly impose behavior; other components are containers used to (indirectly) hold the subprograms and data instances. More semantics can be derived from the AADL standard to complement simulation of architectures.

For example, the AADL standard describes complex behavior for thread scheduling and dispatching in the form of a state machine. These semantics can be used in the simulation. Also, port connections have different characteristics, such as immediate and delayed transfer of data values. In the current approach, all port connections are considered to be delayed port connections.

Additionally, modes and mode transitions play an important role in the specification of the dynamicity of an architecture. The simulator can be extended to take this into account.

Supporting events from the environment

A scenario describes the initial state of the simulation. This initial state describes the location of data, after an event has been detected by the architecture. During the simulation itself, no other external events can be introduced into the simulation. This limits the simulation of the architecture, in the sense that it is harder/impossible to validate the correctness of the architecture with regard to concurrency and synchronization. Allowing for events to occur during the simulation itself would allow for more elaborate behavioral analysis possibilities.

In order to achieve this, the parallel simulation can be extended. An extra transition function can be added which describes the detection of an occurrence by the architecture. The Alloy model can be restricted such that the transition function is taken at most once during simulation of the architecture. The resulting state space describes the occurrence of the original detection of the occurrence of the event through the initial state, and the detection of the occurrence of the event through the additional transition function.

Subprograms as white-boxes

The behavior of subprograms is specified as pre- and post-conditions. While effective, as shown in chapter 5, it only allows for simple behavioral specification. It is impossible to model complex behavior. As a result, it is harder to reason about the complete behavior of the architecture. A different approach to model the complex behavior of a subprogram is to use a process algebra. Algebras such as Milner's π-calculus[Mil99] or Hoare's CSP[C.A78] provide formal semantics which can be simulated/executed.

Another approach to model complex behavior is to use finite state machines. The AADL Behavioral Annex[BDF+06] provides an annex used to describe the behavior of components through finite state machines. The semantics of the state machines of the AADL Behavioral Annex can be transformed into an Alloy model. The current version of the simulation can be extended such that state machines are used to specify the behavior of subprograms.

Optimizing the simulation

It is possible that techniques such slicing and/or chopping, as shown in chapter 8, can be applied to the architecture as a preprocessing step. Using these techniques, we can identify the relevant components, features, and connection of the architecture. Only the selected components, features, and port connections have to be transformed to a simulatable architecture. The benchmarks from chapter 5 show that decreasing the size of the architecture increases the speed of the simulation. Also, decreasing the size of the simulated architecture allows for a simulation with more states, as shown in chapter 5.

Another possibility is to simulate the architecture iteratively. Instead of simulating the architecture using a large number of states, we can instruct the Alloy Analyzer to only simulate two states. The first state would be the initial state, or the last discovered state from the last iteration. The newly discovered state is the next state, from the current state. After each simulation-iteration, the next state is extracted from the Alloy Analyzer, and set as the initial state/current state for the next simulation-iteration. This way, the number of states for each iteration is always two. The benchmarks from chapter 5 show that simulation using a small number of states is fast. The downside of this approach is that the model discovered by the Alloy Analyzer has to be interpreted after each iteration.

9.3.2 Stepping away from Alloy

In section 5 we described the simulation of architecture using the Alloy Analyzer. The benchmarks provided in that chapter give insight into the speed of the

simulator. The results show an exponential growth in time required to perform the simulation when the number of simulated states is increased linearly. Also, the Alloy Analyzer introduces an upper limit for the size of the architecture and the number of states used for simulation. Complex behavior, as proposed in subsection 9.3.1, can result in a state space explosion. The result of the state space explosion is that simulation of the architecture will become infeasible, if possible at all, given the limits of the Alloy Analyzer. On the other hand, it may be argued that an architecture is supposed to be abstract and simulation should be possible using a small number of states.

Other approaches to perform simulation and model checking seem to be more suited for the purpose of behavior analysis. Examples include SPIN[Hol03, Hol97], LTSA[MKG99] and Maude[CDE+02, CDE+08]. Although a SPIN specification only describes processes and not architecture structure, as in our approach, process analysis can be used to validate requirements. The functional requirements would be translated to LTL formulas, which are validated over the state space by the SPIN model checker. SPIN returns a counter example if the LTL property is violated. LTSA came forward in chapter 8. LTSA provides a counter example if one of the safety properties is violated. It provides a means to validate properties over labeled transition systems. Maude is a rewriting logic programming language used to model systems and actions within those systems. LTL is also used to specify properties which must hold. Maude returns a counter example if an LTL property is violated.

Abstract away from Alloy-related notation

Currently, the formalized requirements are specified as Alloy predicates. This requires knowledge of the Alloy notation (and workings) to specify the formalized requirements. The same holds for the specification of the behavior of subprograms, and the invariants for the subprograms and data instances.

Another, more intuitive, notation can be used to specify the required functionality of the architecture. One possibility is Linear Temporal Logic (LTL). LTL is a model temporal logic to describe temporal properties over paths in graph structures. Another possibility, as seen in chapter 8, is the use of sequence diagrams. Sequence diagrams provide a graphical notation to specify interactions between components. This notation is very intuitive, and does not require knowledge of a formal notation which can be hard to understand.

The behavior of the subprograms is also expressed as Alloy-statements. AADL provides the AADL Behavior Annex[BDF+06] to describe the behavior of individual components. The annex makes use of the state chart notation. State charts are commonly used to describe behavior, for example in UML.

9.3.3 Further automating the process

The process can be further automated. A meta-model can be created which allows one to record the traces between the requirements, requirement predicates, and architecture components. To record the requirements, the meta-model defined by Goknil et al.[GKvdB08] can be used. Either this meta-model can be extended to support the recording of the requirement predicates, or a new meta-model can be defined for this purpose. AADL is used to record the architecture. A model which records the traces between the requirements, requirement

predicates, and architecture components, would reference the requirements, requirements predicates, and architecture models.

Using the information from the previously defined models, one could further automate the process of performing change impact analysis. A tool can be developed which reads the models, transform the architecture model - using the existing transformation - to an Alloy model. The tool then calls Alloy to validate the architecture, and, in case of a counter example, queries the counter example to identify the unsatisfied requirement predicates. The model containing the traces from requirement predicates to architecture components is used to identify the failing architecture components, given the unsatisfied requirement predicates.

The tool could be used to automate the activities *simulate architecture, verify formalized requirements*, and *find impacted components* found in chapters 3 and 6.

Bibliography

[AG96] Robert Allen and David Garlan. The Wright architectural specification language. CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[All09] Tutorial for alloy analyzer 4.0, 07 2009.

[AMBD04] Alain Abran, James W. Moore, Pierre Bourque, and Robert Dupuis. Guide to the software engineering body of knowledge. Technical report, IEEE Computer Society, 2004.

[BC84] J. Banks and J. S. Carson. *Discrete-Event System Simulation*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

[BDF+06] Jean-Paul Bodeveix, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, and F. Vernadat. AADL behavioural annex. In *Data Systems In Aerospace (DASIA), Berlin-Germany, 22/05/2006-25/05/2006*, page (electronic medium), http://www.esa.int/publications, 2006. European Space Agency (ESA Publications).

[Boh02] Shawn A. Bohner. Software change impacts - an evolving perspective. In *ICSM*, pages 263–272. IEEE Computer Society, 2002.

[C.A78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.

[CDE+02] Clavel, Duran, Eker, Lincoln, Marti-Oliet, Meseguer, and Quesada. Maude: Specification and programming in rewriting logic. *TCS: Theoretical Computer Science*, 285, 2002.

[CDE+08] Manual Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, Jose Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.4)*, October 2008.

[CK03] Paul Clements and Rick Kazman. *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[DN02] Liliana Dobrica and Eila Niemelä. A survey on software architecture analysis methods. *IEEE Trans. Software Eng*, 28(7):638–653, 2002.

[Dro03] R. Geoff Dromey. From requirements to design: Formalizing the key steps. In *SEFM*, page 2. IEEE Computer Society, 2003.

[DV00] Marcio S. Dias and Marlon E. R. Vieira. Software architecture analysis based on statechart semantics. In *IWSSD*, pages 133–140. IEEE Computer Society, 2000.

[DvdHT01a] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, 2001.

[DvdHT01b] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *WICSA*, pages 103–112. IEEE Computer Society, 2001.

[EKHL03] Gregor Engels, Jochen Malte Küster, Reiko Heckel, and Marc Lohmann. Model-based verification and validation of properties. *Electr. Notes Theor. Comput. Sci*, 82(7), 2003.

[EMF] Eclipse modeling framework.

[Fei] Peter Feiler. SAE AADL: An industry standard for embedded systems engineering.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*, 2 2006.

[Fle99] Wolfgang Fleisch. Applying use cases for the requirements validation of component-based real-time software. Universität Stuttgart; Fakultät Informatik, Elektrotechnik und Informationstechnik. Institut für Automatisierungs- und Softwaretechnik, May 1999.

[FM06] Tie Feng and Jonathan I. Maletic. Applying dynamic change impact analysis in component-based architecture design. In Yeong-Tae Song, Chao Lu, and Roger Lee, editors, *Seventh International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2006), 19-20 June 2006, Las Vegas, Nevada, USA*, pages 43–48. IEEE Computer Society, 2006.

[GKvdB08] A. Goknil, I. Kurtev, and K. G. van den Berg. Change impact analysis based on formalization of trace relations for requirements. In J. Oldevik, G. K. Olsen, T. Neple, and R. Paige, editors, *ECMDA Traceability Workshop (ECMDA-TW), Berlin, Germany*, pages 59–75, Trondheim, Norway, June 2008. SINTEF Report.

[GMW97] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[GS92] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–40. World Scientific Publishing Co., 1992.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol03] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2003.

[IEE90] IEEE Standards Board. IEEE standard glossary of software engineering terminology—IEEE std 610.12-1990, 1990.

[IEE00] IEEE. IEEE recommended practice for architectural description for software-intensive systems. Technical report, The Architecture Working Group of the Software Engineering Committee, October 2000.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, second edition, 1990.

[Ken02] Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for architecture evaluation. Technical report, September 05 2000.

[KM85] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, April 1985.

[Kru95] Philippe Kruchten. Architecture blueprints - the "4+1" view model of software architecture. In *TRI-Ada Tutorials*, pages 540–555, 1995.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements Engineering : Processes and Techniques (Worldwide Series in Computer Science).* John Wiley & Sons, September 1998.

[Kur05] I. Kurtev. *Adaptability of Model Transformations.* University of Twente, Enschede, 05 2005.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture—Practice and Promise.* Addison-Wesley, 2003.

[LHS08] Jiexin Lian, Zhaoxia Hu, and Sol M. Shatz. Simulation-based analysis of UML statechart diagrams: methods and case studies. *Software Quality Journal*, 16(1):45–78, 2008.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, 1999.

[MIT] MIT Software Design Group. The Alloy Analyzer homepage.

[MK95] Mike Mannion and Barry Keepence. SMART requirements. *ACM SIGSOFT Software Engineering Notes*, 20(2):42–47, April 1995.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In Patrick Donohoe, editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 35–50. Kluwer, 1999.

[Mon01] Robert T. Monroe. Capturing software architecture design expertise with armani version 2.3, 01 2001.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT FSE*, pages 24–32, 1996.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, jan 2000.

[Muk97] Madhavan Mukund. Linear-time temporal logic and Büchi automata. Tutoriala talk, Winter School on Logic and Computer Science, January 1997.

[MZL07] Chengying Mao, Jinlong Zhang, and Yansheng Lu. Matrix-based change impact analysis for component-based software. In *COMPSAC*, pages 641–642. IEEE Computer Society, 2007.

[Obj03] Object Management Group, Framingham, Massachusetts. *MDA Guide Version 1.0.1*, June 2003.

[OCL06] Object Modeling Group. *Object Constraint Language Specification, version 2.0*, 05 2006.

[ope] openarchitectureware - the leading platform for professional model-driven software development.

[Rob08] William N. Robinson. Extended OCL for goal monitoring. *ECEASST*, 9, 2008.

[RR98] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri nets: advances in Petri nets. Part 1. Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pub-SV:adr, 1998. Springer-Verlag Inc. "Based on the Advanced Course on Petri Nets, held in Dagstuhl (Germany) in September 1996".

[RST+04] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. *ACM SIGPLAN Notices*, 39(10):432–448, 2004.

[SAE04] SAE Aerospace. *Architecture Analysis & Design Language (AADL) AS5506*, 11 2004.

[SAE09] SAE Aerospace. *Architecture Analysis & Design Language (AADL) AS5506A*, 01 2009.

[Sel03] Bran Selic. Model-driven development of real-time software using OMG standards. In *ISORC*, pages 4–6. IEEE Computer Society, 2003.

[Spi87] Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1987.

[UCI] Uci isr software architecture research.

[VDR00] Marlon E. R. Vieira, Marcio S. Dias, and Debra J. Richardson. Analyzing software architectures with argus-I. In *ICSE*, pages 758–761, 2000.

[War62] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[Wie09] Jan Wielemaker. *SWI-Prolog 5.7.11 Reference Manual*. Department of Social Science Informatics (SWI), Universiteit Amsterdam, 2009.

[WSMD08] Guillaume Waignier, Prawee Sriplakich, Anne-Françoise Le Meur, and Laurence Duchien. A model-based framework for statically and dynamically checking component interactions. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2008.

[Zel96] Gregory Zelesnik. *The UniCon Language Reference Manual*. Carnegie Mellon University, May 1996.

[ZYXX02] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance*, 14(5):317–333, 2002.

Appendix A

AADL meta-model in Alloy

Listing A.1: AADL meta-model

```
module aadl_meta

abstract sig DataType {
}

abstract sig Component {
  features: set Feature,
  subComponents: set Component,
}

abstract sig Process extends Component {
} {
  subComponents in DataInstance + Thread
  features in DataPort + EventPort + EventDataPort + DataAccess
}

abstract sig Thread extends Component {
  subprogramCall: lone Subprogram,
} {
  subComponents in DataInstance + Subprogram
  features in DataPort + EventPort + EventDataPort + DataAccess
}

abstract sig DataInstance extends Component {
} {
  no subComponents
  no features
}

abstract sig Subprogram extends Component {
  next: lone Subprogram
} {
  no subComponents
  features in EventPort + EventDataPort + Parameter + DataAccess
  all p: features & (EventPort + EventDataPort) | p.direction =
      PortDirection_Out
  all ca: features & DataAccess | ca.direction =
      AccessDirection_Required
}

abstract sig Device extends Component {
} {
```

```
    subComponents = none
    features in DataPort + EventPort + EventDataPort
}

abstract sig System extends Component {
} {
    subComponents in DataInstance + Process + Device + System
    features in DataPort + EventPort + EventDataPort + DataAccess
}

abstract sig Feature {
}

enum PortDirection {
    PortDirection_In,
    PortDirection_Out
}

abstract sig Port extends Feature {
    direction: one PortDirection,
}

abstract sig DataPort extends Port {
}

abstract sig EventPort extends Port {
}

abstract sig EventDataPort extends Port {
}

abstract sig Parameter extends Port {
}

abstract sig ComponentAccess extends Feature {
    direction: one AccessDirection,
}

enum AccessDirection {
    AccessDirection_Provided,
    AccessDirection_Required
}

abstract sig DataAccess extends ComponentAccess {
}

abstract sig Connection {
}

abstract sig PortConnection extends Connection {
    src: one Port,
    dst: one Port,
}

abstract sig DataAccessConnection extends Connection {
    src: one DataInstance + DataAccess,
    dst: one DataAccess,
}

fact NoDescendentOfSelf {
    all c: Component | c not in c.^subComponents
}
```

Appendix B

Remote Patient Monitoring

An existing project, called Remote Patient Monitoring, is used as an example throughout the thesis. From this project, a number of selected requirements and the architecture are used. As the name suggests, the system monitors patients. It does this by measuring temperature at patients, and when required sends an alarm to the doctor(s). Doctors can also view the historical data from the patients.

B.1 Informal requirements

Table B.1 lists a subset of the requirements for the RPM system. Duplicate requirements, mostly providing similar functionality, are omitted.

B.2 Architecture

The original architecture is presented in figure B.1. The architecture is not specified using any ADL, and thus hardly, if any at all, semantics can be extracted from it. While different figures are used for different elements, no concrete facts are attached to these figures. As a result, the architecture is only usable for communication and informal reasoning.

ID	Requirement
R1	The system must measure temperature from a patient
R2	The system must transfer temperature measurements (R1) from the patient to the central system
R3	The system must store temperature measurements (R1) for a patient in a central database
R4	The system must generate an alarm if the temperature threshold is violated
R5	The system must transfer the temperature alarm (R4) from the patient to the central system
R6	The system must store the generated temperature alarm (R4) when a temperature threshold is violated in a central database
R7	The system must transfer the temperature alarm (R4) form the patient to the doctors computers
R8	The system must show the doctor the temperature alarm (R4) at the doctors computers
R9	A doctor must be able to set the temperature threshold at a patient
R10	The system must transfer the temperature threshold (R9) from the doctors computer to the patient
R11	The system must store the temperature threshold (R9) for a patient at the patients sensor device
R12	The system must use ZigBee for the wireless connection between the sensor and central database
R13	The system must use a ZigBee network coordinator
R14	Communication between the SDC and SDM occurs through the use of RS-232
R15	A doctor must be able to retrieve all the stored temperature measurements for a patient
R16	A doctor must be able to retrieve all the stored temperature alarms for a patient

Table B.1: Selected requirements for RPM system

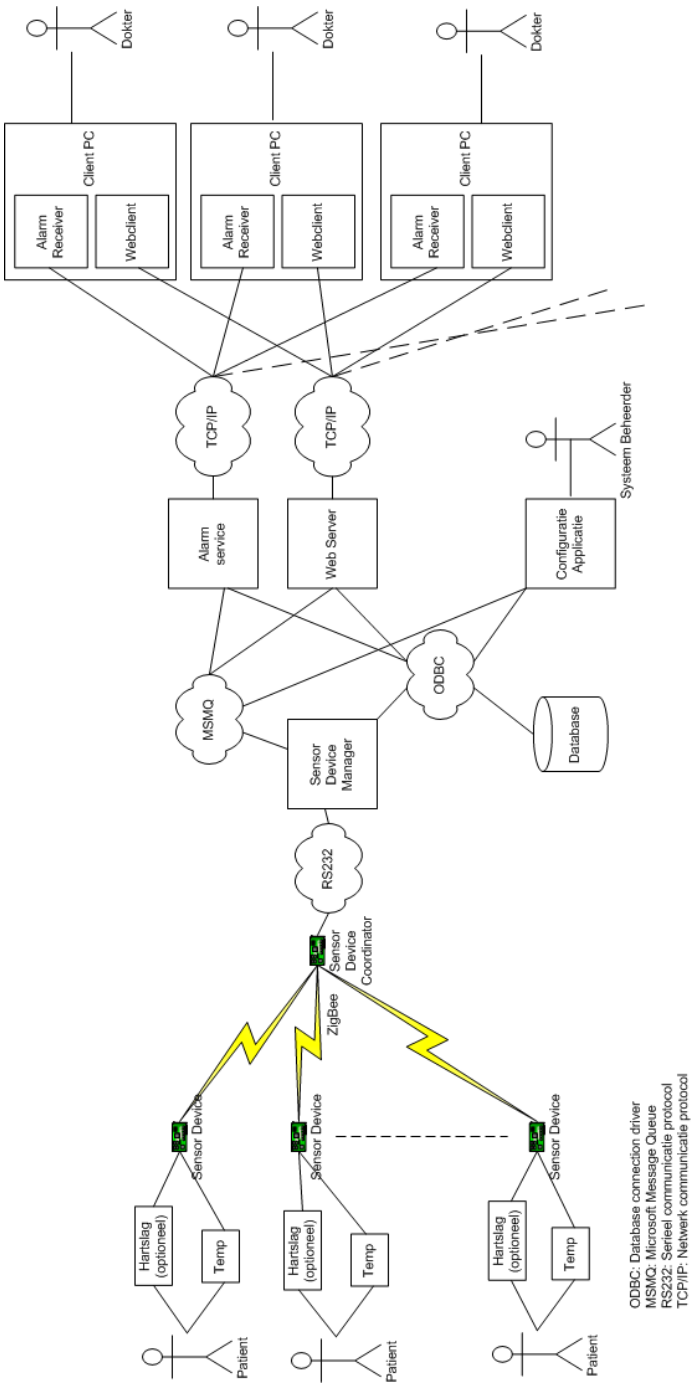


Figure B.1: Original RPM architecture

B.2.1 AADL description

AADL is used to re-specify the architecture. Additional properties are added to give more semantics to the architecture, such as the behavior of certain

components. By adding these additional properties, we make the architecture executable. The text and figures below give a brief overview of the architecture.

On the other hand, certain details are omitted from the architecture, such as hardware specific details. An example of this is the use of RS-232. The original RPM architecture specifies the use of RS-232 as a cloud-figure, while this is represented by a simple port connection in the re-specified architecture.

The architecture can be found in program listing B.1.

Listing B.1: RPM architecture in AADL

```
data GenerateAlarm
end GenerateAlarm;

data implementation GenerateAlarm.i
end GenerateAlarm.i;

data ZigBeeAddress
end ZigBeeAddress;

data implementation ZigBeeAddress.i
end ZigBeeAddress.i;

data Dev
end Dev;

data implementation Dev.i
end Dev.i;

data Patient
end Patient;

data implementation Patient.i
end Patient.i;

data Measurement
end Measurement;

data implementation Measurement.i
  properties
    Alloy::fields => "
      sensorM: one Device,
      zaM: lone ZigBeeAddress_i,
      devM: lone Dev_i,
      patM: lone Patient_i,
    ";
    Alloy::facts => "one (zaM + devM + patM)";
end Measurement.i;

data Alarm
end Alarm;

data implementation Alarm.i
  properties
    Alloy::fields => "
      sensorA: one Device,
      zaA: lone ZigBeeAddress_i,
      devA: lone Dev_i,
      patA: lone Patient_i,
    ";
    Alloy::facts => "one (zaA + devA + patA)";
end Alarm.i;
```

```
data ZigBeeDevice
end ZigBeeDevice;

data Threshold
end Threshold;

data implementation Threshold.i
  properties
    Alloy::fields => "
      sensorT: one Device,
      zaT: lone ZigBeeAddress_i,
      devT: lone Dev_i,
      patT: lone Patient_i,
    ";
    Alloy::facts => "one (zaT + devT + patT)";
end Threshold.i;

data implementation ZigBeeDevice.i
  properties
    Alloy::fields => "zigBeeToDevice: ZigBeeAddress_i -> one Dev_i"
        ;
end ZigBeeDevice.i;

data DevicePatient
end DevicePatient;

data implementation DevicePatient.i
  properties
    Alloy::fields => "deviceToPatient: Dev_i -> one Patient_i";
end DevicePatient.i;

data MeasurementsRequest
end MeasurementsRequest;

data implementation MeasurementsRequest.i
  properties
    Alloy::fields => "
      sensorMR: one Device
    ";
end MeasurementsRequest.i;

data AlarmsRequest
end AlarmsRequest;

data implementation AlarmsRequest.i
  properties
    Alloy::fields => "
      sensorAR: one Device
    ";
end AlarmsRequest.i;


--- SD ---
system SD
  features
    measurement_out: out event data port Measurement.i;
    alarm_out: out event data port Alarm.i;
    threshold_in: in event data port Threshold.i;
end SD;

system implementation SD.i
  subcomponents
```

```
    sensor1: device Sensor.Temperature;
    proc1: process SD_Proc.i;
    temperature_threshold: data Threshold.i {
      Alloy::invariant => "all t: this.(s.dataInstanceValues) | t.
          sensorT = RPM_RPM_i_Instance_sd_sensor1";
    };
  connections
    pc1: event data port sensor1.measurement -> proc1.sensor1_in;
    pc2: event data port proc1.measurement_out -> measurement_out;
    pc3: event data port proc1.alarm_out -> alarm_out;
    pc4: event data port threshold_in -> proc1.threshold_in;

    dac1: data access temperature_threshold -> proc1.
        temperature_threshold;
end SD.i;

device Sensor
  features
    measurement: out event data port Measurement.i;
  flows
    flow_source_measurement: flow source measurement;
end Sensor;

device implementation Sensor.Temperature
  flows
    flow_source_measurement: flow source measurement;
end Sensor.Temperature;

process SD_Proc
  features
    sensor1_in: in event data port Measurement.i;
    threshold_in: in event data port Threshold.i;
    measurement_out: out event data port Measurement.i;
    alarm_out: out event data port Alarm.i;

    temperature_threshold: requires data access Threshold.i;
end SD_Proc;

process implementation SD_Proc.i
  subcomponents
    thread1: thread SD_Thread.i;
  connections
    pc1: event data port sensor1_in -> thread1.sensor1_in;
    pc2: event data port thread1.measurement_out -> measurement_out
        ;
    pc3: event data port thread1.alarm_out -> alarm_out;
    pc4: event data port threshold_in -> thread1.threshold_in;

    dac1a: data access temperature_threshold -> thread1.
        temperature_threshold;
end SD_Proc.i;

thread SD_Thread
  features
    sensor1_in: in event data port Measurement.i;
    threshold_in: in event data port Threshold.i;
    measurement_out: out event data port Measurement.i;
    alarm_out: out event data port Alarm.i;

    temperature_threshold: requires data access Threshold.i;
end SD_Thread;
```

```
thread implementation SD_Thread.i
  calls
    calls1: {
      c1: subprogram SD_Subp.i;
    };
  connections
    pc1: parameter sensor1_in -> c1.sensor1_in;
    pc2: event data port c1.measurement_out -> measurement_out;
    pc3: event data port c1.alarm_out -> alarm_out;
    pc4: parameter threshold_in -> c1.threshold_in;

    dac1: data access temperature_threshold -> c1.
        temperature_threshold;
end SD_Thread.i;

subprogram SD_Subp
  features
    sensor1_in: in parameter Measurement.i;
    measurement_out: out event data port Measurement.i;
    alarm_out: out event data port Alarm.i;
    threshold_in: in parameter Threshold.i;

    temperature_threshold: requires data access Threshold.i;
end SD_Subp;

subprogram implementation SD_Subp.i
  properties
    Alloy::invariant => "
      all m: RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
          parameterValues) | one m.zaM
      all t: RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s
          .parameterValues) | one t.zaT
    ";
    Alloy::post_edp => "
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and no GenerateAlarm_i =>
      -- send temperature measurement
      { RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out ->
          RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
          parameterValues) }
    else one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and one GenerateAlarm_i =>
      -- send temperature alarm
      { RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out -> { a:
          Alarm_i | a.sensorA =
          RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
          parameterValues).sensorM and one a.zaA } }
    else
      none -> none
  ";
    Alloy::post_diProduced => "
      one RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s.
          parameterValues) =>
      -- store temperature threshold
      { getDataInstance[
          RPM_RPM_i_Instance_sd_proc1_thread1_c1_temperature_threshold
          ] -> RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in
          .(s.parameterValues) }
    else
      none -> none
    ";
end SD_Subp.i;
```

```
-- SDC --
system SDC
  features
    measurement_in: in event data port Measurement.i;
    measurement_out: out event data port Measurement.i;
    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;
end SDC;

system implementation SDC.i
  subcomponents
    proc1: process SDC_Proc.i;

    zd_coupling: data ZigBeeDevice.i;
  connections
    pc1: event data port measurement_in -> proc1.measurement_in;
    pc2: event data port proc1.measurement_out -> measurement_out;
    pc3: event data port alarm_in -> proc1.alarm_in;
    pc4: event data port proc1.alarm_out -> alarm_out;
    pc5: event data port threshold_in -> proc1.threshold_in;
    pc6: event data port proc1.threshold_out -> threshold_out;

    dac1: data access zd_coupling -> proc1.zd_coupling;
end SDC.i;

process SDC_Proc
  features
    measurement_in: in event data port Measurement.i;
    measurement_out: out event data port Measurement.i;
    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    zd_coupling: requires data access ZigBeeDevice.i;
end SDC_Proc;

process implementation SDC_Proc.i
  subcomponents
    thread1: thread SDC_Thread.i;
  connections
    pc1: event data port measurement_in -> thread1.measurement_in;
    pc2: event data port alarm_in -> thread1.alarm_in;
    pc3: event data port thread1.measurement_out -> measurement_out
        ;
    pc4: event data port thread1.alarm_out -> alarm_out;
    pc5: event data port threshold_in -> thread1.threshold_in;
    pc6: event data port thread1.threshold_out -> threshold_out;

    dac1: data access zd_coupling-> thread1.zd_coupling;
end SDC_Proc.i;

thread SDC_Thread
  features
    measurement_in: in event data port Measurement.i;
    measurement_out: out event data port Measurement.i;
    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
```

```
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    zd_coupling: requires data access ZigBeeDevice.i;
end SDC_Thread;

thread implementation SDC_Thread.i
  calls
    calls1: {
      c1: subprogram SDC_Subp.i;
    };
  connections
    pc1: parameter measurement_in -> c1.measurement_in;
    pc2: event data port c1.measurement_out -> measurement_out;
    pc3: parameter alarm_in -> c1.alarm_in;
    pc4: event data port c1.alarm_out -> alarm_out;
    pc5: parameter threshold_in -> c1.threshold_in;
    pc6: event data port c1.threshold_out -> threshold_out;

    dac1: data access zd_coupling -> c1.zd_coupling;
end SDC_Thread.i;

subprogram SDC_Subp
  features
    measurement_in: in parameter Measurement.i;
    measurement_out: out event data port Measurement.i;
    alarm_in: in parameter Alarm.i;
    alarm_out: out event data port  Alarm.i;
    threshold_in: in parameter Threshold.i;
    threshold_out: out event data port Threshold.i;

    zd_coupling: requires data access ZigBeeDevice.i;
end SDC_Subp;

subprogram implementation SDC_Subp.i
  properties
    Alloy::invariant => "
      all m: RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in
          .(s.parameterValues) | one m.zaM
      all a: RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_in.(s.
          parameterValues) | one a.zaA
      all t: RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_in.(
          s.parameterValues) | one t.devT
    ";
    Alloy::post_edp => "
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in.(s.
        parameterValues) =>
      -- send temperature measurement
      { RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_out ->
        { m: Measurement_i | m.devM =
            RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in
            .(s.parameterValues).zaM.(getDataInstance[
            RPM_RPM_i_Instance_sdc_proc1_thread1_c1_zd_coupling].(s
            .dataInstanceValues).zigBeeToDevice) } }
    else one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_in.(s.
        parameterValues) =>
      -- send temperature alarm
      { RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_out ->
        { a: Alarm_i | a.devA =
            RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_in.(s.
            parameterValues).zaA.(getDataInstance[
            RPM_RPM_i_Instance_sdc_proc1_thread1_c1_zd_coupling].(s
```

```
            .dataInstanceValues).zigBeeToDevice) } }
    else one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_in.(
        s.parameterValues) =>
      -- send temperature threshold
      { RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_out ->
        { t: Threshold_i | t.zaT =
            RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_in.(s
            .parameterValues).devT.~(getDataInstance[
            RPM_RPM_i_Instance_sdc_proc1_thread1_c1_zd_coupling].(s
            .dataInstanceValues).zigBeeToDevice) } }
    else
      none -> none
    ";
end SDC_Subp.i;


--- HPC ---
system HPC
  features
    measurement_in: in event data port Measurement.i;

    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;

    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    measurements_request_in: in event data port MeasurementsRequest
        .i;
    alarms_request_in: in event data port AlarmsRequest.i;

    measurements_out: out event data port Measurement.i;
    alarms_out: out event data port Alarm.i;
end HPC;

system implementation HPC.i
  subcomponents
    sdm: process SDM_Proc.i;
    as: process AS_Proc.i;
    ws: process WS_Proc.i;

    temperature_measurements: data Measurement.i {
      Alloy::invariant => "all m: this.(s.dataInstanceValues) | m.
          sensorM = RPM_RPM_i_Instance_sd_sensor1";
    };
    temperature_alarms: data Alarm.i {
      Alloy::invariant => "all a: this.(s.dataInstanceValues) | a.
          sensorA = RPM_RPM_i_Instance_sd_sensor1";
    };

    dp_coupling: data DevicePatient.i {
      Alloy::invariant => "";
    };
  connections
    pc1: event data port measurement_in -> sdm.measurement_in;

    pc2: event data port alarm_in -> sdm.alarm_in;
    pc3: event data port sdm.alarm_out -> as.alarm_in;
    pc4: event data port as.alarm_out -> alarm_out;

    pc5: event data port threshold_in -> ws.threshold_in;
    pc6: event data port ws.threshold_out -> sdm.threshold_in;
```

```
    pc7: event data port sdm.threshold_out -> threshold_out;

    pc8: event data port measurements_request_in -> ws.
        measurements_request_in;
    pc9: event data port alarms_request_in -> ws.alarms_request_in;

    pc10: event data port ws.measurements_out -> measurements_out;
    pc11: event data port ws.alarms_out -> alarms_out;

    dac1: data access temperature_measurements -> sdm.
        temperature_measurements;
    dac2: data access temperature_alarms -> sdm.temperature_alarms;

    dac3: data access temperature_measurements -> ws.
        temperature_measurements;
    dac4: data access temperature_alarms -> ws.temperature_alarms;

    dac5: data access dp_coupling -> sdm.dp_coupling;
end HPC.i;

-- SDM --
process SDM_Proc
  features
    measurement_in: in event data port Measurement.i;
    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    temperature_measurements: requires data access Measurement.i;
    temperature_alarms: requires data access Alarm.i;

    dp_coupling: requires data access DevicePatient.i;
end SDM_Proc;

process implementation SDM_Proc.i
  subcomponents
    thread1: thread SDM_Thread.i;
  connections
    pc1: event data port measurement_in -> thread1.measurement_in;

    pc2: event data port alarm_in -> thread1.alarm_in;
    pc3: event data port thread1.alarm_out -> alarm_out;

    pc4: event data port threshold_in -> thread1.threshold_in;
    pc5: event data port thread1.threshold_out -> threshold_out;

    dac1: data access temperature_measurements -> thread1.
        temperature_measurements;
    dac2: data access temperature_alarms -> thread1.
        temperature_alarms;

    dac3: data access dp_coupling -> thread1.dp_coupling;
end SDM_Proc.i;

thread SDM_Thread
  features
    measurement_in: in event data port Measurement.i;

    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
```

```
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    temperature_measurements: requires data access Measurement.i;
    temperature_alarms: requires data access Alarm.i;

    dp_coupling: requires data access DevicePatient.i;
end SDM_Thread;

thread implementation SDM_Thread.i
  calls
    calls1: {
      c1: subprogram SDM_Subp.i;
    };
  connections
    pc1: parameter measurement_in -> c1.measurement_in;

    pc2: parameter alarm_in -> c1.alarm_in;
    pc3: event data port c1.alarm_out -> alarm_out;

    pc4: parameter threshold_in -> c1.threshold_in;
    pc5: event data port c1.threshold_out -> threshold_out;

    dac1: data access temperature_measurements -> c1.
        temperature_measurements;
    dac2: data access temperature_alarms -> c1.temperature_alarms;

    dac3: data access dp_coupling -> c1.dp_coupling;
end SDM_Thread.i;

subprogram SDM_Subp
  features
    measurement_in: in parameter Measurement.i;
    alarm_in: in parameter Alarm.i;
    alarm_out: out event data port Alarm.i;
    threshold_in: in parameter Threshold.i;
    threshold_out: out event data port Threshold.i;

    temperature_measurements: requires data access Measurement.i;
    temperature_alarms: requires data access Alarm.i;

    dp_coupling: requires data access DevicePatient.i;
end SDM_Subp;

subprogram implementation SDM_Subp.i
  properties
    Alloy::invariant => "
      all m: RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(
          s.parameterValues) | one m.devM
      all a: RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
          parameterValues) | one a.devA
      all t: RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_in.(s.
          parameterValues) | one t.patT
    ";
    Alloy::post_edp => "
      one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
          parameterValues) =>
        -- send temperature alarm
      { RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_out ->
        { a: Alarm_i | a.patA =
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
            parameterValues).devA.(getDataInstance[
```

```
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_dp_coupling].(s.
            dataInstanceValues).deviceToPatient) } }
    else one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_in.(s.
        parameterValues) =>
      -- send temperature threshold
      { RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_out ->
        { t: Threshold_i | t.devT =
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_in.(s.
            parameterValues).patT.~(getDataInstance[
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_dp_coupling].(s.
            dataInstanceValues).deviceToPatient) } }
    else
      none -> none
  ";
    Alloy::post_diProduced => "
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(s.
        parameterValues) =>
        -- store temperature measurement
      { getDataInstance[
          RPM_RPM_i_Instance_hpc_sdm_thread1_c1_temperature_measurements
          ] ->
        { m: Measurement_i | m.patM =
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(s
            .parameterValues).devM.(getDataInstance[
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_dp_coupling].(s.
            dataInstanceValues).deviceToPatient) } }
    else one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
        parameterValues) =>
        -- store temperature alarm
      { getDataInstance[
          RPM_RPM_i_Instance_hpc_sdm_thread1_c1_temperature_alarms]
          ->
        { a: Alarm_i | a.patA =
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
            parameterValues).devA.(getDataInstance[
            RPM_RPM_i_Instance_hpc_sdm_thread1_c1_dp_coupling].(s.
            dataInstanceValues).deviceToPatient) } }
    else
      none -> none
    ";
end SDM_Subp.i;

-- AS --
process AS_Proc
  features
    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
end AS_Proc;

process implementation AS_Proc.i
  subcomponents
    thread1: thread AS_Thread.i;
  connections
    pc1: event data port alarm_in -> thread1.alarm_in;
    pc2: event data port thread1.alarm_out -> alarm_out;
end AS_Proc.i;

thread AS_Thread
  features
    alarm_in: in event data port Alarm.i;
    alarm_out: out event data port Alarm.i;
end AS_Thread;
```

```
thread implementation AS_Thread.i
  calls
    calls1: {
      c1: subprogram AS_Subp.i;
    };
  connections
    pc1: parameter alarm_in -> c1.alarm_in;
    pc2: event data port c1.alarm_out -> alarm_out;
end AS_Thread.i;

subprogram AS_Subp
  features
    alarm_in: in parameter Alarm.i;
    alarm_out: out event data port Alarm.i;
end AS_Subp;

subprogram implementation AS_Subp.i
  properties
    Alloy::invariant => "
      all a: RPM_RPM_i_Instance_hpc_as_thread1_c1_alarm_in.(s.
          parameterValues) | one a.patA
    ";
    Alloy::post_edp => "
      one RPM_RPM_i_Instance_hpc_as_thread1_c1_alarm_in.(s.
          parameterValues) =>
        -- send temperature alarm
        { RPM_RPM_i_Instance_hpc_as_thread1_c1_alarm_out ->
            RPM_RPM_i_Instance_hpc_as_thread1_c1_alarm_in.(s.
            parameterValues) }
      else
        none -> none
    ";
end AS_Subp.i;

-- WS --
process WS_Proc
  features
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    measurements_request_in: in event data port MeasurementsRequest
        .i;
    alarms_request_in: in event data port AlarmsRequest.i;

    measurements_out: out event data port Measurement.i;
    alarms_out: out event data port Alarm.i;

    temperature_measurements: requires data access Measurement.i;
    temperature_alarms: requires data access Alarm.i;
end WS_Proc;

process implementation WS_Proc.i
  subcomponents
    thread1: thread WS_Thread.i;
  connections
    pc1: event data port threshold_in -> thread1.threshold_in;
    pc2: event data port thread1.threshold_out -> threshold_out;

    pc3: event data port measurements_request_in -> thread1.
        measurements_request_in;
    pc4: event data port alarms_request_in -> thread1.
```

```
        alarms_request_in;

    pc5: event data port thread1.measurements_out ->
        measurements_out;
    pc6: event data port thread1.alarms_out -> alarms_out;

  dac1: data access temperature_measurements -> thread1.
      temperature_measurements;
  dac2: data access temperature_alarms -> thread1.
      temperature_alarms;
end WS_Proc.i;

thread WS_Thread
  features
    threshold_in: in event data port Threshold.i;
    threshold_out: out event data port Threshold.i;

    measurements_request_in: in event data port MeasurementsRequest
        .i;
    alarms_request_in: in event data port AlarmsRequest.i;

    measurements_out: out event data port Measurement.i;
    alarms_out: out event data port Alarm.i;

    temperature_measurements: requires data access Measurement.i;
    temperature_alarms: requires data access Alarm.i;
end WS_Thread;

thread implementation WS_Thread.i
  calls
    calls1: {
      c1: subprogram WS_Subp.i;
    };
  connections
    pc1: parameter threshold_in -> c1.threshold_in;
    pc2: event data port c1.threshold_out -> threshold_out;

    pc3: parameter measurements_request_in -> c1.
        measurements_request_in;
    pc4: parameter alarms_request_in -> c1.alarms_request_in;

    pc5: event data port c1.measurements_out -> measurements_out;
    pc6: event data port c1.alarms_out -> alarms_out;

  dac1: data access temperature_measurements -> c1.
      temperature_measurements;
  dac2: data access temperature_alarms -> c1.temperature_alarms;
end WS_Thread.i;

subprogram WS_Subp
  features
    threshold_in: in parameter Threshold.i;
    threshold_out: out event data port Threshold.i;

    measurements_request_in: in parameter MeasurementsRequest.i;
    alarms_request_in: in parameter AlarmsRequest.i;

    measurements_out: out event data port Measurement.i;
    alarms_out: out event data port Alarm.i;

    temperature_measurements: requires data access Measurement.i;
    temperature_alarms: requires data access Alarm.i;
```

```
end WS_Subp;

subprogram implementation WS_Subp.i
  properties
    Alloy::invariant => "
      all t: RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_in.(s.
          parameterValues) | one t.patT
    ";
    Alloy::post_edp => "
      one RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_in.(s.
          parameterValues) =>
        -- send temperature threshold
        { RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_out ->
            RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_in.(s.
            parameterValues) }
      else one
          RPM_RPM_i_Instance_hpc_ws_thread1_c1_measurements_request_in
          .(s.parameterValues) =>
        -- send temperature measurement
        { RPM_RPM_i_Instance_hpc_ws_thread1_c1_measurements_out ->
            getDataInstance[
            RPM_RPM_i_Instance_hpc_ws_thread1_c1_temperature_measurements
            ].(s.dataInstanceValues) }
      else one
          RPM_RPM_i_Instance_hpc_ws_thread1_c1_alarms_request_in.(s
          .parameterValues) =>
        -- send temperature alarm
        { RPM_RPM_i_Instance_hpc_ws_thread1_c1_alarms_out ->
            getDataInstance[
            RPM_RPM_i_Instance_hpc_ws_thread1_c1_temperature_alarms
            ].(s.dataInstanceValues) }
      else
        none -> none
    ";
end WS_Subp.i;


--- CPC ---
system CPC
  features
    alarm_in: in event data port Alarm.i;
    threshold_out: out event data port Threshold.i;

    measurements_request_out: out event data port
        MeasurementsRequest.i;
    alarms_request_out: out event data port AlarmsRequest.i;

    measurements_in: in event data port Measurement.i;
    alarms_in: in event data port Alarm.i;
end CPC;

system implementation CPC.i
  subcomponents
    wc: process WC_Proc.i;
    ar: process AR_Proc.i;
  connections
    pc1: event data port alarm_in -> ar.alarm_in;
    pc2: event data port wc.threshold_out -> threshold_out;

    pc3: event data port wc.measurements_request_out ->
        measurements_request_out;
    pc4: event data port wc.alarms_request_out ->
```

```
        alarms_request_out;

    pc5: event data port measurements_in -> wc.measurements_in;
    pc6: event data port alarms_in -> wc.alarms_in;
end CPC.i;

-- AR --
process AR_Proc
  features
    alarm_in: in event data port Alarm.i;
end AR_Proc;

process implementation AR_Proc.i
  subcomponents
    thread1: thread AR_Thread.i;
  connections
    pc1: event data port alarm_in -> thread1.alarm_in;
end AR_Proc.i;

thread AR_Thread
  features
    alarm_in: in event data port Alarm.i;
end AR_Thread;

thread implementation AR_Thread.i
  calls
    calls1: {
      c1: subprogram AR_Subp.i;
    };
  connections
    pc1: parameter alarm_in -> c1.alarm_in;
end AR_Thread.i;

subprogram AR_Subp
  features
    alarm_in: in parameter Alarm.i;
end AR_Subp;

subprogram implementation AR_Subp.i
  properties
    Alloy::invariant => "
      all a: RPM_RPM_i_Instance_cpc_ar_thread1_c1_alarm_in.(s.
          parameterValues) | one a.patA
    ";
end AR_Subp.i;

-- WC --
process WC_Proc
  features
    threshold_out: out event data port Threshold.i;

    measurements_request_out: out event data port
        MeasurementsRequest.i;
    alarms_request_out: out event data port AlarmsRequest.i;

    measurements_in: in event data port Measurement.i;
    alarms_in: in event data port Alarm.i;
end WC_Proc;

process implementation WC_Proc.i
  subcomponents
    thread1: thread WC_Thread.i;
```

```
  connections
    pc1: event data port thread1.threshold_out -> threshold_out;

    pc2: event data port thread1.measurements_request_out ->
        measurements_request_out;
    pc3: event data port thread1.alarms_request_out ->
        alarms_request_out;

    pc4: event data port measurements_in -> thread1.measurements_in
        ;
    pc5: event data port alarms_in -> thread1.alarms_in;
end WC_Proc.i;

thread WC_Thread
  features
    threshold_out: out event data port Threshold.i;

    measurements_request_out: out event data port
        MeasurementsRequest.i;
    alarms_request_out: out event data port AlarmsRequest.i;

    measurements_in: in event data port Measurement.i;
    alarms_in: in event data port Alarm.i;
end WC_Thread;

thread implementation WC_Thread.i
  calls
    calls1: {
      c1: subprogram WC_Subp.i;
    };
  connections
    pc1: event data port c1.threshold_out -> threshold_out;

    pc2: event data port c1.measurements_request_out ->
        measurements_request_out;
    pc3: event data port c1.alarms_request_out ->
        alarms_request_out;

    pc4: parameter measurements_in -> c1.measurements_in;
    pc5: parameter alarms_in -> c1.alarms_in;
end WC_Thread.i;

subprogram WC_Subp
  features
    threshold_out: out event data port Threshold.i;

    measurements_request_out: out event data port
        MeasurementsRequest.i;
    alarms_request_out: out event data port AlarmsRequest.i;

    measurements_in: in parameter Measurement.i;
    alarms_in: in parameter Alarm.i;
end WC_Subp;

subprogram implementation WC_Subp.i
end WC_Subp.i;


--- RPM ---
system RPM
end RPM;
```

```
system implementation RPM.i
  subcomponents
    sd: system SD.i;
    sdc: system SDC.i;
    hpc: system HPC.i;
    cpc: system CPC.i;
  connections
    pc1: event data port sd.measurement_out -> sdc.measurement_in;
    pc2: event data port sdc.measurement_out -> hpc.measurement_in;

    pc3: event data port sd.alarm_out -> sdc.alarm_in;
    pc4: event data port sdc.alarm_out -> hpc.alarm_in;
    pc5: event data port hpc.alarm_out -> cpc.alarm_in;

    pc6: event data port cpc.threshold_out -> hpc.threshold_in;
    pc7: event data port hpc.threshold_out -> sdc.threshold_in;
    pc8: event data port sdc.threshold_out -> sd.threshold_in;

    pc9: event data port cpc.measurements_request_out -> hpc.
        measurements_request_in;
    pc10: event data port cpc.alarms_request_out -> hpc.
        alarms_request_in;

    pc11: event data port hpc.measurements_out -> cpc.
        measurements_in;
    pc12: event data port hpc.alarms_out -> cpc.alarms_in;
  properties
    Alloy::scenarios => "...";
end RPM.i;
```

B.3 Formalized requirements and scenarios

The informal requirements, are translated into the solution domain. For each requirement (Req.), a refinement (Ref. req.) and a formal behavior description is given. The following format for each formal behavior description is used:

- Data type;
- From port (feature instance in model), or data instance;
- Next or subsequent state ($\rightarrow$ or $\Rightarrow$, respectively);
- To port (feature instance in model), or data instance.

The next or subsequent state specifies whether the data value must be at the to-port in directly the next state, or must be at the to-port in a subsequent state. If the next or subsequent state and to-port are not present, it means that the binding of a data value to the from-port must be satisfied for only one state.

An informal description has been omitted in this table for brevity. One is able to extract an informal description based on the formal behavior description by looking at the involved ports and data instances.

The requirement predicates and scenarios validating the requirement predicates of the RPM architecture are given in the listing below.

Listing B.2: Formalized RPM requirements and scenarios

```
-- Requirements --
pred R1 {
```

Req.	Ref. req.	Formal behavior description
R1		Temperature measurement: sd.sensor1.measurement ⇒ sd.proc1.thread1.c1.measurement_out
	R1_a	Temperature measurement: sd.sensor1.measurement → sd.proc1.thread1.sensor1_in
	R1_b	Temperature measurement: sd.proc1.thread1.sensor1_in → sd.proc1.thread1.c1.sensor1_in
	R1_c	Temperature measurement: sd.proc1.thread1.c1.sensor1_in → sd.proc1.thread1.c1.measurement_out
R2		Temperature measurement: sd.proc1.thread1.c1.measurement_out ⇒ hpc.sdm.thread1.c1.measurement_in
	R2_a	Temperature measurement: sd.proc1.thread1.c1.measurement_out ⇒ sdc.proc1.thread1.c1.measurement_in
	R2_b	Temperature measurement: sdc.proc1.thread1.c1.measurement_in → sdc.proc1.thread1.c1.measurement_out
	R2_c	Temperature measurement: sdc.proc1.thread1.c1.measurement_out ⇒ hpc.sdm.thread1.c1.measurement_in
R3	R3	Temperature measurement: hpc.sdm.thread1.c1.measurement_in → hpc.temperature_measurements
R4		Temperature alarm: sd.sensor1.measurement ⇒ sd.proc1.thread1.c1.alarm_out
	R4_a	Temperature measurement: sd.sensor1.measurement → sd.proc1.thread1.c1.sensor1_in
	R4_b	Temperature alarm: sd.proc1.thread1.c1.sensor1_in → sd.proc1.thread1.c1.alarm_out
R5		Temperature alarm: sd.proc1.thread1.c1.alarm_out ⇒ hpc.sdm.thread1.c1.alarm_in
	R5_a	Temperature alarm: sd.proc1.thread1.c1.alarm_out → sdc.proc1.thread1.c1.alarm_in
	R5_b	Temperature alarm: sdc.proc1.thread1.c1.alarm_in → sdc.proc1.thread1.c1.alarm_out
	R5_c	Temperature alarm: sdc.proc1.thread1.c1.alarm_out → hpc.sdm.thread1.c1.alarm_in
R6	R6	Temperature alarm: hpc.sdm.thread1.c1.alarm_in → hpc.temperature_alarms
R7	R7	Temperature alarm: sd.proc1.thread1.c1.alarm_out ⇒ cpc.ar.thread1.c1.alarm_in
R8	R8	Temperature alarm: cpc.ar.thread1.c1.alarm_in
R9	R9	Temperature threshold: cpc.wc.thread1.c1.threshold_out

Table B.2: Informal RPM requirements translated to solution domain

Req.	Ref. req.	Formal behavior description
R10		Temperature threshold: cpc.wc.thread1.c1.threshold_out ⇒ sd.proc1.thread1.c1.threshold_in
	R10_a	Temperature threshold: cpc.wc.thread1.c1.threshold_out → hpc.ws.thread1.c1.threshold_in
	R10_b	Temperature threshold: hpc.ws.thread1.c1.threshold_in → hpc.ws.thread1.c1.threshold_out
	R10_c	Temperature threshold: hpc.ws.thread1.c1.threshold_out → hpc.sdm.thread1.c1.threshold_in
	R10_d	Temperature threshold: hpc.sdm.thread1.c1.threshold_in → hpc.sdm.thread1.c1.threshold_out
	R10_e	Temperature threshold: hpc.sdm.thread1.c1.threshold_out → sdc.proc1.thread1.c1.threshold_in
	R10_f	Temperature threshold: sdc.proc1.thread1.c1.threshold_in → sdc.proc1.thread1.c1.threshold_out
	R10_g	Temperature threshold: sdc.proc1.thread1.c1.threshold_out → sd.proc1.thread1.c1.threshold_in
R11	R11	Temperature threshold: sd.proc1.thread1.c1.threshold_in → sd.temperature_threshold
R15	R15	Temperature measurement request: cpc.wc.thread1.measurements_request_out ⇒ cpc.wc.thread1.measurements_in
R16	R16	Temperature alarm request: cpc.wc.thread1.alarms_request_out ⇒ cpc.wc.thread1.alarms_in

Table B.3: Informal RPM requirements translated to solution domain (continued)

```
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.^
        next.eventDataPortValues)
}
pred R1_a {
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in.(s.next.
        eventDataPortValues)
}
pred R1_b {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.next.
        parameterValues)
}
pred R1_c {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.
        next.eventDataPortValues)
}

pred R2 {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(s.^
        next.parameterValues)
}
pred R2_a {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in.(s.^
        next.parameterValues)
}
pred R2_b {
  one s: State |
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_out.(s.
        next.eventDataPortValues)
}
pred R2_c {
  one s: State |
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_measurement_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(s.^
        next.parameterValues)
}

pred R3 {
  one s: State |
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_measurement_in.(s.
        parameterValues) and
```

```
    one RPM_RPM_i_Instance_hpc_temperature_measurements.(s.next.
        dataInstanceValues)
}

pred R4 {
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out.(s.^next.
        eventDataPortValues)
}
pred R4_a {
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.next.
        parameterValues)
}

pred R4_b {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out.(s.next.
        eventDataPortValues)
}

pred R5 {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.^next.
        parameterValues)
}
pred R5_a {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_in.(s.next.
        parameterValues)
}
pred R5_b {
  one s: State |
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_out.(s.next.
        eventDataPortValues)
}
pred R5_c {
  one s: State |
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_alarm_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.next.
        parameterValues)
}

pred R6 {
  one s: State |
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_alarm_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_hpc_temperature_alarms.(s.next.
        dataInstanceValues)
```

```
}

pred R7 {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_alarm_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_cpc_ar_thread1_c1_alarm_in.(s.^next.
        parameterValues)
}

pred R8 {
  some s: State |
    one RPM_RPM_i_Instance_cpc_ar_thread1_c1_alarm_in.(s.
        parameterValues)
}

pred R9 {
  some s: State |
    one RPM_RPM_i_Instance_cpc_wc_thread1_threshold_out.(s.
        eventDataPortValues)
}

pred R10 {
  one s: State |
    one RPM_RPM_i_Instance_cpc_wc_thread1_c1_threshold_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s.^
        next.parameterValues)
}
pred R10_a {
  one s: State |
    one RPM_RPM_i_Instance_cpc_wc_thread1_c1_threshold_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_in.(s.next.
        parameterValues)
}
pred R10_b {
  one s: State |
    one RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_out.(s.next.
        eventDataPortValues)
}
pred R10_c {
  one s: State |
    one RPM_RPM_i_Instance_hpc_ws_thread1_c1_threshold_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_in.(s.next.
        parameterValues)
}
pred R10_d {
  one s: State |
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_out.(s.next
        .eventDataPortValues)
}
pred R10_e {
  one s: State |
    one RPM_RPM_i_Instance_hpc_sdm_thread1_c1_threshold_out.(s.
        eventDataPortValues) and
```

```
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_in.(s.
        next.parameterValues)
}
pred R10_f {
  one s: State |
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_out.(s.
        next.eventDataPortValues)
}
pred R10_g {
  one s: State |
    one RPM_RPM_i_Instance_sdc_proc1_thread1_c1_threshold_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s.next
        .parameterValues)
}

pred R11 {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sd_temperature_threshold.(s.next.
        dataInstanceValues)
}

pred R15 {
  one s: State |
    one
        RPM_RPM_i_Instance_cpc_wc_thread1_c1_measurements_request_out
        .(s.eventDataPortValues) and
    one RPM_RPM_i_Instance_cpc_wc_thread1_c1_measurements_in.(s.^
        next.parameterValues)
}

pred R16 {
  one s: State |
    one RPM_RPM_i_Instance_cpc_wc_thread1_c1_alarms_request_out.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_cpc_wc_thread1_c1_alarms_in.(s.^next.
        parameterValues)
}

--- Scenario 1: perform measurement ---
pred scenario1_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch

  one s.eventDataPortValues
  one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
      eventDataPortValues).zaM
  all m: Measurement_i | m.sensorM = RPM_RPM_i_Instance_sd_sensor1
  one m: Measurement_i | m.zaM = ZigBeeAddress_i
  one m: Measurement_i | m.devM = Dev_i
  one m: Measurement_i | m.patM = Patient_i

  no (DataInstance - RPM_RPM_i_Instance_sdc_zd_coupling -
      RPM_RPM_i_Instance_hpc_dp_coupling).(s.dataInstanceValues)
  one RPM_RPM_i_Instance_sdc_zd_coupling.(s.dataInstanceValues)
  one RPM_RPM_i_Instance_hpc_dp_coupling.(s.dataInstanceValues)
```

```
}
Check_scenario1 : check {
  (StateTransitions and scenario1_initial[so/first]) =>
  (ArchitectureInvariants and
  R1 and R2 and R3)
  -- R1_a and R1_b and R2_a and R2_b and R2_c and R3)
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  exactly 25 State,
  exactly 3 Measurement_i, exactly 0 Alarm_i, exactly 0 Threshold_i
      , exactly 0 GenerateAlarm_i, exactly 1 ZigBeeAddress_i,
      exactly 1 Dev_i, exactly 1 Patient_i, exactly 1
      ZigBeeDevice_i, exactly 1 DevicePatient_i, exactly 0
      MeasurementsRequest_i, exactly 0 AlarmsRequest_i

--- Scenario 2: send alarm ---
pred scenario2_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch

  one s.eventDataPortValues
  one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
      eventDataPortValues).zaM

  Measurement_i.sensorM = RPM_RPM_i_Instance_sd_sensor1
  all a: Alarm_i | a.sensorA = RPM_RPM_i_Instance_sd_sensor1
  one a: Alarm_i | a.zaA = ZigBeeAddress_i
  one a: Alarm_i | a.devA = Dev_i
  one a: Alarm_i | a.patA = Patient_i

  no (DataInstance - RPM_RPM_i_Instance_sdc_zd_coupling -
      RPM_RPM_i_Instance_hpc_dp_coupling).(s.dataInstanceValues)
  one RPM_RPM_i_Instance_sdc_zd_coupling.(s.dataInstanceValues)
  one RPM_RPM_i_Instance_hpc_dp_coupling.(s.dataInstanceValues)
}
Check_scenario2 : check {
  (StateTransitions and scenario2_initial[so/first]) =>
  (ArchitectureInvariants and
  R4 and R5 and R6 and R7 and R8)
  -- R4_a and R4_b and R5_a and R5_b and R5_c and R6 and R7 and R8)
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  exactly 25 State, exactly 1 Measurement_i, exactly 3 Alarm_i,
      exactly 0 Threshold_i, exactly 1 GenerateAlarm_i, exactly 1
      ZigBeeAddress_i, exactly 1 Dev_i, exactly 1 Patient_i,
      exactly 1 ZigBeeDevice_i, exactly 1 DevicePatient_i, exactly
      0 MeasurementsRequest_i, exactly 0 AlarmsRequest_i

--- Scenario 3: set threshold ---
pred scenario3_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch
```

```
  one s.eventDataPortValues
  one RPM_RPM_i_Instance_cpc_wc_thread1_c1_threshold_out.(s.
      eventDataPortValues).patT
  all t: Threshold_i | t.sensorT = RPM_RPM_i_Instance_sd_sensor1
  one t: Threshold_i | t.zaT = ZigBeeAddress_i
  one t: Threshold_i | t.devT = Dev_i
  one t: Threshold_i | t.patT = Patient_i

  no (DataInstance - RPM_RPM_i_Instance_sdc_zd_coupling -
      RPM_RPM_i_Instance_hpc_dp_coupling).(s.dataInstanceValues)
  one RPM_RPM_i_Instance_sdc_zd_coupling.(s.dataInstanceValues)
  one RPM_RPM_i_Instance_hpc_dp_coupling.(s.dataInstanceValues)
}
Check_scenario3 : check {
  (StateTransitions and scenario3_initial[so/first]) =>
  (ArchitectureInvariants and
  R9 and R10 and R11)
  -- R9 and R10_a and R10_b and R10_c and R10_d and R10_e and R10_f
       and R10_g and R11)
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  exactly 25 State, exactly 0 Measurement_i, exactly 0 Alarm_i,
      exactly 3 Threshold_i, exactly 0 GenerateAlarm_i, exactly 1
      ZigBeeAddress_i, exactly 1 Dev_i, exactly 1 Patient_i,
      exactly 1 ZigBeeDevice_i, exactly 1 DevicePatient_i, exactly
      0 MeasurementsRequest_i, exactly 0 AlarmsRequest_i

--- Scenario 4: request measurements ---
pred scenario4_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch

  no (EventDataPort -
      RPM_RPM_i_Instance_cpc_wc_thread1_c1_measurements_request_out
      ).(s.eventDataPortValues)
  one RPM_RPM_i_Instance_cpc_wc_thread1_c1_measurements_request_out
      .(s.eventDataPortValues)
  all r: MeasurementsRequest_i | r.sensorMR =
      RPM_RPM_i_Instance_sd_sensor1

  no (DataInstance -
      RPM_RPM_i_Instance_hpc_temperature_measurements).(s.
      dataInstanceValues)
  one RPM_RPM_i_Instance_hpc_temperature_measurements.(s.
      dataInstanceValues)
  all m: Measurement_i | m.patM = Patient_i
}
Check_scenario4 : check {
  (StateTransitions and scenario4_initial[so/first]) =>
  (ArchitectureInvariants and
  R15)
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
```

```
  exactly 25 State, exactly 1 Measurement_i, exactly 0 Alarm_i,
      exactly 0 Threshold_i, exactly 0 GenerateAlarm_i, exactly 0
      ZigBeeAddress_i, exactly 0 Dev_i, exactly 1 Patient_i,
      exactly 0 ZigBeeDevice_i, exactly 0 DevicePatient_i, exactly
      1 MeasurementsRequest_i, exactly 0 AlarmsRequest_i

--- Scenario 5: request alarms ---
pred scenario5_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch

  no (EventDataPort -
      RPM_RPM_i_Instance_cpc_wc_thread1_c1_alarms_request_out).(s.
      eventDataPortValues)
  one RPM_RPM_i_Instance_cpc_wc_thread1_c1_alarms_request_out.(s.
      eventDataPortValues)
  all r: AlarmsRequest_i | r.sensorAR =
      RPM_RPM_i_Instance_sd_sensor1

  no (DataInstance - RPM_RPM_i_Instance_hpc_temperature_alarms).(s.
      dataInstanceValues)
  one RPM_RPM_i_Instance_hpc_temperature_alarms.(s.
      dataInstanceValues)
  all a: Alarm_i | a.patA = Patient_i
}
Check_scenario5 : check {
  (StateTransitions and scenario5_initial[so/first]) =>
  (ArchitectureInvariants and
  R16)
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  exactly 25 State, exactly 0 Measurement_i, exactly 1 Alarm_i,
      exactly 0 Threshold_i, exactly 0 GenerateAlarm_i, exactly 0
      ZigBeeAddress_i, exactly 0 Dev_i, exactly 1 Patient_i,
      exactly 0 ZigBeeDevice_i, exactly 0 DevicePatient_i, exactly
      0 MeasurementsRequest_i, exactly 1 AlarmsRequest_i
```

B.4 Traces from requirements to architecture

Table B.4 gives the trace-relations from the requirements to the component instances of the RPM instance model given in listing B.1. The names of the formalized versions of the requirements are given in the column predicates. Some of the formalized versions of the requirements have been split in two or more predicates, to improve the traceability from the requirements to the architecture.

The requirements R12, R13, and R14 are not traced to any components. These requirements are non-functional requirements, and thus cannot be validated by our approach.

Req.	Pred.	Component
R1	R1_a	sd.sensor1, sd.proc1.thread1.c1 (pc)
	R1_b	sd.proc1.thread1, sd.proc1.thread1.c1 (pc)
	R1_c	sd.proc1.thread1.c1
R2	R2_a	sd.proc1.thread1.c1, sdc.proc1.thread1.c1 (pc)
	R2_b	sdc.proc1.thread1.c1
	R2_c	sdc.proc1.thread1.c1, hpc.sdm.thread1.c1 (pc)
R3	R3	hpc.sdm.thread1.c1, hpc.temperature_measurements
R4	R4_a	sd.sensor1, sd.proc1.thread1.c1 (pc)
	R4_b	sd.proc1.thread1.c1
R5	R5_a	sd.proc1.thread1.c1, sdc.proc1.thread1.c1 (pc)
	R5_b	sdc.proc1.thread1.c1
	R5_c	sdc.proc1.thread1.c1, hpc.sdm.thread1.c1 (pc)
R6	R6	hpc.sdm.thread1.c1, hpc.temperature_alarms
R7	R7	sd.proc1.thread1.c1, sdc.proc1.thread1.c1, hpc.sdm.thread1.c1, hpc.as.thread1.c1, cpc.ar.thread1.c1 (pc, comp)
R8	R8	cpc.ar.thread1.c1
R9	R9	cpc.wc.thread1.c1
R10	R10_a	cpc.wc.thread1.c1, hpc.ws.thread1.c1 (pc)
	R10_b	hpc.ws.thread1.c1
	R10_c	hpc.ws.thread1.c1, hpc.sdm.thread1.c1 (pc)
	R10_d	hpc.sdm.thread1.c1
	R10_e	hpc.sdm.thread1.c1, sdc.proc1.thread1.c1 (pc)
	R10_f	sdc.proc1.thread1.c1
	R10_g	sdc.proc1.thread1.c1, sd.proc1.thread1.c1 (pc)
R11	R11	sd.proc1.thread1.c1, sd.temperature_threshold
R15	R15_a	cpc.wc.thread1.c1, hpc.ws.thread1.c1
R16	R16_a	cpc.wc.thread1.c1, hpc.ws.thread1.c1

Table B.4: Traces from requirements to architecture

Appendix C

SD system instance in Alloy

The architecture given in listing B.1 is transformed into an Alloy model. The part representing the SD system of the resulting Alloy model, together with formalized requirements R1 and R4, and a scenario to validate these requirements is given in the listing below.

Listing C.1: SD system instance represented in Alloy

```
open aadl_meta
open util/ordering[State] as so

--- State machine signatures/facts ---
sig State {
  dataPortValues: DataPort -> lone DataType,
  eventPortValues: EventPort -> lone Event,
  eventDataPortValues: EventDataPort -> lone DataType,
  parameterValues: Parameter -> lone DataType,

  dataInstanceValues: DataInstance -> DataType,

  toDispatch: set Subprogram,
}

one sig Event extends DataType { }

pred StateTransitions {
  -- proper transitions
  all s: State - so/last |
    transition[s, s.next]
}

pred transition[s, s': State] {
  s'.dataPortValues = dp2dp[s.dataPortValues] + p2dp[s.
      parameterValues]
  s'.eventPortValues = e2e[s.eventPortValues] + producedEps[s]
  s'.eventDataPortValues = edp2edp[s.eventDataPortValues] + p2edp[s
      .parameterValues] + producedEdps[s]
  s'.parameterValues = dp2p[s.dataPortValues] + edp2p[s.
      eventDataPortValues] + p2p[s.parameterValues] +
      producedParams[s]
  s'.dataInstanceValues = s.dataInstanceValues +
      producedDataInstanceValues[s] - removedDataInstanceValues[s]
  s'.toDispatch = scheduleDispatchByEventPort[s] +
      scheduleDispatchByEventDataPort[s] + { s.toDispatch.next }
}
```

```
fun dp2dp[r: DataPort -> DataType] : DataPort -> DataType {
  { dp: DataPort, dt: DataType |
    one (dp.~portConnections[] -> dt & r) } +
  { dp: DataPort, dt: DataType | one pc: PortConnection |
    dp = pc.dst and one (pc.src -> dt & r) }
}

fun e2e[r: EventPort -> Event] : EventPort -> Event {
  { ep: EventPort, e: Event |
    one (ep.~portConnections[] -> e & r) } +
  { ep: EventPort, e: Event | one pc: PortConnection |
    ep = pc.dst and one (pc.src -> e & r) }
}

fun dp2p[r: DataPort -> DataType] : Parameter -> DataType {
  { p: Parameter, dt: DataType |
    one (p.~portConnections[] -> dt & r) } +
  { p: Parameter, dt: DataType | one pc: PortConnection |
    p = pc.dst and one (pc.src -> dt & r) }
}

fun p2dp[r: Parameter -> DataType] : DataPort -> DataType {
  { dp: DataPort, dt: DataType |
    one (dp.~portConnections[] -> dt & r) } +
  { dp: DataPort, dt: DataType | one pc: PortConnection |
    dp = pc.dst and one (pc.src -> dt & r) }
}

fun ep2ep[r: EventDataPort -> DataType] : EventPort -> Event {
  { ep: EventPort, dt: Event |
    one (ep.~portConnections[] -> dt & r) } +
  { ep: EventPort, dt: DataType | one pc: PortConnection |
    ep = pc.dst and one (pc.src -> dt & r) }
}

fun edp2edp[r: EventDataPort -> DataType] : EventDataPort ->
    DataType {
  { edp: EventDataPort, dt: DataType |
    one (edp.~portConnections[] -> dt & r) } +
  { edp: EventDataPort, dt: DataType | one pc: PortConnection |
    edp = pc.dst and one (pc.src -> dt & r) }
}

fun edp2p[r: EventDataPort -> DataType] : Parameter -> DataType {
  { p: Parameter, dt: DataType |
    one (p.~portConnections[] -> dt & r) } +
  { p: Parameter, dt: DataType | one pc: PortConnection |
    p = pc.dst and one (pc.src -> dt & r) }
}

fun p2edp[r: Parameter -> DataType] : EventDataPort -> DataType {
  { edp: EventDataPort, dt: DataType |
    one (edp.~portConnections[] -> dt & r) } +
  { edp: EventDataPort, dt: DataType | one pc: PortConnection |
    edp = pc.dst and one (pc.src -> dt & r) }
}

fun p2p[r: Parameter -> DataType] : Parameter -> DataType {
  { p: Parameter, dt: DataType |
    one (p.~portConnections[] -> dt & r) } +
  { p: Parameter, dt: DataType | one pc: PortConnection |
```

```
      p = pc.dst and one (pc.src -> dt & r) }
  }

  fun scheduleDispatchByEventPort[s: State] : set Subprogram {
    { sp: Subprogram | one t: Thread, ep: EventPort |
      sp = t.subprogramCall and ep in t.features and ep.direction =
          PortDirection_In and one ep.(s.eventPortValues)
    }
  }

  fun scheduleDispatchByEventDataPort[s: State] : set Subprogram {
    { sp: Subprogram | one t: Thread, edp: EventDataPort |
      sp = t.subprogramCall and edp in t.features and edp.direction =
          PortDirection_In and one edp.(s.eventDataPortValues)
    }
  }

  fun getDataInstance[da: one DataAccess] : DataInstance {
    da.~^dataAccessConnection & DataInstance +
    { di: DataInstance | one dac: DataAccessConnection | dac.dst = da
        and dac.src = di }
  }


  --- Model ---
  --- Data types ---
  sig ZigBeeAddress_i extends DataType { }

  sig Alarm_i extends DataType {
    sensorA: one Device,
    zaA: lone ZigBeeAddress_i,
    devA: lone Dev_i,
    patA: lone Patient_i,
  } {
    one (zaA + devA + patA)
  }

  sig GenerateAlarm_i extends DataType { }

  sig AlarmsRequest_i extends DataType {
    sensorAR: one Device
  }

  sig Measurement_i extends DataType {
    sensorM: one Device,
    zaM: lone ZigBeeAddress_i,
    devM: lone Dev_i,
    patM: lone Patient_i,
  } {
    one (zaM + devM + patM)
  }

  sig Threshold_i extends DataType {
    sensorT: one Device,
    zaT: lone ZigBeeAddress_i,
    devT: lone Dev_i,
    patT: lone Patient_i,
  } {
    one (zaT + devT + patT)
  }

  sig MeasurementsRequest_i extends DataType {
```

```
  sensorMR: one Device
} {
}

-- Structure --

one sig RPM_RPM_i_Instance extends System {
} {
  subComponents = none
      + RPM_RPM_i_Instance_sd
      + RPM_RPM_i_Instance_sdc
      + RPM_RPM_i_Instance_hpc
      + RPM_RPM_i_Instance_cpc
  features = none
}

one sig RPM_RPM_i_Instance_sd extends System {
} {
  subComponents = none
      + RPM_RPM_i_Instance_sd_sensor1
      + RPM_RPM_i_Instance_sd_proc1
      + RPM_RPM_i_Instance_sd_temperature_threshold
  features = none
      + RPM_RPM_i_Instance_sd_measurement_out
      + RPM_RPM_i_Instance_sd_alarm_out
      + RPM_RPM_i_Instance_sd_threshold_in
}

one sig RPM_RPM_i_Instance_sd_measurement_out extends EventDataPort
    {
} {
  direction = PortDirection_Out
  all s: State | this.(s.eventDataPortValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_alarm_out extends EventDataPort {
} {
  direction = PortDirection_Out
  all s: State | this.(s.eventDataPortValues) in Alarm_i
}

one sig RPM_RPM_i_Instance_sd_threshold_in extends EventDataPort {
} {
  direction = PortDirection_In
  all s: State | this.(s.eventDataPortValues) in Threshold_i
}

one sig RPM_RPM_i_Instance_sd_sensor1 extends Device {
} {
  subComponents = none
  features = none
      + RPM_RPM_i_Instance_sd_sensor1_measurement
}

one sig RPM_RPM_i_Instance_sd_sensor1_measurement extends
    EventDataPort {
} {
  direction = PortDirection_Out
  all s: State | this.(s.eventDataPortValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_proc1 extends Process {
```

```
} {
  subComponents = none
      + RPM_RPM_i_Instance_sd_proc1_thread1
  features = none
      + RPM_RPM_i_Instance_sd_proc1_sensor1_in
      + RPM_RPM_i_Instance_sd_proc1_threshold_in
      + RPM_RPM_i_Instance_sd_proc1_measurement_out
      + RPM_RPM_i_Instance_sd_proc1_alarm_out
      + RPM_RPM_i_Instance_sd_proc1_temperature_threshold
}

one sig RPM_RPM_i_Instance_sd_proc1_sensor1_in extends
    EventDataPort {
} {
  direction = PortDirection_In
  all s: State | this.(s.eventDataPortValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_proc1_threshold_in extends
    EventDataPort {
} {
  direction = PortDirection_In
  all s: State | this.(s.eventDataPortValues) in Threshold_i
}

one sig RPM_RPM_i_Instance_sd_proc1_measurement_out extends
    EventDataPort {
} {
  direction = PortDirection_Out
  all s: State | this.(s.eventDataPortValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_proc1_alarm_out extends EventDataPort
    {
} {
  direction = PortDirection_Out
  all s: State | this.(s.eventDataPortValues) in Alarm_i
}

one sig RPM_RPM_i_Instance_sd_proc1_temperature_threshold extends
    DataAccess {
} {
  direction = AccessDirection_Required
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1 extends Thread {
} {
  subComponents = none
      + RPM_RPM_i_Instance_sd_proc1_thread1_c1
  features = none
      + RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in
      + RPM_RPM_i_Instance_sd_proc1_thread1_threshold_in
      + RPM_RPM_i_Instance_sd_proc1_thread1_measurement_out
      + RPM_RPM_i_Instance_sd_proc1_thread1_alarm_out
      + RPM_RPM_i_Instance_sd_proc1_thread1_temperature_threshold
  subprogramCall = none
      + RPM_RPM_i_Instance_sd_proc1_thread1_c1
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in extends
    EventDataPort {
} {
```

```
    direction = PortDirection_In
    all s: State | this.(s.eventDataPortValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_threshold_in extends
    EventDataPort {
} {
    direction = PortDirection_In
    all s: State | this.(s.eventDataPortValues) in Threshold_i
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_measurement_out extends
     EventDataPort {
} {
    direction = PortDirection_Out
    all s: State | this.(s.eventDataPortValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_alarm_out extends
    EventDataPort {
} {
    direction = PortDirection_Out
    all s: State | this.(s.eventDataPortValues) in Alarm_i
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_temperature_threshold
    extends DataAccess {
} {
    direction = AccessDirection_Required
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_c1 extends Subprogram {
} {
    subComponents = none
    features = none
        + RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in
        + RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in
        + RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
        + RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out
        +
            RPM_RPM_i_Instance_sd_proc1_thread1_c1_temperature_threshold

    next = none
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in extends
    Parameter {
} {
    direction = PortDirection_In
    all s: State | this.(s.parameterValues) in Measurement_i
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in extends
     Parameter {
} {
    direction = PortDirection_In
    all s: State | this.(s.parameterValues) in Threshold_i
}

one sig RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out
    extends EventDataPort {
} {
```

```
    direction = PortDirection_Out
    all s: State | this.(s.eventDataPortValues) in Measurement_i
  }

  one sig RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out extends
      EventDataPort {
  } {
    direction = PortDirection_Out
    all s: State | this.(s.eventDataPortValues) in Alarm_i
  }

  one sig
      RPM_RPM_i_Instance_sd_proc1_thread1_c1_temperature_threshold
      extends DataAccess {
  } {
    direction = AccessDirection_Required
  }

  one sig RPM_RPM_i_Instance_sd_temperature_threshold extends
      DataInstance {
  } {
    all s: State | this.(s.dataInstanceValues) in Threshold_i
  }

  --- Connections ---
  fun portConnections[] : Port -> Port {
    none -> none
      + (RPM_RPM_i_Instance_sd_sensor1_measurement ->
          RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in)
      + (RPM_RPM_i_Instance_sd_proc1_thread1_sensor1_in ->
          RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in)
      + (RPM_RPM_i_Instance_sd_proc1_thread1_threshold_in ->
          RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in)
      + (RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out ->
          RPM_RPM_i_Instance_sd_proc1_thread1_measurement_out)
      + (RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out ->
          RPM_RPM_i_Instance_sd_proc1_thread1_alarm_out)
  }

  fun dataAccessConnection[] : (DataInstance + DataAccess) -> (
      DataAccess) {
    none -> none
      + (RPM_RPM_i_Instance_sd_temperature_threshold ->
          RPM_RPM_i_Instance_sd_proc1_thread1_temperature_threshold)
      + (RPM_RPM_i_Instance_sd_proc1_thread1_temperature_threshold ->

          RPM_RPM_i_Instance_sd_proc1_thread1_c1_temperature_threshold
          )
  }

  --- Predicates for data instances ---
  pred RPM_RPM_i_Instance_sd_temperature_threshold::invariants[s:
      State] {
    all t: this.(s.dataInstanceValues) | t.sensorT =
        RPM_RPM_i_Instance_sd_sensor1
  }

  --- Predicates and functions for subprograms ---
  pred RPM_RPM_i_Instance_sd_proc1_thread1_c1::invariants[s: State] {

  all m: RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
      parameterValues) | one m.zaM
```

```
    all t: RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s.
        parameterValues) | one t.zaT

  }

370 fun RPM_RPM_i_Instance_sd_proc1_thread1_c1::post_eps[s: State] :
        EventPort -> Event {
      none -> none
  }

    fun RPM_RPM_i_Instance_sd_proc1_thread1_c1::post_edps[s: State] :
        EventDataPort -> DataType {
375   one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
          parameterValues) and no GenerateAlarm_i =>
        -- send temperature measurement
        { RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out ->
            RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
            parameterValues) }
      else one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
          parameterValues) and one GenerateAlarm_i =>
        -- send temperature alarm
380     { RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out -> { a:
            Alarm_i | a.sensorA =
            RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
            parameterValues).sensorM and one a.zaA } }
      else
        none -> none
    }

385 fun RPM_RPM_i_Instance_sd_proc1_thread1_c1::post_params[s: State] :
         Parameter -> DataType {
      none -> none
    }

    fun RPM_RPM_i_Instance_sd_proc1_thread1_c1::post_diProduced[s:
        State] : DataInstance -> DataType {
390   one RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s.
          parameterValues) =>
        -- store temperature threshold
        { getDataInstance[
            RPM_RPM_i_Instance_sd_proc1_thread1_c1_temperature_threshold
            ] -> RPM_RPM_i_Instance_sd_proc1_thread1_c1_threshold_in.(s
            .parameterValues) }
      else
        none -> none
395 }

    fun RPM_RPM_i_Instance_sd_proc1_thread1_c1::post_diRemoved[s: State
        ] : DataInstance -> DataType {
      none -> none
    }
400

    fun producedEps[s: State] : EventPort -> Event {
      none -> none
        + (RPM_RPM_i_Instance_sd_proc1_thread1_c1 in s.toDispatch =>
            RPM_RPM_i_Instance_sd_proc1_thread1_c1.post_eps[s] else
            none -> none)
405 }

    fun producedEdps[s: State] : EventDataPort -> DataType {
      none -> none
```

```
      + (RPM_RPM_i_Instance_sd_proc1_thread1_c1 in s.toDispatch =>
          RPM_RPM_i_Instance_sd_proc1_thread1_c1.post_edps[s] else
          none -> none)
}

fun producedParams[s: State] : Parameter -> DataType {
  none -> none
    + (RPM_RPM_i_Instance_sd_proc1_thread1_c1 in s.toDispatch =>
        RPM_RPM_i_Instance_sd_proc1_thread1_c1.post_params[s] else
        none -> none)
}

fun producedDataInstanceValues[s: State] : DataInstance -> DataType
    {
  none -> none
    + (RPM_RPM_i_Instance_sd_proc1_thread1_c1 in s.toDispatch =>
        RPM_RPM_i_Instance_sd_proc1_thread1_c1.post_diProduced[s]
        else none -> none)
}

fun removedDataInstanceValues[s: State] : DataInstance -> DataType
    {
  none -> none
    + (RPM_RPM_i_Instance_sd_proc1_thread1_c1 in s.toDispatch =>
        RPM_RPM_i_Instance_sd_proc1_thread1_c1.post_diRemoved[s]
        else none -> none)
}

--- Architecture invariants ---
pred SubprogramInvariants {
  all s: State |
    RPM_RPM_i_Instance_sd_proc1_thread1_c1.invariants[s]
}

pred DataInstanceInvariants {
  all s: State |
    RPM_RPM_i_Instance_sd_temperature_threshold.invariants[s]
}

pred ArchitectureInvariants {
  SubprogramInvariants
  DataInstanceInvariants
}

--- Commands and predicates ---
ShowArchitecture : run {
  no State.dataPortValues
  no State.eventPortValues
  no State.eventDataPortValues
  no State.parameterValues
  no State.dataInstanceValues
  no State.toDispatch
} for
  0 DataType,
  0 Component,
  0 Feature,
  0 Connection,
  1 State

--- Requirements ---
pred R1 {
  R1_a
```

```
  R1_b
}

pred R1_a {
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.^next.
        parameterValues)
}

pred R1_b {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_measurement_out.(s.^
        next.eventDataPortValues)
}

pred R4 {
  R4_a
  R4_b
}

pred R4_a {
  one s: State |
    one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
        eventDataPortValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.^next.
        parameterValues)
}

pred R4_b {
  one s: State |
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_sensor1_in.(s.
        parameterValues) and
    one RPM_RPM_i_Instance_sd_proc1_thread1_c1_alarm_out.(s.^next.
        eventDataPortValues)
}

--- Scenario 1: perform measurement ---
pred scenario1_initial[s: State] {
  no s.dataPortValues
  no s.eventPortValues
  no s.parameterValues
  no s.toDispatch

  one s.eventDataPortValues
  one RPM_RPM_i_Instance_sd_sensor1_measurement.(s.
      eventDataPortValues).zaM
  all m: Measurement_i | m.sensorM = RPM_RPM_i_Instance_sd_sensor1
  one m: Measurement_i | m.zaM = ZigBeeAddress_i
  one m: Measurement_i | m.devM = Dev_i
  one m: Measurement_i | m.patM = Patient_i

  no (DataInstance - RPM_RPM_i_Instance_sdc_zd_coupling -
      RPM_RPM_i_Instance_hpc_dp_coupling).(s.dataInstanceValues)
  one RPM_RPM_i_Instance_sdc_zd_coupling.(s.dataInstanceValues)
  one RPM_RPM_i_Instance_hpc_dp_coupling.(s.dataInstanceValues)
}

Check_scenario1 : check {
```

```
    (StateTransitions and scenario1_initial[so/first]) =>
    (ArchitectureInvariants and
    R1 and R2 and R3)
  } for
    0 DataType,
    0 Component,
    0 Feature,
    0 Connection,
    exactly 25 State,
    exactly 3 Measurement_i, exactly 0 Alarm_i, exactly 0 Threshold_i
        , exactly 0 GenerateAlarm_i, exactly 1 ZigBeeAddress_i,
        exactly 1 Dev_i, exactly 1 Patient_i, exactly 1
        ZigBeeDevice_i, exactly 1 DevicePatient_i, exactly 0
        MeasurementsRequest_i, exactly 0 AlarmsRequest_i
```