Dynamic Design and Evaluation of Software Architecture in Critical Systems Development

Klaus Marius Hansen **Lisa Wells**

Computer Science Department
University of Aarhus
DK-8200 Aarhus N
Email: {klaus.m.hansen,wells}@daimi.au.dk

Abstract

The software architecture of a computing system is an abstracted structure of the system in terms of elements and relationships. Such structures may be viewed from a number of viewpoints including static/module, dynamic/execution, and deployment viewpoints. Software architecture fundamentally influences systems from all of these viewpoints and designing and implementing proper software architectures is thus critical in many problem domain areas, including the ones that pertain to safety-critical systems.

With respect to safety-critical systems, a particular problem with focusing on software architecture is that there may be a large abstraction gap between an architectural description and an executing system or a formal model thereof thus potentially leading to inconsistencies between models and implementation. Addressing this problem, this paper presents tools and techniques for specifying executable software architectures and for validating these with formal models such as statecharts and Petri nets.

1 Introduction

Safety-critical systems are systems that can cause undesired loss or damage to life, property, or the environment, and safety-critical software is any software that can contribute to such loss or damage [20]. Since safety-critical systems have the potential to cause extensive damage, there are many standards and guidelines describing processes, techniques, and methods for developing such systems. For example, the IEC 61508 [14] is a standard for achieving functional safety of programmable electronic safety-related systems, and the Australian Defence standard 5679 [9] is concerned with the procurement of computer-based safety critical systems. Such standards contain recommendations regarding which techniques and measures should be used when developing software.

One of the techniques that these and other standards recommend or even require is the use of semi-formal or formal methods through various development phases for improving the quality of the safety-critical software. The use of formal methods and supporting tools "provide increased repeatability of analyis, increased soundness and extra assurance" [9]. The IEC 61508 standard recommends that (semi-)formal methods should be used at various development states, including software safety requirement specification, software architecture design, detailed software design and development, and software safety validation. The recommended methods include (semi-) formal models for representing both static and dynamic characteristics of the software. Here we are only interested in models for representing dynamic behaviour of systems. Such models can be used for either specifying desired behaviour of software and/or for validating and verifying that modelled software behaves has desired.

While the standards advocate the use of (semi-) formal models, they do not necessarily make any recommendations about how to ensure consistency between models of software behaviour and the corresponding executable software. It is clearly a good idea to model software behaviour, however, the usefulness of such models will be compromised if it is not possible to ensure some consistency between the model of the behaviour, and the behaviour of the executable software. This paper presents tools and techniques for validating the behaviour of executable software against models of the behaviour of the software, and thereby for reducing the gap between the software and the model.

1.1 Modelling Software Behaviour

Models of software behaviour can be used for many different purposes, such as for specifying software requirements, for designing software, and for analysing the behaviour of software. Since the majority of accidents in which software was involved can be traced to requirements flaws [20], it is of particular importance to develop complete and unambiguous requirement specifications for safety-critical software. Several standards recommend that requirements be specified as (semi-)formal models, and there is even rigorous language and tool support for checking completeness and consistency of software specifications [11]. The behaviour of software can be modelled both by static models, such as decision tables and Unified Modeling Language (UML) sequence diagrams and by dynamic models with executable behaviour, such as finite and timed automata, statecharts, and Petri nets. One of the advantages of using dynamic models is that it is possible to investigate and, in some cases, even verify the behaviour of the model in an appropriate tool.

Dynamic models that represent states of a system and transitions from one state to another can represent either discrete or continuous changes between states. When modelling the behaviour of (safety-

The research presented in this paper has been supported by ISIS Katrinebjerg Competency Centre, http://www.isis.alexandra.dk.

critical) software, it is rarely interesting to have an accurate model of continuous state changes, and in most cases it is sufficient to consider a set of discrete state changes. For example, when modelling software that controls the speed of a conveyor belt, it would not be necessary to model all possible speeds of the conveyor belt, but it would be sufficient to consider a number of different discrete classes of speeds, such as stopped, within range, and above acceptable range.

In this paper we consider only *discrete-state models* which are state-based models with discrete transitions between states. Transitions between states will also be called events. A more formal definition of the kind of models that we are interested in will be provided in Sect. 3.2. As always, when using models it is important to find an appropriate level of abstraction for the models. If the models are too detailed, then it may be too time-consuming to develop them, and it may be difficult, if not impossible, to do reasonable analysis of the behaviour of the model. Discrete-state models are well-suited for specifying fairly high-level requirements, and for analysing the behaviour of relatively small systems.

A variety of tools provide support for creating and analysing different kinds of discrete-state models of software behaviour. For example, SPIN [13] and UPPAAL [19] support model checking of finite and timed automata respectively, visualSTATE [25] and STATEMATE [10] support analysis of statecharts, and CPN Tools [7] supports analysis of a kind of high-level Petri nets which will be introduced in Sect. 4. With some of these tools, it is possible to generate executable code from the models, in which case, it is possible to ensure that there is consistency between the model and the code (assuming that the code is regenerated or updated if the model is modified). However, if code is not or cannot be generated from models, then there is likely to be a large gap between the models of software behaviour and the executable code that is modelled. And in particular, even though code may be generated, it is not certain that it corresponds to a required or desired software architecture. This lets us to consider the concept of *software architecture*.

1.2 Software Architecture

Software architecture is concerned with abstracted structures of software systems. A generally accepted definition of the term 'software architecture' is

Definition 1 (Software Architecture) *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [4]*

The definition implies a number of characteristics of software architecture. First, a system has many structures/views of interest (e.g., module structure, dynamic structure at runtime in terms of processes and communication, and deployment structure in terms of processors and components deployed) [18]. Secondly, software architecture is abstract in the sense that it is only concerned with externally visible properties of elements and relations and thus not concerned with the inner structure of components. Thirdly, all systems have a software architecture whether intended or not.

All of these characteristics are relevant in relation to software safety. Software architecture highly influences various system quality attributes such as performance, modifiability, and testability because these are influenced by structures in various views [4]. A consequence of the second characteristic is that software architecture descriptions may be more manageable than the actual system (or a less abstract description thereof) making the descriptions amenable to, e.g., analyses and communication. And a consequence of the third characteristics (in combination with the above) is that software architecture is well worth to be concerned with in safety-critical system development.

A large number of techniques for software architecture requirements analysis such as Quality Attribute Workshops [1] and Global Analysis [12]; techniques for software architecture design such as Attribute-Driven Design [5] and architecture pattern-base design[6]; and techniques for software architecture evaluation such as the Architecture Tradeoff Analysis Method and Architecture Level Prediction of Software Maintenance [8] have been developed and tested. One characteristic of these are that they are almost all specification-based in that they use and produce *descriptions* of software architectures rather than software architectures of actual systems. Some problematic consequences of basing software architecture work solely on such descriptions can be that the architecture-as-built differs from the architecture-as-designed, that quality attributes are not properly addressed, or that software architects tend to design conservatively even if the conservative choice may not be appropriate.

As a way to mitigate some of these problems, and as a supplement to existing well-documented techniques related to software architecture, we have previously introduced the concept of *architectural prototyping* [2, 3]:

Definition 2 (Architectural Prototype) *An architectural prototype consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development. Architectural prototyping is the process of designing, building, and evaluating architectural prototypes [2]*

Architectural prototypes are characterized by having no functionality per se and thus often being cheap to implement. Often architectural prototypes experiment with and evaluate infrastructure and middleware, e.g., to decide whether a push or a pull message passing architecture is most suitable for an embedded control system [2]. Section 2.4 presents an architectural prototype constructed in a safety-critical system development context.

In this paper we claim that architectural prototypes are useful in safety-critical software development in that the technique promises a cost-effective way to implement various architectural alternatives. Further, we provide a way of validating such executable software architectures. In doing this, we are in line with the views of [21]: what matters more than how or by which principles it was developed is that the designed software architecture is safe.

1.3 Software Architectures and Discrete-State Models

Given the above discussion, there are a number of issues in combining the use of software architecture and discrete-state model in the development of (critical) software systems.

Most fundamental is that software architecture is concerned with structures (of systems) whereas discrete-state models are concerned with behaviour. Further, discrete-state models typically provide one, behavioural view of a system whereas software architecture provides several as discussed in Section 1.2.

An example of why this may be an issue is that a deployment decision (such as about the type of network used in a concrete distributed system) may impact behavioural characteristics such as performance.

This also means that discrete-state models are mostly concerned with runtime system quality attributes (e.g., logical correctness, reliability, performance, or scalability) whereas software architecture is also concerned with development time system qualities (e.g., modifiability, testability, or interoperability).

Finally, discrete-state models and software architectures may also often be orthogonal abstractions of a system. In our case study, presented in Section 2, discrete-state models were used to model requirements of the system where a software architecture is used to represent the system per se.

These problems make, e.g., traceability between software architectures and discrete-state models and reasoning about whether software architectures fulfill requirements modeled by discrete-state models hard. Section 3 introduces our approach to handling parts of these problems.

1.4 Contributions

The main contribution of this paper is the introduction of the Heimdall[1] tool. The tool enables the validation of sequences of program execution events against a discrete-state model. We present a real-life case study in which Heimdall is applied by using aspect-oriented instrumentation to an architectural prototype of a frequency converter for safety-critical applications for which program execution events are then mapped to a formal model of requirements described by a Coloured Petri Net [15].

The rest of the paper is structured as follows. Section 2 describes the case study which emphasised the need for tools like the Heimdall tool. Section 3 describes the architecture and functionality of the Heimdall tool, and it also illustrates the current implementation of the tool. Section 5 discusses ideas for future work and concludes the paper.

2 Frequency Converter Case

Several of the problems and issues that were discussed above were encountered in a collaborative research project between Danfoss Drives[2], Systematic Software Engineering[3], and the Computer Science Department, University of Aarhus[4]. Danfoss Drives produces frequency converters which are used to control the speed of motors, e.g. for elevators, cranes, and conveyor belts. A new generation of frequency converters is being developed in accordance with IEC 61508. One part of the project investigated different (semi-)formal methods for specifying software safety requirements. Another part of the project focused on the design of the software architecture for the frequency converter. In this project we experienced the problem of a large gap between the models specifying the software safety requirements and the executable prototype of the software architecture. This section will briefly present the case study which is described in more detail in [26].

[1]Heimdall is the watchman of the Gods in Norse mythology. Using his excellent hearing and vision he watches the rainbow, Bifrost, that leads to Asgard, the home of the gods, sounding his alarming horn when danger approaches

[2]http://drives.danfoss.com

[3]http://www.systematic.dk

[4]http://www.daimi.au.dk

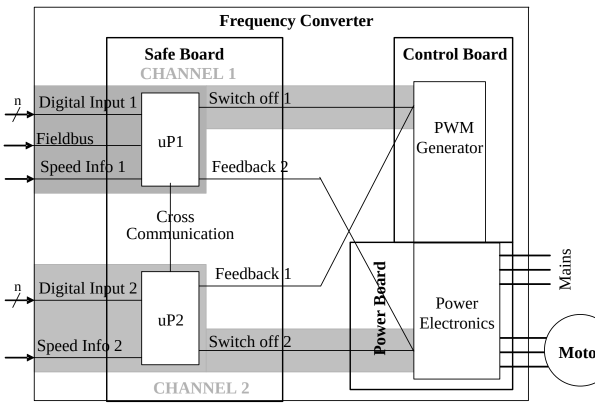

Figure 1: Hardware structure of a frequency converter with safety functions.

2.1 Hardware and Software

In the new generation of frequency converters, safety-critical software runs on two microprocessors. The hardware structure of the frequency converter is shown in Figure 1. The two blocks *PWM Generator* and *Power Electronics* control the speed of the attached motor, and they make up the normal, "non safety-related" part of a frequency converter.

The safety functionality is achieved by an additional subsystem on the *Safe Board* composed of *Channels 1* and *2*, each containing a microprocessor (*uP*), a *Switch-off* path, and three *Digital Inputs*. The two microprocessors can, independently from each other, activate its own switch-off path to stop the motor. The two *Channels* cross-monitor each other through *Feedbacks 1* and *2* and through the *Cross Communication* connection.

A number of so-called *designated safety functions* (DSF, or safety function) are implemented in software that runs on the two microprocessors on the *Safe Board*. The simplest safety function is a so-called 'uncontrolled stop' which immediately stops power supply to the motor. Another safety function is a 'controlled stop' or 'safe delay', where the stopping of the power supply to the motor is delayed, allowing the non-safety-related part of the frequency converter to ramp the motor down in a controlled way. A more complex example is the 'safe speed' where an uncontrolled stop is made if the motor speed exceeds a set limit. A frequency converter is configurable, and users can determine which safety function is associated with each of the *n=3* digital inputs. A specific safety function is activated upon reception of signals at the appropriate digital input at each of the *Channels*.

All diagnostic functionality with respect to cross monitoring and self monitoring of the *Channels* is implemented in software. On detection of a dangerous failure, an appropriate fault reaction is initiated, and the motor is stopped.

2.2 Specifying Safety Requirements

The software that runs on the two microprocessors on the *Safe Board* is safety-critical since it can contribute to loss or damage to the environment of the frequency converter through its effect on the speed and control of the attached motor. System-level safety requirements were already defined at the outset of the project. These requirements addressed issues such as, when output to the motor should be enabled, what should happen when an error occurs (either in hardware or software), how requests for safety functions

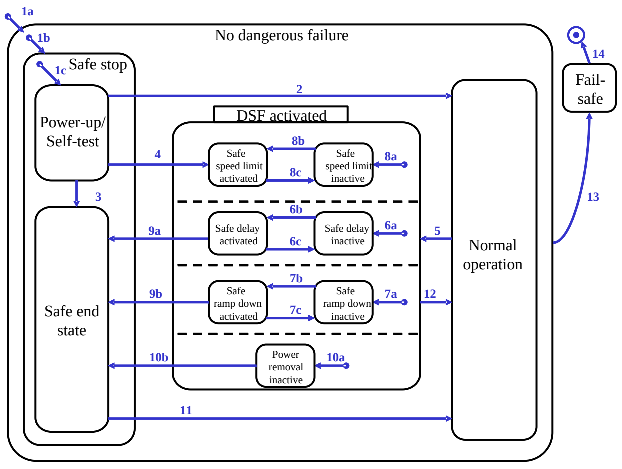

Figure 2: Informal statechart specification of system-level safety requirements.

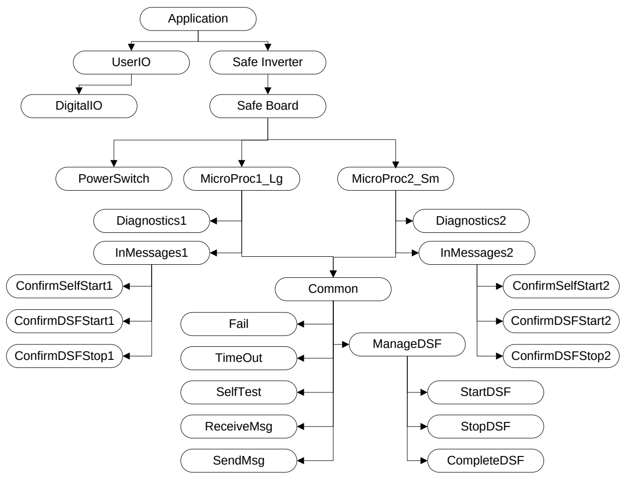

Figure 3: Module hierarchy of the CPN model.

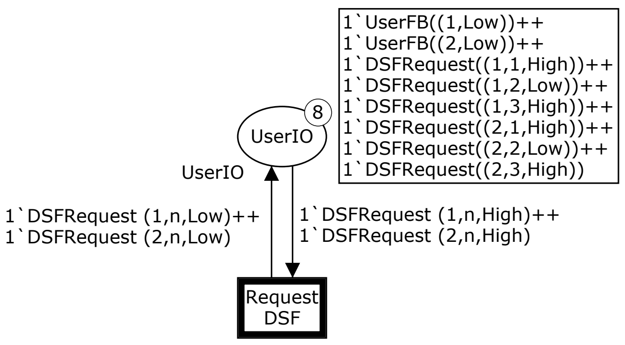

Figure 4: DigitalIO module from the CPN model.

should be made and handled, and what should happen after a safety function completes.

As mentioned previously, one of the recommendations of standard IEC 61508 is that semi-formal methods should be used to specify safety requirements. In order to comply with this recommendation, Danfoss developed an informal statechart model (shown in Figure 2) that was included in the initial product proposal that was approved by the certification authorities. The model is informal in that it was drawn in a generic drawing tool, and the states, transitions, and event triggers are described separately in simple, natural-language texts. It is not important to understand the details of the behaviour specified by the statechart, but it will be briefly explained.

The statechart specifies that the frequency converter must always be in one of three top-level states, namely No dangerous failure, Fail-safe or the Final state (denoted by a dot in a circle in the upper right-hand corner of the figure). If any kind of error is detected, then the frequency converter must enter Fail-safe state, and the power supply to the motor must be stopped. The only way to leave Fail-safe state is to turn the frequency converter off (Transition 14), and thereby enter Final state. If no errors are detected, then the frequency converter must be in No dangerous failure state, and more specifically, in one of its three composed states: Normal operation, DSF activated or Safe stop. In Safe stop state, output to the motor is always disabled.

One of the goals of the project was to specify software safety requirements based on the informal statechart of the system safety requirements. The software safety requirements were a refinement of the system safety requirements. Again, the IEC 61508 standard highly recommended that semi-formal methods should be used to define software safety requirements.

2.3 CPN Model of Requirements

A very detailed model of software safety requirements was developed in the formal modelling language Coloured Petri Nets (CPN or CP-nets) [15, 17]. This section will provide a brief overview of the CPN model of the frequency converter, and the formal definition of CPN will be introduced in Sect. 4. All of the requirements that were specified in the statechart model from Figure 2 are included in the CPN model. Those requirements have been specified more formally, and the specification is much more detailed. In addition, the CPN model specifies requirements that are not addressed in the statechart model, such as diagnositics and synchronisation of the state of the software on the two microprocessors.

Figure 3 provides an overview of the CPN model which was created in CPN Tools. Each node in Figure 3 represents a module in the model, and an arc from one node to another indicates that the source node contains an abstract representation of some behaviour that is specified in more detail in the module of the destination node.

The Application module (at the top of Figure 3) is the most abstract representation of the frequency converter and its environment. This module has two submodule, namely User IO and Safe Inverter, modelling the means for user input/output, i.e. the digital inputs (module Digital IO) shown in Figure 1, and the frequency converter itself, respectively. The software for the two microprocessors is modelled by the modules MicroProc1_Lg and MicroProc2_Sm. Both of these modules share some common functionality as specified by the module Common and its submodules. The two microprocessors send different kinds of messages and have different diagnostic algorithms, which is why there are separate modules for modelling these characteristics.

Figure 4 shows a simplified version of the Digital IO module of the model. Requests for activating safety functions are modelled in this module. The behaviour of the module will be discussed in detail in Sect. 4.1.

Simulations of the model were run for three main purposes: for debugging the model, for analysing the behaviour of the model, and for discussing the software requirement specification with the project team. Even though an exhaustive investigation of the behaviour of the model was not performed, a number of important problems were identified through the

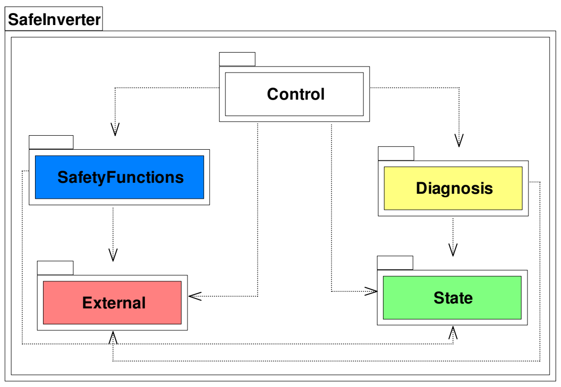

Figure 5: Package diagram for the software architecture.

construction and simulation of the model. Examples of these problems were: a simple diagnostic algorithm could lead to deadlock, and outdated messages in message queues could lead to hazards, such as enabling power supply to the motor after an error had been detected in one of the microprocessors.

2.4 Executable Architecture Prototype

Another goal of this project was to investigate and develop techniques for ensuring that safety-critical software fulfills the corresponding software safety requirements. In other words, we were interested in closing the gap between a semi-formal requirement specification and a software implementation. We focused on techniques for specifying and validating a software architecture (rather than the final software) for the frequency converter. A software architecture was developed and documented using a technique similar to Kruchten's 4+1 technique [18] in which an architecture is described in different views.

The architecture was defined largely by UML diagrams, including class, package, deployment, and sequence diagrams. Figure 5 shows the package diagram for the software architecture. The Control package contains classses for ensuring strict scheduling requirements for the frequency converter, including regular checks for requests on digital inputs, diagnostics, and checking microprocessor state consistency. The Safety Functions package contains classes for the safety functions. The classes in the Diagnosis package initiate, coordinate, and perform diagnostics. The State package is used by software on the two microprocessors to regularly communicate and compare their internal states. The External package contains classes for reading and setting digital input/output values.

An executable architecture prototype was implemented as skeleton classes in Java. Figure 6 shows an abstract class from the Safety Functions package for the architecture prototype. Classes for each of the different safety functions are defined as specialisations of this abstract class. A number of important use scenarios were described as sequence diagrams, such as initialisation during power-up and requesting safety functions. These use scenarios were implemented as simple Java programs that exercised the architecture prototype by emulating external events of the frequency converter, e.g. pressing the power button or requesting a safety function by activating a digital input, by calling appropriate methods in the executable architecture prototype. Given this architecture prototype, Danfoss was interested in developing techniques for ensuring that the architecture fulfilled the software safety requirements, including those specified by the CPN model. An early prototype of the Heimdall tool

```
public abstract class SafetyFunction {
        State state;
        boolean isrequested = false;

        public SafetyFunction (State state) {
                this.state = state;
                selfCheck();
        }
        public abstract void activate();
        public abstract void selfCheck();
        public abstract boolean isRequested();
}
```

Figure 6: Java skeleton class from executable software architecture.

was developed during this project. We introduce the Heimdall tool next.

3 The Heimdall Tool

Informally, the intented function of the Heimdall tool is to map a sequence of well-defined program execution events to a sequence of well-defined model events of a discrete-state model (Figure 7).

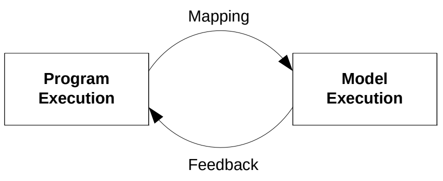

Figure 7: Conceptual overview of the Heimdall tool

The mapping is introduced more formally in Section 3.2 and concrete examples of the specification of mappings is given in Section 4.3. The mapping should be done in such a way that for an implementation that violates the model, the execution should at some point lead to a corresponding sequence of model events that are illegal with respect to requirements and feedback should be given. Conversely, the execution of a correct implementation should not lead to violations in the corresponding sequence of model events.

In the following sections we first present an overview of the architecture of Heimdall followed by a more precise introduction of the mapping of execution events to model events. Next, we present our concrete instantiation of the architecture to be used with Coloured Petri Nets and show how the Heimdall tool has been applied to architectural prototypes in the frequency converter case.

3.1 Heimdall Software Architecture

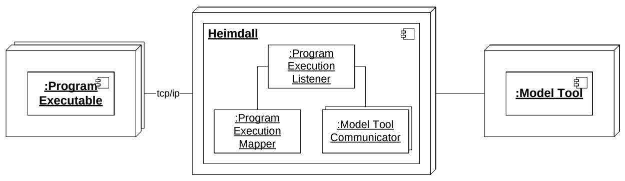

Figure 8: UML deployment overview of software architecture of the Heimdall tool

An overview of the architecture of the Heimdall tool is shown in Figure 8. A *Model Tool* is a tool which can execute and analyse a discrete-state model of the behaviour of an executable program. A set of

Program Executables are instrumented to send execution events to the *Program Execution* Listener of the Heimdall tool. Instrumentation may be done, e.g., by using a debugger, by instrumenting source code with tracing functions, or by using an aspect-oriented approach (such as AspectC++ [24] for C/C++, AspectJ [16] for Java, or (eventually) AspectAda [23] for Ada95). The instrumentation sends information about relevant program execution events to the Heimdall tool instance using a TCP/IP-based protocol.

The Heimdall tool instantiates a *Program Execution Mapper* based on a description of the mapping of execution events to model events. This mapping is described in an XML format of which Figure 10 gives an example. Whenever the Program Execution Listener receives an event from a program execution, it consults the Program Execution Mapper. The Program Execution Mapper maintains traces of program execution events and returns corresponding model events as appropriate (cf. Definition 8 in Section 3.2). Given a match, the *Model Tool Communicator* is used to communicate with a Model Tool in order to examine whether the mapped model events are legal in the model that the program execution is validated against.

The Model Tool Communicators are tool-specific. The requirements on Model Tools that are to be used with Heimdall is provisions for tool integration, e.g., through plug-in capabilities, trace replay, or using a tool-specific protocol. Our current status is that CPN Tools can interact with the Heimdall tool (see Section 4). Also traces of execution events need not be replayed immediately, but may be saved and (re)executed later, meaning that different mapping could be tested against the same program execution trace. Correspondingly, mapped model elements could also be saved for later transfer.

3.2 Mapping from Execution Events to Model Events

The intent in Heimdall is to validate that a sequence of execution events corresponds to a valid sequence of model events. This is achieved, in part, by mapping sequences of *join points* [16] in a *program execution* to a sequence of valid events in a discrete-state model. A join point is a well-defined point in the execution of a program. We are primarily interested in join points corresponding to method/procedure calls, setting field/data values, and getting field/data values.

In Sect. 1.1 we informally described discrete-state models, now we will provide a formal definition of the models in which we are interested. A discrete-state model is a model that is equivalent to a *labelled transition system*:

Definition 3 (Labelled Transition System) *A* labelled transition system *is a tuple LTS=(S,i,Λ,T) where S is a set of states, i$\in$S is the initial state, Λ is a set of labels, and T $\subseteq$ (S$\times\Lambda\times$S) is the set of labelled transitions.*

Note that both the set of states and the set of labels may be uncountable. An LTS is said to be finite if its sets of states and labels are finite. For a labelled transition system with states s_1, s_2, and label l where $(s_1, l, s_2) \in T$, we will write $s_1 \xrightarrow{l} s_2$ and further, for a set of labels, Λ, Λ^* denotes the set of all sequences of labels from Λ. An element of Λ^* is legal or valid if it is a *trace*:

Definition 4 (Trace) *Given a labelled transition system LTS=(S,i,Λ,T), a sequence of labels $l_1 l_2 \ldots l_n \in \Lambda^*$ is a* trace *of LTS if $\exists s_1, s_2, \ldots, s_n \in S$ so that $i \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots \xrightarrow{l_n} s_n$.*

We also consider the set of possible *program executions* of a program as a labelled transition system where the states are program states of interest (which may be discerned by heap contents, stack contents etc.) during execution and where the labels are join point executions and related state, i.e., events of interest in the program execution:

Definition 5 (Program Execution System) *A* program execution system *is a labelled transition system $P = (S_P, i_P, \Lambda_P, T_P)$ which is a representation of all of the possible executions of a program in which execution states are abstracted into S_P and where call, set, and get join point executions are abstracted into Λ_P.*

Note that the defintion of a program execution system is somewhat imprecise in that the definition of the set of states and set of labels is left to the discretion of those who are interested in validating an executable program against a discrete-state model of the behaviour of the software. Thus a reasonable set of "interesting" states and "interesting" join point labels that will be used during the validation process will have to be defined. Section 4 will discuss the states and join point labels that were used when validating the executable architecture prototype of the frequency converter against the CPN model of the software safety requirements.

Recall that the purpose of the Heimdall tool is to provide support for validating an executable program against a model of the behaviour of the program. We have just defined discrete-state models and program execution systems in terms of labelled transition systems. LTSs are quite general in that they allow for non-deterministic behaviour and infinitely many states and labels. All that we need now is a way to show that two labelled transition systems are (more or less) equivalent. A large body of research is concerned with this issue, and bisimulation and weak bisimulation can be used to show equivalences between two LTS. However, the problem with these techniques is that they are difficult, if not impossible, to use for large LTSs with (infinitely) many states and labels.

Many systems, and in particular safety-critical systems, can be represented by finite labelled transition systems, which are somewhat more practical to deal with. Furthermore, the behaviour of safety-critical systems is generally deterministic, which means that the set of transitions for the LTS for the system would be deterministic, in other words, if $s_1 \xrightarrow{l} s_2$ and $s_1 \xrightarrow{l} s_3$ then $s_2 = s_3$. Finite, deterministic labelled transition systems are somewhat easier to deal with, however it is rarely possible to construct and analyse an LTS for complicated, industrial-sized systems. So it is still necessary to develop techniques that can be use to check and validate the behaviour of software for non-trivial systems.

Given the definitions above, our primary interest is now to define mappings between *program executions* and *model executions* that will allow us to validate the behaviour of an executing program against the behaviour of a discrete-state model. Program and model executions are defined as traces of an LTS:

Definition 6 (Program and Model Executions) *Given a program execution system, $P = (S_P, i_P, \Lambda_P, T_P)$, a* program execution *for this system is a trace $p \in \Lambda_P^*$.*

Given a discrete-state model, $M = (S_M, i_M, \Lambda_M, T_M)$, a model execution *is a trace $m \in \Lambda_M^*$.*

In this context, a program execution is considered to be a sequence of execution join points that form a trace. Similarly, a model execution is a sequence of legal model events. For such program executions, we are interested in mappings of these to corresponding events in the discrete-state model that is an abstract representation of the behaviour of the program execution system of the program execution. More precisely, we define an *execution mapping* as:

Definition 7 (Execution Mapping) *Given a program execution system $P = (S_P, i_P, \Lambda_P, T_P)$ and a discrete-state model expressed as an LTS $M = (S_M, i_M, \Lambda_M, T_M)$, an* execution mapping *for P and M is a set $E \subseteq (\Lambda_P^* \times \Lambda_M^*)$.*

In other words, an element e in an execution mapping specifies how sequences of program join points map to sequences of model events. An example of a mapping element would be $(l_{p_1} l_{p_2} l_{p_3}, l_{m_1} l_{m_2})$ meaning that $l_{p_1} l_{p_2} l_{p_3} \in \Lambda_P^*$ maps to $l_{m_1} l_{m_2} \in \Lambda_M^*$. The goals of the validation process will help to determine how detailed the execution mapping should be.

Based on program execution systems and mapping definitions we are now able to define when a program execution may be considered correct:

Definition 8 (Correctness) *A program execution $p = l_{p_1} \dots l_{p_k}$ of a program execution system $P = (S_P, i_P, \Lambda_P, T_P)$ is* correct *with respect to a discrete-state model $M = (S_M, i_M, \Lambda_M, T_M)$ and an execution mapping E if*

1. *$\exists e = (l_{p_1} \dots l_{p_u}, l_{m_1} \dots l_{m_v}) \in E, s_{p_1}, \dots, s_{p_u} \in S_P, s_{m_1}, \dots, s_{m_v} \in S_M : i_P \xrightarrow{l_{p_1}} s_{p_1} \xrightarrow{l_{p_2}} \dots \xrightarrow{l_{p_u}} s_{p_u} \wedge i_M \xrightarrow{l_{m_1}} s_{m_1} \xrightarrow{l_{m_2}} \dots \xrightarrow{l_{m_v}} s_{m_v}$ where $l_{p_1} \dots l_{p_u}$ is a prefix of p and*
2. *$p' = l_{p_u+1} \dots l_{p_k}$ of $P' = (S_P, s_{p_u}, \Lambda_P, T_P)$ is* correct *with respect to $M' = (S_M, s_{m_v}, \Lambda_M, T_M)$ and E where p' is the remainder of p after the prefix $l_{p_1} \dots l_{p_u}$ has been removed.*

Note that P' and M' are essentially the same as P and M — the only difference is the initial states.

In some cases a discrete-state model may contain events that do not correspond to any "interesting" execution events. For example, the model may specify behaviour that is more detailed than what is currently implemented in the software, or there may be model events that are used to initialise parts of the model at the beginning of an execution. Since the definition of an execution mapping allows an empty sequence of join points to be mapped to a non-empty sequence of model events, it is still possible for unmapped model events to occur when checking correctness of a program execution.

Even though a set of program executions are correct with respect to a mapping, they are not necessarily "good" in the sense that they cover all states of the discrete-state model. Ideally, we also want *completeness* for this set of program executions:

Definition 9 (Completeness) *A set of correct program executions are* complete *with respect to an execution mapping and a discrete-state model if the set of all states of the model execution mapped to is the complete set of states of the discrete-state model.*

Ideally we would like to do an exhaustive verification of program executions against discrete-system models, but this is rarely possible in practice which is why there is a need for tools like Heimdall. In a safety-critical system setting, we may aim for establishing that program executions should be correct *and* complete with respect to a set of critical states/states of interest in the labelled transition system.

The next section will give an example of how this is realised in practice with the specific program executions being executions of Java architectural prototypes and where the concrete discrete-state model is a Coloured Petri Net.

4 The Heimdall Tool for Coloured Petri Nets

This section discusses the current implementation of the Heimdall tool that has been used to validate the executable architecture prototype for the frequency converter against the CPN model of the software safety requirements.

4.1 CPN and CPN Tools

Coloured Petri Nets is a formal, graphical modelling language with well-defined syntax and semantics. We will provide a very brief and somewhat informal introduction to CP-nets which is taken from [15]. An example following the formal definition will be used to illustrate several concepts from the defnition. The structure of a non-hierarchical CP-net is formally defined as a tuple:

Definition 10 (Coloured Petri Net) *A* non-hierarchical CP-net *is a tuple CPN = (Σ,P,T,A,N,C,G,E,I), where Σ is a finite set of non-empty types called* colour sets*; P,T, and A are non-empty finite, disjoint sets of* places*,* transitions*, and* arcs*, respectively; N is a* node function *defined from A into $(P \times T) \cup (T \times P)$; C is a* colour function *defined from P into Σ; G is a* guard function *defined from T into boolean expressions; E is an* arc expression function *defined from A into expressions such that the arc expression for an arc evaluates to a multi-set of values from C(p) where p is the place that the arc is connected to; and I is an* initialization function *defined from P into expressions that do not contain variables such that the initialization expression for place p evaluates to a multi-set of values from C(p).*

Note that arc and guard expressions may contain variables. A similar definition exists for hierarchical CP-nets, in which modules are connected via well-defined interfaces.

Recall that Figure 4 shows a simplified version of the `DigitalIO` module for the CPN model described in Section 2.3. The ellipse `UserIO` is a place representing digital inputs and outputs for the two microprocessors. The `UserIO` place acts an interface for this particular module. The colour set for the place is determined by the inscription `UserIO` to the lower left of the place. The states of a CP-net are represented by a number of *tokens* distributed on the places in the model. A token on a place carries a data value, and the type of the data value must correspond the the colour set of the place. Figure 4 shows a state in which there are eight tokens on place `UserIO`, as indicated by the small circle with the number next to the place, and the box next to the small circle shows the values of the eight tokens. Two tokens indicate that the voltage for the digital outputs (which are not shown in Figure 1) for the user feedback (`UserFB`) at microprocessors 1 and 2 are both `Low`. The other six tokens represent the three digital inputs that are used to request safety functions for the two microprocessors. The format for such a data value is `DSFRequest((x,y,voltage))` where `x` indicates the number of the microprocessor (1 or 2), `y` indicates the number of the digital input (1, 2, or 3), and `voltage` indicates the voltage of the digital input where there are three possible values (`High`, `Low`, and `Error`).

The formal semantics of CP-nets determine which events can occur in a given state, and how the state will change when a particular event occurs. Events in a system are modelled by transitions. The rectangle Request DSF is a transition that represents the request for the activation of a safety function. The arc expressions on the arcs between UserIO and Request DSF determine how the state of the model will change when the Request DSF transition occurs. The arc expression on the arc from UserIO to Request DSF contains only one variable which is `n`, and it determines that two tokens will be removed from the place when the transition occurs. A transition together with a binding of all of its variables is known as a *binding element*. This transition can only occur if the voltage of digital input `n` at microprocessors 1 and 2 is `High`. When the transition occurs, two tokens will be added to the place, representing the fact that the voltage of the digital inputs is changed to `Low` which indicates that a request is being made for the safety function that corresponds to input `n`. In Figure 4 the safety function corresponding to digital inputs 2 has been requested (and possibly activated, but this cannot be seen in this module), but it is currently possible to request the safety functions that correspond to inputs 1 and 3.

Coloured Petri Nets have been used to specify software safety requirements, but we have said that the models that are used with Heimdall must be discrete-state models that can be expressed as an LTS. This is not a problem, because it is possible to define a labelled transition system that is equivalent to a CP-net. Given a CP-net CPN, let $LTS_C = (S_C, i_C, \Lambda_C, T_C)$ where S_C is the set of states of CPN that are reachable by sequences of transition occurrences from the initial state of CPN, i_C is the initial state of CPN, Λ_C is the set of binding elements of CPN, and T_C is the set $\{(s, be, s')\}$ where s is a reachable state of CPN, be is a binding element that is enabled in s, and s' is the state that is reached when be occurs in s.

CPN Tools is a tool supporting the construction and analysis of CP-nets. There is support for running two kinds of simulations: interactive and automatic. In interactive simulations, it is possible for the user to select which transitions should occur. The choice of how transition variables should be bound can either be left to the simulator or the user can manually pick among the legal bindings in a given state. In automatic simulations the simulator randomly picks among the events that are enabled in a given state. In either case, the simulator will update the state of the model after each event occurs.

CPN Tools can execute and analyse models that are equivalent to labelled transition systems, and it therefore fulfills some the requirements that must be met by the modelling tools that should interact with Heimdall. In order for the Heimdall tool to work with CPN Tools, it must be possible to run and control simulations without (or with very minimal) manual interaction between a user and the GUI of CPN Tools. The simulator for CPN Tools is implemented in Standard ML [22] which means that arbitrary SML functions can be written to control simulations via the predefined primitives in the simulator. The simulator has primitives for running automatic simulations and for selecting which transitions should occur in a simulation, but it lacks a primitive for selecting a transition together with particular bindings of some or all of the variables of the transition. The existing primitives for selecting a particular binding required manual interaction with the GUI by a user. The simulator has been modified, and a new primitive makes it possible to specify that a transition with a particular binding of some of its variables should occur (assuming that the corresponding event can occur in the current state of the model). Support for communicating with the Heimdall tool and for running simulations based on the information received from Heimdall has been implemented in SML, and it will be discussed in Section 4.3.

```
public aspect SafeInverterTracer extends HeimdallTracer {
       pointcut calls() :
               call(* safeinverter..*(..)) &&
               !call(* safeinverter.Factory.*(..)) &&
               !call(* safeinverter..main(..));
       pointcut initializers() :
               initialization(safeinverter..*.new(..));
}
```

Figure 9: Aspect for extracting join points from executable software architecture.

4.2 Aspects for the Architecture Prototype

As mentioned in Section 2.4 the executable architecture prototype for the frequency converter was implemented as skeleton classes in Java. The classes reflect the design of the software architecture, and they are very simple. Each class contains a number of important methods and, in some cases, some important state variables. The methods are also very simple — they take few, if any, arguments, and the only actions that they perform is that they may update local state variables or call other methods in the architecture prototype.

In order to validate the architecture prototype of the frequency converter, information about join points must be extracted from the prototype during execution, as described in Section 3.1. AspectJ is used for this purpose. We provide an abstract aspect, HeimdallTracer, with functionality for communicating with the Program Execution Listener in the Heimdall tool. The aspect allows for tracing of method calls and object constructors. Object constructors are traced in order to provide a object id to correlated with method calls which is necessary in order to distinguish between instances of classes. The default object id is simply derived from the sequence in which objects of interest are constructed, a default approach that may be reasonable in cases where object creation order is deterministic.

The aspect named SafeInverterTracer (the new frequency converters are also known as safe inverters), shown in Figure 9, determines which method call join points will be sent to the Heimdall tool. Further, it defines which objects should have their ids tracked. In this case the join points that are to be validated are virtually all method calls in the architecture prototype which is defined in the `safeinverter` package. However, join points for calls to methods in the `Factory` class and calls to main `methods` will not be sent to the Heimdall tool. The abstract class `SafetyFunction` from Figure 6 has three method call join points that may be validated, namely when the `selfCheck` is performed during initialization of the frequency converter, whenever a call is made to `activate` the safety function, and whenever a check is made to see if a safety function `isRequested`.

We will now turn our attention to the execution mapping for the architecture prototype and the CPN model.

4.3 Mapping Execution Events to Model Events

In the architecture prototype for the frequency converter, the only join points of interest are method call join points. These join points (and join points for the

```
<element>
  <joinpointevents>
    <callevent>
      <id>4</id>
      <call>safeinverter.external.DigitalIO.requestDSF</call>
    </callevent>
  </joinpointevents>
  <modelevents>
    <modelevent><id>DigitalIO'Request_DSF(n,3)</id></modelevent>
  </modelevents>
</element>
```

Figure 10: An excerpt from the execution mapping.

```
public class DigitalIO extends IO {
        private SafetyFunction safetyFunction;

        public DigitalIO (State state,
                        SafetyFunction safetyFunction) {
                super(state);
                this.safetyFunction = safetyFunction;
        }
        public void selfCheck() {}
        public void requestDSF() {
                safetyFunction.activate();
        }
}
```

Figure 11: The DigitalIO class from the executable software architecture.

construction of objects of interest, cf. Section 4.2) are specified in the aspect in Figure 9. The architecture prototype contains very few join point for getting or setting fields, and none of these join points need to be validated. Currently, the XML file specifying the mapping must be created manually. The mapping was created after a careful and systematic examination of the architecture prototype and the CPN model.

In the execution mapping for the architecture prototype, each method call join point is mapped to one or more events in the CPN model. Many join points are mapped to just a single transition, while one of the join points is mapped to ten model events. Figure 10 shows an excerpt from the execution mapping. This example shows the XML format of the mapping of a single method call join point to a single model event. In this case a call to the `requestDSF` method to the object with id 4 from the class `safeinverter.external.DigitalIO` is mapped to the model event which is the transition Request_DSF (shown in Figure 4) in the module DigitalIO, where the variable n of the transition is bound to 3. In the CPN model, digital input number 3 is associated with the 'controlled stop' or 'safe delay' safety function. In the architectural prototype, the object with id 4 is an instance of DigitalIO that is associated with the safe delay safety function. The DigitialIO class is shown in Figure 11.

Let us consider what steps are taken when validating the architecture prototype and the `requestDSF` method is called in a DigitalIO object. When the method is called, the `SafeInverterTracer` aspect will cause the signature for the method call as well as the id of the target object to be sent to the Program Execution Listener in the Heimdall tool. The Program Execution Listener will then use the Program Execution Mapper to locate the model events (if any) that the program execution event is mapped to. Given the information in Figure 10, we know that this join point is mapped to a model event corresponding to the transition Request DSF with the variable n bound to 3. The textual representation of this model event shown near the bottom of Figure 10 is sent from the Model Tool Communicator to CPN Tools.

A library that allows CPN Tools to interact with Heimdall has been implemented. This library contains functions for sending and receiving data via a TCP connection with a Model Tool Communicator in Heimdall. Additional functions are used to run simulations based on the commands that are received from the Model Tool Communicator. When a specification of a model event is received from the Model Tool Communicator, there are three possible outcomes. If the event specification corresponds to an event in the model and the corresponding event can occur, then the event will occur in the simulator, and an appropriate response will be returned to the Model Tool Communicator. If the event specification corresponds to an event but the event cannot occur in the current state of the model, then the state of the model remains unchanged, and the response to the Model Tool Communicator indicates that the event cannot occur. The fact that a particular event cannot occur may indicate that the behaviour of the executable code is not consistent with the behaviour specified by the model. Finally, the event specification may not correspond to any known events in the model, and this indicates that there is an inconsistency somewhere, i.e. either in the model, in the executable code, or in the mapping from the code to the model. If the Model Tool Communicator requests that the Request DSF transition should occur, then this is a known event in the model, and a response will be sent back to the Model Tool Communicator indicating either that the event has occurred, thus validating the most recent sequence of join points, or that the event cannot occur in the current state of the model. If the event does not occur in the model, then the program execution has performed a sequence of execution events that cannot be validated, and an error has been found.

5 Discussion and Conclusion

This paper has introduced the Heimdall tool and its associated approach to mapping program execution events to events in a discrete-state model. The tool has been integrated with CPN Tools and has been used to validate architectural prototypes. Even though the evaluations have been made in the context of a real safety-critical system development project, the Heimdall tool can still be considered experimental in nature.

First of all, a full validation during a development project is needed. This will stress the usability of the actual mapping mechanism. The current mapping mechanism is essentially simple since one of our goals have been to support experimentation with mapping from architectural prototype executionss and other types of program executions. One area in which the mapping mechanism could be improved is in considering transitions that do not correspond to method calls. It should be possible for them to occur if they are enabled: e.g., if a particular transition that is mapped from a method invocation is not enabled, then it might become enabled if one or more of the unmapped transitions/events occur.

Also a more thorough evaluation could potentially illustrate to which extent architectural prototyping is actually useful and beneficial in safety-critical system development.

Secondly, there is a definite lack of proper tool support for constructing Heimdall mappings. One step in this direction would be to be able to generate a set of possible program execution events/model events to base the mapping construction on. In particular if iterations on the software architecture and models are considered, better tool support is of importance.

Even though the Heimdall tool can in no way prove that a specific architecture will lead to safe software, it may help in doing so by allowing architects to experiment with and partly validate their architectural designs thus potentially leading to better and safer software architectures.

References

[1] M. R. Barbacci, R. Ellison, A. J. Lattanze, J. A. Stafford, C. B. Weinstock, and W. G. Wood. Quality Attribute Workshops (QAWs), Third Edition. Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, 2003.

[2] J. Bardram, H. B. Christensen, and K. M. Hansen. Architectural Prototyping: An Approach for Grounding Architectural Design and Learning. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 15–24, Oslo, Norway, 2004.

[3] J. Bardram, H. B. Christensen, and K. M. Hansen. Exploring quality attributes using architectural prototyping. In *Proceedings of the First International Conference on the Quality of Software Architectures, QoSA 2005*, volume 3712 of *LNCS*, pages 155–170, Erfurt, Germany, 2005.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 2nd edition, 2003.

[5] L. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *Proceedings of the 4th International Workshop on Product Family Engineering*, 2001.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[7] CPN Tools. Online: http://www.daimi.au.dk/CPNTools/.

[8] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.

[9] *DEF (AUST) 5679: The Procurement of Computer-based Safety Critical Systems*, 1998. Australian Defence Standard.

[10] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[11] M. Heimdahl and N. Leveson. Completeness and consistency checking of software requirements. *IEEE Transactions on Software Engineering*, 22(6), 1996.

[12] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture.* Addison-Wesley, 1999.

[13] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[14] International Electrotechnical Commission. *Functional Safety of Electrical/ Electronic/ Programmable Electronic Safety-Related Systems*, 1st edition, 1998-2000. International Standard IEC 61508, Parts 1-7.

[15] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353, 2001.

[17] L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[18] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[19] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[20] N. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[21] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–256, 1986.

[22] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 2nd edition, 1996.

[23] K. H. Pedersen and C. Constantinides. Aspectada: Aspect oriented programming for Ada95. In *SigAda '05: Proceedings of the 2005 annual ACM SIGAda international conference on Ada*, pages 79–92, New York, NY, USA, 2005. ACM Press.

[24] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: an AOP extension for C++. *Software Developer's Journal*, (5):68–76, 2005.

[25] visualSTATE. Online: http://www.iar.com/vs.

[26] L. Wells and T. Maier. Specifying and analyzing software safety requirements of a frequency converter using coloured Petri nets. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 403–422. Springer, 2005.