Model-Based Software Engineering for Embedded Systems

Simon Tjell

Department of Computer Science

University of Aarhus

Progress Report

June 12, 2007

Contents

Chapter 1 -Introduction

This progress report describes the scientific work, which has been done as part of the first one and a half years of my Ph.D. studies. The goal of the report is to document the work and give directions for future work.

The main topic of my work has been the investigation of how formal modeling languages can be used to support the development of embedded systems at different stages in the development process. So far, the work has been manifested by the production of six papers. These papers will be described in the frames of a generic software development process in order to make it easier to understand their mutual relations and their relations to the research field in general. The six papers can be found online: `http://daimi.au.dk/~tjell/artikler`

1.1 Structure of the Report

The report is structured as follows: in Chapter 2, I introduce a generic software development process. This process is used as reference for a description of the typical activities and an introduction to how these activities may be augmented by the use of formal modeling languages. The chapter will introduce work that relates to my own work at a general level as part of the description of the development activities. In the next three chapters (3, 4, and 5), I will describe the contents of the six paper that I have been involved in during the first half of my Ph.D. studies. The papers have been categorized with respect to the activities that are introduced in Chapter 2. A subset of the activities have been covered by the work described in the papers. This results in the following chapters: in Chapter 3, three papers ([14; 15; 42]) are described. These papers are all related to requirements analysis and validation with models. In Chapter 4, two papers are described ([16; 40]). These papers are both related to design verification with models. Finally, Chapter 5 describes a paper ([41]), which is related to the testing activity. The progress report is concluded in Chapter 6 where directions for future work are also discussed.

Chapter 2 -Background

Figure 2.1 informally illustrates a generic software development process. The process is inspired by generic process activities described in [37]. The figure consists of a group of development activities (single-ended arrows), products (rectangles) and Validation & Verification (V&V) activities (double-ended arrows). Basically, an activity is performed based on some input product and the result of the activity is the production of a certain output product. V&V activities are applied in order to investigate the relationships between two products in order to evaluate if a satisfying level of consistency has been achieved. The method for determining this and the meaning of consistency depends on the type of V&V activity.

The following sections give an introductory description of the specific activities and products in the generic development process while referring to the elements of Figure 2.1. This description will serve as a frame for the description of my work.

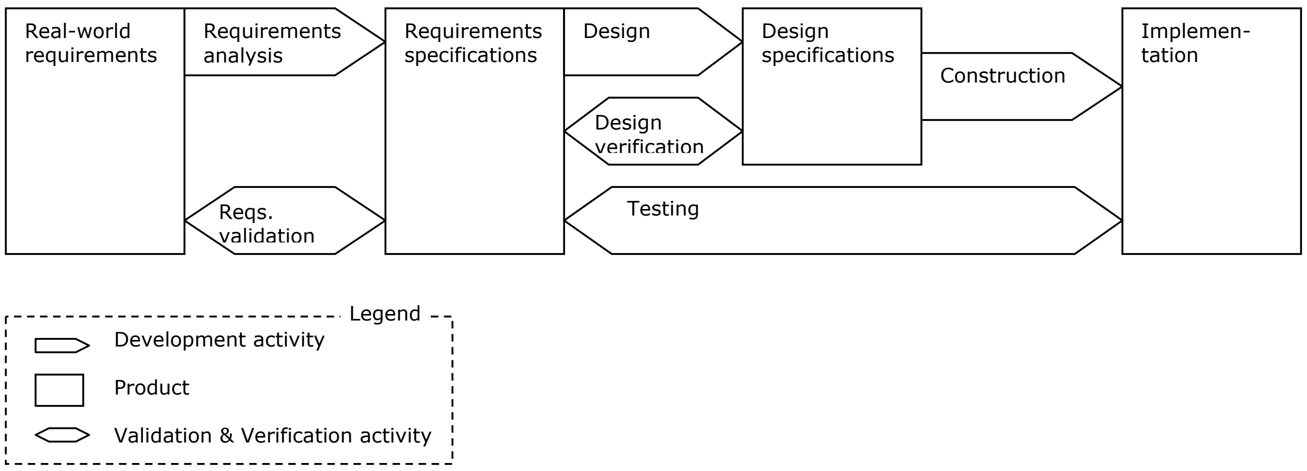


Figure 2.1: Activities and products of a generic development process

2.1 Development Activities and Products

In this section, we describe the development activities and their input and output products. The structure of the section is based on the description of activities. These descriptions are tied together by the description of the products.

2.1.1 Requirements Analysis

The requirements analysis activity[1] is one of the first in a development project. It concerns the identification and analysis of requirements to the final product. The requirements derive from real-world requirements, which are typically determined by a potential client with a desire to have a product developed by a contractor. The requirement analysis activity is concerned with recording and categorizing these requirements. This activity is important since its product (the requirements specification) is the basis for a potential contract and also serves as a medium for communication with the technical employees which will eventually have to design and implement a product satisfying the requirements.

The requirements in a requirements specification can generally be categorized into two disjunct categories: functional and nonfunctional requirements. Functional requirements describe the required behavior of a system; e.g "when the button is pushed, the motor is started". Non-functional requirements describe constraints related to the functional requirements; e.g. "the motor should be started at least 2 seconds after the button push". In general, all requirements in a requirements specification should be *testable*. This is necessary in order be able to verify that the requirements are satisfied by a final implementation. Both of the examples given here are testable - i.e. it is possible to push the button and observe the reaction (and measure the delay). A trivial example of a non-testable requirement could be the following: "The motor should not turn too fast".

2.1.2 Design

The design activity is focused on refining the requirements specification in order to obtain a design specification. Whereas the requirements specification describes *what* should be accomplished by the system being developed, the design specifications describe *how* this should be accomplished at an increasingly concrete level. The specification

[1]The term "requirements analysis" is used to cover a collection of requirements engineering disciplines such as elicitation, analysis, specification, documentation, and management.

typically starts at the architectural level where the overall structure of the system components is determined. In embedded systems, the architectural design is also aimed at the partitioning problem - i.e. deciding which functionality is to be supported by hardware and which is to be supported by software. From there, the design activity move into looking at the interfaces between the system components that are connected according to the architectural design. Next natural step is the design of the internals of system components. At this level, the design is aimed at internal architecture. At the lowest level of abstraction, attention is turned to the design of data structure and algorithms internally in the components.

2.1.3 Construction

The construction activity is concerned with developing the actual product based on the design specifications. For embedded software systems, the construction activity consists of the development of hardware and the programming of software. The output of the construction activity is the final product. In embedded systems, this activity is composed by a wide range of sub-activities in two main categories: hardware and software development. Hardware development is concerned with the implementation of functionality by means of analog and digital electronics. Software development is concerned with the programming of software to be executed in micro controllers and micro processors in the hardware platform of the product. In modern electronics, the division between hardware and software is becoming increasingly blurry with the spreading use of programmable hardware such as Field-Programmable Gate Arrays (FPGAs), System-on-Chip (SoC) etc.

2.2 V&V Activities

In the context of our generic software development process, a V&V activity serves to evaluate the consistency of a chain of development activities by comparing the input product of the first activity and the output product of the last activity based on some criterion. As an example, the design verification activity evaluates the consistency of the design activity while the testing activity evaluates the consistency of the chain formed by the design and the construction activities.

2.2.1 Requirements Validation

The requirements validation activity is concerned with evaluating the correctness of the requirements specification which was produced by the requirements analysis and elicitation activity. The requirements need to be correct with respect to many different properties. Some properties are related to the internal relationships between requirements: e.g. they are not conflicting, they are all testable, they are complete based on some measure, etc. Other properties are related to the relationship between the real-world requirements and the requirements specified in the requirements specification - meaning that the requirements of the requirements specification are correct if they based on some measure represent the real-world requirements in a precise manner.

The requirements validation activity is important in most development processes because it defines the final requirements specification which serves as an input product for the design activity. In a typical waterfall approach, this activity should be completed before the design activity is started. In this case, the validation activity is crucial because all the development activities following it depend on the agreements on requirements, which it represents. In iterative processes, the validation activity will be revisited as more insight about the system is gained through repeated design and construction phases.

Together with the requirements analysis activity, the requirements validation activity forms the field of Requirements Engineering [3].

2.2.2 Design Verification

The design verification activity is concerned with the verification of the design specification, which is the product of the design activity. The contents of the design specification is the result of a number of design choices. The design verification activity takes place at many different levels ranging from the architectural level where the choice of architecture is evaluated to the algorithm level where properties such as the performance and correctness of algorithms are considered. The overall purpose of the design verification activity is to assist in making sure that the right design is chosen before the construction activity is started since the final output product serves as the input product for the construction activity.

2.2.3 Testing

The purpose of the testing activity is to verify that the product of the construction activity is operating as expected - i.e. that the implementation satisfies the requirements that were stated in the requirements specification as the output of the requirements analysis/elicitation activity.

Apart from very small projects, the construction activity will typically result in the production of a number of software components based on the architectural design manifested in the design specification. These components are often tested individually and in groups based on their specifications in the design specification with respect to interface and behavioral properties. This type of activity is not seen in the generic process, which we use as reference here. The testing activity seen here represents the activity of verifying that the composite behavior of the implementation in whole behaves as required in the requirements specification. This activity is also known as acceptance testing.

An alternative to testing is static analysis. In this approach to the testing activity, the source code (or hardware design) of the implementation is investigated in search for malfunctions without actual execution.

2.3 Formal Modeling in the Development Process

"Thou shalt formalize but not overformalize" [6]

Many years of research have gone into investigating how the software development process can be augmented and enforced by the use of formal modeling techniques. This section gives an example-based overview of how formal modeling can be applied to the activities of the development process, which were described in the previous sections.

In [6] the authors present a simple classification of levels to which formal methods are used throughout a development process (this paper was followed by a revised version 10 years later: [7]). The three levels are described in Table 2.1.

In recent years, many researchers have argued that the higher levels of formalization are not practically applicable in many real-world development projects [8; 26]. This is partly due to the fact that new software systems are rarely developed from scratch but are rather based on existing legacy components and use of off-the-shelf software components from third party suppliers. Another reason is the classical phenomenon of state

Level	Name	Description
0	Formal specification	Formal modeling is used for specification of requirements. The model can be used for validation of the requirements. The purpose of formal models at this level is to eliminate misunderstandings about requirements in both the requirements analysis/elicitation activity and the design activity. Models at this level also facilitate automated testing.
1	Formal development and verification	Formal verification techniques are applied to abstract models of a system. Synthesis techniques are used for automatically refining a requirements specification model toward an executable implementation. Verification results from abstract level analysis can be inferred to more concrete levels because of the automated refinement.
2	Machine-checked proofs	This approach is similar to that of level 1 with a fundamental difference: at this level the verification is based on mathematical proofs about system properties. In contrast, the approach at level 1 is based on exhaustively identifying all possible states of a system (the state space) and use this as a basis for the verification analysis.

Table 2.1: Levels of formalization

explosion which relates to the fact that real-world systems are often too complex to be analyzed by exhaustive methods - i.e. methods where the entire state space of a system is generated and analyzed - because the number of possible states is too big. This is particularly the case for embedded systems where properties such as real-time operation, interaction with and dependency of the environment through sensors/actuators result in very complex state spaces. A reaction to this has been the focusing on the lowest level of formalization. This level has been given the name *formal methods light* [25]. The work being presented in this progress report is exclusively focused at exploiting formal modeling techniques at the *light* level as will be seen when the six papers are introduced.

We will now have a look at some concrete usages of models in the activities of the generic software development process, which was introduced in the beginning of this chapter.

2.3.1 Requirements Analysis and Validation

"One of the major problems in developing new computer applications is specifying the user's requirements such that the Requirements Specification is correct, complete and unambiguous." [19]

In classical software engineering, requirements specifications are often expressed in natural language documents possibly including some informal or semi-formal figures. A typical combination is Use Case diagrams and Sequence Diagrams (the UML 2.0 superset of Message Sequence Charts) [33]. In this combination, the Sequence Diagrams are used for describing scenarios of interaction between the system and the environment in which it is operating. The behavior described at this level is observable behavior - i.e. in contrast to internal behavior described in the more detailed design specifications. Since one of the main purposes of the requirements specification is to provide an unambiguous description of the requirements, it is sometimes useful to use more precise methods for such as formal modeling languages. The benefit of using formal modeling for requirements specifications comes with one of the key properties and commonalities of formal modeling languages: the syntax (and typically the semantics) are formally defined. This makes formal models precise and usable for automated analysis and transformation [47].

On the other hand, they may be harder to comprehend by non-technical stakeholders - and this is in particular a problem in the requirements analysis and validation activities where a client is typically involved [17].

Below, two specific examples are given of the use of formal methods for requirements specification and validation:

Formal specifications as a means of communication In a very general view on the requirements activities, four types of stakeholders are involved in or have are affected by the specification: clients, specifiers, verifiers, and implementers [47]. These stakeholders need to communicate in a lot of combinations: the client and the specifier need to agree on some mutually understandable description of the client's requirements to the system. This description must be understandable to the implementer, who is going to (design and) implement the system. When the system has been implemented, the requirements specification, which was developed by the specifier should be usable as a basis for testing, which is performed by the verifier. Formal (modeling) languages can be an important means of communication between all these stakeholders because of the characteristics of preciseness. Furthermore, a formal specification of the requirements may serve as a supplement to a contract between the client and the company developing the system.

Validation by simulation/animation A helpful technique for validation of requirements specifications is the use of graphical animations for visualization of the required behavior of the system [44]. Many formal modeling languages are executable and if tool-support is available it is possible to simulate scenarios based on a formal requirements specification. The execution of the requirements specification may be used for driving a domain-specific[1] graphical animation, which is shown to the client. Examples of this approach can be found in [28; 45].

Apart from these examples of specific applications of formal methods in the requirements activities, it is commonly known that the process of specifying system properties in a formal language in itself is helpful in understanding the requirements. Wing describes this experience in the following quote:

> *"The greatest benefit in applying a formal method often comes from the process of formalizing rather than from the end result. Gaining a deeper understanding of the specificand by forcing yourself to be abstract yet precise about desired system properties can be more rewarding than having the specification document alone."* [47]

2.3.2 Design and Design Verification

It is important to maintain consistency between the requirements specification and the design specification during the design activity. In informal approaches to the development process, it may be difficult to achieve a satisfactory level of confidence in the consistency between these two products (because of missing precision). The design verification is aimed at evaluating the level to which the design specification complies with the restrictions defined in the requirements specification. This activity can be supported when a formal modeling language is used for expressing at least one of the products. Adrion et. al. express this property in the following way:

> *"The more formal and precise the statement of the [...] product, the more amenable it is to the analysis required to support verification"* [4]

[1] a graphical animation is domain-specific if it is composed by entities that are well-known in the domain of the system being.

If the design specification is expressed by means of a formal modeling language, its consistency may be verified with respect to both functional and nonfunctional requirements in the requirements specification. Please note, that the term *verification* is used to denote both proof by formal verification and empirical analysis by simulation.

Some examples of how a formally expressed design specification can be utilized are given here:

Performance analysis, design space exploration etc. If the design specification is represented by a formal model, which is executable it may be used in general to conduct experiments in order to investigate non-functional properties such as performance, delays, queue lengths etc [5; 13]. The simulations can be repeated and statistically analyzed. It may also be possible to sweep the space of parameters in order to estimate consequences of design configurations and optimize these at an early stage.

Model checking An alternative to simulation-based analysis is model checking [11]. In model checking, the basic approach is to generate a large collection of possible states in which the system may exist based on a model. This collection (the state space) can then be investigated. The analysis may involve investigation of trivial properties such as deadlock situation, fairness, and liveness as well as more complex properties. The latter are often expressed as assertions in some variant of temporal logic [27; 29; 38] and these assertions are then evaluated against the design specification.

2.3.3 Construction and Testing

If the requirements and design specifications of a system have been expressed in terms of formal models, these models are usable for several purposes when the system is implemented. A list of examples are given here:

Automatic generation of executable code If the design specifications are represented by an executable model it may be possible to generate (parts of) the executable implementation automatically. A classical paradox in this technique is the gap between different levels of abstraction in the model and in the implementation: abstraction is a key property of a useful model [36] while much more details are described in the implementation. One approach is to generate a skeleton of the implementation automatically and leave the finalization to be made manually by a developer. When this is approach is taken, care must be taken in the interpretation of results of analysis based on the model.

Model-based testing A formal model of the specifications for a system can be used for testing the implementation. In model-based testing [12; 35; 43], this is done by automatically deriving a testing oracle from the formal model. The oracle can be used for automated generation of large collections of test cases with which the implementation is validated. In its basic form, a test case is a relation between test input and test output. Given a test input, the oracle defines the correct output (or outputs in the case of nondeterministic systems. The test inputs can itself be derived from the model based on some definition of coverage objective such as state coverage, module-coverage etc.

Automatic generation of assertions If the formal model contains axiomatic expressions (such as preconditions, postconditions, and invariants) these may be adopted in the implementation as assertions that are used for monitoring the implementation at runtime. The approach is similar to that of model-based testing. It differs by integrating the functionality similar to the oracle into the implementation itself. Examples of this technique can be found in [9; 32; 34]

2.4 A Formal Modeling Language: Coloured Petri Nets

In this section, we look at a specific formal modeling language, which has form part of most of the work I have done so far: Coloured Petri Nets. Here, the language will be introduced and the following descriptions of my papers will contain examples of its application for formal modeling in the activities found in Figure 2.1.

Coloured Petri Nets (CPN) [24] is a graphical modeling language in which models are expressed by means of basic graphical elements in combination with textual declarations and inscriptions.

Coloured Petri Nets (or CPN) is a graphical modeling language with formally defined syntax and semantics. A CPN model expresses both the states and the events of a system. Events are represented by *transitions* (drawn as rectangles) and the states are represented by *places* (drawn as ellipses). These two types of nodes are connected by *arcs* that lead either from a transition to a place or from a place to a transition. The graphical structure is supplemented by declarations of functions, constants etc. in a functional programming language. This language is called CPN ML [1] and is an extension of the more commonly known Standard ML [31]. Expressions in this language are also used for annotations with different purposes in the graphical nodes of the model.

We will now look into how state is represented and how it is changed. Each place is able to hold a (possibly empty) collection of values of a given data type. We call these values *tokens*. The current state of a given place is determined by the collection of tokens at a given point in (model) time. We can see this as a local state while the global state of a model is the composition of local states. The local states and thereby the global change when transition *fires*. A transition must become *enabled* before it is ready to fire. There are several restrictions involved in the determination of which transitions are enabled. The main restriction is a quantitative one: the transition must be able to consume a number of tokens from all its *input place*. An input place to a transition is a place from which an arc leads to the given transition. A transition also have *output places*. Those are the places to which an arc leads from the given transition. The quantitative restriction is defined by the expressions in the arcs connecting the input places to the transition. A restriction is purely quantitative if it only expresses a required number of tokens to be *consumed* from each of the input places. A more precise restriction is a qualitative one, where requirements to the values of the tokens being consumed are expressed. Finally, the enabling of a transition could be further restricted by use of a *guard*. This is an expression that evaluates to a boolean value over the values of all tokens being consumed. A transition is only enabled if its guard evaluates to true. When a transition fires, the result of the state change is manifested by the production of new tokens in its output places. The values of these tokens may or may not depend on the values of the consumed tokens and depend on constants or function that are expressed in the outgoing arcs.

CPN is only one of many high-level extensions of classical Petri Nets. The main difference between classical Petri Nets and CPN is related to the tokens: in classical Petri Nets, tokens are indistinguishable (anonymous) and a local state is determined by the number of tokens in a place. In CPN, tokens have values (colors) of potentially complex data types that make them distinguishable and a local state is the combination of the number of tokens in a place and the values of these tokens. The integer arc weights of classical Petri Nets have been extended in CPN by terms of a full functional programing language (CPN ML [1] which is an extension of Standard ML [31]). The CPN ML inscriptions of a CPN model are used for things such as selection and manipulation of tokens, restriction of enabling of transitions based on token values, specification of initial states and more. The CPN formalism also extends classical Petri Nets by means of a method for specifying model as a hierarchical composition of a collection of modules. Finally, the CPN models can be annotated with real-time inscriptions that specify timing properties of the model. This makes it possible to use a CPN model for simulation of real-time systems.

Chapter 3 -Requirements Analysis and Validation with Models

The sections 2.1.1 and 2.2.1 introduced the activities related to requirements analysis and validation while Section 2.3.1 gave an overview of how formal methods may be applied to these activities. This chapter presents three papers that are all related to the usage of formal models in the requirement activities.

3.1 CPN Models as Requirements Specification for Reactive Systems

This section introduces a technical report [15] in which we propose an approach to how informal functional requirements for a reactive system may be used as a basis for the generation of a formal requirements specification that consists of a composite CPN model.

The work presented in the paper relates to the activities and products of Figure 2.1 in the following way: the requirements analysis activity is enforced by the use of a CPN model as a representation of the resulting requirements specification product. The approach tempts to introduce formalization at an intermediate stage during the requirements analysis activity (Section 2.1.1) at which some informal requirements specification product is assumed to be available. We describe how this can be brought to a more formal form (a CPN model with some specific properties) - i.e. a more formal requirements specification. At the same time, the resulting requirements specification model is expected to be usable during the requirements validation activity (Section 2.2.1) since it is an early-stage executable representation of the system being developed.

The presentation of the approach is based on an elevator controller case study. The functional requirements to this controller have initially been expressed at a low level of formality by means of use cases with scenarios described in UML 2.0 sequence diagrams. The elevator controller is a classical example of a *reactive system* and as such it complies well with Wieringas definition of this class of systems:

> *"A reactive system is a system that, when switched on, is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment."* [46]

This definition implies what other researchers such as Jackson [21] have recognized: it is useful and often necessary, to describe not only the system itself, but also the environment in which the system must operate.

In our approach, we construct an executable model that is, in the first place, *controller-and-environment-partitioned*. This key property means that it is a description of (1) the desired behavior of the controller itself; (2) the desired behavior of the composite system that is made up of the controller, plus relevant external entities in its environment; and (3) the assumed behavior of these external entities. Additionally, the description must clearly distinguish between the controller and the environment and also between desired and assumed behavior. For example, desired behavior is that an elevator car stops when it arrives at a floor for which there is a request; assumed behavior is that the motor starts and sends the car downwards when it receives a certain signal. The reason that we emphasize the distinction between desired and assumed behavior is that developers have freedom to make design choices regarding the former, while regarding the latter they "just" have to accept and understand what is given and act accordingly.

The second key property of the model is that it must be *use case-based*, which means that it is constructed from a given use case diagram and can reproduce the behavior described in accompanying descriptions of scenarios for the use cases. The reason that we enforce this property is that use cases are a convenient and widely-used technique. We would like our approach to be useful in projects that apply use cases for requirements engineering, because this may increase the possibility of its industrial adaptation. Our approach expands, refines, and supplements use case descriptions through the creation of a model, which can be seen as an executable version of a given use case diagram.

3.1.1 The Elevator Case Study

This case study considers a simplified version of an elevator system with two elevator cars in a six-floor building. The main responsibility of the elevator controller is to control the movements of the cars, which are triggered by passengers pushing buttons. On each floor, there are hall buttons, which can be pushed to call the elevator; a push indicates whether the passenger wants to travel up or down. Inside each car, there are floor buttons, which can be pushed to request the car movement to a particular floor, and there is one button to force the car door to open. The controller is also responsible for updating a floor indicator inside each car that displays the current floor of the car. Similarly, a direction indicator must be updated. Figure 3.1 shows the use case diagram for the elevator controller. As usual, the use case diagram identifies and names the use cases that the elevator controller must support, and shows which use cases each external actor participates in. The actors in the use case diagram are the external entities that the controller interacts directly with. These entities are given, and we cannot change them or affect them in our development project, but we must know how they behave. This conception of actors in a use case diagram may deviate from more common conventions.

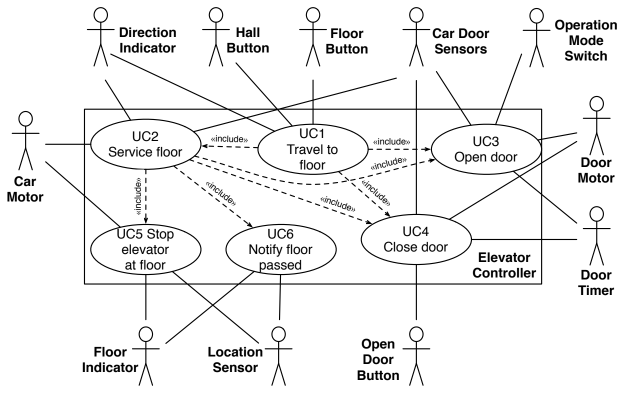


Figure 3.1: The use case diagram of the elevator controller system

Each use case is described by a combination of natural language text and one or more sequence diagrams that describe the desired behavior of the elevator controller in combination with the environment. One sequence diagram always describes the main scenario of the use case. An example of such a sequence diagram can be seen in Figure 3.3. The sequence diagram in question depicts the main scenario of UC1. In UC1, the elevator handles the scenario of a passenger requesting an elevator by pushing a hall button. The elevator controller sends an elevator car to the origin floor, the passenger enters and selects a destination floor, the elevator is moved to the destination floor where the passenger leaves.

Similar descriptions exist for the other use cases. There are some dependency relationships among the use cases (for example, to complete its execution, UC1 needs the functionalities provided by UC2, UC3, and UC4). They are specified in the use case diagram by *include* relations and in the sequence diagrams by *ref* operators.

3.1.2 The CPN Model

This section gives an overview of the CPN model that has been created for the elevator controller case study, based on the artifacts presented in the previous section.

The CPN model is structured in a hierarchy of which the topmost modules are shown in Figure 3.2. The boxes in the figure represent modules while the connecting lines represent relationships in the hierarchy. For example, the `Top` module contains two *substitution transitions* [1]: one bound to the `Controller` module and one bound to the `Environment` module.

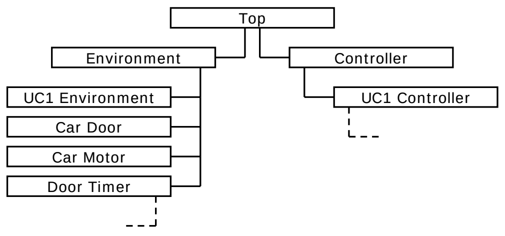


Figure 3.2: Structure of the CPN model.

The topmost module of the CPN model (Figure 3.4) is structured to describe two domains: the controller and the environment. In this way, the CPN model as a whole is constructed to ensure the first key property we pursue; it is controller-and-environment-partitioned (the distinction between desired and assumed behavior in the environment is dealt with on the lower levels in the hierarchy). Each of the two domains is represented at the top level by a substitution transition as seen in 3.4.

The two domains communicate through an interface formed by a collection of shared phenomena [21]. A *shared phenomenon* is a state or an event that is observable by both domains while being controlled by only one domain. In contrast, a *private phenomenon* is only observable within the controlling domain (not to be confused with the controller domain).

In our case study, an example of a shared phenomenon is the event of a passenger pushing a request button. This event is initiated by the environment and is observable by both the controller and the environment. The controller might also record all pending requests in an internal data structure, and this would be a private phenomenon of the controller. In CPN models, places are not only used for holding information about states, but also for exchanging information about the occurrence of events. Thus, a place can be seen as a communication channel between two modules rather than a state holder. In the

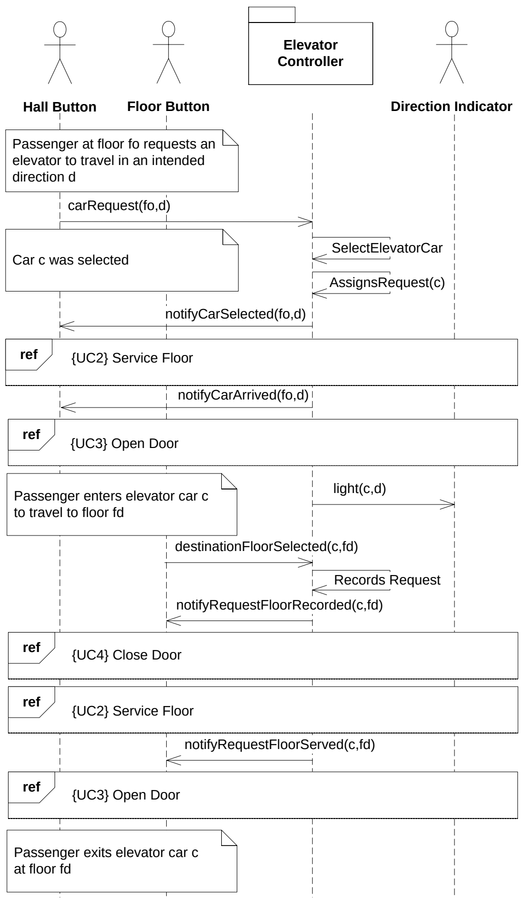


Figure 3.3: A sequence diagram for UC1

[1]Substitution transitions are used in hierarchical CPN models to represent instances of modules

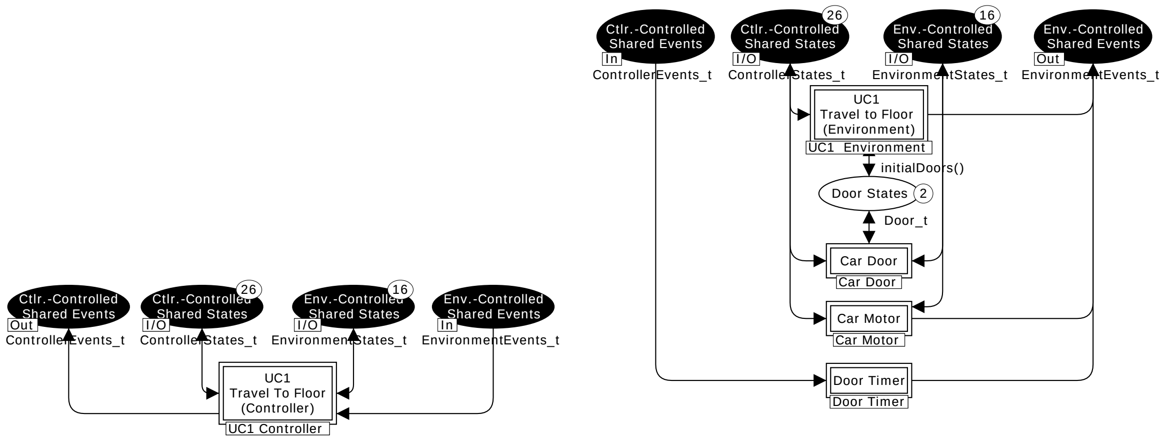

Ctlr.-Controlled Shared Events
Out
ControllerEvents_t
Ctlr.-Controlled Shared States
26
I/O
ControllerStates_t
Env.-Controlled Shared States
16
I/O
EnvironmentStates_t
Env.-Controlled Shared Events
In
EnvironmentEvents_t
UC1 Travel To Floor (Controller)
UC1 Controller
Ctlr.-Controlled Shared Events
In
ControllerEvents_t
Ctlr.-Controlled Shared States
26
I/O
ControllerStates_t
Env.-Controlled Shared States
16
I/O
EnvironmentStates_t
Env.-Controlled Shared Events
Out
EnvironmentEvents_t
UC1 Travel to Floor (Environment)
UC1 Environment
initialDoors()
Door States
2
Door_t
Car Door
Car Door
Car Motor
Car Motor
Door Timer
Door Timer

Figure 3.5: The `Controller` (left) and `Environment` (right) CPN modules.

model, we enforce a distinction between shared events and shared states and between phenomena controlled by either the controller or the environment. This results in four places that are shown in Figure 3.4. The interface between the controller and the environment is represented by these places. Since the shared phenomena places only contain tokens that represent shared phenomena, the structure defined in the `Top` module helps to ensure that the CPN model possesses the environment-and-controller-partitioned property.

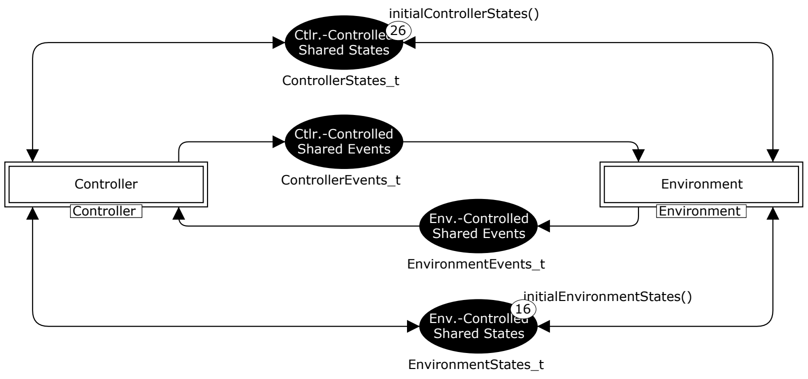

initialControllerStates()
Ctlr.-Controlled Shared States
26
ControllerStates_t
Ctlr.-Controlled Shared Events
ControllerEvents_t
Controller
Controller
Environment
Environment
Env.-Controlled Shared Events
EnvironmentEvents_t
initialEnvironmentStates()
Env.-Controlled Shared States
16
EnvironmentStates_t

Figure 3.4: The `Top` module in the CPN model.

Fig. 3.5 depicts the `Controller` and the `Environment` modules that, as can be seen in fig. 3.2, are sub-modules of the `Top` module. Both modules contain a substitution transition used in the modelling of UC1. The substitution transition `UC1 Travel to Floor (Controller)` represents all the actions initiated by the controller, while the substitution transition `UC1 Travel to Floor (Environment)` represents all the actions initiated by the environment as part of UC1. Both figures include black places holding representations of shared phenomena; they are conceptually glued together with the places with the same names in the `Top` module. Thus, the two modules for UC1 communicate through the interface formed by the black places exclusively. This is an example of how the controller-and-environment-partitioned property guides and restricts the way a use case is modelled.

In the `Environment` module, three external entities (Car Door, Car Motor, and Door Timer) are represented by the correspondingly named substitution transitions.

All use cases of the use case diagram are modeled as separate CPN modules and the result is a model that satisfies the two key properties: it is use case-based and environment-and-controller-based. The main justification for the development of such a model is derived from the benefits of clearly non-ambiguously expressed requirements specifications as discussed in 2.3.1. Since the CPN model is executable, it can be used for multiple purposes. In

[15], we list some examples: simulation of scenarios with potential interleaving of concurrently executed instances of use cases and simulation of failures in terms of unexpected behavior of the environment.

3.2 Defining the Environment-and-Controller-partitioned Property

As described in [15] and summarized in Section 3.1, we find the clear partitioning between environment and controller to be of great importance when formal models such as the one of the elevator controller are used in the requirement activities. In this section, some of the guidelines used informally in the [15] are expressed formally. This is done as the first steps toward making it possible to develop a tool that semi-automatically verifies the compliance with these guidelines based on structural analysis of CPN models. This section describes a paper [42], which has been accepted for a special session on behavioral modeling of embedded systems at SIES 2007. The paper introduces a formalized representation of the guidelines for domain partitioning in CPN models of reactive systems. So far, only non-hierarchical models are considered. Theoretically, all hierarchical CPN models can be unfolded to semantically equivalent non-hierarchical CPN models. However, the contribution of [42] should be seen as a manifestation of the ideas about partitioning which will later be extended to cover hierarchical models without this unfolding. Please note, that this section uses the term *system* as a more general denotion of what we called *controller* in Section 3.1.

The work presented in the paper relates to the activities and products of Figure 2.1 in the following way: the guidelines that are introduced apply to the structure of the model used as a representation of (parts of) the requirements specification product. Since a specific general structure is proposed, the guidelines also affect the work taking place during the requirements analysis activity when this activity is enforced by the use of formal modeling - i.e. the guidelines should be taken into account when the requirements specification model is developed.

In [20], Gunther et al. formally define a reference model for specifying requirements to software system. This model is used as the foundation of the documentation of the guidelines that are presented in this section.

The reference model is composed by five artifacts (W, R, S, P, and M) with the following relationships: (1) The purpose of the program (P) is to implement the specifications (S) and this implementation is restricted by the programing platform (M) and (2) if the program implements the specifications (S), then it will satisfy the requirements (R) if the environment is described by domain knowledge about the world (W).

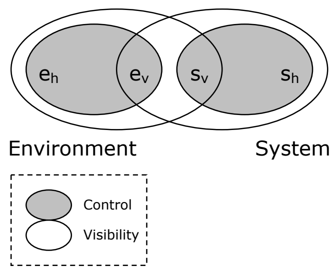


Figure 3.6: Designated terms

The specifications provide a description of the interface between the environment and the system by means of a common terminology based on a collection of *shared phenomena*. A phenomenon is an event or a state, which is controlled by one of the domains and which is either visible to or hidden from the other domain. Figure 3.6 illustrates the different categories of phenomena.

A phenomenon is *controlled* by a domain if the domain causes changes to the phenomenon (in the case of states) or generates the phenomenon (in the case of events). As seen in Figure 3.6, the phenomena in e_v and e_h are controlled by the environment while the phenomena in s_v and s_h are controlled by the system.

If a phenomenon is *visible* in a domain, this means that reactive behavior in that domain could be initiated by the phenomenon. On the other hand, a domain is not able to react based on *hidden* phenomena. To pin this out, the system is only able to react on phenomena that exist in e_v, s_v and s_h while the environment is only able to react on phenomena in e_h, e_v and s_v. This distinction is important in order to be able to identify the interface between the system and the environment.

The interface between the environment and the system is formed by two collections of phenomena: e_v and s_v. These two collections are the *shared phenomena*. In an *environment-and-system-partitioned* model, all interaction between the two domains is performed by means of shared phenomena. [42] focuses on the identification of shared phenomena in CPN models.

Here follows the formal definition of the syntax of non-hierarchical CPN models. This definition is important here, because the following restrictions will be based on it. As described in [23], the syntax of a non-hierarchical CPN is given by a tuple: $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ satisfying the following properties: Σ is a finite set of non-empty *types* (colour sets), P is the finite set of all *places*, T is the finite set of all *transitions*, A is the finite set of all *arcs*, N is the *node* function, C is the *colour* function, G is the *guard* expression function, E is the *arc expression* function, and I is the *initialization* function. For details, please refer to [23] where the semantics of CPN are also defined.

We define the A' relation that relate all pairs of nodes that are connected by arcs s.t.: $A' \subseteq (P \cup T) \times (P \cup T)$, $A' = \{(n_1, n_2) | \exists a \in A : N(a) = (n_1, n_2)\}$ The first component of an element in the relation is the source node and the second element is the destination node.

3.2.1 Expressing the Reference Model in CPN

This section describes how the reference model is to be expressed by means of subsets of the nodes in a CPN model. We start out with a brief introduction of an example model that will be used to illustrate the principles being presented here. Figure 3.7 shows an example of a very simple non-hierarchical CPN model. In this case, a vending machine has been modeled at a rather abstract level. The left-most white nodes model the behavior of the environment, which consists of a person who wants to buy a product from the vending machine. On the right-hand side, the white nodes model the behavior of the vending machine. In between the two domains, we see four dark-colored places (the colors have no syntactical meaning). These places form the interface between the system and its environment - i.e. the interface S, which was described as an important part of the reference model. We will next look into how the phenomena in S are identified and categorized.

We start out by linking the two groups of hidden phenomena to the nodes of the CPN model:

$e_h \subseteq P \cup T$ and $s_h \subseteq P \cup T$

We do the same for the groups of visible phenomena:

$e_v \subseteq P$ and $s_v \subseteq P$

For convenience in the following, we define four subsets that define all hidden, all visible, all environment and all system nodes respectively:

$H = e_h \cup s_h$, $V = e_v \cup s_v$, $E = e_h \cup e_v$, and $S = s_h \cup s_v$

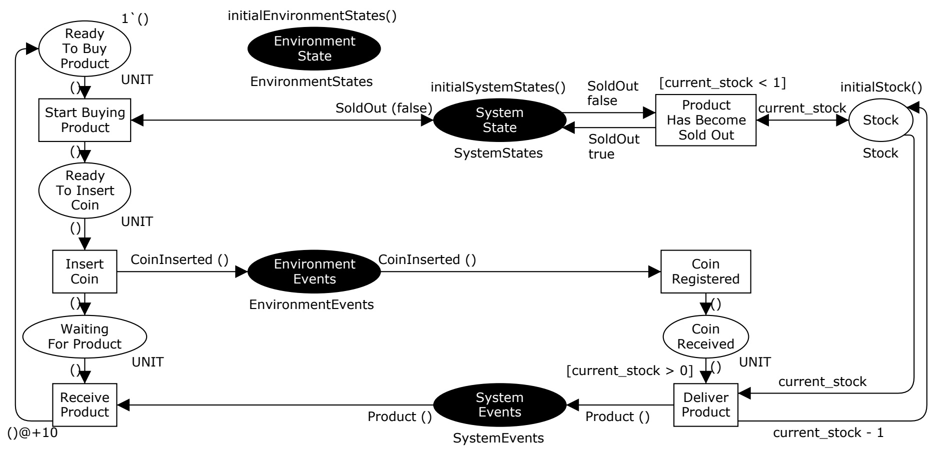


Figure 3.7: A CPN model of a vending machine

We define a collection of very trivial rules that restrict the categorizations of nodes in a model: (1) No nodes belong to both the system and the environment ($S \cap E = \emptyset$), (2) No nodes are both hidden and visible ($V \cap H = \emptyset$), and (3) All nodes can be categorized into two of these subsets ($S \cup E = V \cup H = P \cup T$)

These rules are examples of the rules that are checked automatically by a tool of which a prototype has been implemented as is described in Section 3.2.3. Based on the categorization into the system and environment subsets, we define the relation D, which is used in the following rules for identifying nodes belonging to each their domain:

D is the cross-domain relation s.t.

$D \subseteq (T \cup P) \times (T \cup P)$, $D = \{(n_1, n_2) | (n_1 \in S \wedge n_2 \in E) \vee (n_1 \in E \wedge n_2 \in S)\}$

We write $D(n_1, n_2)$ as shorthand for $(n_1, n_2) \in D$

3.2.2 Communication

We identify two more sets of places which are the places being used for communication between the two domains through shared events and shared states. A shared event is an event that is generated by one domain and observed by the other. A shared state is a state that is controlled by one domain and observed by the other. All interaction between the two domains is performed through the collection of shared events and states. Figure 3.8 show examples of the two types of communication.

We start out by identifying two sets of places; one for representing shared states and one for shared events s.t. $P_{SS} \subseteq V \cap P$ is the set of places representing shared states and $P_{SE} \subseteq V \cap P$ is the set of places used for exchanging shared events. These two sets of places hold all places through which the communication between the environment and the system is performed. We now move on to defining some restrictions to this communication. These restrictions are defined by identifying a collection of arc categories. All arcs to and from P_{SE} and P_{SS} should belong to exactly one of these categories. We also require that no place is used for both events and states: $P_{SS} \cap P_{SE} = \emptyset$.

Communication Through Shared States

Shared states are represented by tokens on places. The value of a token represents the current state of a given property. The state is changed by the controlling domain by changing the value of the token. The state is readable by the other domain. Since the state is reflected by the value of a token, this token should always be available at the place. Firstly, we define a group of categories of arcs that are involved in reading and writing the shared state:

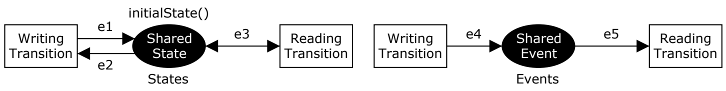


Figure 3.8: Communication through a shared states (left) and events (right)

- A_{SSW_1} is the set of arcs used for writing new values of shared states s.t. $A_{SSW_1} = \{(t, p) \in A' | t \in T, p \in P_{SS}, \neg D(p, t)\})$
- A_{SSW_2} is the set of arcs used for removing the token representing the current value of a state before writing the new value by placing a new token s.t. $A_{SSW_2} = \{(p, t) \in A' | t \in T, p \in P_{SS}, \neg D(p, t)\}$
- A_{SSR_1} is the set of arcs used for reading the current values of a shared states s.t. $A_{SSR_1} = \{(p, t) \in A' | t \in T, p \in P_{SS}, D(p, t))\}$
- A_{SSR_2} is the set of arcs used for reproducing tokens of shared states after reading the current values s.t. $A_{SSR_2} = \{(t, p) \in A' | t \in T, p \in P_{SS}, D(p, t))\}$
- A_{SS} is the set of all arcs being used for communication through shared states s.t. $A_{SS} = A_{SSW_1} \cup A_{SSW_2} \cup A_{SSR_1} \cup A_{SSR_2}$

A shared state is controlled by one domain and read by the other domain. This means that only the controlling domain is able to change the current value of the state. The controlling domain is the domain in which the place holding the token representing the shared value belongs. If such a place belongs to the environment, it means that the shared state is controlled by the environment while it might be observed by the system. To ensure the visibility (and availability) of such states, the places holding the state values must be part of V. In Figure 3.8 (left), the writing transition is in the controlling domain and the reading transition is in the observing domain. The double arc with the $e3$ annotation is shorthand for two arcs with the same expression. Since the expression ($e3$) is the same in both the arc removing a token from the shared state place and the arc placing a token back on the place, the value of the token will not be changed by the firing of the reading transition.

It should be ensured manually or automatically that places holding shared states will always (in any marking) contain a *multiset* (a bag) of constant cardinality (equal to the cardinality of the multiset created by the initial marking of the place. The reason why this is important is that the states should always be available for reading by the observing domain. This is important in order to ensure the intended level of separation between the controlling and the observing domain. Provided that cardinality of the collection of tokens that are produced by output arcs can be predicted statically, the following rule should hold in order to ensure the availability of shared state tokens:

$$|A_{SSR_1}| = |A_{SSR_2}| = |\{((p, t), (t, p)) \in A_{SSR_1} \times A_{SSR_2} | E(p, t) = E(t, p)\}|$$

This property could be evaluated by use of place invariants or by bounding analysis of full state space. Alternatively, the property could be dynamically observed by monitoring the number of tokens on the relevant places runtime - i.e. during tool-based execution of the CPN model. We will not discuss this further here, but it remains as an open topic for future work.

Communication Through Shared Events

Like shared states, shared events are also represented by tokens on places - but in the case of shared events, these tokens are consumed when transitions representing the reading of events fire. The values of the tokens specify the type and possible data the events. Figure 3.8 (right) shows an example of a place holding tokens representing shared events. The occurrence of a visible event in one domain is modeled by the firing of the writing transition. This results in the production of a token identifying the event in the shared event place. In the other domain, this can cause a reaction as modeled by the reading transition.

- A_{SEW} is the set of all arcs used for writing (generating) shared events s.t.
  $A_{SEW} = \{(t,p) \in A' | t \in T, p \in P_{SE}, \neg D(t,p)\}$
- A_{SER} is the set of all arcs used for reading (consuming) shared events s.t.
  $A_{SER} = \{(p,t) \in A' | t \in T, p \in P_{SE}, D(t,p)\}$
- A_{SE} is the set of all arcs being used for communication through shared events s.t.
  $A_{SE} = A_{SEW} \cup A_{SER}$

Internal Communication

We have now identified the communication that crosses the interface between the two domains and thereby conducts all interaction between the system and the environment. In both domains, the internal house keeping is performed by what we call internal communication. In this context, the term *communication* should be interpreted with a bit more abstraction since it covers both communication between internal components and the update and reading of internal (hidden) states. The common denominator is that this kind of communication is hidden and only visible from within the domain where it takes place. This is one the most important restriction that the work in [42] aims at enforcing by automation. We define A_{Int} to be the set of all arcs used for internal communication s.t.

$A_{Int} = \{(n_1, n_2) \in A' | \neg D(n_1, n_2)\}$

Legal Communication

After having identified three different types of communication (internal, through shared events and through shared states) it is now possible to sum up by identifying a basic rule over the set of arcs: $A' \backslash (A_{Int} \cup A_{Ext}) = \emptyset$, where A_{Ext} is the set of all arcs used for external communication s.t. $A_{Ext} = A_{SS} \cup A_{SE}$ This rule is satisfied when all arcs in the model can be categorized by the three categories of legal communication. If this is the case, the model is said to be *environment-and-system-partitioned.*

3.2.3 Implementing a Prototype

This section describes the development of an application that validates a CPN model with respect to the *environment-and-system-partitioned* property based on structural analysis. We will call this application the *validator*. One of the most commonly used tools for developing and analyzing CPN models is a graphical tool called CPN Tools [2]. We will call this tool the *editor*. This tool uses a XML file format for storing and reading the structure of models (nodes, arcs, declarations, graphical layout, inscriptions etc.). The overall purpose of the validator is to analyze such a file with respect to the guidelines defined in [42] and provide the user with information about the level of compliance with the guidelines for the given model. This process is a performed as a combination of manual and automatic operations. The model is defined in the editor as the user is used to doing it. This means that the user will not experience any changes to the way the editor is used. At any point during the specification of a model, it should be possible to execute the validator.

The response when an automated check of a model is performed is composed by a collection of error messages. In many situations, multiple error messages will occur in the response. When the checking of a model results in one or more error messages, this is a symptom of the existence of one or both of two possible problems: (1) the nodes of the model have been incorrectly categorized in the validator or, (2) the structure defined in the editor contains arcs that represent illegal communication. A prototype of the validator has been implemented and is presented in [42].

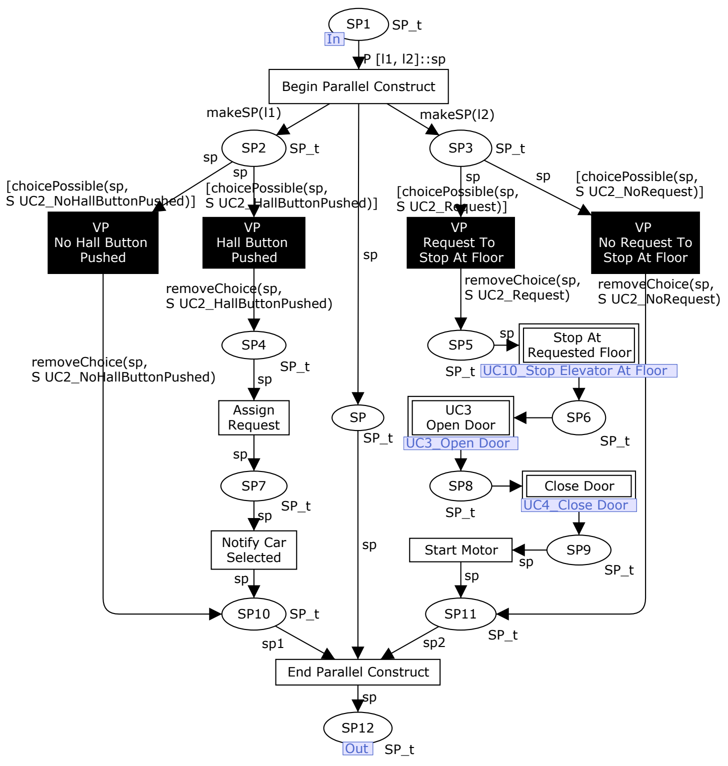
3.3 Replaying Scenarios with CPN Models of Requirements

Our initial work on transformation of use cases for reactive systems into CPN models is documented in a paper [14], which has been accepted for SCESM 2007. In this work, we focus on developing a general approach to the transformation as the first steps toward the design of a tool-based transformation. The basic idea is to automate part of the requirement analysis activity. The result of the transformation is a formally specified model representing the requirements that are informally expressed by means of use case descriptions and sequence diagrams. In this paper, we use the same elevator controller case study as described in 3.1.1. The transformation process is described in its details in [14] and in the rest of this section, we will give a brief overview of potential uses of the CPN model.

The work presented in the paper relates to the activities and products of Figure 2.1 in the same way as the paper described in Section 3.1 by augmenting the requirements specification activity and its out-coming product (the requirements specification) with formal modeling. Still, the two papers address the formalization of the requirements analysis activity differently. While the work presented in Section 3.1 is focused at the partitioning between interacting domains (controller and environment), the work, which will be presented here is focused at representing scenarios found in the informally specified requirements specification.

A CPN model generated by our approach can be executed in three different ways. Two of them are directly supported by CPN Tools [2] and are known as *interactive* and *automatic* simulation. During interactive simulation, the user selects which transitions should be fired, and thereby which execution path to follow. This selection is performed nondeterministically by the tool during automatic simulation. A third possibility is to specify a scenario as a sequence of choices which are determined in advance. This solution allows for some choices to be performed by the user or nondeterministically by the tool, while others are fixed. We call this *scenario simulation.*

Figure 3.9 shows how one of the use cases has been modeled in the CPN as a result of the transformation from a sequence diagram. This is a module in a hierarchical model; it includes instances of use cases at lower levels in the hierarchy specified by the use case description and it itself is included in higher-level use cases. The inclusion of use cases is represented by substitution transitions in the CPN model. The VP annotations in the four dark transitions is an acronym for *variation point.* In the specific case, the variation points form two choices (if a hall button is pushed or not and if a request exists or not). Variation points are identified during the transformation of sequence diagrams into the CPN model and serve as an important property when the CPN model is used for simulating scenarios. A scenario can be seen as a sequence of choices made at the variation points. When the CPN model is used for scenario simulation, the scenario to simulate is expressed as a list of choices. This list is carried in a special token, which is moved around in the model and used for evaluating the enabling of the variation point transitions.



SP1
SP_t
In
P [l1, l2]::sp
Begin Parallel Construct
makeSP(l1)
makeSP(l2)
SP2
SP_t
SP3
SP_t
sp
[choicePossible(sp, S UC2_NoHallButtonPushed)]
[choicePossible(sp, S UC2_HallButtonPushed)]
[choicePossible(sp, S UC2_Request)]
[choicePossible(sp, S UC2_NoRequest)]
VP No Hall Button Pushed
VP Hall Button Pushed
VP Request To Stop At Floor
VP No Request To Stop At Floor
removeChoice(sp, S UC2_HallButtonPushed)
removeChoice(sp, S UC2_Request)
removeChoice(sp, S UC2_NoRequest)
removeChoice(sp, S UC2_NoHallButtonPushed)
SP4
SP5
Stop At Requested Floor
UC10_Stop Elevator At Floor
Assign Request
SP
UC3 Open Door
UC3_Open Door
SP6
SP7
SP8
Close Door
UC4_Close Door
Notify Car Selected
Start Motor
SP9
SP10
SP11
sp1
sp2
End Parallel Construct
SP12
Out

Figure 3.9: The CPN module representing a variation of UC2

When a scenario is described as a sequence of choices, it is often beneficial to be able to adjust the level of restriction of these choices, i.e., to make some choices free, while fixing others. In the approach described so far, all choices are either free and made by the user during execution of the model, or predetermined by the sequence specified in the list of choices. To cover the gap between these two extrema, wildcards have been introduced to the specification of the scenarios. This is done by including the keywords `ANY` and `SEQ` as accepted elements in the sequences. These operators have simple semantics as described informally with the next examples. The `ANY` operator replaces a single choice. In the sequence `[a,ANY,b]`, `a` and `b` are fixed choices while the choice in-between these choices is performed freely. After one free choice, the ANY operator is consumed from the sequence, leaving only the `b` choice. The `SEQ` operator replaces a sequence of choices until the choice following the `SEQ` operator. In the sequence `[a,SEQ,b,c]`, `a` and `c` are fixed choices while any sequence of choices is allowed to be performed until a `b` choice is performed. At that point, the `SEQ` operator will be consumed from the sequence leaving a sequence containing only the `c` choice. With the sequence of choices and the wildcard solution, scenarios can be described with a varied level of restriction. Thereby it is possible to describe either specific scenarios or families of scenarios in which some variation points are common and others are different among the members of a given family. The following expression defines a scenario: `[S UC1, S SEQ, S UC2_Variation1, S SEQ, ...]`, where UC1 is selected. This is followed by an arbitrary sequence of choices until a specific variation for UC2 is chosen. This is followed by a new sequence of free choices and so on.

Chapter 4 -Design Verification with Models

In this chapter, we consider two examples of how a formal design specification may be used to gain knowledge about the expected properties of a potential design before it is constructed. In both cases, formal models represent focused areas of the design specifications and these models are executed in order to make it possible to evaluate the consequences of design choices based on simulation of typical scenarios.

4.1 Model-Based Analysis of Real-Time Critical Soft State Signaling

This Section describes work, which was initially part of my Master's Thesis [39] and later summarized and published as a paper [40] in DIPES 2006.

The work presented in the paper relates to the activities and products of Figure 2.1 in the following way: the work gives an example of how the product of the design activity (the design specification) may be expressed as a formal model, which can be used for early simulation-based analysis of certain properties of interest in the design before the construction activity leading to the implementation product is started.

In the specific case, the design specification of a communication component for a windmill controller has been modified in an attempt to overcome problems in an existing design and implementation. A CPN model is developed of the revised design for the communication component and this CPN model is used for model-based analysis of the design based on verification against informally specified non-functional requirements to the real-time performance of the system.

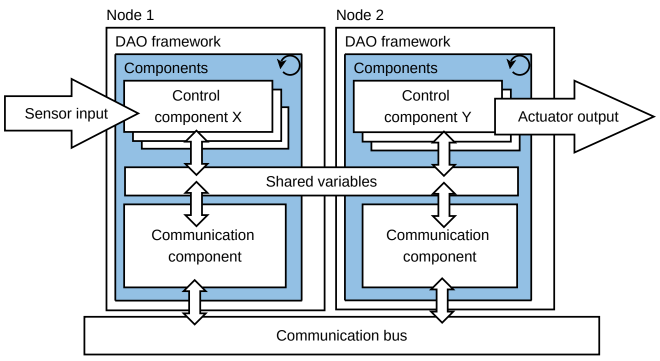


Figure 4.1: DAO

The communication component is part of a framework (Distributed and Active Objects - DAO) for execution of distributed control algorithms in a windmill. Figure 4.1 shows a conceptual overview of the DAO framework. A system consists of a number of *nodes* (embedded computers) connected by a communication bus (Ethernet). Each node houses an instance of DAO, which executes a group of (software) control components. A control component may be connected to physical sensors and actuators through which the operation of the windmill is monitored and controlled. A control component may also be tied together with other control components through *shared variables* to form distributed control algorithms across the nodes. To make this possible, the shared variables need to exist in updated copies in all relevant nodes. The task of updating these variables is handled by a single *communication component* in each node. This is a special component, which has access to both the variables and the communication bus. All components in each node (the control components and the communication component) are periodically activated as part of a repeated cycle. The communication component is activated during every cycle and the control components are activated with precision that varying periods (derived from the cycle period). The timeliness of the activation of components is important since it directly affects the level of precision observed in the output of the components. In order to be able to obtain timely activation, it is important to be able to have some assumptions about the maximum execution time of each component. This is relatively straight-forward with respect to the control components because of their pure transformational nature. It is a much harder problem wrt. the communication component. The existing design of the communication component is based on *change-triggered communication* - i.e. a broadcast message is sent by a communication component when the value of a *local variable* is changed by a local component. This message is received by the communication components in the other nodes, which will update their local copies of the variable in question in order to maintain an updated view of the collection of shared variables. Two severe problems have been revealed after the implementation of the system:

Change-burst ⇒ message-burst ⇒ unpredictability: If a lot of variables are modified within a short period of time, this will require the communication components to exchange a lot of messages. If too many messages are exchanged, this may not leave enough time for the timely activation of the control components.

Message-loss ⇒ inconsistency The update messages are exchanged through an unreliable protocol (UDP). If a message is lost, this will result in an inconsistent view of the variable that has been modified. The view will remain inconsistent until the variable is modified at a later point in time.

It has been proposed to implement a reliable multicast protocol to solve the second problem. This is not a feasible solution though, since it will make the effect of the first problem even worse (because it would require even more messages to be exchanged). Based on measurements, it is known that around 1 ms is used for sending and receiving update messages. At the same time, it has been determined that at most 20% of the cycle period, which is typically in the order of 20-50 ms may be used for communication. This limit is important in order to leave sufficient cycle time for the control components to operate properly. Another non-functional requirement determines that the maximum delay related to the update of the distributed view of a modified shared variable be 50 ms - i.e. when a variable is modified in one node, the new value should be reflected in the local copies of all relevant nodes within a period of at most 50 ms. A system typically contains 3-5 nodes and several thousands of variables.

4.1.1 Soft State Signaling

As an alternative to the existing change-triggered approach to communication, a time-triggered approach based on *Soft State signaling* is proposed in [40]. The proposed communication protocol is based on a generic Soft State principle [10]. Soft State messaging is a variant of protocols within the family of signaling protocols. This family of protocols spans within two generically defined poles; Soft State and Hard State protocols. Signaling protocols are applied in a wide range of applications. Common to those applications and the specific application in relation to DAO is the need to maintain a consistent view of a shared state - where the state could be a routing table, a list of shared files or a set of variables. Soft State protocols differ from Hard State protocols by the fact that they rely on the exchange of messages using best-effort-semantics as opposed to reliably exchanging messages. The difference is observable in the way, in which the distributed views of the state are updated across the network. In both cases, messages are sent from the sender, which is where the changing of the state occurs to a receiver (or a group of receivers), where the state is observed. Also, in both cases, the sender is the initiator of the message exchange. The difference here lies in which event triggers the sending of the message with the changed state from the sender to the receiver. In the case of the Hard State protocols, the triggering event is the change itself - i.e. when the sender detects a change in its local view of the state, this causes a message with the updated state to be sent to the receiver. On the other hand, in the case of the Soft State protocols, the triggering event is a temporal event caused by a timer running out. Each state is associated with an update timer that causes a periodic event trigger.

Based on the properties of the proposed communication protocol, it is considered a sane choice for replacing the existing protocol, because the proposal is suited for addressing the two main problems with the existing protocol (the two issues refer to the two problems described earlier):

1. The periods for sending messages are planned offline. This makes it possible to predict the maximum amount of messages being exchanged within an activation cycle and thereby predict processing overloads.

2. All messages are periodically retransmitted. This resolves the problem of potentially long periods of inconsistency in the case of lost messages.

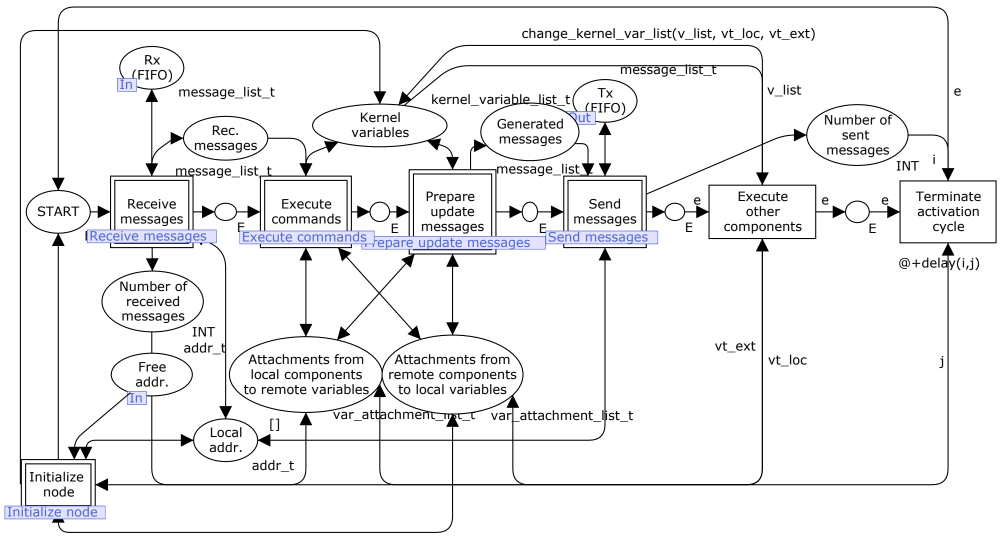


Figure 4.2: The behavioral specification of a node

4.1.2 Analysis of the Design Proposal

The DAO framework has been modeled in order to simulate and thereby evaluate the real-time performance of the proposed design of the communication component. This has resulted in a CPN model, which represents the entire system but is highly focused at the communication component. The control components are implicitly modeled at a high level of abstraction by random modifications of the shared variables while the communication components are modeled in more detail. Figure 4.2 shows a module from the hierarchical CPN model. This module represents the cyclic behavior of a DAO node. The model has been instrumented with real-time properties and is used for simulation execution during which log files are generated. These log files contain timed information about messages being exchanged combined with time stamped events representing modification of variables. The implications of the design proposal can be evaluated informally by visualization of the measurements in a standard spreadsheet application. Figure 4.3 shows an example of this. In this case, the maximum and average update delays are shown as a function of the percentage of the activation cycle used by the communication component. Measurements such as these can be collected for

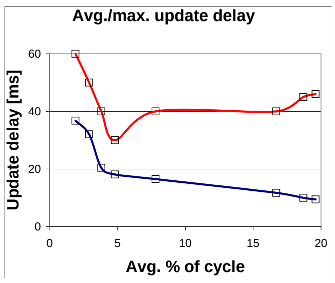


Figure 4.3: Measurements

varying settings of parameters, which are made adjustable in the CPN model. These parameters include the number of nodes, the probability of message loss, the activation period of the communication components etc. In this way, the model-based approach serves to deliver early estimates about the implications of design decisions.

4.2 Real-Time Logic for Analysis and Visualization of VDM++ Execution Traces

This section describes a technical report [16] in which an extension to VDM++ is proposed. The extension makes it possible to express non-functional requirements to the real-time behavior of a system represented by a formal design specification (a VDM++ model). The properties are expressed as assertions (logic predicates) that are used to evaluate designs based on execution traces (i.e. log files) obtained from execution of the formally expressed design specification. In this way, the extension provides a method for formally expressing (some classes of) non-functional requirements combined with tool support for the validation [1] of design specifications with respect to such requirements.

The work presented in the paper relates to the activities and products of Figure 2.1 in the following way: the language extension makes formal expression of non-functional requirements possible in the requirements specification product while the implemented tool support makes it possible to automate part of the design verification activity.

The technique is demonstrated on an example based on an in-car radio navigation system consisting of several software applications running on a common distributed hardware platform. Each application has individual requirements and the design challenge is to develop an architecture capable of satisfying all requirements. In developing such an architecture, the designer will want feedback on the system-level timing properties of a selected model. The model presented here reflects one of the proposals that was considered during the design, consisting of three processing units connected through an internal communication bus. We use the terms *application* and *task* to informally describe the case study. An overview is presented in Figure 4.4.

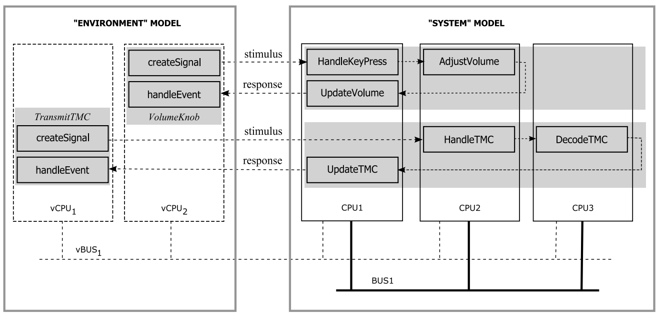


Figure 4.4: An overview of the car radio system

There are two applications: `ChangeVolume` and `ProcessTMC`, each consisting of three individual tasks. The `ChangeVolume` application, represented by the top right gray box, controls the radio volume. The task `HandleKeyPress` takes care of all user interface input handling, `AdjustVolume` modifies the volume accordingly and `UpdateVolume` displays the new volume setting on the screen. The `ProcessTMC` application, indicated by the bottom right gray box in Figure 4.4, handles all Traffic Message Channel (TMC) messages. TMC messages arrive at the `HandleTMC` task where they are checked and forwarded to the `DecodeTMC` task to be translated into human readable text which is displayed on the screen by the `UpdateTMC` task.

Two additional applications represent the environment of the system: `VolumeKnob` and `TransmitTMC`. The former simulates the behavior of a user turning the volume knob at a certain rate and the latter simulates the

[1] In this section, the term *validation* is used for describing non-exhaustive verification. The terminology is adopted from [16]

behavior of a radio station that transmits TMC messages. Both applications inject stimuli into the system (`createSignal`) and observe the system response (`handleEvent`).

The model is executed and stimulated with predefined sequences of input events in order to simulate scenarios. The result of this kind of execution is the generation of execution traces containing detailed timestamped information about events and states changes that occurred during the execution. We define a simple language, which is used to specify non-functional requirements to the system as assertions about the timing properties of the behavior of the system. We call these assertions *validation conjectures*. Before we started this work, tool support was available for graphical visualization of execution traces by means message sequence charts. We have implemented tool support for (1) checking specific execution traces against a collection of validation conjectures and (2) visualizing potential conflicts graphically as an extension to the existing execution trace visualization. It is our hope that this approach will give important information during the design activities. The validation of execution traces does not replace formal verification: a failing conjecture is the symptom of a possible design defect but a passing conjecture is not a proof of correctness with respect to the real-time constraints.

4.2.1 A VDM++ Model of the Case Study

In the system model of our example, there are two independent applications that consist of three tasks each. Tasks can either be triggered by external stimuli or by receiving messages from other tasks. A task can also actively acquire or provide information by periodically checking for available data on an input source or delivering new data to an output source. All three notions of task activation are supported by our approach. Note that task activation by external stimuli can be used to model *interrupt handling*. The `HandleKeyPress` and `HandleTMC` tasks belong to this category. The other tasks in our system model are message triggered. We already used periodic task activation in the environment model (`createSignal`).

```
 class Radio
  values
    public MAX : nat = 20

  instance variables
    private volume : nat := 0

  operations
    async public AdjustVolumeUp: nat ==> ()
    AdjustVolumeUp ( pno) == ( if  volume<= MAX then ( volume :=  volume + 1;
      RadNavSys‘mmi.UpdateScreen(1, pno)));

    async public HandleTMC: nat ==> ()
    HandleTMC (pno) == RadNavSys‘navigation.DecodeTMC(pno);
    ...
end Radio
```

Figure 4.5: Part of the *Radio* class

Application tasks are modeled by asynchronous operations in VDM++. Figure 4.5 presents the definition of `AdjustVolumeUp`. and `HandleTMC`, which are grouped together in the `Radio` class.

4.2.2 Validation Conjectures

Validation conjectures describe the temporal relationships between system-level events that can be observed in an execution trace. An execution trace may be thought of as a finite sequence of records, one for each time unit. Each record contains a set of event names for the events that occurred at that time, and a snapshot of the values of the instance variables in the system model at that time. Events are simply temporal markers; they use up no system resources. Each event has a unique name and may occur many times in an execution trace. However, at any one time, there may be at most one occurrence of a given event. Two kinds of system-level event are detectable in an execution trace generated from a VDM++ model: *operation events* and *state transition events*. Operation events occur when operations are requested, activated, or terminated (denoted `#req(Op)`, `#act(Op)` and `#fin(Op)` respectively). State transition events occur when a predicate over the instance variables of a model becomes true. Validation conjectures are predicates over execution traces. We will write $\mathcal{O}(e, i, t)$ to indicate that the ith occurrence of event e takes place at time t. The variable i ranges over the non-zero natural numbers $\aleph_1$, and t ranges over the indices of the trace. For example, the simple conjecture $\mathcal{O}(\texttt{\#fin(MMI`UpdateScreen)}, 1, 50)$ is true in a trace where the first occurrence of the event marking the termination of the `UpdateScreen` operation is at exactly time unit 50. Note that distinct occurrences of an event must happen at different times and that the occurrence numbers increase incrementally over time. The relation $\mathcal{O}$ is similar to the occurrence relation Θ in Real-Time Logic (RTL) [22], but we do not claim here to be using full RTL. The formal definition of conjecture evaluation is given in VDM-SL for uniformity with the tools framework [16].

It is often necessary to check a conjecture that relates to the specific values of some instance variables. For example, a designer may wish to check that a variable reaches a certain value at a specified time. In order to do this conveniently, we introduce the notion of a state predicate. A state predicate is a predicate over the instance variables of the system model. We will write $\mathcal{E}(p, t)$ to mean that the state predicate p is true of the variables in the execution trace at time t. In stating a conjecture, it may be necessary to mark the times at which a predicate becomes true. In order to support this, we introduce the notion of a *state transition event* which contains a predicate and which occurs at any time when the predicate becomes true.

It would be possible to construct validation conjectures just using the concepts defined so far. However, in order to promote ease of use, several higher level validation conjecture patterns are defined in order to allow common forms of conjecture to be expressed or composed. Three simple forms of conjecture are currently supported: *separations*, *"required" separations* and *deadlines*. Intuitively, separation conjectures describe a minimum separation between specified events, should the events occur. Required separations are separations in which the second event is required to occur at or after the minimum separation. Deadline conjectures state that the second event must occur before a deadline is reached after the occurrence of the first event. These three simple forms are not intended to be exhaustive, but could provide a basis for a language of more sophisticated conjectures. We will now look at how separation conjectures are defined. A separation conjecture is a 5-tuple $Separate(e_1, c, e_2, d, m)$ where e_1 and e_2 are the names of events, c is a state predicate, d is the minimum acceptable delay between an occurrence of e_1 and any following occurrence of e_2 provided that c evaluates to true at the occurrence time of e_1. If c evaluates to false when e_1 occurs, the validation conjecture holds independently of the occurrence time of e_2. The Boolean flag m is called the *match flag*, when set to true, indicates a requirement that the occurrence numbers of e_1 and e_2 should be equal. This allows the designer to record conjectures that describe some coordination between events. For example, we may wish to state that a stimulus and response events occur together in pairs within some time bounds, so the ith occurrence of the stimulus is always followed by the ith occurrence of the response.

A validation conjecture $Separate(e_1, c, e_2, d, m)$ evaluates true over an execution trace if and only if:

$\forall i_1, t_1 \cdot \mathcal{O}(e_1, i_1, t_1) \wedge \mathcal{E}(c, t_1) \Rightarrow$

$\neg\exists i_2, t_2 \cdot \mathcal{O}(e_2, i_2, t_2) \wedge t_1 \leq t_2 < t_1 + d \wedge (m \Rightarrow i_1 = i_2) \wedge (e_1 = e_2 \Rightarrow i_2 = i_1 + 1)$

The *required separation* conjecture is similar to the separation conjecture but additionally requires that the e_2 event does occur. A conjecture $SepRequire(e_1, c, e_2, d, m)$ evaluates to true over an execution trace if and only if:

$\forall i_1, t_1 \cdot \mathcal{O}(e_1, i_1, t_1) \wedge \mathcal{E}(c, t_1) \Rightarrow$

$\neg\exists i_2, t_2 \cdot \mathcal{O}(e_2, i_2, t_2) \wedge t_1 \leq t_2 < t_1 + d \wedge (m \Rightarrow i_1 = i_2) \wedge (e_1 = e_2 \Rightarrow i_2 = i_1 + 1) \wedge$

$\exists i_3, t_3 \cdot \mathcal{O}(e_2, i_3, t_3) \wedge (m \Rightarrow i_1 = i_3 \wedge (e_1 = e_2 \Rightarrow i_3 = i_1 + 1)$

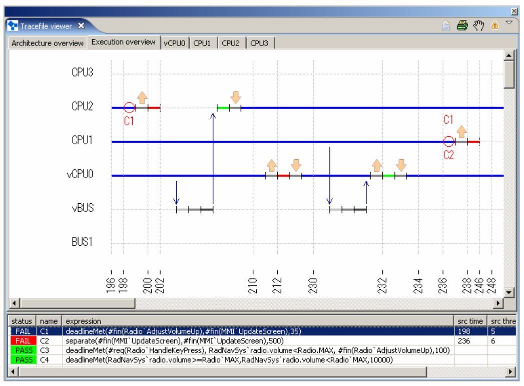
The *Deadline* conjecture places a maximum delay on the occurrence of the reaction event. Again, the *match* option may be used to link the occurrence numbers of the stimulus and reaction events. A validation conjecture $DeadlineMet(e_1, c, e_2, d, m)$ consists of a stimulus event, condition and reaction event; if c holds, d is the maximum tolerable delay between stimulus (e_1) and reaction (e_2). The conjecture evaluates true over an execution trace if and only if:

$\forall i_1, t_1 \cdot \mathcal{O}(e_1, i_1, t_1) \wedge \mathcal{E}(c, t_1) \Rightarrow$

$\exists i_2, t_2 \cdot \mathcal{O}(e_2, i_2, t_2) \wedge t_1 \leq t_2 \leq t_1 + d \wedge (m \Rightarrow i_1 = i_2) \wedge (e_1 = e_2 \Rightarrow i_2 = i_1 + 1)$

These basic forms of validation conjecture might be used to build up a more sophisticated language. For example, a conjecture to validate the periodic character of an event might take the form $Periodic(e, p, j)$ where e is the periodic event, p is the period and j the allowable jitter. The conjecture might be defined to be true in a given execution trace if and only if: $DeadlineMet(e, true, e, p + j, false) \wedge Separate(e, true, e, p - j, false)$ evaluates to true over the same trace.

We now have a look at an examples of a validation conjectures that is used to gain confidence in the quality of the design specification before the actual construction activity is initiated. More examples can be found in [16]. In the example, we want to require that when the volume is changed, this should be reflected by an update to the display no later than 35 ms after the button push. This requirement is expressed by the following validation conjecture in concrete VDM++ syntax: `deadlineMet(#fin(Radio‘AdjustVolumeUp), #fin(MMI‘UpdateScreen),35)`
where the first and the second parameters represent the stimulus and the reaction respectively.



status	name	expression	src time	src thre
FAIL	C1	deadlineMet(#fin(Radio`AdjustVolumeUp),#fin(MMI`UpdateScreen),35)	198	5
🟥 FAIL	C2	separate(#fin(MMI`UpdateScreen),#fin(MMI`UpdateScreen),500)	236	6
🟩 PASS	C3	deadlineMet(#req(Radio`HandleKeyPress), RadNavSys`radio.volume<Radio.MAX, #fin(Radio`AdjustVolumeUp),100)		
🟩 PASS	C4	deadlineMet(RadNavSys`radio.volume>=Radio`MAX,RadNavSys`radio.volume<Radio`MAX,10000)		

Figure 4.6: Trace view showing conjecture violations

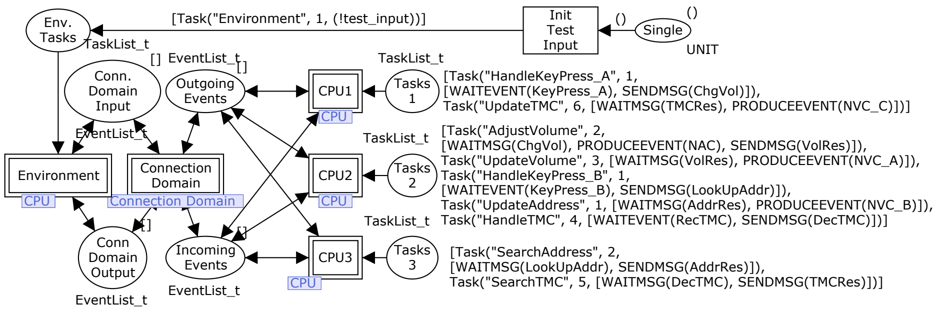
When validation conjectures have been expressed, they may be used for analysis of execution traces in the tool that has been extended as part of the work presented in [16]. Figure 4.6 shows an example of the graphical visualization. In this case, four validation conjectures are evaluated. C3 and C4 pass with respect to the specific execution trace while C1 and C2 fail. The text in the bottom part shows the validation conjectures express by means of the concrete VDM++ syntax. The circles in the graphical part of the window point out violations of the validation conjectures.

Chapter 5 -Testing with Models

This section very briefly introduces a paper [41], which was presented at MOTES 2006. The paper documents a very initial approach to model-based testing with CPN models of reactive systems.

The work presented in the paper relates to the activities and products of Figure 2.1 in the following way: the requirements analysis activity leads to an agreement about the expected behavior of the system under development. This agreement is represented in some form in the requirements specification product. The paper presented here is aimed at evaluating the final implementation of the system with respect to these requirements about behavior.

The approach is explained by use of the in-car radio navigation system described in Section 4.2. Figure 5.1 shows the topmost module of the CPN model. A representation is connected to three CPUs by an interface formed by places holding in- and out-going events. Each CPU is represented by an instance of the CPU module. Each instance is configured with a set of tasks according to the case study description. As a an example, `CPU1` is configured to execute two tasks: `HandleKeyPress_A` and `UpdateTMC`. Each task is defined by three parameters: a name, a priority (an integer), and a sequence of operations. The operations are high-level abstractions of atomic behavior found in the system and the environment. These operations including sending/receiving messages and generating/waiting for events. Informally explained, the `UpdateTMC` is specified to wait for a message (`TMCRes`) and react by producing an event (`NVC_C`) as a response to receiving the message. When the end of a sequence of operations is reached, the sequence is restarted - i.e. the tasks run as endless loops unless explicitly stopped.



```
[Task("HandleKeyPress_A", 1,
[WAITEVENT(KeyPress_A), SENDMSG(ChgVol)]),
Task("UpdateTMC", 6, [WAITMSG(TMCRes), PRODUCEEVENT(NVC_C)])]

[Task("AdjustVolume", 2,
[WAITMSG(ChgVol), PRODUCEEVENT(NAC), SENDMSG(VolRes)]),
Task("UpdateVolume", 3, [WAITMSG(VolRes), PRODUCEEVENT(NVC_A)]),
Task("HandleKeyPress_B", 1,
[WAITEVENT(KeyPress_B), SENDMSG(LookUpAddr)]),
Task("UpdateAddress", 1, [WAITMSG(AddrRes), PRODUCEEVENT(NVC_B)]),
Task("HandleTMC", 4, [WAITEVENT(RecTMC), SENDMSG(DecTMC)])]

[Task("SearchAddress", 2,
[WAITMSG(LookUpAddr), SENDMSG(AddrRes)]),
Task("SearchTMC", 5, [WAITMSG(DecTMC), SENDMSG(TMCRes)])]
```

Figure 5.1: The *Top* module

The CPN module shown in 5.2 specifies the dynamic behavior of the CPU module and thereby the semantics for execution of the tasks. The structure is a high-level representation of a typical operating system [18] in which tasks have priorities and are preemptively scheduled.

The CPN model is used in a simplistic approach to model-based testing, which is based on parameterized state space generation in order to obtain collections of legal execution traces (sequences of input and output events). The parameter in this context is a sequence of stimuli given to the system (the `testinput` parameter seen in

Figure 5.1). By state space generation, it is possible to collect the set of all legal sequences of events observed in the connection domain as a result of a given sequence of input events.

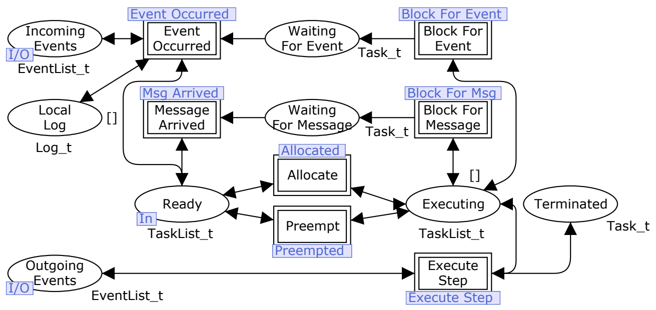


Figure 5.2: The *CPU* module

The main draw-back of this approach is caused by the well-known state explosion phenomenon which causes the complexity of the state space to grow rapidly as the length of the sequence of stimuli is increased. An appropriate alternative would be on-line model-based testing [30; 43]. In this approach, the model is executed in parallel with the implementation. The output from the implementation is evaluated dynamically thus making the generation of a large amount of test cases unnecessary.

Chapter 6 -Future Work and Conclusions

This progress report will now be concluded by an outlook based on an introduction to potential areas of work during the second half of my Ph.D. studies and some concluding remarks. I will briefly introduce some potential areas of future work. In some of these areas, work has already been initiated while other areas are merely proposed because I believe some interesting potential for future work may be found there.

6.1 Directions for Future Work

I would find it interesting to work in some of the following areas:

On-line Model-Based Testing with Coloured Petri Nets

As suggested in Section 5, it would be interesting to investigate methods for performing on-line model-based testing of reactive systems with CPN models. The basic idea of this approach would be to dynamically generate small parts of the state space for a model representing both the environment and the system being tested. In this context, the separation between the two domains in the model is important and thus the work will depend on the definitions of this partitioning as described above. The partitioning is important because it will be necessary to distinguish observable events and states in the system from internal (hidden) ones. Tool-based on-line testing will be based on an algorithm, which is able to validate the correctness of behavior of the system being tested and generate new stimuli for the event at the same time. The stimuli will be generated for both the part of the model that represents the system and for the actual implementation. When the implementation reacts to the stimuli, a part of the state space will be calculated based on the current (observable) state of the model (which is expected to reflect the state of the implementation). The result of this calculation is a set of next possible states (in the case of non-deterministic systems). If this set is empty, the reaction from the model is considered erroneous.

Semi-Automated Generation of Code for Reactive Systems Based on Coloured Petri Net Models

One of the activities that have not been covered by my work during the first half of the Ph.D. is the construction activity. When this activity is part of a development process augmented by formal methods, it is important that the construction activity produces an implementation, which is a consistent representation of the design specification. If this is not the case, all effort of analysis and verification will be a waste of work. An obvious approach to raising the level of consistency at this point is to automatically (or systematically) generate part of the implementation based the design specification. This is far from a new area of research, but it would be interesting to look into how CPN one may take advantage of the system-and-environment-partitioned modeling approach as a foundation for generating skeletons of code for reactive/embedded systems. I am currently taking the first steps of exploration in this field with a fellow Ph.D. student who has work in the field of model transformation.

Language Integration of Real-Time Logic Assertions in VDM++

Section 4.2 describes an approach to analysis of execution traces generated through execution of VDM++ models of reactive systems. We are currently discussing future possibilities for this work and we have decided to look into how the assertions about real-time properties can be integrated directly into the VDM++ language. The language already contains extensive support of expression of assertions about the states of models through preconditions, postconditions, and invariants that are used internally in classes. We would like to develop a natural and integrated method for expressing real-time assertions as part of these assertions. The purpose of this would be to make it possible to dynamically evaluate the assertions *while* executing the model rather than based on execution traces that are analyzed *after* the execution of the model. This would make it possible to interpret real-time failures as runtime failures as is already done for failures relating to functional requirements.

Monitoring Execution of Coloured Petri Net Models with Real-Time Logic Assertions

It would be very interesting to investigate how the dynamic approach to validation of real-time requirements may be adopted to monitor the execution of CPN models. Since tool support is being developed for VDM++, a feasible approach might be to develop a consistent mapping between CPN log files, which are generated automatically by CPN Tools and VDM execution traces. In this case, assertions about real-time behavior would be expressed by means of the high-level constructs we have developed for VDM++, an automated translation of the log file is applied, and the existing tool-based analysis is performed. The translation process is not trivial since it is necessary to develop a method that maintains a consistent link between the existing CPN model and the VDM++ representation even though the two languages are of very different natures. This consistency is important in order to obtain understandable results from the VDM++-based analysis.

Formalization of the System-and-Environment-Partitioned Property for Hierarchical Coloured Petri Nets

Section 3.2 introduces an approach to partitioning the system and environment domains in CPN models of reactive systems. As described, the definition is the initial work on the formal definition of the partitioning. A natural next step is to extend the definitions to cover hierarchical CPN models. The modeling approach used for the model of the elevator controller system described in 3.1 gives a practical demonstration of how the partitioning approach can be applied to hierarchical models. This gives some evidence of the feasibility of extending the definitions for structural analysis of CPN model to cover hierarchical models. The extension will also include an extension of the prototypical implementation of the validator tools. This work is important since it will hopefully serve as part of the foundation for other possible directions for the future work as described below.

6.2 Concluding Remarks

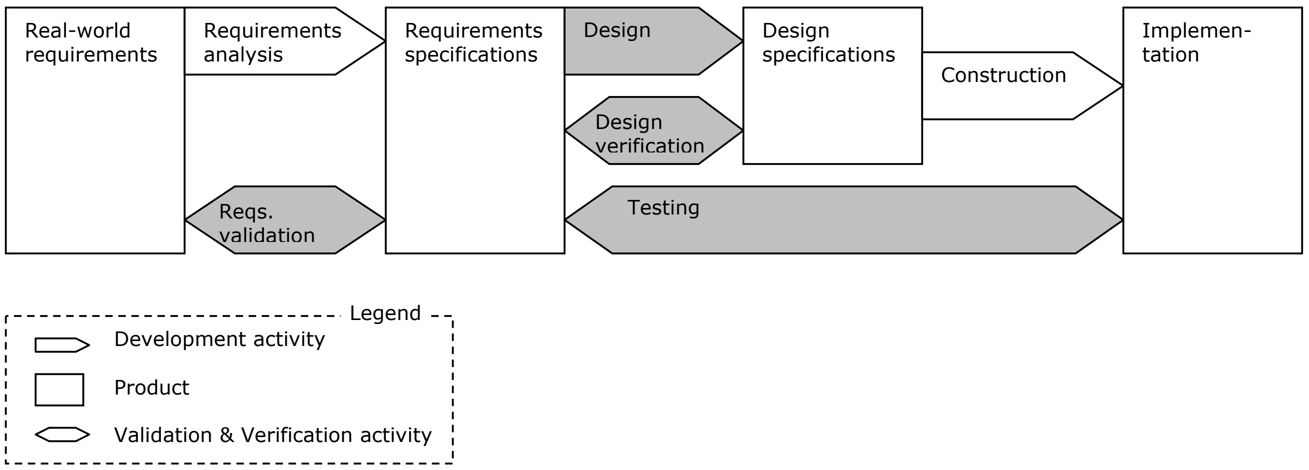


Figure 6.1: The generic process revisited. My work has touched the gray activities.

This report has introduced the basic activities and products of a generic software development process. It has been described how the process may be augmented by the introduction of formal methods at various points. Concrete examples of work within this area has been given through the description of six paper that have been produced during the first half of my Ph.D. studies. Figure 6.1 provides a graphical overview of the activities that have been touched by these six papers based on the generic software process introduced in the beginning of this report.

I have described the work that I have done so far and given proposals for potential future areas of work. Most of these areas are closely related to the work I have done so far. They are not focused on one specific activity in the development process and this corresponds with my general interest in investigating how formal modeling applies throughout the entire software life cycle. Still, it is evident - both in the proposed topics for future work and in the topics to which the work that has already been done relates - that my work has and will be focused on the light-weight approaches to formal methods. I aim to maintain that focus.

Bibliography

[1] CPN ML. http://wiki.daimi.au.dk/cpntoolshelp/cpn_ml.wiki. 8

[2] CPN Tools. http://wiki.daimi.au.dk/cpntools/cpntools.wiki. 17, 18

[3] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. *SWEBOK, Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, California, USA, May 2004. `http://www.swebok.org/`. 4

[4] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, 1982. 6

[5] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, New York, NY, USA, 1998.

[6] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995. 4

[7] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods ...ten years later. *Computer*, 39(1):40–48, 2006. 4

[8] Michael Breen. Experience of using a lightweight formal specification method for a commercial embedded system product line. *Requirements Engineering*, 10(2), 2005. 4

[9] Jean-Yves Brunel, Marco Di Natale, Alberto Ferrari, Paolo Giusto, and Luciano Lavagno. Softcontract: an assertion-based software development process that enables design-by-contract. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10358, Washington, DC, USA, 2004. IEEE Computer Society. 7

[10] David D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM*, pages 106–114, Stanford, CA, August 1988. ACM. 21

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000. 7

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press. 7

[13] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. pages 86–107, 2002.

[14] João Miguel Fernandes, Jens Bæk Jørgensen, and Simon Tjell. Requirements engineering for reactive systems: Coloured petri nets for an elevator controller. Technical report, 2007. 1, 18

[15] João Miguel Fernandes, Simon Tjell, Jens Bæk Jørgensen, and Óscar Ribeiro. Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. Technical report, 2007. Accepted for SCESM 2007. 1, 9, 13

[16] John Fitzgerald, Peter Gorm Larsen, Simon Tjell, and Marcel Verhoef. Validation support for distributed real-time embedded systems in vdm++. Technical Report CS-TR-1017, Newcastle University, School of Computing Science, May 2007. 1, 23, 25, 26

[17] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. Softw. Eng.*, 17(5):454–466, 1991. 6

[18] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with Uml*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 27

[19] Hassan Gomaa and Douglas B.H. Scott. Prototyping as a tool in the specification of user requirements. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 333–342, Piscataway, NJ, USA, 1981. IEEE Press. 5

[20] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, 2000. 13

[21] M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001. 9, 11

[22] Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.*, 12(9):890–904, 1986. 25

[23] K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1992. 14

[24] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992. 8

[25] Cliff B. Jones, Daniel Jackson, and Jeannette Wing. Formal methods light. *Computer*, 29(4):20–22, 1996. 5

[26] John C. Knight, Colleen L. DeJong, Mathew S. Gibble, and Luís G. Nakano. Why are formal methods not USED more widely? In C. Michael Holloway and Kelly J. Hayhurst, editors, *Fourth NASA Langley Formal Methods Workshop*, number 3356, pages 1–12, Hampton, Viginia, 1997. 4

[27] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[28] Jeff Magee, Nat Pryce, Dimitra Giannakopoulou, and Jeff Kramer. Graphical animation of behavior models. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 499–508, New York, NY, USA, 2000. ACM Press. 6

[29] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[30] Mikucionis, Larsen, and Nielsen. Online on-the-fly testing of real-time systems. Technical Report RS-03-49, 2003.

[31] R. Milner, R. Harper, and M. Tofte. *The Definition of Standard ML*. MIT Press, 1990. 8

[32] Aloysius K. Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 252, Washington, DC, USA, 1997. IEEE Computer Society. 7

[33] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press. 5

[34] Dennis K. Peters and David L. Parnas. Requirements-based monitors for real-time systems. *SIGSOFT Softw. Eng. Notes*, 25(5):77–85, 2000. 7

[35] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real: The inhouse card case study. *Int. J. Softw. Tools Technol. Transf.*, 5(2):140–157, 2004. 7

[36] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–15, 2003. 7

[37] Ian Sommerville. *Software Engineering*. Addison Wesley, 2007. 1

[38] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[39] Simon Tjell. Modeling and analysis of a communication protocol for windmills (danish only). Master's thesis, University of Aarhus, 2005. (http://daimi.au.dk/˜tjell/thesis.pdf). 20

[40] Simon Tjell. Model-based analysis of a windmill communication system. In *Proc. 5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, Braga, Portugal, Oct. 2006. 1, 20, 21

[41] Simon Tjell. Model-based testing of a reactive system with coloured petri nets. In Christian Hochberger and Rüdiger Liskowsky, editors, *Proceedings of INFORMATIK 2006*, volume 94 of *Lecture Notes in Informatics (LNI)*, pages 274–281, Bonn, Germany, 2006. Gesellschaft für Informatik. 1, 27

[42] Simon Tjell. Distinguishing environment and system in coloured petri net models of reactive systems. Technical report, 2007. Accepted for SIES 2007. 1, 13, 14, 17, 18

[43] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[44] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press. 6

[45] Margaret M. West and Barry M. Eaglestone. Software development: two approaches to animation of z specifications using prolog. *Software Engineering Journal*, 7(4):264–276, 1992. 6

[46] R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003. 9

[47] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990. 5, 6