Abstract

EMBEDDED SYSTEMS are becoming increasingly common in our everyday lives. As technology progresses, these systems become more and more complex. Designers handle this increasing complexity by reusing existing components. At the same time, the systems must fulfill strict functional and non-functional requirements.

This thesis presents novel and efficient techniques for the verification of component-based embedded system designs. As a common basis, these techniques have been developed using a Petri net based modelling approach, called PRES+.

Two complementary problems are addressed: component verification and integration verification. With component verification the providers verify their components so that they function correctly if given inputs conforming to the assumptions imposed by the components on their environment.

Two techniques for component verification are proposed in the thesis. The first technique enables formal verification of SystemC designs by translating them into the PRES+ representation. The second technique involves a simulation based approach into which formal methods are injected to boost verification efficiency.

Provided that each individual component is verified and is guaranteed to function correctly, the components are interconnected to form a complete system. What remains to be verified is the interface logic, also called glue logic, and the interaction between components.

Each glue logic and interface cannot be verified in isolation. It must be put into the context in which it is supposed to work. An appropriate *environment* must thus be derived from the components to which the glue logic is connected. This environment must capture the essential properties of the whole system with respect to the properties being verified. In this way, both the glue logic and the interaction of components through the glue logic are verified. The thesis presents algorithms for automatically creating such environments as well as the underlying theoretical framework and a step-by-step roadmap on how to apply these algorithms.

Experimental results have proven the efficiency of the proposed techniques and demonstrated that it is feasible to apply them on real-life examples.

This work has been supported by SSF (Swedish Foundation for Strategic Research) through the INTELECT and STRINGENT programmes, as well as by CUGS (National Computer Science Graduate School).

Acknowledgements

MANY PEOPLE HAVE EITHER directly or indirectly contributed to this thesis. I would here like to take the opportunity to thank them all.

I would first like to sincerely thank my suvervisors Professor Petru Eles and Professor Zebo Peng for their invaluable guidance and support during these years. I will particularly remember our fruitful and sometimes lively discussions at our regular meetings. They have hopefully taught me something about what it is to be a true researcher.

I would also like to thank all my other colleagues at the Department of computer and information science, and in particular at the Embedded systems laboratory, for the happy and joyful time we have spent together. In this context, I would like to include my fellow students and teachers at CUGS (National Computer Science Graduate School), I shall never forget the time we spent taking courses together at various kursgårdar away from civilisation - including the social activities in the evenings.

During the years, a number of master thesis students have implemented various parts of the techniques presented in the thesis. Thank you for your effort, you made my days easier with subsequent experimental work.

As my artistic talents leave a huge room for improvement, I am very grateful to Liana Pop for investing the effort in designing the nice cover of this thesis.

Last, but not the least, I would like to express my gratitude and appreciation to my family who is always there for me. In particular I would like to mention my beloved wife, Zhiping Wang, who always gives me more care than I deserve.

Linköping, April 2006

Daniel Karlsson

Contents

PART I
Preliminaries

Chapter 1
Introduction

VERIFICATION IS AN IMPORTANT aspect of embedded system development. This thesis addresses verification issues with a particular emphasis on component reuse. Although the thesis concentrates on formal verification, in particular model checking, it also covers issues related to simulation of component-based embedded systems.

This introductory chapter presents the motivation behind our work, problem formulation and contributions. In the end follows an overview of the thesis.

1.1 Motivation

Electronic devices increasingly penetrate and become part of our everyday lives. Such are, for instance, cell phones, PDAs, and portable music devices, such as Mp3-players. Moreover, other, traditionally mechanical, devices, such as cars, are becoming more and more computerised. The computer system inside this kind of devices is often referred to as an *embedded system*.

There does not exist a universal definition of an embedded system. However, there exists a certain consensus that the following features are common to most embedded systems [Cam96]:

- **They are part of a larger system (host system)**, hence the term *embedded*, with which they continuously or frequently interact. Usually, the embedded system serves as a control unit inside the host system.
- **They have a dedicated functionality** and are not intended to be reprogrammable by the end-users. Once an embedded system is built, its functionality does not change throughout its lifetime. For example, a device controlling the engine of a car will probably never be reprogrammed to decode Mp3s. A desktop computer, on the other hand, has a wide range of functionality, including web browsing, word processing, gaming, advanced scientific calculator, etc.
- **They have real-time behaviour.** The systems must, in general, respond to their environment in a timely manner.
- **They consist of both hardware and software components.** In order to cope with the wide and unpredictable range of applications, the hardware of a general purpose computer has to be generously designed with the risk of wasting resources. However, since the set of applications to be run on an embedded system is known at design-time, including their performance requirements, the hardware can be tuned at design-time for best performance at minimal cost. Similarly, software must also be optimised to build a globally efficient HW/SW system.

It is both very error-prone and time-consuming to design such complex systems. In addition, the complexity of today's designs (and the *manufacturing capability*) increases faster than what the designers can handle (*design productivity*). On top of this, the ability to verify the systems (*verification productivity*) increases even slower than the design productivity. Thus, proportionally more and more effort has to be put on verifying these complex systems [Ber05]. The difference between the manufacturing capability and design productivity is called the *design productivity gap* (or just *productivity gap*), and the difference between manufacturing capability and verification productivity is called *verification productivity gap* (see Figure 1.1).

In order to manage the design complexity and to decrease the development time, thereby reducing the design productivity gap, designers usually resort to reusing existing components (so called IP blocks) so that they do not have to develop certain functionality themselves from scratch. These components are either developed in-house, by the same company, or acquired from specialised IP vendors [Haa99], [Gaj00].

Not discovering a fault in the system in time can be very costly. Reusing predesigned IP blocks introduces the additional challenge that the exact behaviour of the block is unfamiliar to

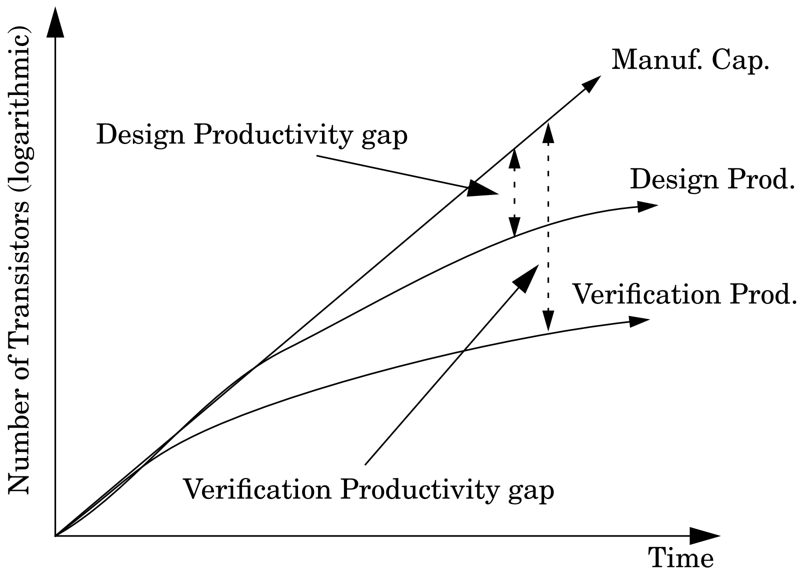


Figure 1.1: Productivity gap

the designer, for which reason design errors that are difficult to detect can easily occur. Discovering such faults only after the fabrication of the chip can easily cause unexpected costs of US$500K - $1M per fault [Sav00]. In many projects, the verification related activities may consume 50-70% of the total design effort [Dru03]. This suggests the importance of a structured design methodology with a formal design representation, and in particular it suggests the need for efficient verification. In highly safety-critical systems, such as aeroplanes or medical equipment, it is even more evident that errors are not tolerable since it is not only for economic reasons that they have to be considered, but also in order to avoid loss of human lives. In such cases, the use of formal methods is required.

Verification tools analyse the system model, captured in a particular design representation, to find out whether it satisfies certain properties. In this way, the verification tool can trap many design mistakes at early stages in the design, and thereby reduce cost significantly.

Increasing both the design and verification productivity are consequently very important. In this thesis, focus will be placed on the verification aspect with an emphasis on formal verification of component-based designs.

1.2 Problem formulation

The previous section stated that designers increasingly often build systems using reusable components due to the complexity of their designs. Therefore, there is an increasing need to efficiently and effectively verify such systems. Verification methodologies, in particular formal ones, which can effectively cope with this situation and take advantage of the component-based structure, need to be developed.

There are two aspects of this task:

- Verify that each component is correct
- Verify that the interconnection (integration) of components is correct.

This thesis solves problems related to both aspects. The following subsections will shortly present a few problems which have been addressed by this thesis.

1.2.1 COMPONENT VERIFICATION

In the case of component verification, the component itself is verified that it fulfils the specification with respect to its interface.

It is convenient for the designers to use the same language for simulation and synthesis as well as for formal verification. SystemC gains popularity partly due to its simulation and synthesis capabilities [Bai03]. However, formal verification techniques applied on SystemC designs are less developed, in particular concerning designs at levels above Register-Transfer Level (RTL). It is, therefore, important to develop techniques so that also designs at higher levels of abstraction can be formally verified.

It is sometimes the case that the component models are too big and complex to verify formally due to state space explosion. In such cases, designers normally resort to simulation. However, simulation only partially covers the total state space and potentially requires long time in order to obtain the appropriate degree of coverage. Injecting formal methods into the simulation process could lead to higher coverage and a shorter total validation time.

1.2.2 Integration Verification

It can often be assumed that the design of each individual component has been preverified [See02] and can be supposed to be correct. What furthermore has to be verified is the interface logic, also called glue logic, and the interaction between components [Alb01].

Each glue logic and interface cannot be verified in isolation. It must be put into the context in which it is supposed to work. An appropriate *environment* must thus be derived from the components to which the glue logic is connected. This environment must capture the essential properties of the whole system with respect to the properties being verified. In this way, both the glue logic and the interaction of components through the glue logic are verified.

1.3 Contributions

This thesis deals with issues related to verification of component-based embedded systems. The main contributions are summarised below:

Integration Verification

- **Theoretical framework.** A theoretical framework underlying the proposed integration verification methodology has been developed which is based on the notion of *stubs*, as an interface model of components. Theoretical results are used in order to improve the efficiency of the verification process [Kar02].
- **Automatic generation of stubs.** An algorithm which, given a model of a component, generates a stub, has been developed. The algorithm builds on the theoretical framework mentioned above. It furthermore removes the obliga-

tion of the IP provider to build appropriate stubs [Kar04a], [Kar04b].

- **Translation of logic formulas into the Petri-net based design representation.** In certain situations it is desired to incorporate logic formulas (other than those being verified) into the verification process, as assumptions about the rest of the system. In order to do so, they must be translated into the design representation used. An algorithm for doing this is proposed [Kar03].

Component Verification

- **Translation of SystemC into a Petri-net based design representation.** Translating SystemC into a well-defined design representation makes it possible to formally analyse and verify designs specified in SystemC. Given this translation, all other techniques discussed in the thesis can also be applied to designs formulated in SystemC [Kar06].
- **Formal method-aided simulation.** Sometimes, the models under verification are too big and complex to be successfully verified formally in a reasonable amount of time. In such cases, we propose a simulation methodology where model checking is invoked in order to improve coverage. The invocation of the model checker is controlled dynamically during verification in order to minimise total verification time [Kar05].

Although these items are contributions by themselves and presented in Part II (component verification) and Part III (integration verification) respectively, they can also be considered as part of one single proposed verification methodology. The components are first verified individually, to guarantee the correct behaviour for each one of them. As a second step, assuming the correctness of the reusable components, their interconnection (integration) is furthermore verified in order to guarantee the overall correctness of the system.

1.4 Thesis Overview

The thesis is divided into four parts. Part I introduces the area of embedded system design with focus on verification. It furthermore presents the background needed to understand the thesis and a high-level overview of the proposed methodology. Part II continues with presenting techniques which can be used for verification of reusable components. Part III introduces a formal verification process aimed at verifying the integration of component-based designs. Part IV concludes the thesis and points out a few areas for future work.

The four parts are, in turn, divided into twelve chapters as follows:

Part I: Preliminaries

- Chapter 1 shortly motivates the importance of the area of formal verification in a component-based context. It furthermore introduces the problems discussed as well as the structure of the thesis.
- Chapter 2 provides a more thorough background of the research area as well as related work.
- Chapter 3 addresses several concepts and definitions which are necessary for understanding the contents of this thesis.
- Chapter 4 presents a high-level overview of the verification methodology proposed in this thesis.

Part II: Component Verification

- Chapter 5 describes a translation mechanism from SystemC into the Petri-net based design representation which is used throughout the thesis.

- Chapter 6 discusses two methods in which components can be verified: formally (model checking) or by simulation. In the second case, emphasis is put on enhancing the coverage obtained from simulation by using formal methods.

Part III: Integration Verification

- Chapter 7 introduces the big picture in which context the chapters in this third part should be put. The main features of the proposed integration verification methodology are presented in this chapter.
- Chapter 8 presents the theoretical framework and the fundamental properties of stubs.
- Chapter 9 describes algorithms used for automatically generating stubs. Additional theory related to these algorithms is also given.
- Chapter 10 presents an algorithm for generating a Petri-net model which corresponds to a given temporal logic formula. The resulting model is able to produce all outputs consistent with the formula. Such models are useful when making assumptions about system properties.
- Chapter 11 illustrates the whole verification methodology by a case study, a mobile telephone design.

Part IV: Conclusions and Future Work

- Chapter 12 concludes the thesis and discusses possible directions for future work.

A summary of abbreviations and notations has also been included at the end of the thesis.

Chapter 2 Background

THE PURPOSE OF THIS CHAPTER is to introduce the context in which the work presented in this thesis belongs. First, a general system-level design flow is introduced. Aspects related to verification of IP blocks, from the perspective of both the IP provider and the IP user, are then presented. This is followed by a section introducing both simulation and formal verification. In the end, related work concerning verification of IP-based designs is presented.

2.1 Design of Embedded Systems

Designing an embedded system is a very complicated task. Therefore, in order to manage the complexity, it is necessary to break down this task into smaller subtasks. Figure 2.1 outlines a typical embedded systems design flow, with emphasis on the early stages from the system specification until the model where the system is mapped and scheduled (the part above the dashed line). This is the part of the design flow, the system-level, to which the work presented in this thesis belongs.

The input to the design process is a specification of the system, usually written in an informal language. The specification contains information about the system, such as its expected functionality, performance, cost, power consumption etc. It does not specify how the system should be built, but only what system to build [Kar01]. Given this document, the designer has to gradually transform, or refine, its contents into a finished product.

When an appropriate system model has been obtained [Var01], it must be validated to make sure that it really corresponds to the initial specification. That can be done either by simulation, formal verification or both.

Having obtained a system model, the designer must decide upon a good architecture for the system. This stage includes finding appropriate IP blocks in the library of components, for instance processors, buses, memories and application specific components, such as ASICs.

The next step is to determine which part of the design (as captured by the model) should be implemented on which processing element (processor, ASIC or bus). This step is called mapping.

If several processes are mapped onto the same processor, these processes need to be scheduled. Possible bus accesses and similar resource usage conflicts need either to be staticly scheduled or a dynamic conflict management mechanism has to be implemented. Constraints given in the original specification, e.g. response times, must be satisfied after scheduling. This must also be verified, either by simulation or formal verification.

Later stages of the design flow deal with synthesis of hardware and software components, as well as their communication, and fall out of the scope of system-level design.

If at a certain stage the designer finds out that an improper design decision was taken at an earlier stage, typically discovered in a verification phase, the design has to reiterate from a point where the problem can be fixed. Such iterations are very costly, especially if errors are detected at late design steps, e.g. at prototyping, when a physical model of the product has already

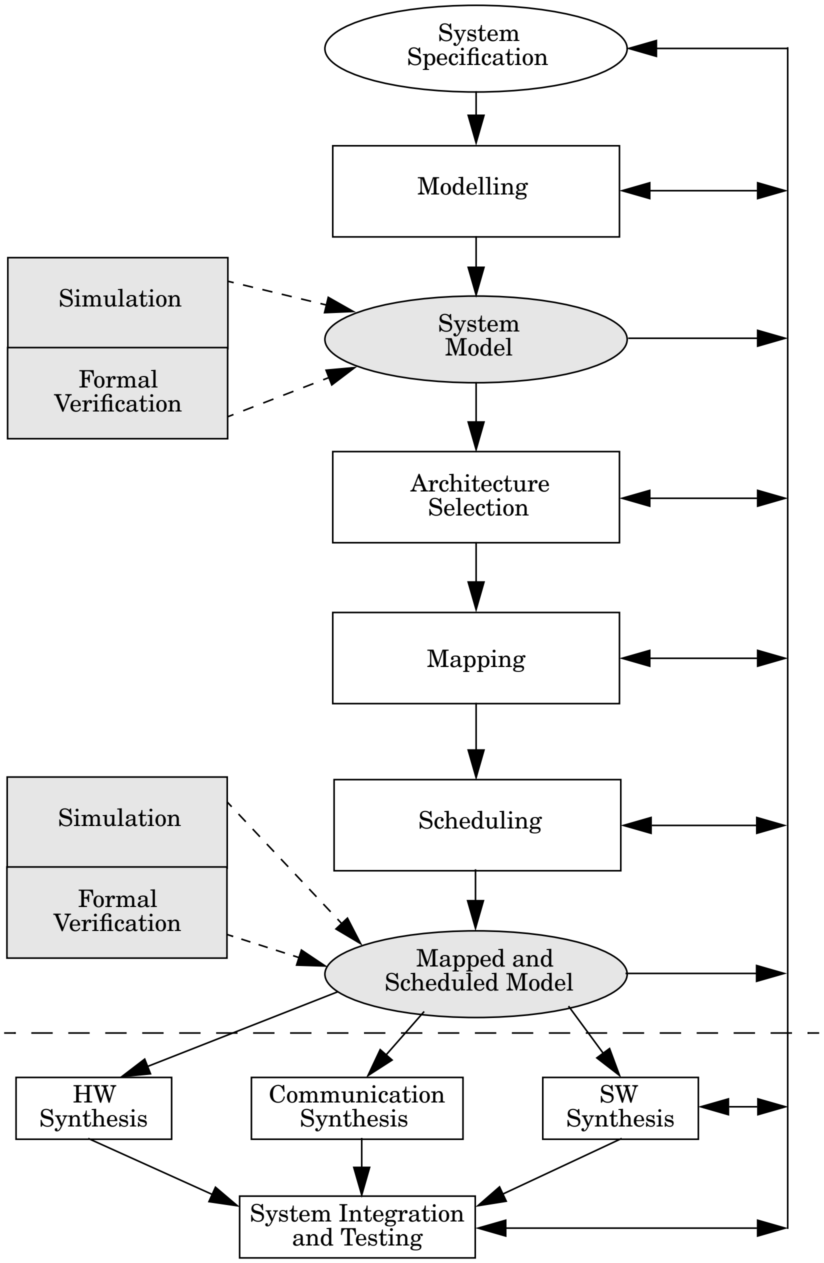


Figure 2.1: Embedded systems design flow

been built. Therefore, it is necessary, not to say crucial, to perform the validation steps, simulation and formal verification, in order to detect errors as early as possible in the design flow.

This thesis addresses the shadowed activities in Figure 2.1, i.e. verification, with emphasis on formal verification.

2.2 IP Reuse

By introducing reusable components, so called IP (intellectual property) blocks, several problems which would otherwise be absent, arise [Kea98], [Lo98]. On the other hand, using predesigned IP blocks is an efficient way for reducing design complexity and time-to-market [Gir93].

Developing a reusable IP block takes approximately 2.5 times more effort compared to developing the same functionality in a classical design [Haa99]. Therefore, the designer must think carefully, if it is worth this effort or not. Will the same functionality be used often enough in the future or in other designs? Does there already exist a suitable block developed by a third party? However, once the block is developed, the design time for future products is decreased significantly.

There are in principle two categories of actors in IP-based design: the IP provider and the IP user [Gaj00]. The following subsections describe problems, related to verification, faced by the two categories respectively.

2.2.1 IP Provider

The task of the IP provider is to develop new IP blocks. Anyone who has performed this task is an IP provider. It is not necessary that this person is someone in an external company, it might as well be the colleague in the office next door.

The first problem encountered by the IP provider is to define the exact functionality of the IP. As opposed to designing a specific system (without using IP), the IP provider must imagine every possible situation in which the IP block may be utilised, in order to maximise the number of users. At the same time, efficiency, verifiability, testability etc. must be kept at a reasonable level [Gaj00]. In general, as a block is made more and more general and includes more and more functionality, these parameters will suffer, as illustrated in Figure 2.2. At a certain point, if the IP is too general, it practically becomes useless.

The component must furthermore be verified thoroughly, considering all possible environments, conditions and situations in which the component might be used. The success of the IP block might critically depend on the effort put on verification.

In order to facilitate for the IP user to reduce the verification productivity gap, information which speeds up the verification effort also needs to be provided together with the IP block. Some elements used for verification of the IP might also be useful when verifying the system. Such elements could, for instance, be monitors, stimuli and response vectors and scripts [And02b]. In the case of formal verification, such information could be formal descriptions of the component and temporal logic formulas of assertions and assumptions.

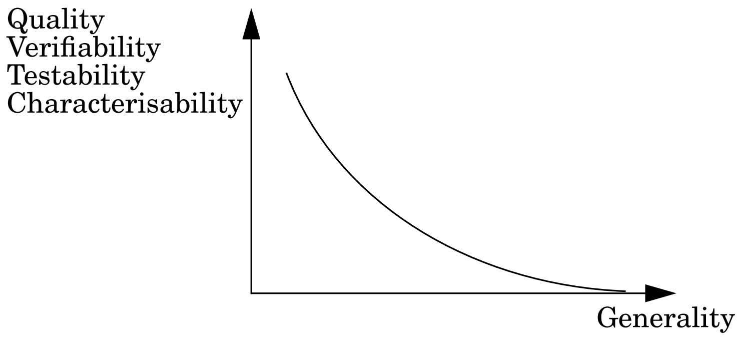


Figure 2.2: Impact of IP generality on various other parameters [Gaj00]

2.2.2 IP USER

The IP user is the person who uses the IP blocks designed by the IP provider. The main task of the IP user is to choose the appropriate blocks and to integrate them. The components may be designed by different providers, in which case their interfaces might not exactly match. Therefore glue logic has to be added between the components to adapt their interfaces in a way that they are able to properly communicate with each other. The glue logic is sometimes also called wrapper [Spi03]. Ideally, the components should be chosen in such a way that the size of the glue logic is minimised. This process of inserting glue logics for interconnecting components is called *integration*.

Keeping the model small facilitates verification, both by simulation and formal verification. Besides trying to find as compatible components as possible in order to keep the glue logic small, it is also favourable to find small such components. The components must provide the requested functionality, but contain as little extra functionality, that will not be used in the design, as possible. The extra functionality will only contribute to the already big verification complexity. This aspect should be contrasted with the goals of the IP provider, who would like to make the component as general as possible in order to maximise the number of potential users.

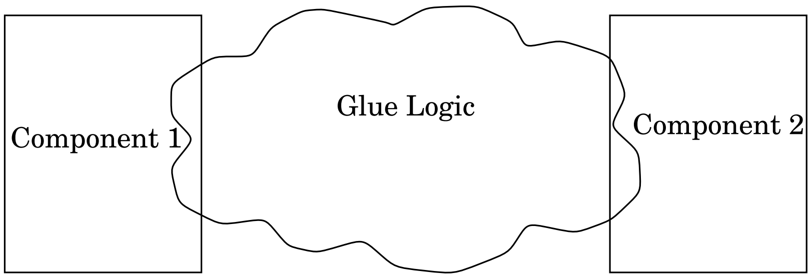


Figure 2.3: Two components interconnected by a glue logic

2.3 Verification

The goal of verification is to find discrepancies between the designer's intent and the implementation (possibly a model) of the design. In order to accomplish that, the designer's intent must be documented in a written specification. Verification then compares the specification with the implementation of the design. Since there might be a discrepancy between the specification and the designer's intent, the result from verification does not necessarily reflect exactly what the designer might think. It is important that designers are aware of this fact. An illustration over this situation is shown in Figure 2.4 [Piz04].

The figure shows three circles representing design intent, specification and implementation respectively. In the ideal case, there should be a complete overlap of these circles. The design intent should be equal to the specification, which in turn should be equal to the final implementation. However, in practice this is rarely the case. There is always a discrepancy between the design intent and the specification. It is very difficult to specify exactly every aspect of the system, and to do it in such a way that the message is correctly conveyed to implementors. Furthermore, the design intent only exists in the minds of the designers, or, even worse, in the minds of the customers or marketing people. Several designers might have different concepts and understanding about the same system, a fact which might

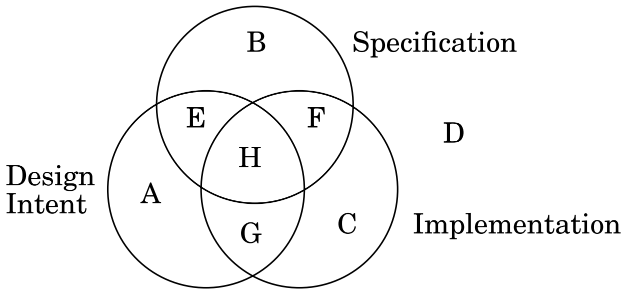


Figure 2.4: Verification Intent Overview [Piz04]

influence the resulting specification. There will, consequently, always (except for very trivial systems) be some parts of the intended system which are never specified and implemented (area A). Other unintended parts are specified but, luckily, not implemented (B), whereas yet other unintended and unspecified parts were implemented (C). These aspects can be furthermore combined, i.e. intended and specified behaviour does not end up in the implementation (E) or that unintended behaviour was specified and implemented (F).

As mentioned previously, the final aim of verification is to ensure that area H is as big as possible, while minimising the other areas. It should be remembered that verification is a comparative technique. If the specification with which the implementation is compared has flaws, then so does the results of the verification. The results of verification techniques cannot have higher quality than that of the specification.

There exist two types of verification techniques: formal and informal. Formal verification techniques search exhaustively, but intelligently, the state space of the designed system. This means that all possible computation paths will be checked. Formal verification is generally based on mathematical (logical) models, methods and theorems. Several techniques exist, such as language containment, model checking, equivalence checking, symbolic simulation and theorem proving [Swa97]. This section will give a quick overview of three of them: model checking, equivalence checking and theorem proving.

The informal verification techniques of interest in our context are based, in principle, on simulation. The main difference to formal verification is that informal techniques only search a limited part of the total state space. They can therefore not *guarantee* correctness of the system, only falsify. On the other hand, such techniques do not suffer from the major disadvantages of formal techniques, e.g. state space explosion.

2.3.1 MODEL CHECKING

Model checking is perhaps the most common type of formal verification used in industry, due to its proven efficiency and relatively simple use.

In model checking, the specification is written as a set of temporal logic formulas. In particular, Computational Tree Logic (CTL) is usually used [Cla86]. CTL is able to express properties in branching time, which makes it possible to reason about possibilities of events happening in different futures. The logic has also been augmented with time (Timed CTL [Alu90]) to allow definition of time bounds on when events must occur. Section 3.3 will present more details about these logics.

The design, on the other hand, is usually given by a transition system. The exact approach may vary between different model checking tools, but a common formalism, also including timing aspects, is timed automata [Alu94].

The model checking procedure traverses the state space by unfolding the transition system [Cla99]. Working in a bottom-up approach, it marks the states in which the inner-most subformulas in the specification are satisfied. Then, the states for which outer subformulas are satisfied are marked based on the sets of states obtained for the subformulas. In the end, a set of states where the whole formula is satisfied is obtained. If the initial state of the transition system is a member of this set, the design satisfies the requirements of the specification. On the other hand, if the initial state is not a member, the specification is not satisfied in the design.

If a universally quantified formula was found to be unsatisfied, the model checker provides a counter-example containing a sequence of transitions leading to a state which contradicts the specification formula. In case an existentially quantified formula is satisfied in the model, a witness showing a sequence of transitions leading to a state which confirms the validity of the formula is given. A common name for counter-example and witness is *diagnostic trace*.

The time complexity of model checking is linear in terms of the state space to be investigated. However, the state space generally grows exponentially with the size of the transition system. This problem is usually referred to as the *state space explosion problem*. A major consequence of the state space explosion problem is that many designs are difficult to formally verify in a reasonable amount of time.

As, basically, every reachable state in the state space is visited one by one by the classical model checking algorithm, it is not feasible to check very large systems with a reachable state space of above 10^6 states. In fact, for a long time, people did not believe that formal verification (and, in particular, model checking) had any practical future because of this problem. However, later on, more efficient data structures to represent sets of states have evolved to allow state spaces of over 10^{20} states to be investigated [Bur90]. In particular, states are not visited or represented one by one, but states with certain common properties are processed symbolically and simultaneously as if they were one entity. The data structure for such efficient representation of state spaces is called Binary Decision Diagrams (BDD) [Bry86]. Model checking using BDDs is called *symbolic model checking*.

2.3.2 EQUIVALENCE CHECKING

Equivalence checking is typically used during the design refinement process. When a new, refined, design is obtained, it is desired to check that it is equivalent with the old, less refined, version. The old, less refined, version can be said to serve as the specification. The method requires the input/output correspondences of the two designs. In the context of digital system design, there exist two distinct types of equivalence checking, depending on the type of circuits to compare: combinational and sequential.

Combinational equivalence checking is relatively simple, checking that the two designs, given a certain input, produce the same output. This is usually accomplished by graph matching and functional comparison [Bra93].

Sequential equivalence checking is more difficult since we need to verify that given the same sequence of inputs, the designs produce the same sequence of outputs. A well-known method is to combine the two designs into one and traverse the product to ensure equivalence [Cou90].

2.3.3 THEOREM PROVING

Formal verification by theorem proving takes a different approach from both model and equivalence checking. The state space as such is not investigated, but a pure mathematical or logical approach is taken. Theorem provers try to prove that the specified properties are satisfied in the system using formal deduction techniques similar to those used in logic programming [Rus01]. The prover needs the following information as input: background knowledge, the environment in which the system operates, the system itself and the specification. Equation 2.1 expresses the task of theorem proving mathematically.

$$background + environment + system \vdash specification \tag{2.1}$$

The main problem of theorem proving is its extremely high computational complexity (sometimes even undecidable). Consequently, human guidance is often needed, which is prone to error and often requires highly skilled personnel [Cyr94].

One attractive solution to this problem is to mix theorem proving and model checking. A simplified model, still preserving the property in question, is developed. Theorem proving is used to verify that the property really is preserved. The property is then verified with the simpler model using model checking. This method moreover allows diagnostic trace generation in applicable situations. Work has been done to automate the property-preserving simplification of the model [Gra97].

The advantage of theorem proving over other techniques is that it can deal with infinite state spaces and supports highly expressive, yet abstract, system models and properties.

2.3.4 SIMULATION

Simulation-based techniques operate with four entities: the model under verification (MUV), the stimulus generator, the assertion checker (or monitor) and coverage measurement. Figure 2.5 illustrates how these entities cooperate [Piz04].

The stimulus generator feeds the model under verification with input stimuli. It is important that the input stimuli are generated in such a way that as much as possible of the model is exercised. Therefore, the stimuli cannot be totally randomly generated. There must hence be a bias, for instance towards corner cases.

The very same set of stimuli, which is given to the MUV, is also given to the assertion checker. The output of the MUV provides additional input to the assertion checker. The assertion checker compares the input and output sequences of the MUV in order to check for any inconsistencies between the specification

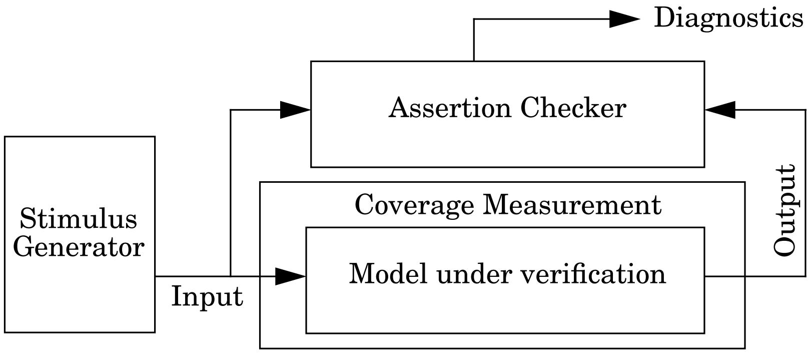


Figure 2.5: Simulation overview

and the implementation. The result is then forwarded to the verification engineer.

Coverage is a measure to indicate the completeness of the verification. 100% coverage indicates that all aspects supposed to be of interest are verified. An implication of this is that once such a coverage is obtained, there is no point in continuing the verification.

In order to state something about the achieved coverage, a coverage metrics has to be defined. Two types of metrics can be defined: implementation specific and specification specific. Implementation specific metrics refer to structures in the MUV, such as the number of covered lines of code, paths, transitions etc. Specification specific metrics, on the other hand, refer to the assertions checked by the assertion checker, such as the number of covered antecedents of temporal logic implication formulas. It is greatly recommended to define a combined coverage metrics, where coverage from the two types are weighted against each other.

The coverage measurement is surveying the whole process, investigating which parts of the MUV and/or the specification has been exercised by the generated stimuli, with respect to the defined coverage metrics. As hinted previously, the stimulus generation should be biased to maximise coverage. From this point of view, one can say that the coverage metrics actually guides the whole simulation process. The results of the simulation process are satisfactory only to the degree indicated by the obtained coverage.

2.4 Verification of IP-based Designs

This section will describe a few techniques where the component based structure can be utilised in order to perform verification more efficiently. The components are assumed to be preverified by their respective designers and thus to be correct. What furthermore has to be verified is the interconnection of components and interaction between components.

2.4.1 ASSUME-GUARANTEE REASONING

Assume-guarantee reasoning [Cla99] is not a methodology, in the sense described in earlier sections in this chapter. It is rather a method of combining the results from the verification of individual components to draw a conclusion about the whole system. This has the advantage of avoiding the state explosion problem by not having to actually compose the components, but each component is verified separately.

The correct functionality of a component, M, does not only depend on the component itself, but also on its input environment. This is expressed as $\langle g\rangle M\langle f\rangle$, where g is what M expects from the environment, and M guarantees that f holds. A typical proof shows that both $\langle g\rangle M'\langle f\rangle$ and $\langle True\rangle M\langle g\rangle$ hold and concludes that $\langle True\rangle M \parallel M'\langle f\rangle$ is true, where $\parallel$ is component composition. M and M' are two different but interacting components. The result of a component composition $M \parallel M'$ is a new component behaving in the same way as M and M' together. Equation 2.2 expresses this statement as an inference rule.

$$\frac{\begin{array}{l}\langle True\rangle M\langle g\rangle \\ \langle g\rangle M'\langle f\rangle\end{array}}{\langle True\rangle M \parallel M'\langle f\rangle} \tag{2.2}$$

Equation 2.3 shows another common inference rule which is very powerful in the context of assume-guarantee reasoning.

$$\frac{\begin{array}{l}\langle g\rangle M\langle f\rangle \\ \langle f\rangle M'\langle g\rangle\end{array}}{M \parallel M' \models f \wedge g} \tag{2.3}$$

It expresses that if M and M' are each other's specification, i.e. fulfils the assumptions of the other component, then their composition will satisfy the whole specification. This type of reasoning is often referred to as *circular* assume-guarantee reasoning [Mis81], [Loc91], [Hen02].

The environment in assume-guarantee reasoning is provided in terms of logic formulas. This is probably acceptable in the cases when verifying the functionality of a single component. However, when verifying the interaction of several components through a glue logic, interconnecting the components, several drawbacks arise. The environment of a given component, in this case, consists of models of the glue logic and of other components, expressed in the particular design representation used. Therefore, assumption formulas have to be extracted from these models with respect to the property to be verified. That is not always easy, especially considering that the environment components, in turn, depend on yet other components. In our approach, on the other hand, we directly involve the environment components into the verification process, though in an adapted form where the dependency with other components is abstracted away. The adapted forms of the components may be obtained automatically.

2.4.2 MODELLING THE ENVIRONMENT IN THE PROPERTY FORMULAS

Another approach, different from assume-guarantee reasoning, is to include the environment of the model to verify in the property formula [Cha02]. The advantage with this approach is that the designer can express the correctness property and the environment under which it is expected to hold in a unified way.

Assume that the possible input to our system is $\{i_1, i_2\}$. Equation 2.4 expresses a property stating that always within 4 time units a state where f is satisfied is reached. This formula should be checked assuming the environment described by $i_1 \wedge i_2$, i.e. both input signals are present.

$$\mathbf{AF}^{i_1 \wedge i_2}_{\leq 4} f \tag{2.4}$$

The authors of [Cha02] call this logic *Open-RTCTL* and they have also developed a model checking algorithm for it.

However, as with assume-guarantee reasoning, the environment (input) must be given as a logic formula. The problems are therefore similar. In addition, this technique targets in particular verification of communication protocols.

2.5 Remarks

In this chapter, issues concerning IP reuse from a verification point of view have been discussed, as well as several verification techniques. However, these techniques are developed with respect to component verification without taking integration into account. Moreover, there does not exist any work that provides a holistic approach to verifying component-based systems.

The rest of this thesis will discuss issues related to applying these techniques (with emphasis on model checking and, to a lesser extent, simulation) to IP based designs. A roadmap guiding the designer through the verification process, facilitating decision-making, will also be provided.

The thesis will, in addition, touch upon issues, related to component verification, which add to its practicality. This includes a translation procedure from SystemC to the Petri-Net based design representation used, and a simulation approach enhanced with model checking to make it more feasible to verify large components.

The final implementation of embedded systems usually consists of both hardware and software parts. The proposed modelling approach is appropriate for representing both the functionality which is going to be implemented in hardware as well as the functionality which is going to be implemented in software. At the beginning of the design process (Figure 2.1) where the actual mapping has not yet been decided, such a distinction cannot be made. At later design steps (in particular mapping), certain parts of the functionality (model) are decided to be implemented in hardware and software respectively. One consequence of such a decision is, for example, that actual estimated execution time intervals can be associated to certain elements of the model and, consequently, timing related properties can be verified.

Chapter 3 Preliminaries

THIS CHAPTER PRESENTS the necessary background concepts in order to fully understand the rest of this thesis. First, important aspects of SystemC will be presented, followed by an introduction of the design representation which will be used throughout the thesis. Finally, a brief introduction to Computation Tree Logic (CTL) follows.

3.1 SystemC

Designing complex embedded systems stresses the need of an intuitive and easy-to-use design language with effective support for component-based design. One such language, gaining popularity, is SystemC [Bai03].

SystemC is, in fact, a C++ class library containing class definitions corresponding to structures (buses, processes, signals, channels, 4-valued logic, etc.) used in embedded system and digital system design. A SystemC program is, in principle, a C++ program. As such, ordinary C++ development tools and compilers can be used. Both hardware and software can therefore be tightly developed using the very same language. Codevelopment and coverification of these two parts are therefore relatively straightforward tasks. Executing a SystemC program corresponds to simulating the model.

The following SystemC concepts are important in the context of this thesis:

- Processes
- Scheduler
- Channels and signals
- Events
- wait statements
- Transaction-level modelling

Each concept will be elaborated in the following subsections.

3.1.1 PROCESSES

SystemC models consist of a collection of processes. Each process belongs to one of three types: METHOD, THREAD and CTHREAD.

Processes of type METHOD are used to model combinational circuits. They are typically set to execute once each time at least one of their input values changes. METHOD processes always execute in zero time.

THREAD processes behave as an ordinary process, as can be found in mainstream programming languages. This is the most general process type. CTHREAD (clocked threads) is similar to THREADs, except that they are activated periodically according to a clock.

Processes of both type METHOD and CTHREAD can be modelled as processes of type THREAD without loss of generality. Therefore, in the rest of this chapter, only processes of type THREAD will be considered.

3.1.2 SCHEDULER

The SystemC scheduler orchestrates the execution of the model. It synchronises the different entities in the model so that they interact according to the correct semantics.

According to the SystemC semantics, only one process may execute at a time. It is the task of the scheduler to decide which process, in a set of ready processes, to execute at a certain time moment. When a process has received control, it retains it until it executes a wait statement. Processes, thus, retain control until they explicitly give it up (yield).

The scheduler furthermore divides the execution into *delta cycles*. A delta cycle is finished when there are no more processes ready to execute. Between two delta cycles, new processes may become ready and execution can progress.

In Section 5.4.1, a more detailed description of the SystemC execution mechanism is given.

3.1.3 CHANNELS AND SIGNALS

Processes communicate through channels. A channel is an object which implements an arbitrarily complex communication protocol. Normally, a channel has at least one write method and one read method. Blocking calls are realised by wait statements inside the methods of the channels.

Signals are a special type of channel. When a new value is written to a signal, that value is not visible to any reader until the next delta cycle.

Processes can register themselves to signal value changes. As a consequence, when the value of a signal changes, the registered processes are declared ready in the next delta cycle.

3.1.4 EVENTS

Events are a mechanism for one process to notify one or several other processes that something has happened which other processes are interested in. There are two ways in which processes can listen, or subscribe, to an event: staticly or dynamically.

Processes listening staticly to an event must declare this in conjunction with the creation of the process by including the event in a *sensitivity list*. Such processes will always be notified upon the particular event.

In addition to static subscription to events, processes can temporarily listen to events dynamically. This is useful when an event only casually has significance to a process. Dynamic listening is performed using wait statements.

When a process is notified, the scheduler adds that process to its pool of ready processes. The scheduler will then eventually give control to that process.

Signals actually use events to notify other processes when their values have changed. Consequently, registering processes to signals comes down to subscribing them to the event connected to the signal.

3.1.5 wait STATEMENTS

wait statements suspend the calling process and give control back to the scheduler which chooses another ready process for execution. The wait statements come in a few different variants. Their difference lies in the way the process should be reactivated after suspension. The following lists the most important variants:

- Time:
  The process is declared ready again when the specified amount of simulated time has elapsed.
- Event:
  The process is declared ready again when the specified event

has occurred (see dynamic subscription to events in Section 3.1.4).

- Event with time-out:
  The process is declared ready again when the specified event has occurred *or* the specified simulated time has elapsed, whichever comes first.

Using wait statements with timing is the only way to specify time, or make time advance. All other statements are considered to be instantaneous.

3.1.6 TRANSACTION-LEVEL MODELLING

At early stages in the design process, designers wish to focus on the functionality rather than low-level communication details. For this purpose, transaction-level modelling (TLM) [Ros05] has been developed. Using TLM, the designer can concentrate on what to transmit, rather than how to transmit.

In TLM, all messages are encapsulated in transactions and sent by one process to another through a channel. If the properties of a selected channel were found unsatisfactory during simulation, this channel can easily and straight-forwardly be changed, so that it finally satisfies the requirements. This is due to imposed standardised interfaces on channels.

TLM has shown to be an efficient approach to refining designs in the development process.

Although SystemC can be used for modelling at various levels of abstraction, it is particularly suitable for TLM. This level of abstraction is of main interest throughout this thesis.

3.2 The Design Representation: PRES+

In this work, we use a Petri-net based model of computation called *Petri-net based Representation for Embedded Systems* (PRES+) [Cor00].

This design representation was chosen because of its expressiveness and intuitivity. It is capable of handling concurrency as well as timing aspects. It is also suitable for describing IP blocks, since they can be well delimited in space and be assigned a well-defined interface. The models can be provided at any desired level of granularity. Moreover, it is possible to verify designs expressed with this formalism using existing model checking tools [Cor00].

3.2.1 STANDARD PRES+

Definition 3.1: PRES+. A PRES+ model is a 5-tuple $\Gamma = \langle P, T, I, O, M_0 \rangle$ where
P is a finite non-empty set of *places*,
T is a finite non-empty set of *transitions*,
$I \subseteq P \times T$ is a finite non-empty set of *input arcs* which define the flow relation from places to transitions,
$O \subseteq T \times P$ is a finite non-empty set of *output arcs* which define the flow relation from transitions to places, and
M_0 is the initial *marking* of the net (see Item 2 in the list below).

■

We denote the set of places of a PRES+ model Γ as $P(\Gamma)$, and the set of transitions as $T(\Gamma)$. We furthermore define $V(\Gamma) = P(\Gamma) \cup T(\Gamma)$.

The following notions of classical Petri Nets and extensions typical to PRES+ are the most important in the context of this thesis (a PRES+ example is illustrated in Figure 3.1):

1. A token k has values and timestamps, $k = \langle v, r \rangle$ where v is the value and r is the timestamp. In Figure 3.1, the token in place p_1 has the value 4 and the timestamp 0. When the timestamp is of no significance in a certain context, it will often be omitted from the figures.

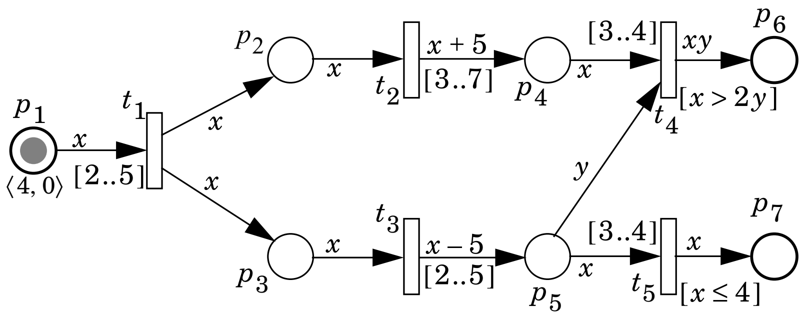


Figure 3.1: A simple PRES+ net

2. A marking M is an assignment of tokens to places of the net. The marking of a place $p \in P$ is denoted $M(p)$. A place p is said to be marked iff $M(p) \neq \varnothing$.
3. A transition t has a function (f_t) and a time delay interval ($[d_t^-..d_t^+]$) associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp of the new tokens is the maximum timestamp of the enabling tokens increased by an arbitrary value from the time delay interval. The transition must fire at a time before the one indicated by the upper bound of its time delay interval (d_t^+), but not earlier than what is indicated by the lower bound (d_t^-). The time is counted from the moment the transition became enabled. In Figure 3.1, the functions are marked on the outgoing edges from the transitions and the time interval is indicated in connection with each transition.
4. The transitions may have guards (g_t). A transition can only be enabled if the value of its guard is true (see transitions t_4 and t_5).
5. The preset $°t$ (postset $t°$) of a transition t is the set of all places from which there are arcs to (from) transition t. Similar definitions can be formulated for the preset (postset) of

places. In Figure 3.1, ${}^{\circ}t_4 = \{p_4, p_5\}$, ${t_4}^{\circ} = \{p_6\}$, ${}^{\circ}p_5 = \{t_3\}$ and ${p_5}^{\circ} = \{t_4, t_5\}$.

6. A transition t is enabled (may fire) iff there is at least one token in each input place of t and the guard of t is satisfied.

3.2.2 DYNAMIC BEHAVIOUR

Figure 3.2 illustrates the dynamic behaviour of the example given in Figure 3.1. In the situation of Figure 3.1, there is an initial token with value 4 and timestamp 0 in place p_1. Moreover, this token enables transition t_1, which can fire at any time between 2 and 5. The associated function of t_1 is the identity function. Assuming that the transition fires at time 3, the situation in Figure 3.2(a) is reached, where two identical tokens with value 4 and timestamp 3 are situated in places p_2 and p_3 respectively. Both transitions t_2 and t_3 are now enabled. t_2 can fire after 3 but before 7 time units after it became enabled and t_3 after between 2 and 5 time units. This means that we have two simultaneous flows of events. If t_2 fires after 4 time units and t_3 after 5 time units, the situation in Figure 3.2(b) is obtained, where the new token in p_4 has value $4 + 5 = 9$ and timestamp $3 + 4 = 7$ and the token in p_5 has value $4 - 5 = -1$ and timestamp $3 + 5 = 8$. In this case, both t_4 and t_5 are enabled since their guards are satisfied. Figure 3.2(c) shows the situation after t_4 has fired after 3 time units. The resulting token in p_6 will have value -9 and timestamp $max(7, 8) + 3 = 11$.

3.2.3 FORCED SAFE PRES+

In the scope of this thesis, a modification of the semantics of PRES+ is made in order to reduce complexity and to guarantee verifiability. The modification lies in the enabling rule of transitions (item 6 in the list defining standard PRES+, Section 3.2.1).

- A transition is enabled iff there is one token in each input place, there is no token in any of its output places and its

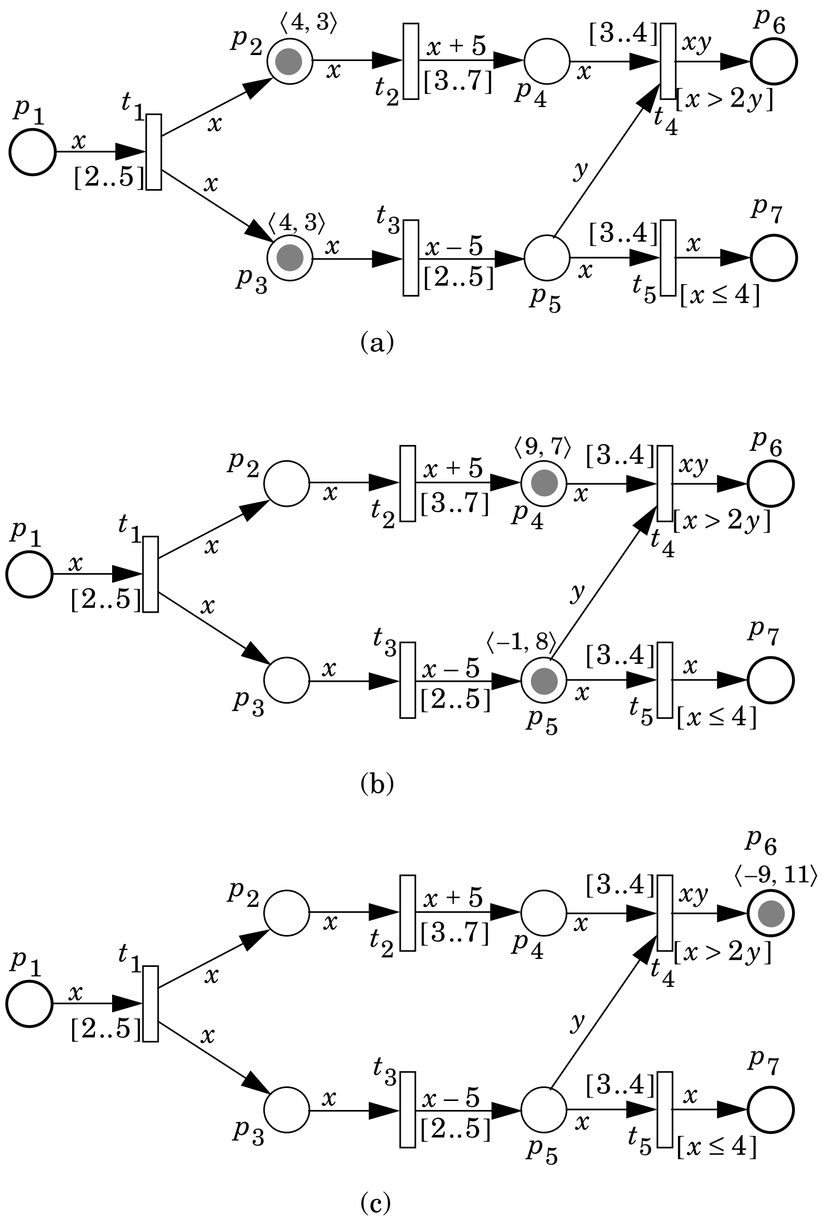


Figure 3.2: Examples of the dynamic behaviour of PRES+

guard is satisfied.

The modification guarantees *safeness* of the Petri-net. A Petri-net is *safe* if there is at most one token in each place for any firing sequence of the net. With this rule, there cannot possibly be two tokens in one place, since each transition is disabled if there is a token in an output place.

Forced safe PRES+ nets can be translated into standard PRES+ using the following translation rules, also illustrated in Figure 3.3.

1. Each place p in the net is duplicated. Label the duplication p'. If p has an initial token, then p' has not and vice versa.
2. For each input arc $\langle p, t\rangle$, where $p \in P$ and $t \in T$, an output arc $\langle t, p'\rangle$ is added.
3. For each output arc $\langle t, p\rangle$, where $p \in P$ and $t \in T$, an input arc $\langle p', t\rangle$ is added.
4. An exception to 2 and 3 is if p is both an input place and an output place of t, $p \in {}^{\circ}t \wedge p \in t^{\circ}$, in which case no arc is added (see arcs $\langle p_3, t_3\rangle$ and $\langle t_3, p_3\rangle$ in the figure.)

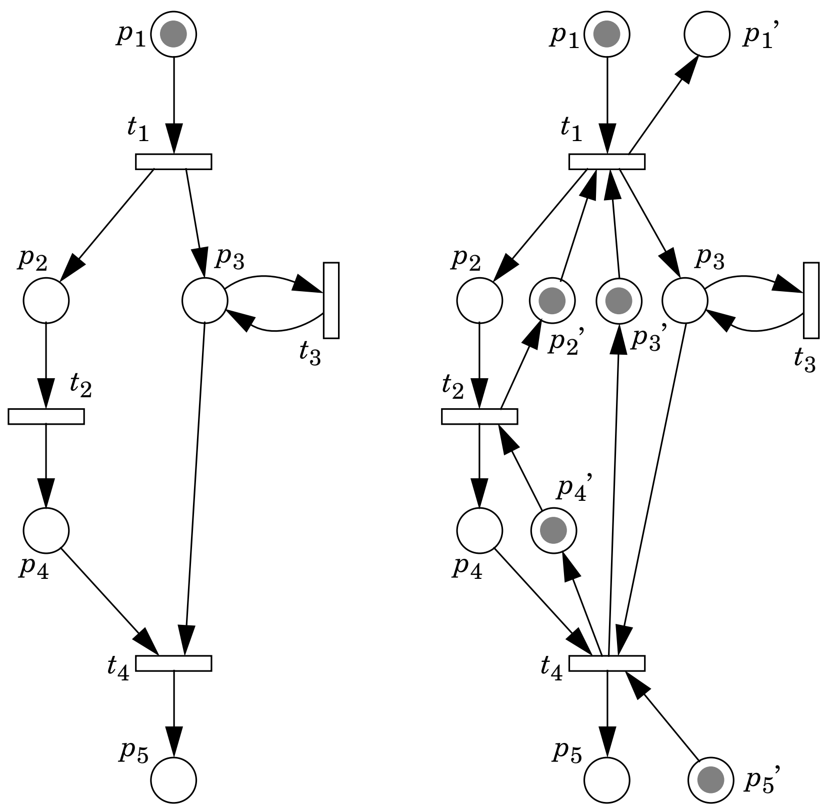
In the rest of the thesis, it will be assumed that forced safe nets are used.

3.2.4 COMPONENTS IN PRES+

We will now define a few concepts related to the component-based nature of our methodology, in the context of the PRES+ notation.

Definition 3.2: Union. The union of two PRES+ models $\Gamma_1 = (P_1, T_1, I_1, O_1, M_{01})$ and $\Gamma_2 = (P_2, T_2, I_2, O_2, M_{02})$ is defined as $\Gamma_1 \cup \Gamma_2 = \langle P_1 \cup P_2, T_1 \cup T_2, I_1 \cup I_2, O_1 \cup O_2, M_{01} \cup M_{02}\rangle$ ■

Definition 3.3: Component. A component is a subgraph of the graph of the whole system Γ such that:



(a) Forced safe PRES+ (b) Equivalent standard PRES+

Figure 3.3: Example of a PRES+ net with forced safe semantics and its equivalent in standard PRES+

1. Two components $C_1, C_2 \subseteq \Gamma$, $C_1 \neq C_2$ may only overlap with their ports (Definition 3.4), $V(C_1) \cap V(C_2) = P_{con}$, where $P_{con} = \{p \in P(\Gamma) | (p^\circ \subseteq T(C_2) \wedge {}^\circ p \subseteq T(C_1)) \vee (p^\circ \subseteq T(C_1) \wedge {}^\circ p \subseteq T(C_2))\}$

2. The pre- and postsets ($°t$ and $t°$) of all transitions t of a component C, must be entirely contained within the component, $t \in T(C) \Rightarrow °t, t° \subseteq P(C)$.

■

Definition 3.4: Port. A place p is an out-port of component C if $p \in P(C)$ and $(p° \cap T(C) = \emptyset) \wedge (°p \subseteq T(C))$. A place p is an in-port of C if $p \in P(C)$ and $(°p \cap T(C) = \emptyset) \wedge (p° \subseteq T(C))$. p is a port of C if it is either an in-port or an out-port of C.

■

Assuming that the net in Figure 3.1 is a component C, p_1 is an in-port and p_6 and p_7 are out-ports.

It is assumed that a component is interacting with other components placing and removing tokens in/from the in-ports and out-ports respectively. Hence, tokens can appear in in-ports at any time with any value. Dually, tokens can disappear from out-ports at any time.

Definition 3.5: Interface. An interface of component C is a set of ports $I = \{p_1, p_2, ..., p_n\}$ where $p_i \in P(C)$.

■

Returning to the example in Figure 3.1, the following sets are all examples of interfaces: $\{p_1\}$, $\{p_6\}$, $\{p_1, p_6\}$, $\{p_6, p_7\}$, $\{p_1, p_6, p_7\}$. The following sets are, on the other hand, *not* interfaces with respect to the example: $\{p_2\}$, $\{p_2, p_3\}$, $\{p_1, p_2, p_6\}$.

A component will often be drawn as a box surrounded by its ports, as illustrated in Figure 3.4(a), in the examples throughout the thesis. Ports will be drawn with bold circles. Modelled in this way, a component can be replaced with its PRES+ model, as indicated by Figure 3.4(b), without change in semantics.

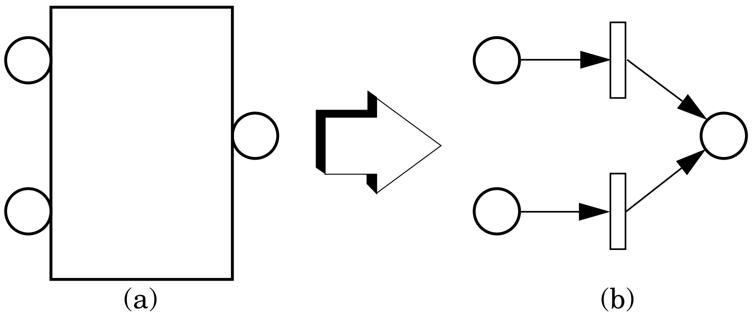


Figure 3.4: Component substitution

3.3 Computation Tree Logic

In model checking, the specification of the system (i.e. the set of properties to be verified) is written as a set of temporal logic formulas. Such formulas allow us to express a behaviour over time. For model checking, Computation Tree Logic (CTL) is particularly used [Cla86]. CTL is able to express properties in branching time, which makes it possible to reason about the possibility, and not only the necessity, of a state occurring in a certain timed manner.

CTL formulas consist of atomic propositions, boolean connectives and temporal operators. The temporal operators are **G** (globally), **F** (future), **X** (next step), **U** (until) and **R** (releases). These operators must always be preceded by a path quantifier **A** (all) or **E** (exists).

The universal path quantifier **A** states that the subsequent property holds in all possible futures (computation paths), whereas **E** states that there exists at least one future (computation path) in which the subsequent property holds. The following paragraphs will give a short explanation of the semantics of the temporal operators, also illustrated in Figure 3.5.

The operator **G** (globally) states that the particular property will always be true in every state along a certain future, i.e. a certain branch of the computation tree (including the initial state). **F** (future) states that the particular property will be true some time in the future (including the initial state), whereas **X** (next) only looks one step ahead in the future (not including the initial state).

As opposed to the previously described operators, **U** and **R** are binary. $\mathbf{Q}[p\mathbf{U}q]$ (p until q), for any path quantifier **Q**, means

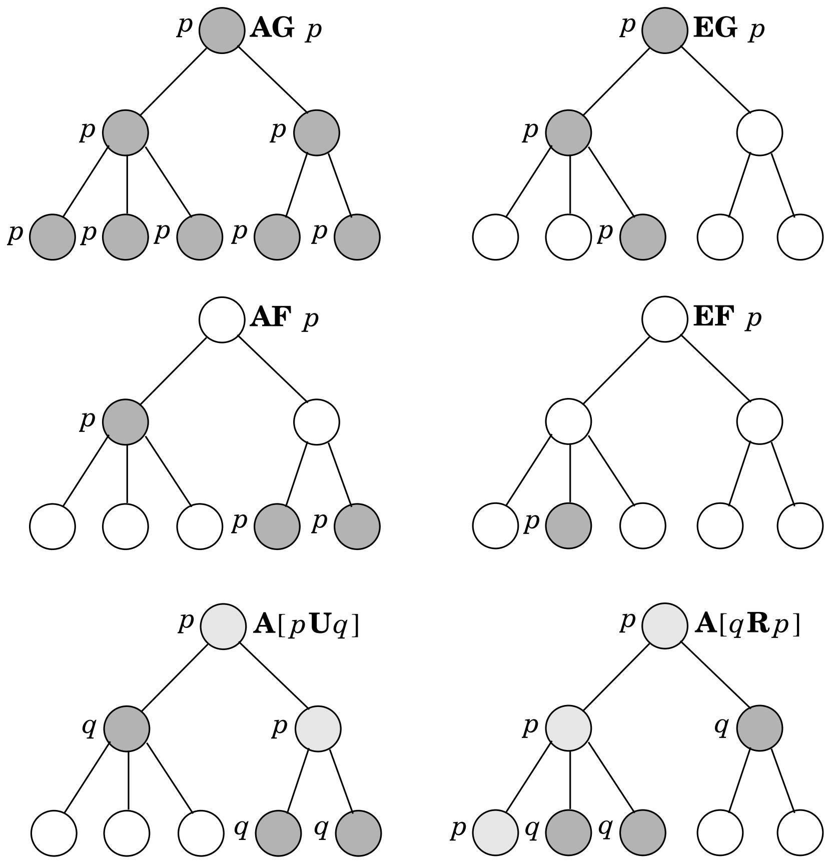


Figure 3.5: Illustration of different CTL formulas

that q must be true at some time in the future. Until the moment when q is true, p must be true in every state up until, but not necessarily including, q. It is not specified how many steps away the future when q is true is, but it must be a finite number of steps. Hence $\mathbf{QF}\ q$ is also true.

$\mathbf{Q}[q\mathbf{R}p]$ (q releases p) has a similar meaning as $\mathbf{Q}[p\mathbf{U}q]$. In fact, the two operators are duals. The difference is that it is not necessary that q will be true in the future. In that case, p must be true globally. However, if q is true at a certain point in the future, then p needs only to be true in every state up until that point. Note that the order of the arguments is reversed.

Formulas can be nested to express more complicated properties. For example, $\mathbf{AG}\ (p \to \mathbf{EF}\ q)$ means that once p is true, then it must be *possible* that q is true in the future. $\mathbf{AG}\ (p \to \mathbf{A}[p\mathbf{U}q])$ means that once p is true, it remains true until q becomes true and $\mathbf{AGAF}\ p$ states that p must be a recurring event, i.e. always be true in the future from any state no matter what happens.

The atomic propositions must be mapped to a (set of) marking in the intended PRES+ model. Every place in the Petri-net has a label. A CTL formula consisting of an atomic proposition which is identical to the label of a place, e.g. p, is true if there exists a token in that place, p. Due to the safeness assumption, there can be maximum one token in a place. A negated label, $\neg p$, is true if there does not exist any token in the corresponding place, p.

In order to verify token values, place labels can be used together with a relation, e.g. $p\Re v$, where $\Re$ is a relation and v is a value. Such a proposition is true if there is a token in the place, p, and its token value is in the relation $\Re$ with v. Hence, $p\Re v \Rightarrow p$. In Figure 3.2(b), both $p_4 = 9$ and $p_5 \le 0$ are true.

The negation of an atomic proposition with relation, $\neg p\Re v$, states that there is no token in that place, p, with a value related in the particular way. Consequently, $\neg p\Re v \Leftrightarrow \neg p \vee p\overline{\Re}v$, where $\overline{\Re}$ is the complementary relation of

$\Re$. Note that $\neg p\Re v \not\Leftrightarrow p\overline{\Re}v$, since $p\overline{\Re}v$ means that there must be a token in p with a token value in the relation $\overline{\Re}$ with respect to v.

$\mathbf{AG}\ (p_1 \to \mathbf{AF}\ (p_6 \vee p_7))$ and $\mathbf{AG}\ (p_1 \to p_1 \le 10)$ are both examples of CTL formulas referring to the example net in Figure 3.1.

It is also useful to define a subset of CTL which has particular properties (discussed in later chapters), ACTL. ACTL formulas do not have any existential path quantifiers and negation only occurs in front of atomic propositions. Hence, $\mathbf{AGAF}\ p$ and $\mathbf{AF}\ \neg p$ are ACTL formulas, whereas $\mathbf{AGEF}\ p$ and $\neg\mathbf{AF}\ p$ are not.

As mentioned previously, CTL can only express relative time, such as "p must be true some time in the future". In many applications, however, it is desired to set a time limit within which a certain property must become true. That would allow to express properties like "p must be true in the future within at least x time units." This time limit is indicated by a subscript on the temporal operators. $\mathbf{AF}_{\sim x}p$, where $\sim \in \{<, \le, =, \ge, >\}$ intuitively indicates the relationship between the time of the current state and the time point x when p must be true. For instance, $\mathbf{AF}_{\le 5}p$ means that p must always be true within (or equal to) 5 time units. The logic allowing such time relations is call Timed CTL, or TCTL [Alu90]. ACTL augmented with time is called TACTL.

Chapter 4 Verification Methodology Overview

THIS CHAPTER PRESENTS an overview of the overall verification methodology with the purpose of integrating the research described in the different chapters of this thesis and put them into a context.

As mentioned in Chapter 1, the methodology is divided into two parts: Component verification and Integration verification. Figure 4.1 illustrates the relationship of the two parts. The shadowed boxes represent tasks discussed in this thesis.

The component providers first design and implement their components. Having a complete model of the component, it needs to be verified. Figure 4.2 provides an illustration on how components can be verified, in the scope of this thesis.

The component model is assumed to be expressed either in SystemC, or in PRES+. SystemC models have to be translated into a formal representation before it is possible to formally verify. We perform such a translation from SystemC to PRES+. In the remaining part of the methodology, only PRES+ models are discussed.

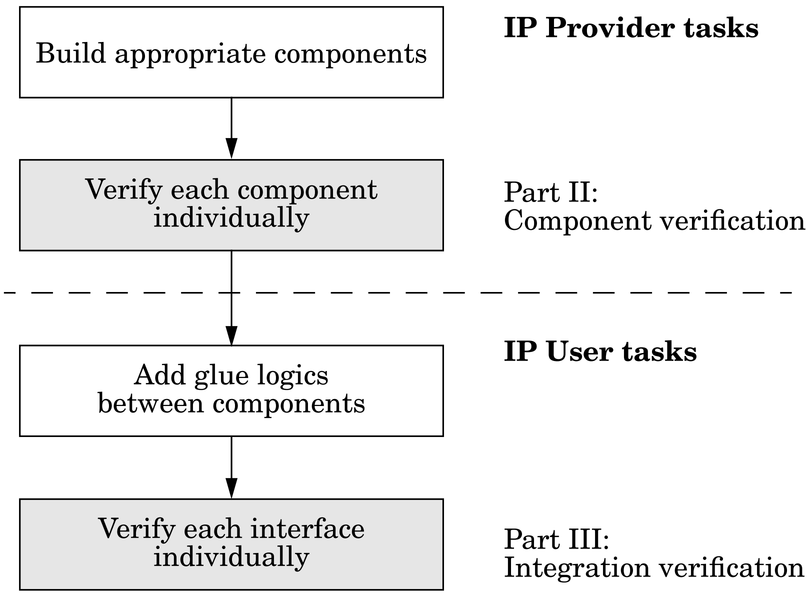

Build appropriate components
IP Provider tasks
Verify each component individually
Part II: Component verification
Add glue logics between components
IP User tasks
Verify each interface individually
Part III: Integration verification

Figure 4.1: Relationship between component and integration verification

The verification can be performed either using model checking or using a mixed simulation/model checking technique. Both methods verify a model with respect to a set of temporal logic properties. For components that are not too large, model checking is the best choice since the results will be guaranteed. However, for larger components, a simulation based technique has to be used, since model checking of a complex system will take a lot of time. Both the SystemC to PRES+ translation as well as the mixed simulation and model checking technique are presented in Part II, Chapter 5 and Chapter 6 respectively. Chapter 6 also discusses briefly model checking of PRES+ models.

After selecting a proper set of components for the design at hand, the designer has to connect them to each other in order to obtain a functional system. As illustrated in Figure 4.3, glue log-

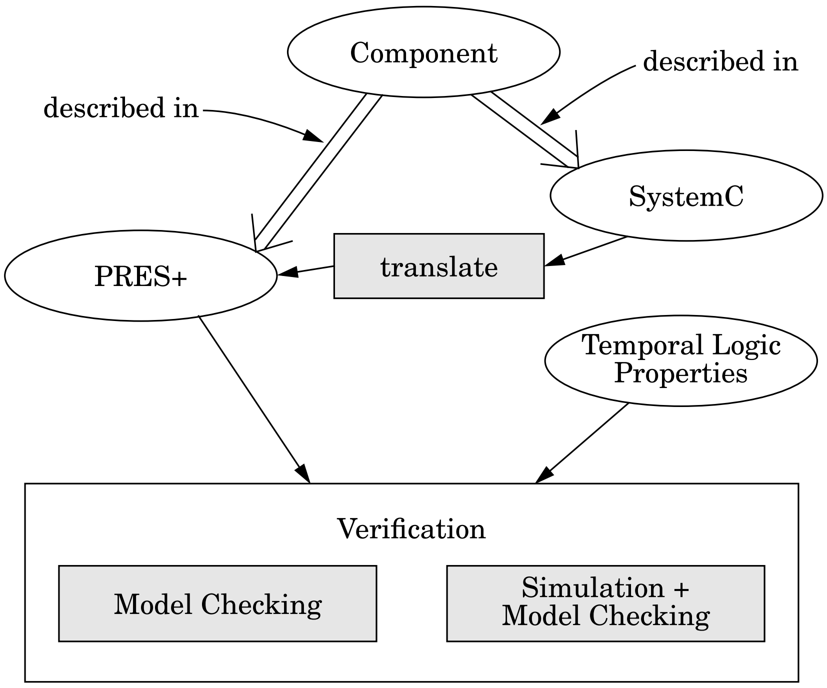


Figure 4.2: Component verification overview

ics are added between the components in order to interconnect them and adapt their interfaces so that they become compatible to each other.

The components and the glue logics together form a complete system. Since the components are now already verified, it is safe to assume that they are correct. What has not been verified is the added glue logic and the interaction of the components via their interfaces. Therefore, each component interface has to go through the verification process described in Part III, where it is verified using a representation of its *environment*. This verification process includes methods for choosing and deriving a proper environment consisting of the glue logic and a representation of the components connected to the glue logic. The term *interface*

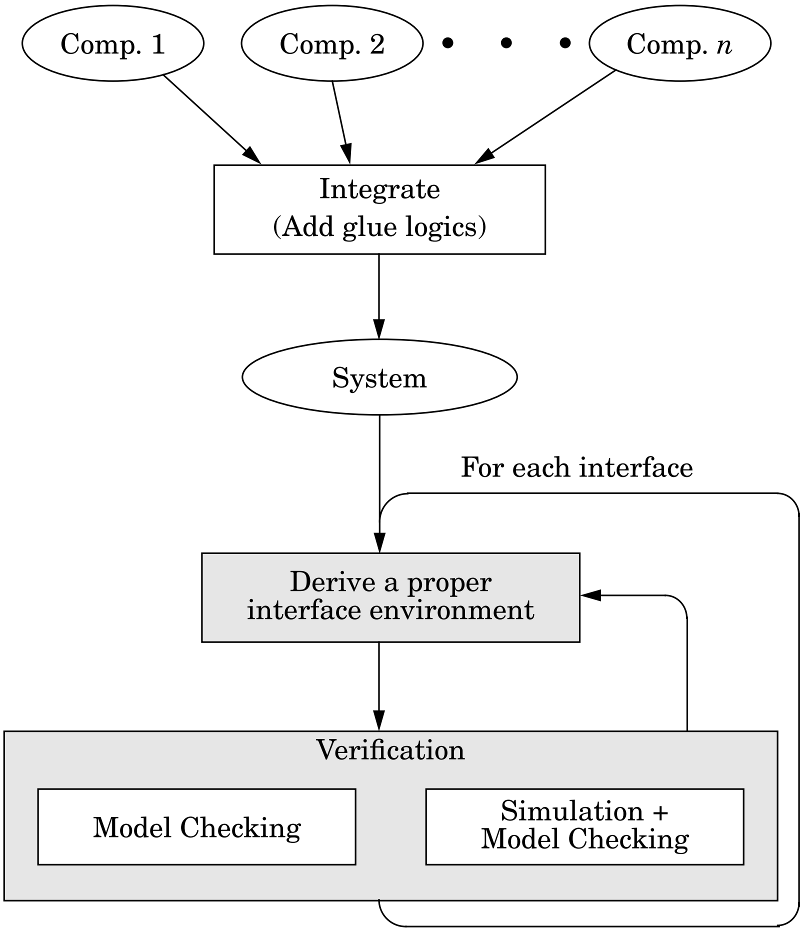
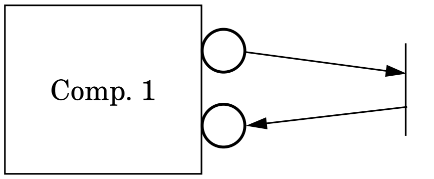


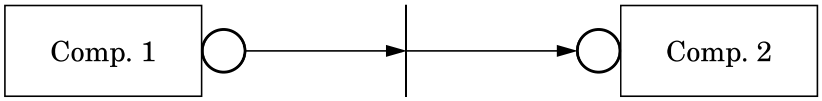
Figure 4.3: Integration verification overview

environment will be used to denote such an environment related to an interface, and the term *stub* will denote the representation of a component in an interface environment.

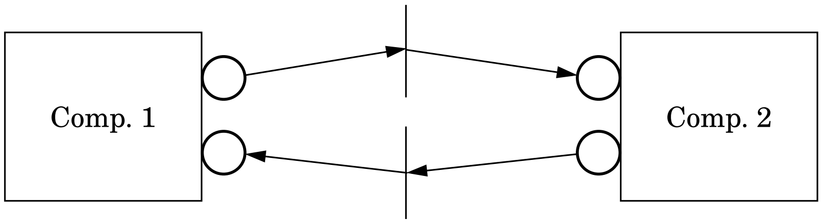
A glue logic is an unverified part of the model, connecting one or more preverified components. The complexity of a glue logic may vary from a single wire to an arbitrarily complex part of the



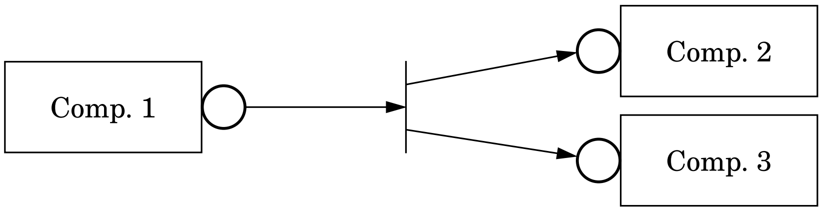
(a) A simple glue logic connecting two ports of one component



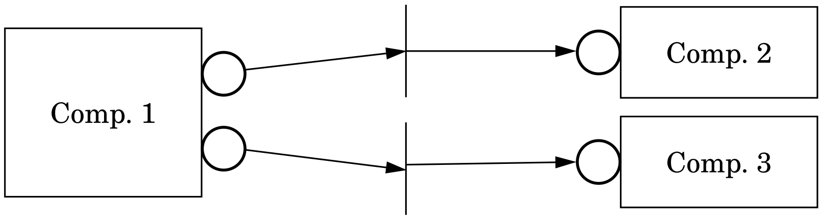
(b) A simple glue logic connecting two components



(c) One *or* two simple glue logics connecting two components

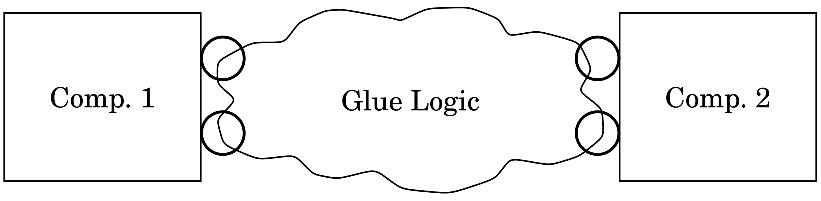


(d) A simple glue logic connecting three components

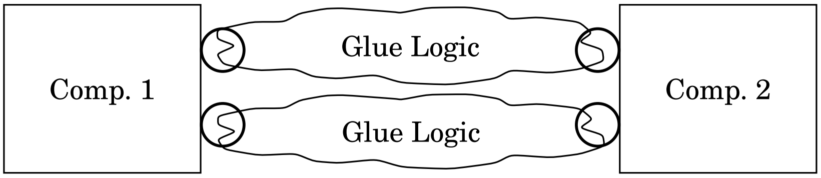


(e) One *or* two simple glue logics connecting three components

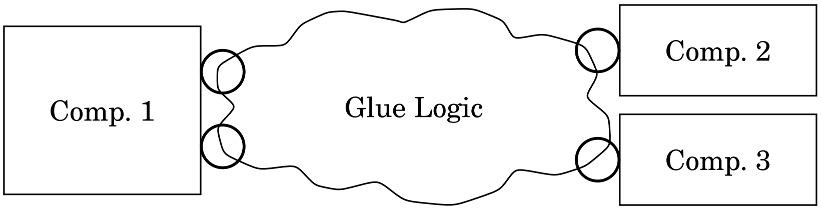
Figure 4.4: Several examples of glue logics constellations



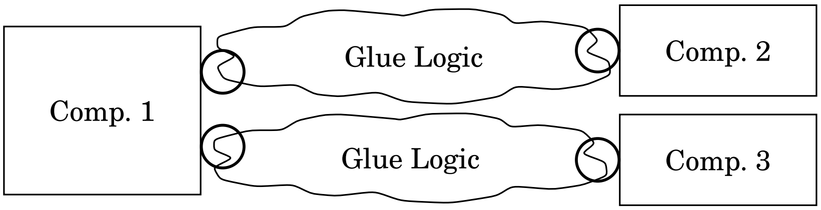
(a) One glue logic connecting two components



(b) Two glue logics connecting two components



(c) One glue logic connecting three components



(d) Two glue logics connecting three components

Figure 4.5: Several ways how to view the glue logics illustrated in Figure 4.4 (c) and (e) respectively

design. Figure 4.4 illustrates several examples of how glue logics can be used to connect one or more components. The glue logics in these figures only consist of one or two transitions, but the examples may straight-forwardly be generalised to arbitrarily complex.

As shown in Figure 4.4(a), a glue logic can be used to interconnect one port (interface) of a component with another port (interface) of the same component.

Figure 4.4(b) presents the common scenario where a glue logic interconnects two components. Figure 4.4(c) also interconnects two components. However, that glue logic consists of two disjunct parts. This means that the glue logic can either be regarded as being one glue logic (as clarified by Figure 4.5(a)), or as two parallel glue logics (as in Figure 4.5(b)). How to view the design in a particular situation is the designer's decision. Often, design and verification parameters, such as the property to be verified, influence the designer in the decision. Assume that the ports of Component 1 in Figure 4.4(c) are labelled p and q. If there is an interface property like $\mathbf{AG}(p \rightarrow \mathbf{AF}q)$ (if there is a token in p, then in the future there must also arrive a token in q), which involves both p and q, this suggests that the glue logic should be treated as one (Figure 4.5(a)). However, from the point of view of properties like $\mathbf{AG}(q \rightarrow even(q))$ (the values of tokens in q must be even), the glue logic can be considered to be either one or two (Figure 4.5(a) or Figure 4.5(b)).

Figure 4.4(d) and Figure 4.4(e) show two cases where glue logic interconnects three components. The glue logic in Figure 4.4(d) can only correspond to the situation in Figure 4.5(c), since it cannot be divided into disjunct parts. The glue logic in Figure 4.4(e) consists, on the other hand, of two disjunct parts. Similar to the situation in Figure 4.4(c), depending on the designer's intent these can be regarded as one glue logic interconnecting three components, or two separate glue logics which interconnect two components each. These scenarios are illustrated in Figure 4.5(c) and Figure 4.5(d) respectively.

Figure 4.6 illustrates the verification procedure of Figure 4.3. In order to verify the interfaces, it is needed to integrate the glue logic connected to the interface with stubs of all components connected to that glue logic. These stubs capture the characteristics of the outputs produced by the components as a result of the given input.

Preverified components have associated to their interfaces a set of (T)CTL formulas, expressing requirements which its environment has to satisfy in order to guarantee correct functional-

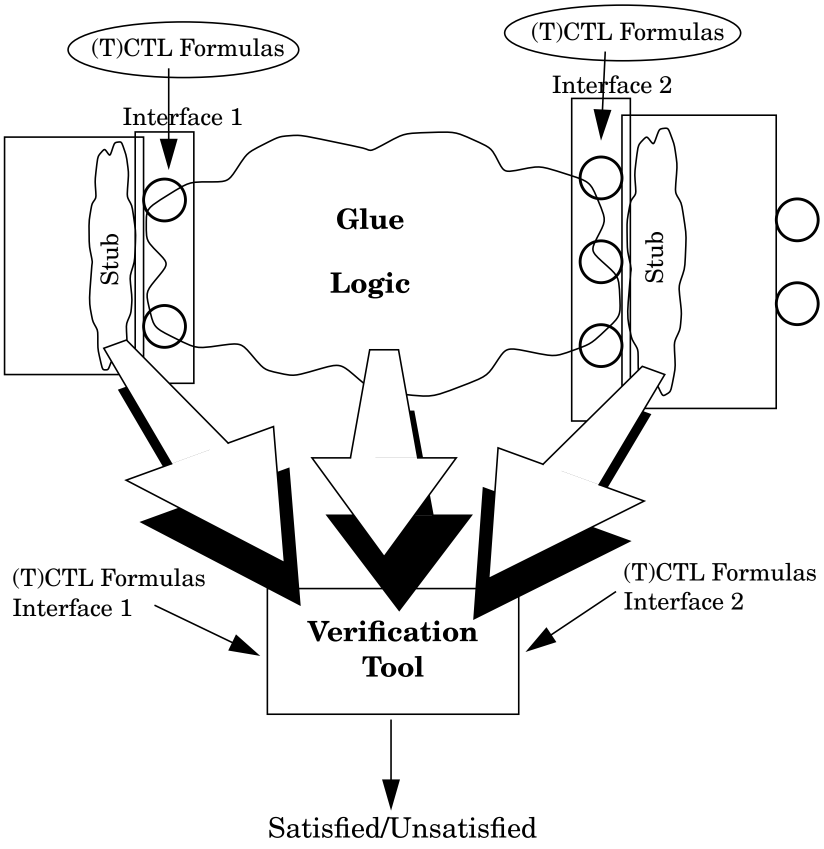


Figure 4.6: Overview of the proposed methodology

ity of the component. The model composed of one or more stubs and the glue logic (the interface environment) is then passed to the verification tool (model checker or simulator) together with the (T)CTL formulas associated to the involved interfaces of the components. The verification tool then answers whether or not the given properties are satisfied.

Part III will present the integration verification methodology in detail, including techniques to choose and automatically generate proper stubs for a particular verification.

PART II
Component Verification

Chapter 5 PRES+ Representation of SystemC Models

SYSTEMC HAS GAINED popularity in recent years as a design language for embedded and digital systems. Using SystemC, developers can easily and quickly create a working model of the system at a functional level. More details can then gradually be added in order to refine the model until it becomes cycle-accurate. Each level of refinement can be simulated.

The big advantage of SystemC over other hardware description languages (HDL's) is its close relationship with C++. Actually, SystemC is a C++ library and can be used with any C++ compiler or C++ development environment. This allows designers to use the same language for all phases in the design, from the initial sketch to the actual implementation of both the software and hardware in the final product. In addition, SystemC offers great ability to reuse existing components thanks to its object oriented nature.

With SystemC, designers normally use simulation to trap design errors. Despite the efficient implementation of the simulators, it is not always feasible to find and cover all situations and corner cases necessary to trap all errors. This is not acceptable, particularly in critical parts of a design. Therefore, there is a need to resort to formal methods. In order to use formal methods, the model needs to be translated into a formal design representation, which in the context of this thesis is PRES+. Translating the model into PRES+ furthermore enables all other verification techniques presented in the thesis, so that they can be applied also to SystemC designs.

5.1 Related Work

Several attempts, though very few, have been made to capture the formal semantics of SystemC in a representation suitable for formal verification by model checking. Kroening and Sharygina [Kro05] translate SystemC models into labelled Kripke structures (LKS). However, their approach does not handle time. Nor does it handle signals, since their scheduler lacks the notions of delta cycles and signal updates. Their work is furthermore focused on an abstraction-refinement approach based on automatic hardware/software partitioning.

Drechsler and Große [Dre02], [Gro03], [Gro05] capture gate-level SystemC specifications in netlists and RTL specifications are transformed into finite state machines. Their approach is, however, quite restricted and limited, in the sense that it can only handle specifications at these two levels. As a consequence, they can, in particular, not capture models using SystemC channels, necessary for transaction-level modelling (TLM). Nor can they handle continuous time, rather than clock cycles.

A complementary work has been performed by Habibi and Tahar [Hab05]. Given a UML description of the system and certain properties, they translate this model and these properties into Abstract State Machines (ASM). The properties are verified on the fly while generating the ASMs. In the end, the ASMs are translated into SystemC. They consequently translate from a formal representation into SystemC, the opposite direction to our work. However, in their paper [Hab06] published after ours [Kar06], they briefly mention translating from SystemC to ASMs. The procedure is not further discussed in the paper.

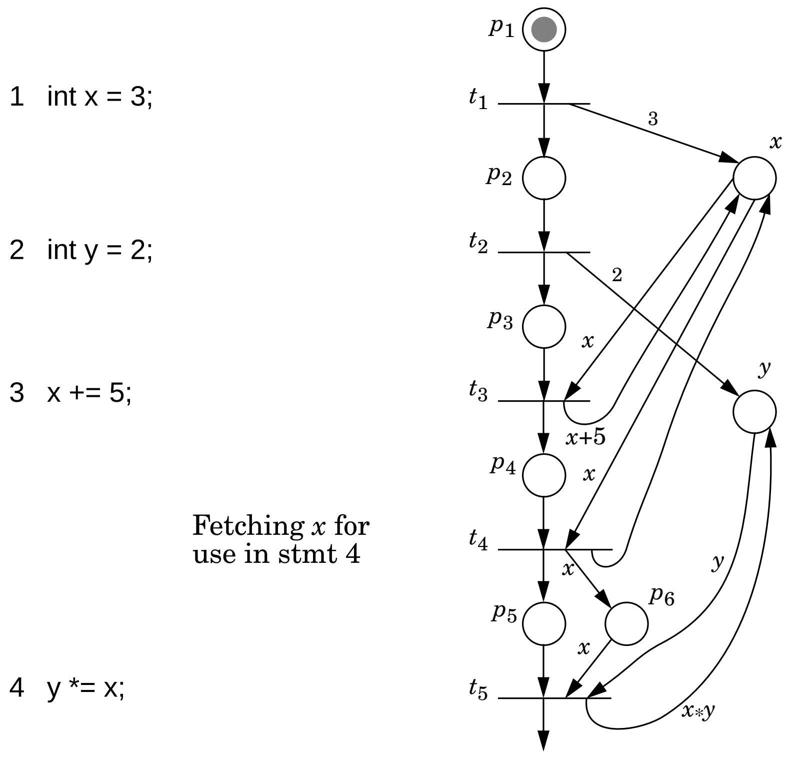
The approach described in this chapter [Kar06] removes the constraints of the described approaches. Most importantly, it can handle models at levels from the initial functional specification to cycle-accurate RT-level, including Transaction-Level Modelling (TLM). Time is treated continuously. Dynamic structures are handled to the extent that an upper bound on the sizes of those structures must be known. Loops may have variable upper bounds and SystemC channels are allowed, and encouraged (core part of TLM).

5.2 Basic Concepts

In PRES+, a SystemC statement is, in the simplest case, represented by one place and one transition. The transition performs the actual statement, whereas a token in the place enables the execution of the statement. Variables are also represented by places. There must be a token in the place during the whole lifespan of that variable. Statements assigning a value to a variable put tokens in the variable's place, and the token is removed when the execution has reached a statement out of its scope. Places for global variables always contain tokens, and places corresponding to fields in objects contain tokens as long as the object exists.

Figure 5.1 provides an example. All transitions have time delay interval [0..0], but the delays are omitted in the figure to avoid clutter.

Statements 1 and 2 introduce and initialise new variables. Transitions t_1 and t_2 reflect this by adding tokens with the initial values in the places corresponding to the variable. At the



```
1  int x = 3;

2  int y = 2;

3  x += 5;

4  y *= x;
```

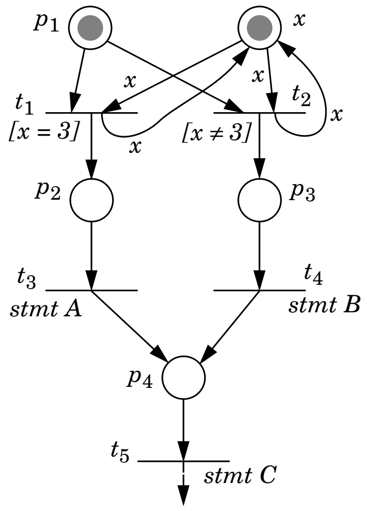
(a) SystemC (b) PRES+

Figure 5.1: Translation of statements and variables

same time they put a token in the places corresponding to the next statement to be executed, i.e. p_2 and p_3 respectively. Statement 3 updates x, which is straightforwardly reflected in transition t_3. This straightforward translation works well if only one variable is involved in the assignment statement. Statement 4, however, involves two variables. Since PRES+ transitions only can produce one output value, one of the variables must be explicitly fetched and a copy must be temporarily stored in a dedicated place (p_6). Without this procedure, variable x would be erroneously updated to the same value as y.

```
if (x == 3)
    stmt A;
else
    stmt B;
stmt C;
```

(a) SystemC



(b) PRES+

Figure 5.2: Example of an if statement

Control statements, such as if and while statements follow the same principle. An example of an if statement is given in Figure 5.2. The if statement itself is translated into one place (p_1) and two transitions (t_1 and t_2). The place, if marked, enables the two transitions. The transitions have guards which reflect the condition of the if statement. One transition (t_1) introduces the true branch, and the other transition (t_2) corresponds to the false branch. The guard of t_1 is thus identical to the condition in the if statement, whereas the guard of t_2 is the negation of that condition. After the last statement in each branch, the two branches join and the execution continues with the statement following the if statement (p_4 and t_5).

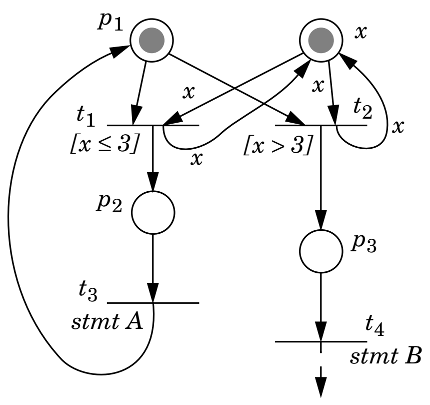
A while statement is constructed partly using the structure of an if statement, since it can be rewritten as an if statement combined with a loop back (goto), as indicated in Figure 5.3. The difference is that after the last statement of the true branch,

```
1  while (x <= 3)
2      stmt A;
3  stmt B;
```

is equivalent to

```
1  if (x <= 3) {
2      stmt A;
3      goto 1;
4  }
5  stmt B;
```

(a) SystemC



(b) PRES+

Figure 5.3: Example of a while statement

execution is transferred back to the beginning of the if statement. If the condition is false, execution continues directly with the next statement after the while statement (t_4).

5.3 Method Calls and Interfaces

Calling a method (function) involves three steps: transfer of parameter values, transfer of control and return of control. Each of these steps must be performed explicitly in the PRES+ model. Figure 5.4 presents the whole scheme. The code in the figure can be divided into two parts. Lines 1 to 5 declare the method addmult and lines 6 and 7 introduce code that invokes addmult.

Each part is translated into a PRES+ model surrounded by a box. The places between the boxes constitute the interface of the method. These places are called ports (see Section 3.2.4). By looking only at the method header (Line 1), it is possible to deduce these ports. Each parameter *par* needs two ports, *setpar* and *setparret*. If *par* is declared as a reference, two additional

```
int addmult(int v, int& xv) {
    int z = xv;
    xv *= v;
    return z + v;
}
int x = 0;
int y = addmult(2, x);
```

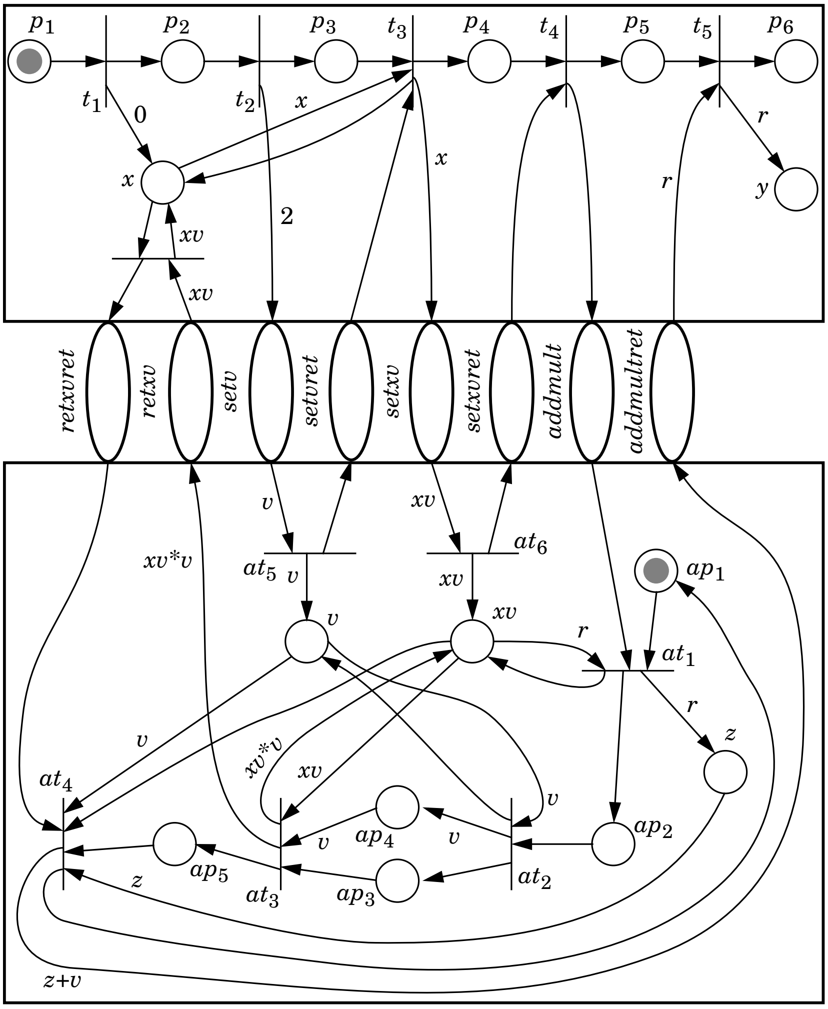


Figure 5.4: Translation of method calls

ports are needed, *retpar* and *retparret*. Finally, the method addmult itself needs two ports, *addmult* and *addmultret*.

The method call itself is realised by transitions t_2, t_3, t_4 and t_5, where t_2 and t_3 transfer the actual parameters 2 and x to formal parameters v and xv respectively. As can be seen in the figure, the transfer uses the *setpar* ports (*setv* and *setxv*) to actually pass the value and the *setparret* ports (*setvret* and *setxvret*) are used to ensure that the transfer is completed before continuing, thereby maintaining the sequential execution semantics inside a process. After the parameter transfer, transition t_4 makes the actual transfer of control to the method. Control is returned back from the method through t_5. The return port, *addmultret*, carries the return value of the method, which is stored in variable y.

A closer look at the formal parameters of addmult reveals that xv is passed by reference, while v is not. This means that whenever the value of xv is modified, so must the corresponding actual parameter. For this reason, two additional ports (*retxv* and *retxvret*) are added to the interface.

The method call structure, in particular the part that each port must be paired with a return port, occurs in many situations in this translation procedure. Without this structure, sequentiality of a process execution cannot be maintained and the model will not reflect the SystemC semantics correctly.

On the method side, assigning a formal parameter with an actual parameter is straightforward. A transition putting a token in the place of the parameter is added, for instance at_5 and at_6. The method body starts with transition at_1 which assigns the value of xv to z, according to Line 2. Since Line 3 involves more than one variable (v and xv), one variable has to be explicitly fetched. Transition at_2 consequently fetches v and the multiplication is performed by at_3. Since at_3 updates xv, a parameter passed by reference, the actual parameter must also be updated. This is performed through ports *retxv* and *retxvret*. Lastly, transition at_4 returns both control and the return value.

5.4 Scheduler

5.4.1 SYSTEMC EXECUTION MECHANISM

The main task of the scheduler is to give control to processes ready for execution, according to SystemC semantics. Processes can be declared ready for execution in one of three modes, all of which have to be supported by the scheduler:

- in the current delta cycle (immediate)
  The new process is immediately added to the set of ready processes in the scheduler. The process will, in particular, execute before any signal is updated.
- in the next delta cycle
  The new process will be added to the set of processes in the next delta cycle, after first updating the signals. The process will be executed at the same simulated time moment as the previous ones.
- at a specified time moment
  The new process will be put in a priority queue sorted on starting time. The process will execute when the simulated time has reached the specified starting time of the process. This happens when the process reaches the head of the queue.

Another important task of the scheduler is to update the signals between two successive delta cycles, so that values written to signals during one cycle are available for reading in the following delta cycle. The details regarding signal updates will be described in Section 5.7.

The scheduler repeatedly performs the following major steps:

1. Select a process ready for execution and give control to it. New processes may be declared ready for execution during the execution of the process (*immediate notification*). Repeat for each ready process until no more ready processes exist.
2. Update all signals.

3. Let time advance to the ready time of the earliest pending process. Go to step 1.

In step 1, processes ready to run are selected for execution. Which process to be selected is undefined, and therefore treated as a non-deterministic choice by the translation procedure. However, during the execution of one process, other processes may become ready during the same delta cycle (immediate notification). These processes must also be executed before steps 2 and 3 may be performed.

In step 3, if the earliest pending ready time is identical with the current time, a new *delta cycle* is introduced. If this is not the case, a new *cycle* (as well as a new delta cycle, the first in the new cycle) is introduced. A cycle, consequently, consists of one or more delta cycles.

5.4.2 PRES+ Model

The PRES+ scheduler model provides the following services through the ports depicted in Figure 5.5.

1. Give execution control to processes.
2. Receive notice of a process becoming ready.
3. Update signals.

Execution control (service 1) is given to processes through the ports labelled *trigger*. There is one trigger port for each process in the system, each dedicated to its specific process. A token in

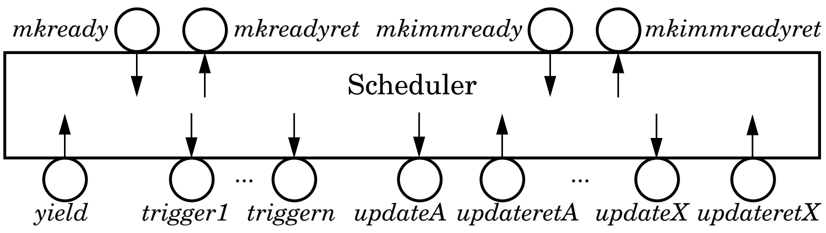


Figure 5.5: The interface of the scheduler

trigger1 signifies that process 1 may execute. When a process releases control, which it must explicitly do, it puts a token in *yield*, a port common to all processes.

Ports *mkready*, *mkreadyret*, *mkimmready* and *mkimmready-ret* are used for notifying the scheduler about the fact that a process became ready (service 2). There is only one instance of each such port. A process can become ready in either of two modes:

- in the current delta cycle (immediate)
- in the next delta cycle.

A token with a unique process identifier (pid) associated to the process to become ready is placed in *mkimmready* or *mkready* depending on the intended mode. The other two ports are return ports (see Section 5.3).

Signal updates (service 3) are performed through the ports *update* and *updateret*. There is one pair of these ports for each signal in the system. A token in *update* causes the associated signal to be updated (see Section 5.7).

Figure 5.6 shows a scheduler able to handle two processes (1 and 2) and two signals (A and B). The model can be divided into three parts, separated with dashed lines in the figure.

Processes can only be given control if they are ready for execution. The scheduler must consequently keep a record for each process about its ready state. Places *ready* and *nready* in the upper part of the figure are dedicated to this. These places are updated either through port *mkimmready* or through places *willmkready* (implicitly through port *mkready*) in the lower part, depending on if the ready notification is immediate or not. There is one *ready* place and one *nready* place for each process in the system.

A token in place *arbiter* signifies that a new process may be given control. The scheduler is, in this case, able to fire any of the *ttrigger* transitions associated to a ready process. This will

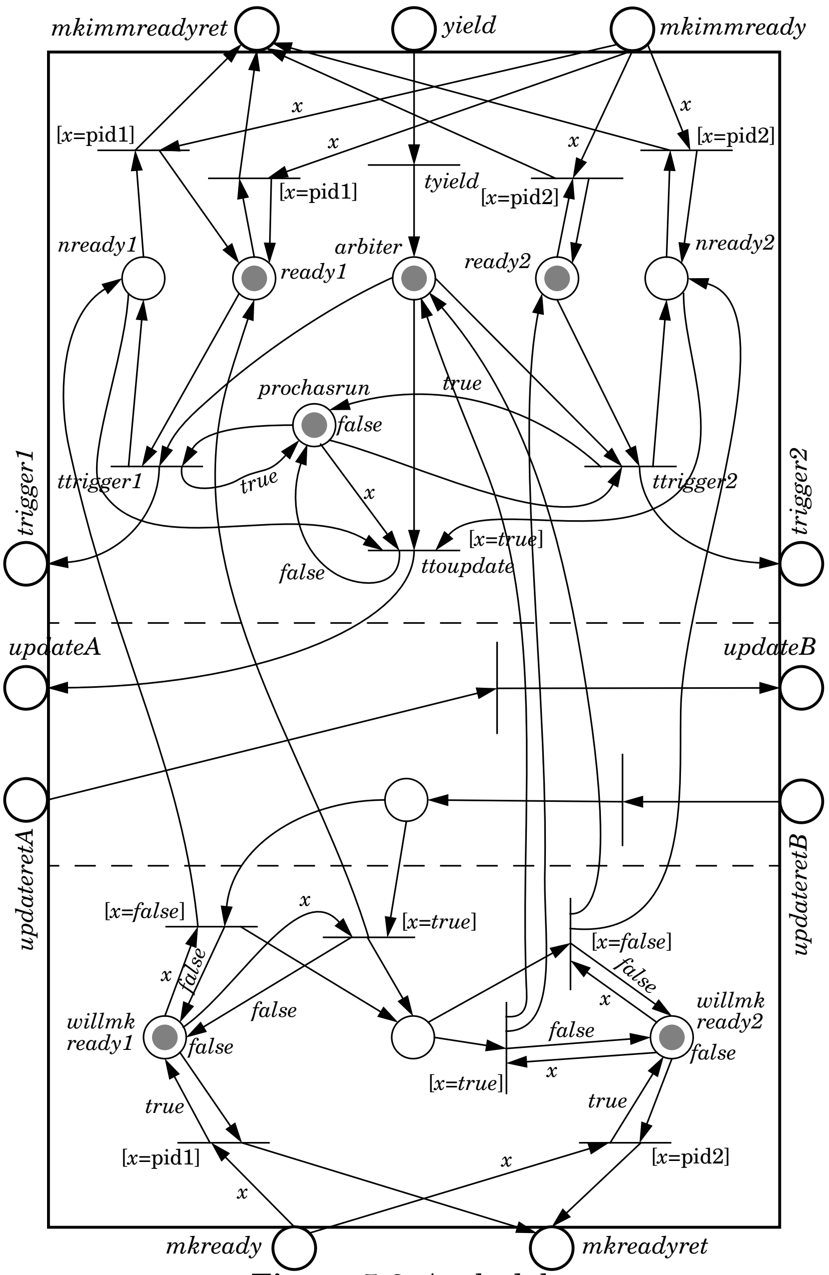


Figure 5.6: A scheduler

put a token in a trigger port and give control to the selected process. When the process finishes, it gives control back to the scheduler, by putting a token in the *yield* port. As a result, the *arbiter* place again holds a token and a new process may be selected for execution. When there are no more ready processes, the scheduler enters the middle part by firing transition *ttoupdate*.

In case no process will ever become ready at the current time, the scheduler will loop through the three parts of the scheduler indefinitely. This is due to the fact that all places *nready* will, in this scenario, always contain a token, and when a token arrives in *arbiter*, the only enabled transition is *ttoupdate*. The scheduler therefore has to fire that transition and continue to the middle and lower parts before returning to the upper part. This infinite loop takes zero time. Since the scheduler loops indefinitely, and the loop is instant, no other transitions have any chance to fire. Thus, this loop prevents time to advance. Transition *ttoupdate* should, therefore, only fire if at least one process has executed in the current delta cycle. In order to prevent the infinite loop, the place *prochasrun* is introduced. It always contains a token with a boolean value, initially false. The value in this token becomes true (see transition *ttrigger*) only if a process has executed in the current delta cycle. The guard on *ttoupdate* only allows the transition to fire if at least one process has executed. *ttoupdate* furthermore restores the value in *prochasrun* when fired. Using this mechanism, the scheduler will remain in the upper part if there does not exist any ready process, thereby breaking the loop. The scheduler can only continue when a process is notified as immediately ready again. If there is no ready process (which also means that no zero time delay transition is enabled), then time can advance. As will be shown in Section 5.6, this allows processes which are waiting for a particular period, to become ready.

The middle part of the scheduler notifies all signals that the system now enters a new delta cycle. This is performed by putting a token into the ports *update*. The signals are updated serially, one after the other. This is acceptable, since the update mechanisms of the signals do not affect each other. Section 5.7 provides a more detailed example of signal updates.

The lower part makes those processes ready that are marked to become ready in the next delta cycle. The truth value of the token in *willmkready* indicates if the associated process should be made ready or not (if a token should be placed or not in places *ready1* and *ready2* respectively). After having made the proper processes ready, a token is again placed in place *arbiter*, and the cycle is closed.

According to the execution mechanism described in Section 5.4.1, processes can be made ready in three different modes. The PRES+ model in Figure 5.6 only handles two: in the current delta cycle and in the next delta cycle. It does not make processes ready at arbitrary time moments. The advance of time is handled by the processes themselves, as will be discussed in Section 5.6.

5.5 Events

Events provide the service of making processes ready with one method call, notify. Each process interested in listening to the event must subscribe to it. Figure 5.7 presents the PRES+ interface of an event.

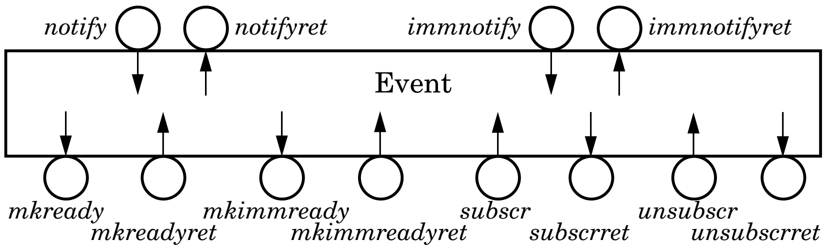


Figure 5.7: The interface of an event

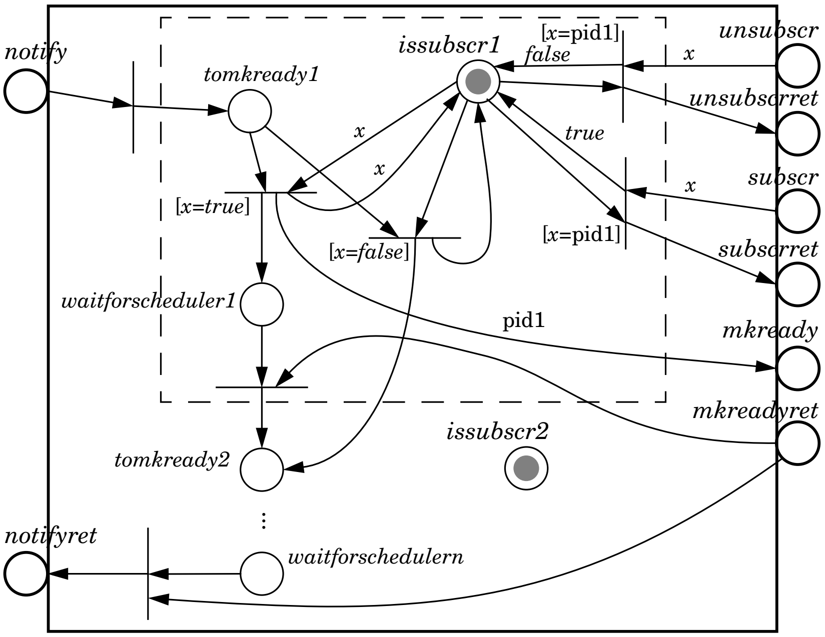


Figure 5.8: An event

Event notifications can be carried out in one of two modes: immediately in the current delta cycle or in the next delta cycle. A notification is invoked by a process by putting a token in either port *immnotify* or *notify*. When a notification is invoked, the event object takes all processes subscribed to the event and makes them ready. This is done by notifying the scheduler through the ports *mkready* or *mkimmready*, depending on whether it is an immediate notification or not. The tokens placed in those ports contain the pid of the process in question. Ports *subscr* and *unsubscr* are used to dynamically subscribe and unsubscribe a process to/from the event by placing a token with the pid of that process in the respective place.

The internal structure of an event is indicated in Figure 5.8. The structure inside the dashed square is repeated for each process subscribing to the event. The figure, furthermore, only presents notification in the next delta cycle. Immediate notification can, however, straightforwardly be extrapolated from the figure using a similar structure.

The token in place *issubscr* records if the corresponding process currently subscribes to the event or not. This value can be changed by a process through ports *subscr* and *unsubscr* respectively. Upon notification, the event model checks for each process if it is subscribed. If subscribed, the event notifies the scheduler to make the associated process ready. If not subscribed, as might be the case of dynamic events, the next process is checked. When all associated processes have been checked and, possibly, made ready, the notify request returns.

5.6 wait Statements

There are three types of wait statements which can be executed by a process:

- waiting for a certain amount of time (time-triggered).
- waiting for an event (event-triggered)
- waiting for an event with timeout.

This section will only describe the first two, since the third type is a combination of the others. Figure 5.9 depicts the PRES+ model of the first type.

Transition t_1 first hands back the control to the scheduler by putting a token in the port *yield* of the scheduler. While other processes are allowed to execute, transition t_2 measures the specified amount of time, x. When the time has elapsed, t_3 noti-

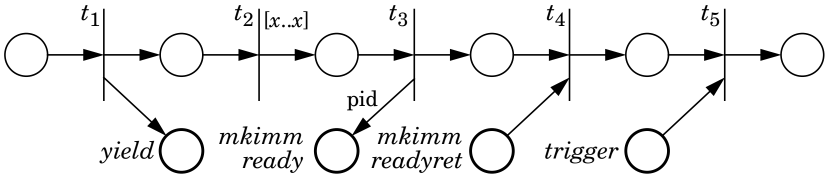


Figure 5.9: Translation of a time-triggered wait statement

fies the scheduler by placing a token containing the process identifier in the port *mkimmready* of the scheduler, thereby notifying the scheduler that the process is again ready to execute. Note that at this point, time has advanced according to step 3 in Section 5.4.1. Transition t_4 ensures that the scheduler has received the notification and finally, the process has to wait until it actually regains control from the scheduler (port *trigger*) in transition t_5.

The second type, waiting for an event, implements dynamic events. In this case, the process must first subscribe to the particular event, and afterwards it must unsubscribe from it. The whole procedure is depicted in Figure 5.10.

Transition t_1 subscribes the process to the particular event given as a parameter to the wait statement call, by putting a token containing its pid in the *subscr* port of the event. The subscription request is returned by the event and accepted by transition t_2. Next, in transition t_3, the process hands back control to the scheduler. When another process notifies the event, the scheduler marks this process as ready and is eventually granted the control. Transition t_4 accepts the grant and the process unsubscribes from the event in transitions t_5 and t_6.

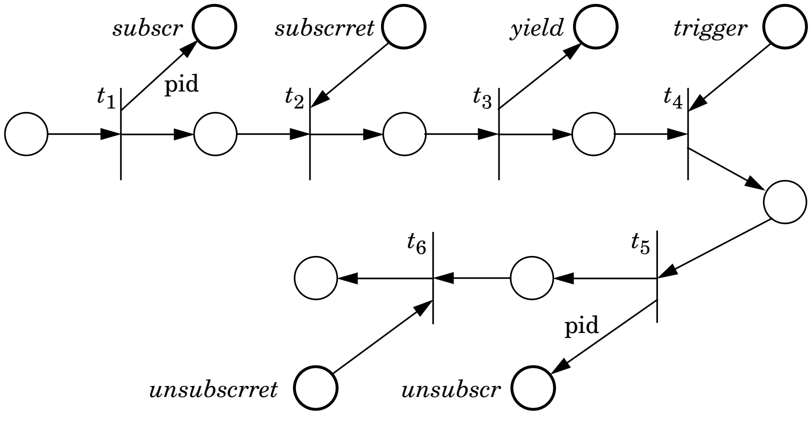


Figure 5.10: Translation of an event-triggered wait statement

A complication of these schemes occurs if the wait statement is located inside a method, for instance within a read or write of a communication channel. Such wait statements are common in the sense that they implement blocking communication protocols. The complication is twofold:

- The method does not know which pid to put in *mkimmready* (actually the pid depends on which process has called the method)
- The scheduler does not know which method the particular process has called, and hence cannot return control directly to that particular method. It can only return control to the process itself.

As a consequence, when the process calls a method, the process must also transfer its pid. This can be conveniently done through the port transferring control to the method. An example, consisting of a process invoking a method (methodWithWait) containing a wait statement, is presented in Figure 5.11.

As demonstrated by the figure, the pid is stored in a variable inside the method for retrieval by the wait statement. When a process invokes the method, it puts a token with its pid in the port transferring control to the method. In the figure, this is done by transition t_1 putting a token containing the pid into the port *methodWithWait*. Inside the method, transition at_1 consumes that token and stores the value in the place called *pid*. At the same time the method gives up control, as a part of the wait statement. At this moment, the method is suspended for the specified amount of time, in this case 2 time units. When this time has elapsed, the method will notify the scheduler in order to make the process ready again. It now uses the pid which it has stored in the dedicated variable, and passes it to the scheduler. It should be noted that the scheduler is not aware that the process is currently executing a method. Therefore, when the scheduler transfers the control back to the process, it does so via

```
int methodWithWait() {
    wait(2);
    return 1;
}
int x = methodWithWait();
```

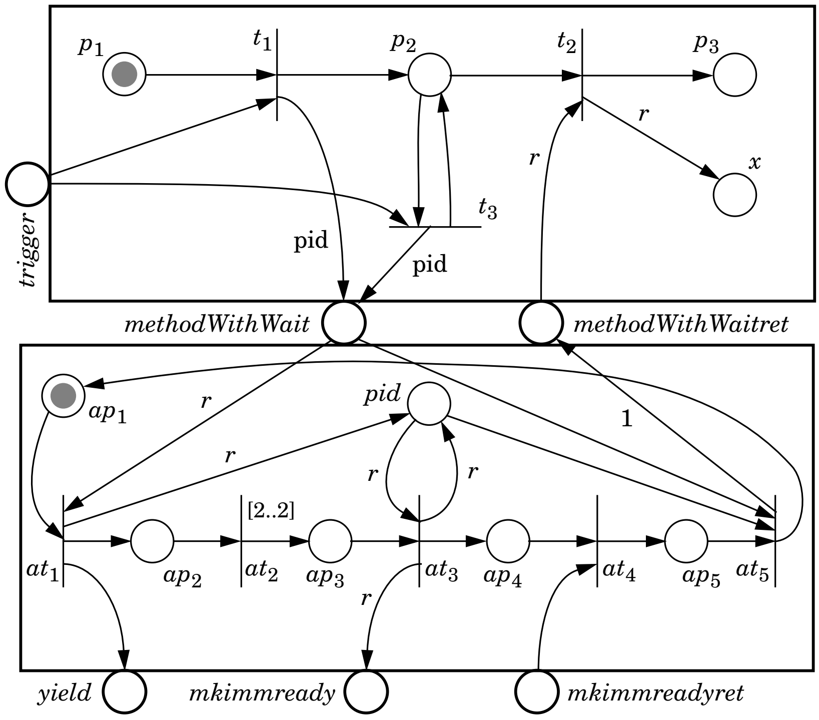


Figure 5.11: Invocation of a method containing a wait statement

port *trigger* and the main method of the process. The main method identifies that control has to be further passed to the invoked method. This identification is straight-forward, since there is a token in the place corresponding to the waiting of the method to return, in this case p_2. The additional transition t_3 performs the forwarding of control. The method can now con-

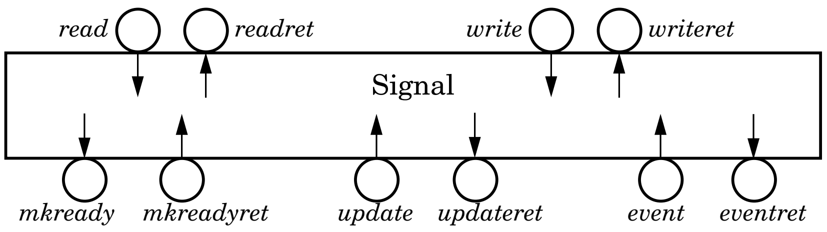


Figure 5.12: The interface of a signal

tinue executing the statements following the wait statement. In this example, this is a return statement.

5.7 Signals

As mentioned before, signals are a special type of communication channels, which require additional synchronisation with the scheduler after each delta cycle. They have to maintain two variables, *curval* and *newval*. When writing to a signal, *newval* is modified, whereas reading is performed on *curval*. After each delta cycle, *curval* must be updated to the same value as *newval*. At the same time, processes subscribing to value changes on the signal must be notified if *curval* has changed. This is done through an event, which is located inside the signal. Figure 5.12 shows the interface of a signal.

Signals have four methods: read, write, event and update. The read and write methods read and write from/to the signal respectively. The method event is a boolean method, which returns true if and only if the signal value was changed during the previous delta cycle. A dedicated variable, *isevent*, keeps track of this status. Port *update* is used by the scheduler to announce a new delta cycle. The ports *mkready* and *mkreadyret* belong to an event located inside the signal. They are used for making subscribing processes ready upon value changes.

Figure 5.13 shows the PRES+ model of a signal. The following explanation will focus on the update mechanism. The read,

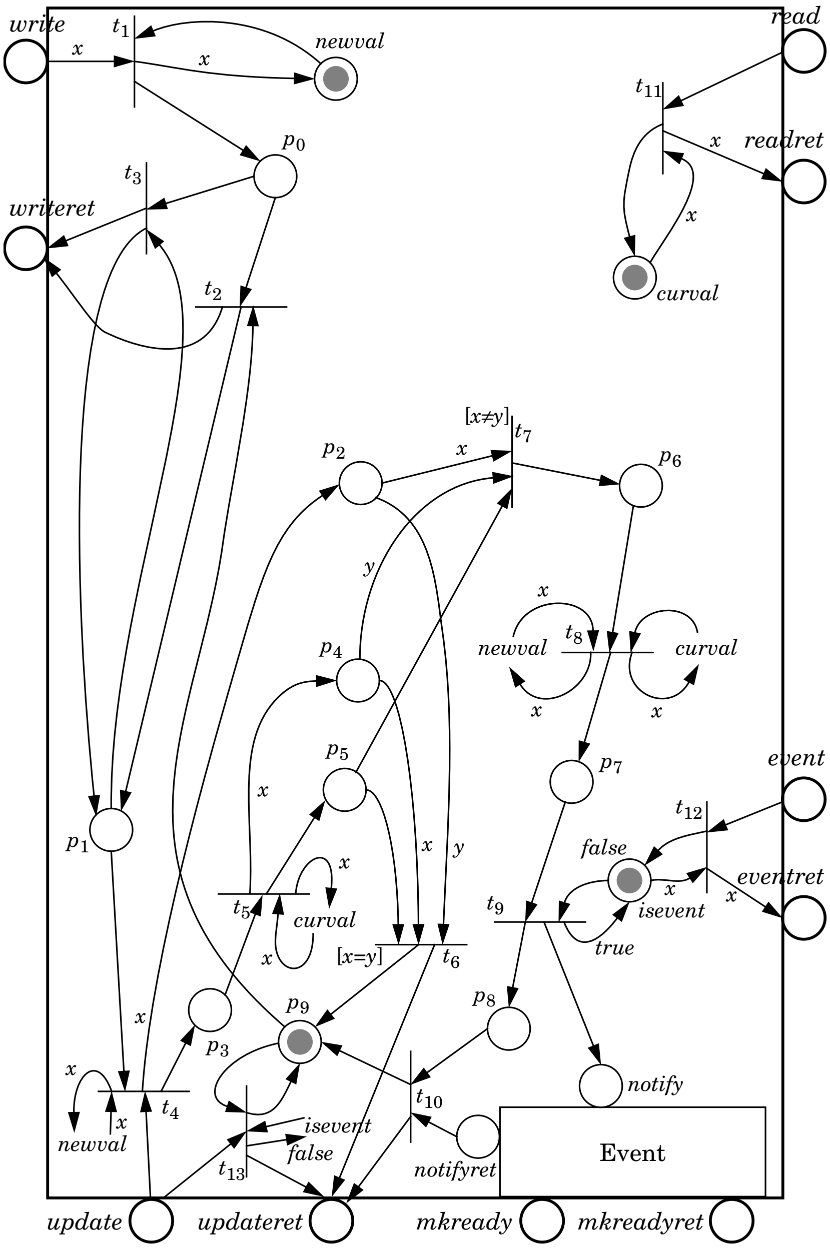


Figure 5.13: Translation of a signal

write and event operations are relatively straightforwardly implemented as method calls. They update or retrieve the values of *curval*, *newval* and *isevent* respectively.

Places p_1 and p_9 record whether there has been a write in the current delta cycle. A token in p_1 means that a write has taken place, a token in p_9 has the opposite meaning. Transitions t_2, t_6 and t_{10} update these places to reflect this situation. Depending on p_1 and p_9, either transition t_4 or t_{13} is fired upon an update request from the scheduler. Transition t_{13} is fired if there was no write. It immediately returns from the update operation and sets *isevent* to false. Transition t_4, on the other hand, is the start of a longer chain of transitions. Transitions t_4 and t_5 serve the purpose of fetching the values of *newval* and *curval*, placing copies in p_2 and p_4 respectively, in preparation for comparison by t_6 and t_7. According to the SystemC semantics, processes subscribing to value changes on the signal should not be notified unless $newval \neq curval$. Hence, if these two values are equal, then no update and notification should be done (t_6). However, if they are not equal (t_7), *curval* is updated to the same value as *newval* (t_8) and all subscribing processes are notified via an event (t_9). Finally, the signal update returns (t_{10}).

Chapter 6 Verification

As mentioned earlier, the verification of PRES+ models can be done either formally or by simulation, depending on the size and complexity of the model. This chapter will present one formal verification technique and one simulation technique where the performance of simulation is improved by injecting formal methods into the process. Both techniques are applied to PRES+ models.

6.1 Model Checking PRES+ Models

Section 6.1.1 introduces an overview of the model checking environment for PRES+ models, and Section 6.1.2 presents experimental results obtained from applying this technique on models originating from SystemC specifications.

6.1.1 Overview of our Model Checking Environment

Figure 6.1 presents an overview of our model checking environment. As input to our model checking approach, the designer has a PRES+ model and a set of (T)CTL properties. In order to reuse efficient existing model checkers, the PRES+ model first has to be translated into the input language of the model checker. In our case, we have used the UPPAAL model checking environment [UPP], since it has proven to be efficient on timed models. The input language of UPPAAL is timed automata [Alu94]. Consequently, the PRES+ models to be formally verified need to be translated into timed automata.

Having obtained a timed automata model of the system, this model is input, together with the (T)CTL properties, to the model checker. The model checker then analyses the model. If the properties were not satisfied, the model checker also provides a diagnostic trace (counter-example) indicating to the designer what part or aspect of the model caused the contradiction. The remainder of this section will focus on the translation

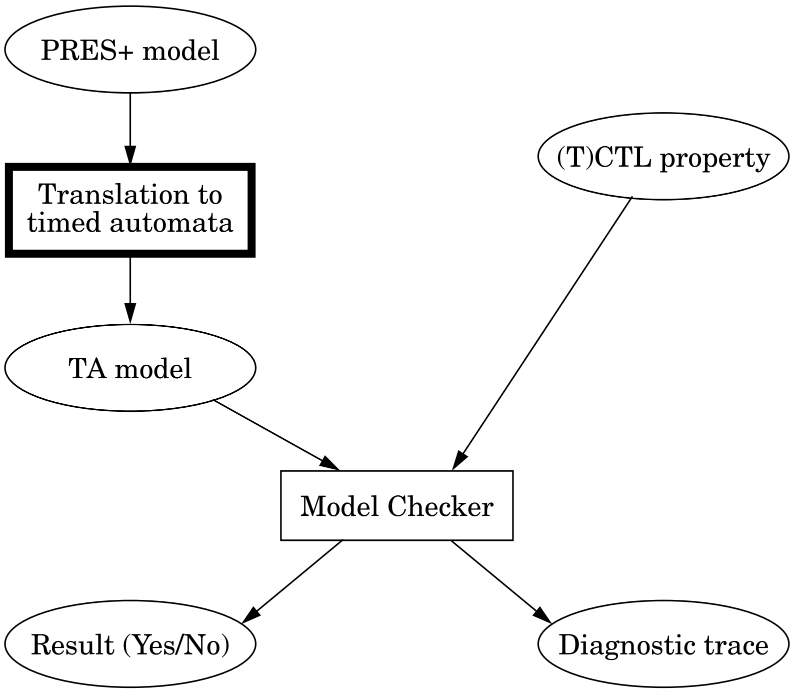


Figure 6.1: Model checking environment overview

from PRES+ to timed automata [Cor00]. The procedure only assumes safe PRES+ models. It will be illustrated with the example in Figure 6.2. Figure 6.3 presents the resulting system of timed automata.

Each place in the PRES+ model corresponds to one global variable in the timed automata. The variable contains the value of the token in the corresponding place. If there is no token in the place, the variable is assigned a default value, $\varnothing$.

One timed automaton is created for each transition t in the PRES+ model. A clock variable, ct_x, is also instantiated for each PRES+ transition. The timed automaton for a transition t has, in general, $n = |{}^{\circ}t| + 1$ locations, i.e. one more location than the number of input places of the transition. However, if t has a guard, an additional location is added to the automaton. All transitions in the example (Figure 6.2) only have one input place, except t_5 which has two. Therefore, automata t_1 and t_4 have two locations and t_5 has three. Automata t_2 and t_3 also have three locations since their corresponding transitions have guards. The locations are labelled s_0, s_1, ..., $s_{n\text{-}1}$ and en. Automata corresponding to transitions with a guard additionally have

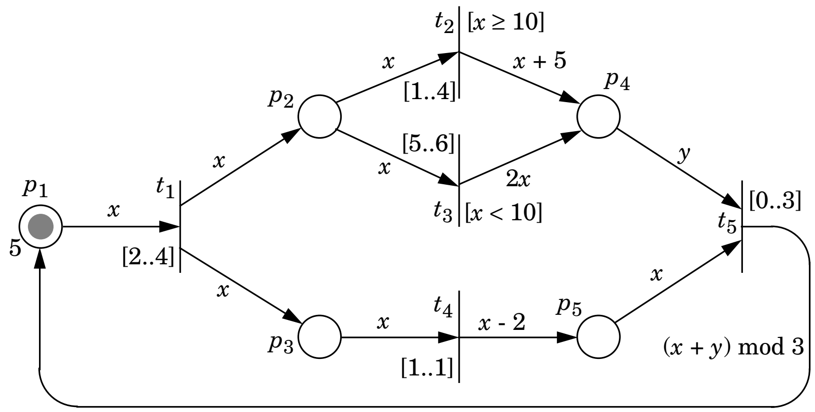


Figure 6.2: A example PRES+ model to be translated into timed automata

a location labelled *enc*. A token in *en* means that the corresponding transition is enabled, i.e. there are tokens in all input places of *t* and, if there is a guard, the guard is satisfied. Location *enc* captures the same situation, but when the guard is unsatisfied. A token in s_0 means that no input place of *t* contains a token, s_1 that one of the input places contains a token, etc.

All transitions in the timed automata have synchronisation labels. The labels are identical to transition names and have either a "!" or "?" attached to them. When a transition with a "!" synchronisation label is taken, all transitions in the whole sys-

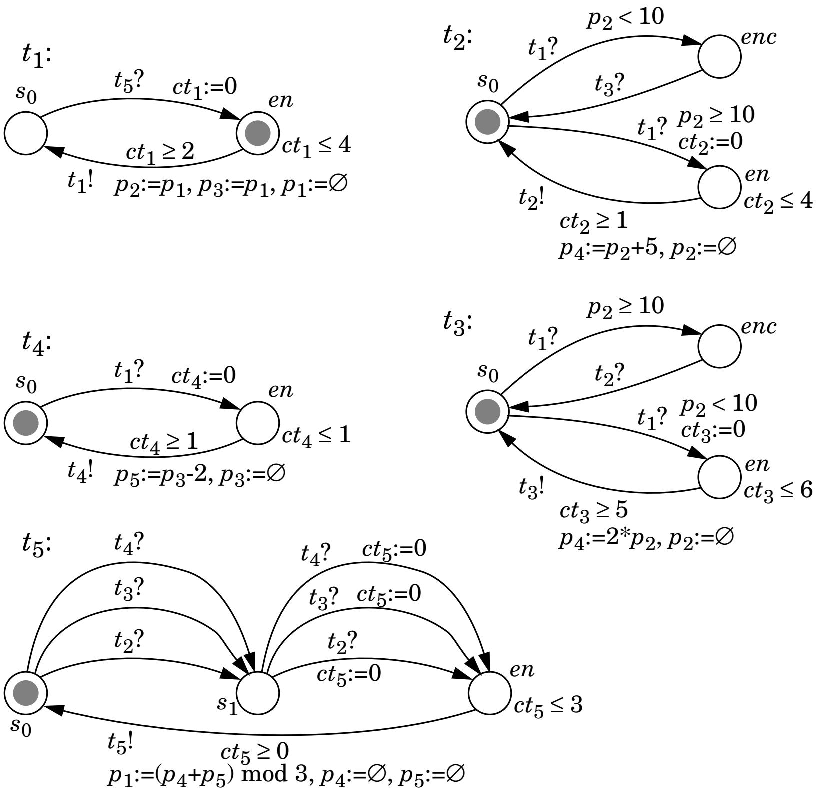


Figure 6.3: A system of timed automata corresponding to the PRES+ model in Figure 6.2

tem with the "?" version of the same label must also be taken *simultaneously*. Transitions are added in the automata so that they reflect the token situation of the input places in the PRES+ model.

Initially, only transition t_1 is enabled in the PRES+ model. This is reflected in the automata by the fact that only the automaton corresponding to t_1 has a token in location *en*. The transition from *en* to s_0 encodes the transition firing in the PRES+ model. When taken, it synchronises with those other automata that are affected by the fact that t_1 fires, so that their states are kept up-to-date with the new situation. The affected automata are those corresponding to transitions with at least one input place being the output place of t_1, i.e. t_2, t_3 and t_4. In addition to the synchronisation, the variables corresponding to the output places are updated according to the PRES+ transition function. The variables corresponding to the input places are assigned the default value, indicating that they no longer contain tokens.

The upper bounds of the time delay intervals of PRES+ transitions are reflected in the automata as a location invariant at *en*, and the lower bounds appear as guards on the transitions from *en* to s_0.

In addition to these basic translation rules, measures can be taken to make the resulting automata more efficient for model checking. One such technique involves reducing the number of clocks, so that the automata corresponding to several transitions may share one clock under certain circumstances [Cor03]. By doing this, the automata sharing the same clock can be merged, thereby also reducing the number of automata.

6.1.2 EXPERIMENTAL RESULTS

The verification efficiency of PRES+ has been demonstrated in several papers [Cor03]. In this section, the verification results of a few models obtained by translating SystemC models into PRES+, as discussed in Chapter 5, are presented. The experiments were conducted on machines with Intel Xeon 2.2GHz processors and 2GB of primary memory running the Linux operating system.

Router

This example, modelling a router at transaction level, is taken from the TLM reference implementation [Ros05]. The router forwards messages from one master to one of two slaves.

The model was verified for four different properties.

1. If a request is issued, then a response must come in the future.
2. If a message is sent to slave 1, it will arrive there.
3. If a message is sent to slave 2, it will arrive there.
4. If a message is sent to slave 2, it will not arrive at slave 1.

Table 6.1 presents the results. All properties were found satisfied within a few seconds' time.

Packet switch

The packet switch example is a slight modification of the pkt_switch example shipped with the SystemC reference implementation [Bai03]. One or more masters send messages to one or more slaves. The switch distributes the messages to their right destinations. In order to cope with messages coming in bursts, the switch model contains one FIFO queue for each master and each slave.

Table 6.1: Results from the Router example

Property	Verification time (s)
1	3.6
2	1.2
3	1.2
4	1.5

Four properties were verified:

1. No deadlock.
2. All messages sent by a master will be received by a slave.
3. Slaves may receive messages.
4. The switch will forward every message it receives.

Property 2 turned out to be false. The reason lies in the semantics of the signals connecting the switch with masters and slaves. An event notification only occurs in the case when, during an update, the new value is not equal to the current one. If several consecutive messages are identical, only the first message will actually pass the signal and reach the switch. Consequently, the subsequent (identical) messages will not reach their destinations. The model, as given in [Bai03], misses such consecutive messages that are identical.

The experiments were conducted using several different combinations on the number of masters and slaves. Table 6.2 shows the verification times. Verifying 2 masters and 2 slaves for properties 1 and 4 took between 4 to 5 hours. The reason for this explosion is that the complexity of this model is double exponential. The first exponential comes from the fact that the model checker has to investigate all execution sequences of all processes. All concurrent models have this problem. The other exponential is due to the fact that messages may have multiple destinations. By adding one more slave, the number of possible

Table 6.2: Results from the Packet Switch example

	Verification time (s)			
Property	**1m 1s**	**1m 2s**	**2m 1s**	**2m 2s**
1	1.1	58.54	39.55	18080.6
2	0.53	1.64	3.13	9.46
3	0.44	0.9	1.48	3.71
4	0.72	28.74	19.11	15375.0

destinations of a message doubles. Verification of the other properties only took a few seconds since property 2 was false (and the model checking is interrupted prematurely) and property 3 is of existential nature.

AMBA bus

An AMBA bus consists of three entities: arbiter, address bus and data bus. Masters sending on the bus must first request access to it through the arbiter, and the arbiter will then eventually grant access. Slaves have the possibility of delaying or temporarily blocking (splitting) incoming messages. However, they must eventually accept all messages. Messages are transmitted pipelined on the buses. First, the address is sent on the address bus. The associated data is sent on the data bus only in the next clock cycle. As data is sent, the address of the next message is simultaneously transmitted on the address bus, hence the pipeline.

The AMBA bus example has been modelled in two versions at different levels of detail, transaction-level and signal-level. At the transaction level, communication is implemented in a channel and transmissions are method calls with the outside behaviour of a real AMBA bus. The signal implementation, on the other hand, explicitly implements all signal exchanges between the bus, arbiter, and masters and slaves. The signal implementation is consequently more detailed than the transaction-level model.

The properties verified on both models are:

1. No deadlock.
2. If a master requests the bus, the request will eventually be granted.
3. A master may request access to the bus.
4. Messages sent by a master will always eventually be read and acknowledged by a slave.

Table 6.3: Results from the TL AMBA example

	Verification time (s)			
Property	**1m 1s**	**1m 2s**	**2m 1s**	**2m 2s**
1	8.95	86.88	81.65	7358.26
2	19.17	182.16	219.94	3281.34
3	1.00	2.58	2.88	8.34
4	13.16	90.95	115.46	3408.00

Table 6.4: Results from the SL AMBA example

	Verification time (s)			
Property	**1m 1s**	**1m 2s**	**2m 1s**	**2m 2s**
1	34.54	506.09	129.73	4339.27
2	21.57	328.79	81.52	3328.71
3	10.20	64.73	35.95	219.41
4	35.83	449.45	139.47	4212.40

Table 6.3 and Table 6.4 present the results of the transaction-level and signal-level AMBA bus examples respectively.

6.1.3 DISCUSSION

Most of the results presented in Section 6.1.2, and also elsewhere in this thesis, show that PRES+ models can be formally verified in reasonable time. However, as the models grow bigger, the state space explosion problem becomes evident, as could be seen in Table 6.2 for the 2-master-2-slave model. In such cases, formal methods are difficult to apply or even infeasible due to the required amount of time. Therefore, simulation based methods have to be used. Section 6.2 presents such a simulation based method, but where formal methods are used to boost performance.

6.2 Formal Method Aided Simulation

Designers using IP blocks in their complex designs must be able to trust that the functionality promised by the IP providers is indeed implemented by the IP block. For this reason, the IP providers must thoroughly validate their blocks. This can be done either using formal methods, such as model checking, or using informal methods, such as simulation.

Both methods, in principle, compare a model of the design with a set of properties (assertions), expressed in a temporal logic (for instance (T)CTL), and answer whether they are satisfied or not. With formal methods, this answer is mathematically proven to be guaranteed. However, using informal methods, this is not the case. The reliability of the result is indicated by a coverage metrics [Piz04]. Unfortunately, formal methods such as, for example, model checking, suffer from state space explosion. Although there exist methods to relieve this problem [Wan93], [Daw96], [Bal96], for very big systems simulation-based techniques are needed as a complement. Simulation techniques, however, are also very time consuming, especially if high degrees of coverage are required.

This section presents a validation technique combining both simulation and model checking. The basis of the approach is simulation, but where model checking is added to reach uncovered parts of the state space, thereby enhancing coverage.

6.2.1 Related Work

Combining formal and informal techniques is however not a new invention. One idea involves using simulation as a way to generate an abstraction of the simulated model [Tas04]. This abstraction is then model checked. The output of the model checker serves as an input to the simulator in order to guide the process to uncovered areas of the state space. This will create a new abstraction to model check. If no abstraction can be generated, it is concluded that the specification does not hold. As opposed to this approach, the technique presented here does not iteratively model check a series of abstractions, but tries to maximise simulation coverage given a single model. There is hence a difference in emphasis. They speed up model checking using simulation, whereas this work improves simulation coverage using model checking.

Another approach uses simulation to find a "promising" initial state for model checking [Syn03]. In this way, parts of the state space, judged to be critical to the specification, are thoroughly examined, whereas other parts are only skimmed. The approach is characterised by a series of partial model checking runs where the initial states are obtained with simulation. The technique presented in this chapter is, on the other hand, coverage driven in the sense that model checking is used to enhance the coverage obtained by the simulation.

Formal methods have also been used for test case generation [Hes03]. The main idea is to apply model checking on a correct model of the system and extract test cases from the diagnostic trace. The test cases are then applied to the actual implementation. The approach is guided by a certain coverage metrics and the resulting test cases are guaranteed to minimise test time. Although their work bears certain similarities with the work presented in this section, it solves a different problem. They assume that the model is correct while the implementation is to be tested. In our case, however, it is the model that has to be proven correct.

As opposed to existing simulation-based methods, the presented approach is able to handle continuous time (as opposed to discrete clock ticks) both in the model under validation and in the assertions. It is moreover able to automatically generate the "monitors", which are used to survey the validation process, from assertions expressed in temporal logic. In addition to this, the method dynamically controls the invocation frequency of the model checker, with the aim of minimising validation time while achieving reasonable coverage.

6.2.2 VERIFICATION STRATEGY OVERVIEW

The basic principle of simulation based validation methods is to fire one transition of the model under validation (MUV) at a time and check whether any assertions imposed on the model have been violated. At the same time, the MUV must be fed with inputs (stimuli) consistent with constraints imposed by the model on its environment. The following three assumptions are consequently imposed:

- The MUV is modelled as a transition system, e.g. PRES+ (see Section 3.2).
- Assertions, expressed in temporal logics, stating important properties which the MUV must not violate, are provided.
- Assumptions, expressed in temporal logics, stating the conditions under which the MUV shall function correctly according to its assertions, are provided.

Note that assertions and assumptions constrain the behaviour on the interface of the MUV. They do not state anything about the internal state. The assumptions constrain the input and the assertions state what the MUV must guarantee, i.e. constrain the output.

The result of the verification is only valid to the extent expressed by the particular coverage metrics used. Therefore, certain measures are normally taken to improve the quality of the results. This could involve finding corner cases which only rarely occur under normal conditions. Simulation-based techniques consequently consist of the following three parts: assertion checking, stimulus generation and coverage enhancement.

The proposed strategy consists of two phases, as indicated in Figure 6.4: simulation and coverage enhancement. These two phases are iteratively and alternately executed. The simulation phase performs traditional simulation activities, such as transition firing and assertion checking. When a certain stop criterion is reached, the algorithm enters the second phase, coverage enhancement. The coverage enhancement phase identifies a part of the state space that has not yet been visited and guides the system to enter a state in that part of the state space. After that, the algorithm returns to the simulation phase. The two phases are alternately executed until the coverage enhancement phase is unable to find an unvisited part of the state space.

In the simulation phase, transitions are repeatedly selected and fired at random, while checking that they do not violate any assertions (Line 4 to Line 6). This activity goes on until a certain stop criterion is reached (Line 3). The stop criterion used in this work is, in principle, a predetermined number of fired transitions without any coverage improvement. This stop criterion will be further elaborated in Section 6.2.8.

When the simulation phase has reached the stop criterion, the algorithm goes into the second phase where it tries to further enhance coverage. An enhancement plan, consisting of a sequence of transitions, is obtained and executed while at each step checking that no assertions are violated (Line 8 to Line 11).

```
1   initialise;
2   while coverage can be further enhanced do
3       while not stop criterion reached do                     ┐
4           select r randomly among the enabled transitions;    │ Simulation
5           fire r;                                             │
6           check that no assertion was violated;               ┘
7
8       obtain a coverage enhancement plan P;                   ┐
9       for each transition r∈P in order do                     │ Coverage
10          fire r;                                             │ Enhancement
11          check that no assertion was violated;               ┘
```

Figure 6.4: Verification Strategy Overview

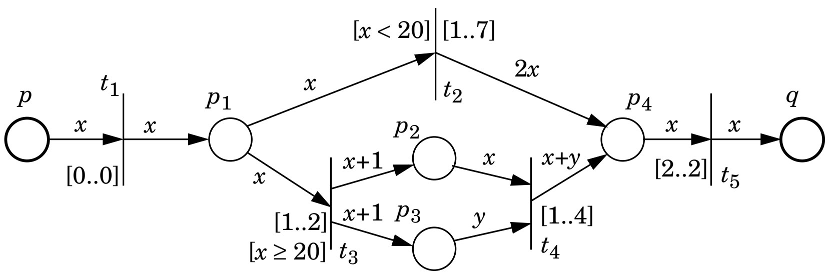


Figure 6.5: An example PRES+ model

It is at this step, obtaining the coverage enhancement plan, that a model checker is invoked (Line 8).

The two phases, simulation and coverage enhancement, are iteratively executed until coverage is considered unable to be further enhanced (Line 2). This occurs when either 100% coverage has been obtained, or when the uncovered aspects, with respect to the coverage metrics in use, have been targeted by the coverage enhancement phase at least once, but failed.

Stimulus generation is not explicitly visible in this algorithm, but is covered by the random selection of enabled transitions (Line 4) or as part of the coverage enhancement plan (Line 8). Subsequent sections will go into more details about the different parts of the overall strategy.

The model in Figure 6.5 and the assertion in Equation 6.1 serve as an example throughout this chapter, in order to clarify the details of the methodology. The assertion states that if p contains a token with a value less than 20, a token will appear in q (regardless the value) within 10 time units.

$$\mathbf{AG}\ (p < 20 \rightarrow \mathbf{AF}_{<\,10}\,q) \tag{6.1}$$

6.2.3 COVERAGE METRICS

Coverage is an important issue in simulation-based methods. It provides a measure of how successful a particular validation is. As mentioned in Section 2.3.4, it is advantageous to use a coverage metrics which refers both to the implementation and specification. A combination of two coverage metrics is therefore used throughout this chapter: assertion coverage and transition coverage.

Definition 6.1: Assertion coverage. The assertion coverage (cov_a) is the percentage of assertions which have been activated (defined shortly in Section 6.2.4) during the validation process (a_{act}) with respect to the total number of assertions (a_{tot}), as formalised in Equation 6.2.

$$cov_a = \frac{a_{act}}{a_{tot}} \tag{6.2}$$

■

Definition 6.2: Transition coverage. The transition coverage is the percentage of fired distinct transitions (tr_{fir}) with respect to the total number of transitions (tr_{tot}), as formalised in Equation 6.3.

$$cov_{tr} = \frac{tr_{fir}}{tr_{tot}} \tag{6.3}$$

■

Definition 6.3: Total coverage. The total coverage (*cov*) (*coverage* for short) is computed by dividing the sum of activated assertions and fired transitions with the sum of the total number of assertions and transitions, as shown in Equation 6.4.

$$cov = \frac{a_{act} + tr_{fir}}{a_{tot} + tr_{tot}} \tag{6.4}$$

■

Assuming, in Figure 6.5, that for a particular validation session transitions t_1, t_2 and t_5 have been fired and the assertion in Equation 6.1 has been activated, the assertion, transition and total coverage are 100%, 60% and 67% as computed in Equation 6.5, Equation 6.6 and Equation 6.7 respectively.

$$cov_a = \frac{1}{1} = 1 \tag{6.5}$$

$$cov_{tr} = \frac{3}{5} = 0.6 \tag{6.6}$$

$$cov = \frac{3+1}{5+1} \approx 0.67 \tag{6.7}$$

6.2.4 Assertion Activation

During simulation, a record of fired transitions and activated assertion has to be kept, in order to compute the achieved coverage. As for recording fired transitions, the procedure is straightforward: For each transition fired, if it has never been fired before, add it to the record of fired transitions. However, when it comes to assertions, the procedure is not so obvious. Intuitively, an assertion is activated when all (relevant) aspects of it have been observed or detected during simulation. In order to formally define the *activation* of an assertion, the concept of *assertion activation sequence* needs to be introduced.

The purpose of an activation sequence is to provide a description of what markings have to occur before the assertion is considered activated. As will be demonstrated shortly, the order between some markings does not matter, whereas it does between other markings. For this reason, an activation sequence is not a pure sequence, but a partial sequence defined as a set of $\langle$number, markings$\rangle$-pairs, where the number denotes the order in which at least one of the associated markings has in the sequence. The order between markings in pairs with the same order number is undetermined, whereas markings with different order numbers have to appear in the order indicated by the number.

The set of markings in a pair is denoted by a place name possibly augmented with a relation. This place name represents all markings with a token in that place, and whose value satisfies the relation. Below follows a formal definition of assertion activation sequence.

Definition 6.4: Assertion activation sequence. An assertion activation sequence is a set of pairs $\langle d, K\rangle$, where d is an integer and K is a (T)CTL atomic proposition, representing a set of markings.

■

Equation 6.8, given below, shows an example of an activation sequence. The order between p and $q = 5$ is irrelevant. However, they must both appear before r. p stands for the set of all markings with a token in place p, the value of the token does not matter. $q = 5$ represents the set of all markings with a token in q with a value equal to 5. Lastly, r denotes the markings with a token in place r.

$$\{\langle 1, p\rangle, \langle 1, q = 5\rangle, \langle 2, r\rangle\} \tag{6.8}$$

To use activation sequences for detecting assertion activations, a method to derive such a sequence given an assertion (ACTL formula) must be developed. In the following discussion, it is assumed that the assertion only contains the temporal operators **R** and **U**. Negation is only allowed in front of atomic propositions. Any formula φ can be transformed to satisfy these conditions. The following function, $A(\varphi)$ returns a set of activation sequences corresponding to the formula φ.

Definition 6.5: $A(\varphi)$. The function $A(\varphi) = A(\varphi, 0)$ returning a set of activation sequences given an ACTL formula is recursively defined as:

- $A(p, d) = \{\{\langle d, p\rangle\}\}$, $A(\neg p, d) = \{\{\langle d, \neg p\rangle\}\}$
- $A(p\Re v, d) = \{\{\langle d, p\Re v\rangle\}\}$

  $A(\neg(p\Re v), d) = A(\neg p \vee p\overline{\Re} v, d)$
- $A(false, d) = \emptyset$, $A(true, d) = \{\emptyset\}$
- $A(\varphi_1 \vee \varphi_2, d) = A(\varphi_1, d) \cup A(\varphi_2, d)$
- $A(\varphi_1 \wedge \varphi_2, d) = \bigcup_{a \in A(\varphi_1, d)} \bigcup_{b \in A(\varphi_2, d)} (a \cup b)$
- $A(\mathbf{Q}[\varphi_1\ \mathbf{R}\ \varphi_2], d) = A(\varphi_1, d+1) \cup A(\neg\varphi_2, d+1)$
- $A(\mathbf{Q}[\varphi_1\ \mathbf{U}\ \varphi_2], d) = A(\neg\varphi_1, d+1) \cup A(\varphi_2, d+1)$

■

It should be noted that $A(\varphi)$ returns a set of activation sequences. The interpretation of this is if any of these sequences have been observed during simulation, the corresponding assertion is considered activated. The function $A(\varphi)$ is moreover provided with an auxiliary parameter d, initially 0, in order to keep track of the ordering between the markings in the resulting sequence.

The activation sequence corresponding to $\varphi = p$, an atomic proposition, is the singleton sequence containing markings with a token in place p. Detecting a token in p is consequently sufficient for activating that property. Similarly $\varphi = \neg p$ and $\varphi = p\Re v$ are activated by markings where there is no token in p and where there is a token in p with a value satisfying the relation, respectively. $\varphi = \neg(p\Re v)$ is activated if there either is no token in p or the token value is outside the specified relation.

Since there is no marking which satisfies $\varphi = false$, there cannot exist any activating sequence. $\varphi = true$, on the other hand, is activated by all markings. There is consequently no constraint on the marking. This situation is denoted by an empty sequence. As will be explained shortly, such a property is therefore immediately marked as activated.

Disjunctions introduce several possibilities in which the property can be activated. It is partly for the sake of disjunctions that $A(\varphi)$ returns a set of sequences, rather than a single one. The function returns the union of the sequences of each individual disjunct. It is sufficient that one of these sequences is detected during simulation to consider the property activated.

In conjunctions, the activation sequences corresponding to both conjuncts must be observed. Since both conjuncts may correspond to several activation sequences, the two sets of sequences must be interleaved so that all possibilities (combinations) are represented in the result.

The formula $\mathbf{Q}[\varphi_1 \mathbf{R} \varphi_2]$, for any $\mathbf{Q} \in \{\mathbf{A}, \mathbf{E}\}$, is considered activated when either of the following two scenarios occurs. After φ_1 is detected, then from the point of view of this property, the following observations are of no significance any more. Therefore, detecting φ_1 is sufficient for activating this property. A similar situation applies when φ_2 no longer holds, therefore also $\neg\varphi_2$ is sufficient for activation. Both situations refer to future markings, for which reason the order number (parameter d) is increased with 1.

The $\mathbf{U}$ operator follows a similar pattern as the $\mathbf{R}$ operator. An important characteristics of a $\mathbf{Q}[\varphi_1 \mathbf{U} \varphi_2]$, for any $\mathbf{Q} \in \{\mathbf{A}, \mathbf{E}\}$, formula is that φ_2 must appear in the future. The property does not specify anything about what should happen after φ_2. Therefore, φ_2 is considered sufficient for activating the property. Similarly, the property does not specify what should happen when φ_1 no longer holds. Detecting $\neg\varphi_1$ is therefore also sufficient for activating the property. Since both situations refer to the future, the order number (parameter d) is increased with 1.

In the computation of activation sequences, the time bounds on the temporal operators of TCTL formulas, e.g. <10 in $\mathbf{AF}_{<10}q$, are dropped by convention.

Consider the example assertion in Equation 6.1, presented in a normalised form in Equation 6.9.

$$\begin{aligned} &\mathbf{AG}(p < 20 \rightarrow \mathbf{AF}\ q) \Leftrightarrow \\ &\mathbf{A}[\mathit{false}\ \mathbf{R}\ (\neg(p < 20) \vee \mathbf{A}[\mathit{true}\ \mathbf{U}\ q])] \end{aligned} \tag{6.9}$$

The set of activation sequences corresponding to this formula is computed as follows:

$$A(\mathbf{A}[\mathit{false}\ \mathbf{R}\ (\neg(p < 20) \vee \mathbf{A}[\mathit{true}\ \mathbf{U}\ q])], 0) =$$

$$A(\mathit{false}, 1) \cup A(\neg(\neg(p < 20) \vee \mathbf{A}[\mathit{true}\ \mathbf{U}\ q]), 1) =$$

$$\varnothing \cup A(p < 20 \wedge \mathbf{E}[\mathit{false}\ \mathbf{R}\ \neg q], 1) =$$

$$\bigcup_{a \in A(p < 20, 1)} \bigcup_{b \in A(\mathbf{E}[\mathit{false}\ \mathbf{R}\ \neg q], 1)} (a \cup b) =$$

$$\bigcup_{a \in \{\{\langle 1, p < 20\rangle\}\}} \bigcup_{b \in \{\{\langle 2, q\rangle\}\}} (a \cup b) = \{\{\langle 1, p < 20\rangle, \langle 2, q\rangle\}\}$$

with the following auxiliary computations:

$$A(p < 20, 1) = \{\{\langle 1, p < 20\rangle\}\}$$

$$A(\mathbf{E}[\mathit{false}\ \mathbf{R}\ \neg q], 1) = A(\mathit{false}, 2) \cup A(q, 2) = \{\{\langle 2, q\rangle\}\}$$

As can be seen in the computation, the activation sequence $\{\langle 1, p < 20\rangle, \langle 2, q\rangle\}$ is the only one activating $\mathbf{AG}(p < 20 \rightarrow \mathbf{AF}\ q)$. According to the sequence, a token in p with a value less than 20, which is eventually followed by a token in q, activates the assertion.

As will be seen in Section 6.2.5, activation sequences are not only used for computing the assertion coverage, but they are also useful for biasing the input stimuli to the MUV in order to boost assertion coverage.

6.2.5 Stimulus Generation

The task of stimulus generation is to provide the model under validation with input consistent with the assumptions given by the model on its environment. In the presented approach, the stimulus generator consists of another model, expressed in the same design representation as the MUV, i.e. PRES+. The exact procedure on how to obtain such a model, given an assumption ACTL formula, will be presented in Chapter 10. In this chapter, it is just assumed that it is possible to derive such a PRES+ model corresponding to an ACTL formula. This model encodes all possible behaviours which a PRES+ model can perform without violating the property for which it was created.

The stimulus generator and the MUV are then connected to each other during simulation to form a closed system. For this reason, the stimulus generator is not explicit in Figure 6.4. An enabled transition selected on Line 4 might belong to the MUV as well as to the stimulus generator.

As an example, let us assume that only even numbers are accepted as input to port p in the model of Figure 6.5. This assumption is formally expressed in Equation 6.10. Following the discussion above, a model capturing this assumption is generated and attached to the MUV. The result is shown in Figure 6.6. A transition, which immediately consumes tokens, is attached to port q, implying the assumption that output on q must immediately be processed.

$$\mathbf{AG}(p \rightarrow even(p)) \tag{6.10}$$

It was mentioned previously that activation sequences can be used to boost assertion coverage during the simulation phase. This can be achieved by not letting the algorithm (Figure 6.4) select a transition to fire randomly (Line 4). The transition selection should be biased so that transitions leading to a mark-

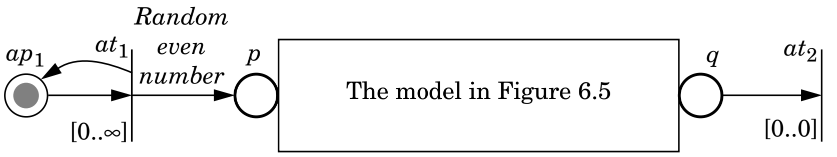


Figure 6.6: A MUV with stimulus generators

```
1  function selectTransition(MUV: PRES+,
         actseqs: set of activation sequences) returns transition
2      entrans := the set of enabled transitions in MUV;
3      p := random[0..1];
4      if p < $p_c$ then
5          if ∃ t ∈ entrans,
               such that t leads to the first marking in any seq. in actseqs then
6              return t;
7      return any transition in entrans;
```

Figure 6.7: The transition selection process

ing in an activation sequence are selected with preference, thereby leading the validation process to activating one more assertions. When all markings in the sequence have been observed, the corresponding assertion is considered activated.

$A(\varphi)$ translates a logic formula φ into a set of sequences of markings (represented by a place name, possibly augmented with a relation on token values). The transition selection algorithm should select an enabled transition which leads to a marking which is first (with lowest order number) in any of the sequences. However, only selecting transitions strictly according to the activation sequences could lead the simulator into a part of the state space, from which it will never exit, leaving a big part of the state space unexplored. Therefore, an approach is proposed in which the transition selection is only guided by the activation sequences with a certain probability. The proposed transition selection algorithm is presented in Figure 6.7.

A random value p, denoting a probability, is chosen between 0 and 1 (Line 3). If that value is less than a user-defined parameter p_c (Line 4), a transition is selected following an activation sequence if such transition exists (Line 5 and Line 6), otherwise a random enabled transition is selected (Line 7). The algorithm in Figure 6.7 is called on Line 4 in Figure 6.4.

The user-defined parameter p_c controls the probability to select a transition which fulfils the activation sequence. This value introduces a trade-off which the designer has to make. The lower the value of p_c is, the higher is the probability to enter unexplored parts of the state space. On the other hand, a too low value of p_c might lead to the situation where the assertions are rarely activated.

6.2.6 ASSERTION CHECKING

The objective of validation is to ensure that the MUV satisfies certain desired properties, called assertions. The part of the simulation process handling this crucial issue is the assertion checker, also called monitor. Designers often have to write such monitors manually, which is a very error-prone activity. The key point is to write a monitor which accepts the model behaviour if and only if the corresponding assertion is not violated.

A model created from an ACTL formula was introduced for stimulus generation in Section 6.2.5, based on the results which will be presented in Chapter 10. The same type of models can also be used for assertion checking as monitors. The assertion in Equation 6.1 will be used to illustrate the operation of a monitor throughout this section and Figure 6.8 shows the essential part of the monitor corresponding to that assertion. For the sake of clarity, the ports p and q are omitted. Arrows connecting to these ports are labelled with the name of the corresponding port.

The structure of monitors, created with the technique in Chapter 10, follows a certain pattern. All transitions in Figure 6.8, except one, interact directly with a port, either p or q. The exception is transition mt_6, whose purpose is to ensure that a token is put in q before the deadline, 10 time units after p<20. Such transitions, watching a certain deadline, are called *timers*. This observation is important when analysing the output of the MUV.

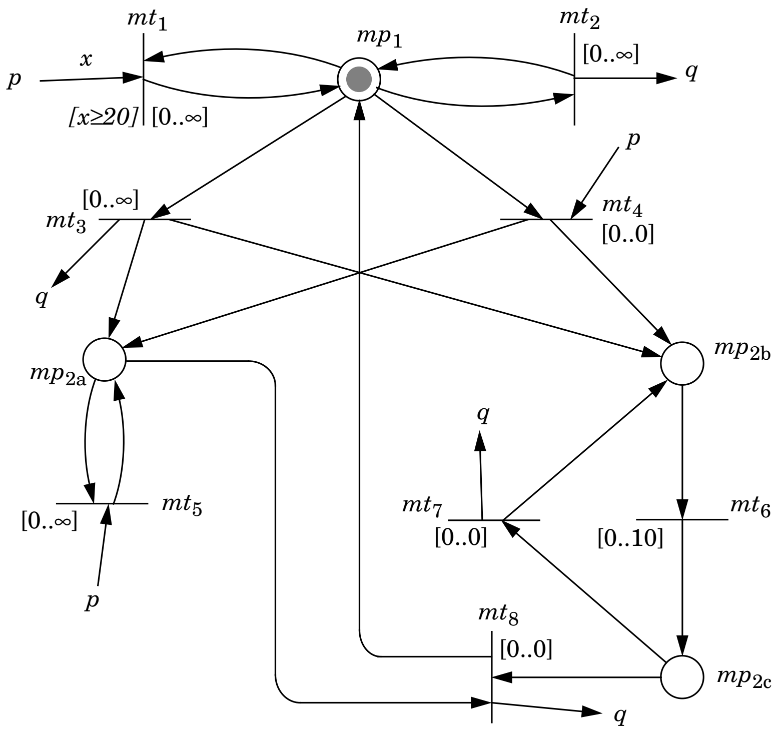


Figure 6.8: Part of an example monitor

Figure 6.9 illustrates the intuition behind assertion checking. Both the input given by the stimulus generator and the output from the MUV are fed into the assertion checker. The assertion checker then compares this input and output with the monitor model generated from the assertion (like the one in Figure 6.8). For satisfiability, there must exist a sequence of transitions in the monitor leading to the same output as provided by the MUV, given the same input. This method works based on the fact that the monitor model captures all possible interface behaviours satisfying the assertion, including the interface behaviour of the MUV. The essence is to find out whether the MUV behaviour is indeed included in that of the monitor.

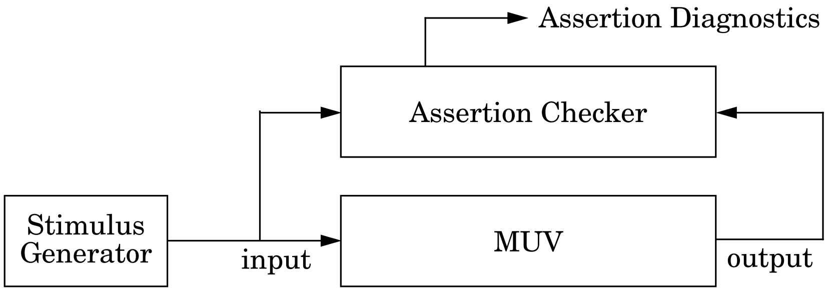


Figure 6.9: Assertion checking overview

As indicated in the figure, the input given to the MUV is also given to the assertion checker. That is everything that needs to be performed with respect to the input. As for the output sequence, on the other hand, the assertion checker has to perform a more complicated procedure. It has to find a sequence of transitions producing the same output.

It was mentioned previously, that all transitions are directly connected to a port. Due to this regularity, the stipulated output can always be produced by (at least) one of the enabled transitions in the monitor. If not, the assertion does not hold. The exception is timers. If, at the current marking, a timer is enabled, the timer is first (tentatively) fired before examining the enabled transitions in the same manner as just described. Successfully firing the timer signifies that the timing aspect of the assertion is correct.

Several enabled transitions may produce the same output. In the situation in Figure 6.8, for example, both transitions mt_2 and mt_3 are enabled and can produce the output q. However, firing either of them will lead to different markings and will constrain the assertion checker in future firings. The monitor has several possible markings where it can go, but the marking of the MUV only corresponds to one (or a few) of them. The problem is that the assertion checker cannot know which one. Therefore, the assertion checker has to maintain a *set* of possible

```
1  monitor: PRES+ := model corresponding to the assertion to be checked;
2  curmarkings: set of markings := { initial marking of monitor };
3  newmarkings: set of markings;
4  ...
5      oldtime := current time in MUV;
6      fire r; -- Line 5 or Line 10 in Figure 6.4
7      newtime := current time in MUV;
8      curmarkings :=
           validateTimeDelay(newtime - oldtime, curmarkings, monitor);
9      if r provided MUV with an input then
10         put the tokens produced by r as input to
               each marking in curmarkings;
11     if r provided MUV with an output then
12         e := marking in the out-ports of MUV;
13         newmarkings := ∅;
14         for each m ∈ curmarkings do
15             set marking of monitor to m;
16             newmarkings := newmarkings ∪ findOutput(e, monitor);
17         curmarkings := newmarkings;
18     if curmarkings = ∅ then
19         abort; -- Assertion not satisfied
20 ...
```

Figure 6.10: The assertion checking algorithm in the context of Figure 6.4

current markings, rather than one single marking. The assertion checker, thus, has to find an output from each marking in the set. If a contradiction is reached, that marking is removed from the set. The assertion is found unsatisfied when the set of current markings is empty. Figure 6.10 presents the assertion checking algorithm. It replaces Line 6 and Line 11 in Figure 6.4. Line 1, Line 2 and Line 3 (Figure 6.10) are, however, part of the initialisation step at Line 1 in Figure 6.4. The algorithm uses the auxiliary function in Figure 6.11, which validates the timing behaviour of the model, and the function in Figure 6.12, which implements the output matching procedure.

Throughout the validation process, the simulator must maintain a global variable, on behalf of the assertion checking, containing the set of possible current markings in the monitor. In Figure 6.10, the variable *curmarkings* is used for this purpose. The variable *newmarkings* is an auxiliary variable whose use will soon become clear.

The assertion checking algorithm must, at a certain moment, know how long (simulated) time a transition firing takes in order to detect timing faults. The variable *oldtime* contains the current time before the transition was fired and *newtime* the time after. The difference between these two variable is the time it took for the transition, denoted *r*, to fire. This value is passed to the function validateTimeDelay (Figure 6.11) which validates the delay with respect to the assertion. The function examines all markings in *curmarkings* and returns the subset which still satisfy the assertion. The function will be explained in more detail shortly.

If the fired transition, *r*, provides an input to the MUV, that input is also added to each marking in *curmarkings*, so that the monitor is aware of the input (Line 9 and Line 10). As input counts either putting a token in an in-port of the MUV or consuming a token from an out-port.

If the fired transition, *r*, provides an output from the MUV (Line 11), that output must be compared with the monitor model in the assertion checker. As output counts either putting a token in an out-port of the MUV or consuming a token from an in-port.

Since the monitor potentially can be in any of the markings in *curmarkings*, all of these markings have to be examined (Line 14), one after the other. The monitor is first set to one of the possible current markings, after which the enabled transitions are examined with the function in Figure 6.12 (Line 16). The function returns a set of markings which successfully have produced the output. The members of this set are added to the auxiliary set *newmarkings*. Later, when all current markings

```
1  function validateTimeDelay(d: delay, curmarkings: set of markings,
           monitor:PRES+) returns set of markings
2      newmarkings : set of markings := $\varnothing$;
3      for each m $\in$ curmarkings do
4          set marking of monitor to m;
5          let time advance in monitor with d;
6          if not monitor exceeded the upper bound of the time delay interval
                   of any enabled transition then
7              newmarkings := newmarkings $\cup$ { m };
8      return newmarkings;
```

Figure 6.11: Algorithm to check the timing aspect of an assertion

have been examined, the new markings are accepted as the current markings (Line 17).

If, at this point, the set of current markings is empty, no monitor marking can produce the output and the assertion is concluded unsatisfied (Line 18 and Line 19).

The function in Figure 6.11 validates the timing aspects of an assertion. It examines the markings in *curmarkings* one after the other (Line 3). At each iteration, time is advanced in the monitor (Line 5). As a consequence, all enabled transitions are checked, so that the upper bound of their time delay interval is not exceeded (Line 6). If at least one transition exceeded its time bound, the marking currently under examination does not agree with the stipulated delay, and is skipped. Otherwise (Line 6), the marking is added to the result set *newmarkings* (Line 7) which later is returned (Line 8) and becomes the new set of current markings (Line 8 in Figure 6.10).

Let us now focus on the auxiliary function in Figure 6.12. Given an output marking (the marking in the out-ports of the MUV) and a monitor, the function returns the set of markings which satisfy the given output.

At this point, it can be assumed that the timing behaviour of the assertion has not been violated, since the function in

```
1   function findOutput(e: output marking, monitor: PRES+)
            returns set of markings
2       newmarkings : set of markings := ∅;
3       fire all enabled timers;
4       entrans := the set of enabled transitions in monitor;
5       initmarking := the current marking of monitor;
6       for each t ∈ entrans do
7           fire t in monitor;
8           if output marking of monitor = e then
9               if a timer has a token in its output place then
10                  move the token to the input place;
11              newmarkings := newmarkings ∪ { current marking of monitor };
12          set marking of monitor to initmarking;
13      return newmarkings;
```

Figure 6.12: Algorithm for finding monitor transitions fulfilling the expected output

Figure 6.11 was called prior to this one. Thus the timers do no longer play any roll. Because of this and the fact that the lower bound of the time delay interval of timers is 0, it is safe to fire all enabled timers (Line 3). At this moment, all enabled transitions are directly connected with a port. Each of these enabled transitions are fired one after the other (Line 6 and Line 7). The result after each firing is checked whether it matches the desired output, denoted *e* (Line 8). If it does, the new marking should be stored in *newmarkings* in order to later be returned. There may, however, be tokens in the output place of some timers, e.g. mp_{2c} in Figure 6.8, which were never used for producing the output. This signifies that it was not yet time to fire those timers, i.e. the timer was fired prematurely. Before storing the new marking, the unused timer must therefore be "unfired" to reflect the fact that it was never used (Line 9 and Line 10), i.e. move the token from mp_{2c} back to mp_{2b}. The monitor is now ready again to be checked with respect to the timing behaviour of this marking in the next invocation of the assertion checker, according to the same procedure. After storing the new marking, the monitor has to restore the marking to the original situation (Line 5 and Line 12) before examining another enabled transition.

If the transition does not result in the desired output, the marking is restored (Line 12) and another enabled transition is examined. When all enabled transitions have been examined, the set *newmarkings* is returned (Line 13).

The algorithm will be illustrated with the sequence of inputs and outputs given in Equation 6.11 with respect to the monitor depicted in Figure 6.8 for the assertion in Equation 6.1. p is an in-port and q is an out-port. Initially, the set of current markings only consists of one marking, which is the initial marking of the monitor, formally denoted in Equation 6.12.

$$\begin{aligned}&[p = 30, \neg p, \langle \text{delay:}20, p = 5\rangle, \neg p,\\ &\langle \text{delay:}2, p = 7\rangle, \neg p, \langle \text{delay:}5, q = 10\rangle, \neg q]\end{aligned} \tag{6.11}$$

$$curmarkings = \{\{mp_1 \mapsto \langle 0, 0\rangle\}\} \tag{6.12}$$

The first transition puts a token in place p with the value 30. Since time has not elapsed, this operation is fine from the timing point of view. Putting tokens in an in-port is considered to be an input. The token is therefore just added to each possible current marking. The resulting set is shown in Equation 6.13.

$$curmarkings = \{\{mp_1 \mapsto \langle 0, 0\rangle, p \mapsto \langle 30, 0\rangle\}\} \tag{6.13}$$

In the next round, the token in p is consumed by the MUV. That is considered as an output, since it is an act by the MUV on its environment. According to the algorithm (Figure 6.10), the next step is to examine the enabled transitions and record the new possible markings. In this situation, four transitions are enabled in the monitor, mt_1, mt_2, mt_3 and mt_4. Firing mt_1 leads to a marking identical to the initial one, Equation 6.12, and mt_2 leads to the same marking but where a token has appeared in out-port q. However, this is not the output marking stipulated by the MUV (no token in neither p nor q). For that reason, this marking is discarded. A similar argument holds for mt_3. Firing mt_4, on the other hand, leads to a marking where both mp_{2a} and mp_{2b} are marked and the output is the same as that of the MUV. Two markings are consequently valid considering the input and output observed so far. This is reflected in that *curmarkings* will contain both markings, as shown in Equation 6.14.

$$\begin{aligned} curmarkings = \{&\{mp_1 \mapsto \langle 0,0\rangle\}, \\ &\{mp_{2a} \mapsto \langle 0,0\rangle, mp_{2b} \mapsto \langle 0,0\rangle\}\} \end{aligned} \tag{6.14}$$

The next input comes after 20 time units, when a new token appears in p, this time with the value 5. At this moment, time has elapsed since the previous transition firing. When the monitor is in the first marking, with a token in mp_1, only transitions mt_2 and mt_3 are enabled prior to giving the input to the monitor. Those transitions do not have an upper bound on their time delay interval. Therefore, in this marking, time can elapse without problem. However, in the second marking, with tokens in mp_{2a} and mp_{2b}, one transition is enabled, mt_6. Moreover, the upper time bound of that transition is 10 time units. Delaying for 20 time units will exceed this bound. As a conclusion, this marking is not valid and removed from the set of current markings. The input is then added to the remaining marking. The result is shown in Equation 6.15.

$$curmarkings = \{\{mp_1 \mapsto \langle 0,0\rangle, p \mapsto \langle 5,20\rangle\}\} \tag{6.15}$$

With no delay, the token in p is then consumed. As discussed previously this is considered to be an output. In this case, since the value of the token is 5, only three transitions are enabled, mt_2, mt_3 and mt_4. Transition mt_1, is disabled since its guard is not satisfied. Transitions mt_2 and mt_3 do not produce the same output as the MUV (consume the token in p), so they are ignored. Only mt_4 satisfies the output. The resulting set of markings is shown in Equation 6.16.

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 20\rangle, mp_{2b} \mapsto \langle 0, 20\rangle\}\} \tag{6.16}$$

After 2 time units, another token arrives in p, this time with value 7. Advancing time by 2 time units is acceptable from the point of view of the monitor, since the only enabled transition, mt_6, has a higher upper bound, $10 > 2$. It is not explicit in Figure 6.11, but it is now necessary to mention that 2 time units are already used from mt_6, leaving only 8 time units before it has to be fired. The input is added to each (only one in this case) set of current markings, as shown in Equation 6.17.

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 20\rangle, mp_{2b} \mapsto \langle 0, 20\rangle, p \mapsto \langle 7, 22\rangle\}\} \tag{6.17}$$

Next, that new token disappears. Before examining the enabled transitions, all enabled timers must first tentatively be fired, leading to the markings in Equation 6.18.

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 20\rangle, mp_{2c} \mapsto \langle 0, 22\rangle, p \mapsto \langle 7, 22\rangle\}\} \tag{6.18}$$

Next, the token in p is consumed. Three transitions are enabled, mt_5, mt_7 and mt_8. However, only mt_5 satisfies the output. The resulting marking should consequently be stored. Firing transition mt_5 did not involve the timer (mt_6), so before storing the marking, the timer must be unfired, i.e. moving the token from mp_{2c} back to mp_{2b}. This was apparently not the right moment to fire the timer. Equation 6.19 shows the resulting marking.

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 22\rangle, mp_{2b} \mapsto \langle 0, 20\rangle\}\} \tag{6.19}$$

After 5 time units, the MUV produces the output q with value 10. Again, the timer mt_6 is tentatively fired. Three transitions are now enabled, mt_5, mt_7 and mt_8, but only the latter two can produce a valid output. They lead to two different markings. The token in the output place of the timer, mp_{2c}, was consumed so no timer needs to be unfired. The result in shown in Equation 6.20.

$$curmarkings = \{\{mp_1 \mapsto \langle 10, 27\rangle, q \mapsto \langle 10, 27\rangle\}, \{mp_{2a} \mapsto \langle 0, 27\rangle, mp_{2b} \mapsto \langle 10, 27\rangle, q \mapsto \langle 10, 27\rangle\}\} \quad (6.20)$$

The output, q, is then consumed by the environment of the MUV (stimulus generator). Removing a token from an out-port is considered as an input, for which reason it is removed from each marking in *curmarkings*. The remaining set of markings is shown in Equation 6.21.

$$curmarkings = \{\{mp_1 \mapsto \langle 10, 27\rangle\}, \{mp_{2a} \mapsto \langle 10, 27\rangle, mp_{2b} \mapsto \langle 10, 27\rangle\}\} \quad (6.21)$$

The following example will demonstrate how an unsatisfied assertion is detected. Consider the sequence of inputs and outputs in Equation 6.22 and the assertion and monitor in Equation 6.1 and Figure 6.8 respectively.

$$[p = 5, \neg p, \langle \text{delay:}20, q = 10\rangle] \quad (6.22)$$

When the input p, with the value 5, and the output "consuming the token in p" have been processed, the set of current markings in the assertion checker has reached the situation in Equation 6.23.

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 0\rangle, mp_{2b} \mapsto \langle 0, 0\rangle\}\} \quad (6.23)$$

After 20 time units a token in out-port q with value 10 is produced. First, time is elapsed in the monitor. One transition, mt_6, is enabled, and it has an upper time bound of 10 time units. The transition thus exceeds this bound, which makes the marking being discarded. The set of current markings is now empty, which signifies that the assertion is violated.

6.2.7 COVERAGE ENHANCEMENT

The previous sections have discussed issues related to the simulation phase (see Figure 6.4). The simulation phase ends when the stop criterion, which will be discussed in Section 6.2.8, is reached. After that, the validation algorithm enters the coverage enhancement phase, which tries to deliberately guide the simulation into an uncovered part of the state space, thereby boosting coverage. As indicated on Line 8 in Figure 6.4, a coverage enhancement plan has to be obtained. This plan describes step by step how to reach an uncovered, with respect to the particular coverage metrics used, part of the state space. This section describes the procedure to obtain the coverage enhancement plan. Obtaining this plan is the core issue in the coverage enhancement phase.

A model checker returns a counter-example when a property is proven unsatisfied. That is true for ACTL formulas. However, for properties with an existential path quantifier, the opposite holds. A witness is returned if the property is satisfied. A common name for both counter-examples and witnesses is diagnostic trace. For instance, when verifying the property $\mathbf{EF}\varphi$, the model checker provides a trace (witness) which describes exactly step by step how to reach a marking where φ holds, starting from the initial marking. This observation is the centrepiece in the coverage enhancement procedure. The trace constitutes the coverage enhancement plan mentioned previously.

What φ represents depends on the particular coverage metrics used. In our case, the coverage metrics is a mix of assertion coverage and transition coverage as described in Section 6.2.3. The following two sections will go into the details of the peculiarities of enhancing both assertion and transition coverage respectively.

Enhancing Assertion Coverage

Each assertion has an associated activation sequence, as described previously. During the simulation phase, the first markings in the sequence are removed as they are observed in the MUV. When the validation algorithm (Figure 6.4) reaches the coverage enhancement phase, only partial activation sequences remain. The first marking in the sequence with the least number of remaining markings is chosen as an objective, φ, for coverage enhancement.

Assume that no marking in the sequence corresponding to the property in Equation 6.9 has been observed, then the objective would be $p < 20$, i.e. to find a sequence of transitions, such that when fired, would lead to a marking where $p < 20$. The property given to the model checker would therefore be $\mathbf{EF}\ p < 20$. The model checker will then automatically provide the requested sequence of transitions in the diagnostic trace.

Enhancing Transition Coverage

Enhancing transition coverage is about finding a sequence of transitions leading to a marking where a previously unfired transition is enabled and fired. Having found a previously unfired transition, *t*, the property $\mathbf{EF}\ \mathit{fired}(t)$ is given to the model checker. The model checker will then automatically provide a sequence of transitions, which, when fired, will lead to a marking where *t* is enabled and fired. In this way, transition coverage is artificially improved.

The time that the model checker takes to find a coverage enhancement plan depends heavily on which previously unfired transition is chosen for coverage enhancement. It is therefore worth the effort to find a transition which is “close” to the current marking, in the sense that the resulting enhancement plan is short, and hence also the model checking time. Definition 6.6 defines a measure of distance between a marking and a transition or place in PRES+ models. The measure can be used to heuristically find an appropriate transition which leads to a short trace. The measure, in principle, estimates the number of transition firings needed to fire the transition given the marking.

Definition 6.6: Distance. Let M be a marking, V a set of values, $\mathbf{U}$ a universe containing all possible values which can occur in the model ($V \subseteq \mathbf{U}$), t a transition, p a place and c_1 and c_2 predefined constants. $P_t(i)$ denotes the ith input place of t. $dist(t, V, M)$ is recursively defined as:

- If $\{f_t(x_1, \ldots, x_n) | g_t(x_1, \ldots, x_n)\} \cap V = \varnothing$, then

  $dist(t, V, M) = c_1$

  Otherwise,

  $$dist(t, V, M) =$$

  $$\sum_{i=1}^{|{}^\circ t|} dist(P_t(i), \{v \in \mathbf{U} | g_t(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_{|{}^\circ t|}) \wedge$$

  $$f_t(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_{|{}^\circ t|}) \in V \wedge$$

  $$x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{|{}^\circ t|} \in \mathbf{U}\}, M)$$

- If $M(p) \neq \varnothing \wedge M(p)_v \in V$, then $dist(p, V, M) = 0$

  If $M(p) \neq \varnothing \wedge M(p)_v \notin V$, then

  $$dist(p, V, M) = c_2 + \min_{t \in {}^\circ p} \{dist(t, V, M)\}$$

If $M(p) = \varnothing$, then

$$dist(p, V, M) = 1 + \min_{t \in {}^{\circ}p} \{dist(t, V, M)\}$$

■

V is an auxiliary parameter with the initial value $V = \mathbf{U}$. In the case of measuring the distance from a transition t, the set V contains all possible values which can be produced by the transition function f_t. Similarly, in the case of measuring the distance from a place p, V contains all possible values which a token in p can carry.

The distance between a transition t and a marking is defined in two different ways, depending on if there exist parameters to the function f_t, $x_1, \ldots, x_n$, which satisfy the guard g_t, such that the function can produce a value in the set V. If such parameters do not exist, it means that the transition cannot produce the specified values. The distance is then considered to be infinite, which is reflected by the constant c_1. c_1 should be a number bigger than any finite distance in the model.

Otherwise, if at least one value in V can be produced by f_t, the distance of t is the same as the sum of all distances of its input places. The set V contains, in each invocation corresponding to input place p, the values which the function parameter associated to p may have in order for f_t to produce a value in V.

In the case of measuring the distance between a place p and a marking M, the result depends on whether there is a token in that place, and if so, the value in that token. If there is a token in p and the value of that token is in V, the distance is 0. In other cases, the search goes on in all of the incoming paths. For a token to appear in p, it is sufficient that only one input transition fires. The distance is defined with respect to the shortest (in terms of the distance) of them. This case is further divided into two cases: there is a token in p (but with a value not in V), or there is no token in p. In the latter case, 1 is added to the path to indicate that one more step has to be taken along the way from M to p. However, in the former case, a larger constant c_2 is added to the distance as penalty in order to capture the fact that the token in p first has to disappear before a token with a good value can appear in p. The proposed distance heuristic does not estimate further the exact number of transition firings it takes for this to occur.

Figure 6.13 shows an example which will be used to illustrate the intuition of the distance metrics. In the example, the distance between transition t_6 and the current marking (tokens in p_1 and p_2) will be measured. All transition functions are considered to be the identity function.

Transition t_6 has two input places p_6 and p_7. Consequently, in order to fire t_6 there must be tokens in both of these places. The distance of t_6 is therefore the sum of the distances of p_6 and p_7.

In order for a token to appear in p_7, only t_2 needs to be fired. The distance of p_7 is therefore $1 + dist(t_2, \mathbf{U}, M)$. Transition t_2 is already enabled, so the distance of t_2 is 0 (because there is a token in its only input place). The distance of p_7 is hence 1.

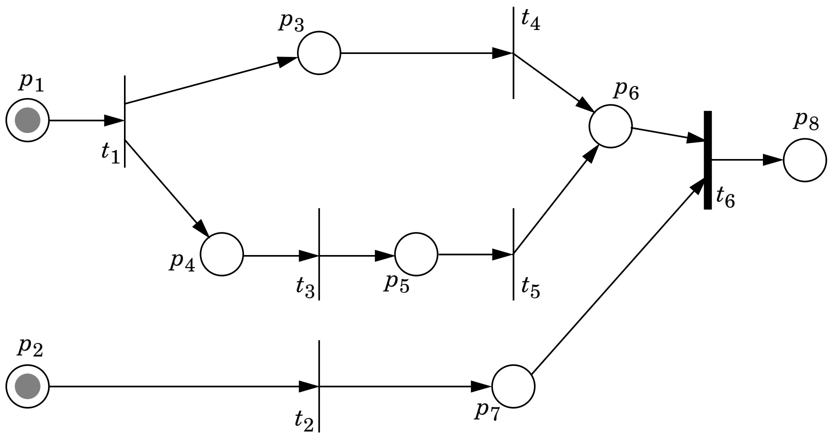


Figure 6.13: Example of computing distance

At p_6, a token may appear from either t_4 or t_5. The distance of p_6 is therefore 1 plus the minimum distance of either transition. The distance of t_5 is 2 (obtained in a similar way as in the case of p_7), while the distance of t_4 is 1. Therefore, $dist(p_6, \mathbf{U}, M) = 1 + \min\{dist(t_4, \mathbf{U}, M), dist(t_5, \mathbf{U}, M)\} = 2$.

Consequently, the distance of t_6 is $dist(t_6, \mathbf{U}, M) = dist(p_6, \mathbf{U}, M) + dist(p_7, \mathbf{U}, M) = 2 + 1 = 3$. Three transition firings are thus estimated to be needed in order to enable t_6.

Given this distance metrics, the uncovered transition with the lowest distance with respect to the current marking may be chosen as a target for coverage enhancement, since it results (heuristically) in the shortest enhancement plan, and it is obtained fast by the model checker.

This procedure can be taken one step further. Not only can the closest transition be chosen, but the smallest transition-marking pair. Among all visited markings and uncovered transitions, the pair with the smallest distance is chosen. When such a pair has been found, the model is reset to the particular marking and the coverage enhancement is performed with respect to that marking.

Although some time has to be spent on finding the transition-marking pair with the smallest distance, it is worth the effort since the time spent in the model checking is reduced significantly. This is the alternative which we have implemented and used in the experiments. However, it is of great importance that the distance computation is as efficient as possible, since it is invoked many times when searching for a good transition-marking pair. In order to avoid long computation times, a maximum search depth can be introduced. When that depth is reached, a constant c_3 is returned, denoting that the distance is big.

Failing to Find a Coverage Enhancement Plan

It might happen that the model checking takes a long time. In such cases, a time-out interrupts this procedure leading to a situation where no coverage enhancement plan could be obtained. When this occurs, the rest of the coverage enhancement phase is skipped, and a new run of the simulation phase is started. The failed assertion or transition will not be target for coverage enhancement again.

6.2.8 STOP CRITERION

Line 3 in Figure 6.4 states that the simulation phase ends when a certain stop criterion is reached. Section 6.2.2 briefly mentioned that the stop criterion holds when a certain number of transitions are fired without any improvement of the coverage. This number is called *simulation length*. It can, however, be very difficult to statically determine a simulation length which minimises total validation time. In this section, a dynamic heuristic approach, where the simulation length is determined at run-time, is presented. Section 6.2.9 will demonstrate that this heuristic yields a comparable coverage as the "optimal" simulation length with little penalty in time for the average case.

Static Stop Criterion

The diagram in Figure 6.14 depicts the relation between the simulation length and the total validation time. The graph shows the result of an example which has been validated with different values on the simulation length. Each diamond (marked "sample") in the figure corresponds to one such run. The example was validated 3 times per simulation length[1]. The average of the three results belonging to the same simulation

1. Since there exists non-determinism both in the MUV and in the methodology, the verification times may differ even if the same simulation length is used.

length is also marked in the graph. The averages for each simulation length are connected by a line, in order to make the trend more visible[1].

For small simulation lengths, the validation time is quite high. This is due to the fact that not many transitions (or assertions) are covered in the simulation phase. Therefore, the model checker has to be invoked frequently to find enhancement plans for each of the uncovered transitions or assertions. This action is expensive. On the other hand, for big simulation lengths, a lot of time is spent in the simulation phase without any contribution to coverage. Between these two extremes, there is a point at which the least possible amount of time is wasted in the simulation phase and coverage enhancement phase respectively. For

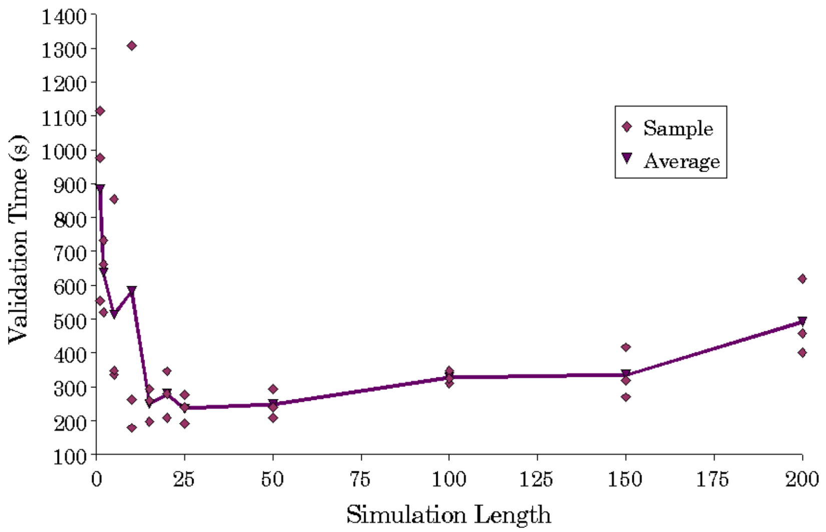


Figure 6.14: Relation between simulation length and validation time

1. All instances have been run until their actual finish according to Line 2 in Figure 6.4. This might, in general, lead to situations where the obtained coverage does not reach 100%. However, in this particular case, 100% coverage is reached everywhere.

the example given in Figure 6.14, that point is somewhere between 15 and 30.

There are basically two factors that influence the location of this point: the size of the MUV (or actually its state space) and the sizes of the assertions (or actually the state space of their monitors). The bigger the size of the MUV is, the longer is the "optimal" simulation length, since more time will be needed for model checking in the coverage enhancement phase. Similarly, the bigger the sizes of the assertions are, the shorter is the "optimal" simulation length, since more time has to be spent in assertion checking in the simulation phase.

However, it is impossible to obtain the best simulation length before-hand without first validating the MUV multiple times. For this reason, a dynamic method which finds this value at runtime, while paying attention to coverage, is desired.

Dynamic Stop Criterion

As concluded in the previous section, the total validation time depends on the two factors: MUV size and assertion size. These factors influence model checking time and transition firing time respectively. The total validation time consequently is directly dependent on these factors. The following discussion will derive a function describing the total validation time in terms of these factors. This function will later be used to analytically determine the simulation length which most likely minimises validation time.

The total time the validation process spends in the coverage enhancement phase is the sum of the times of all model checking sessions. For a given simulation length, σ, and average model checking time (including the time to find a marking-transition pair with small distance), t_{ver}, the time spent in the coverage enhancement phase can be expressed as stated in Equation 6.24.

T is the set of transitions in the MUV and A is the set of assertions. $cov(\sigma)$ is a function expressing how big coverage would have been achieved at a certain simulation length σ, if the verification only is performed with the simulation phase, i.e. no coverage enhancement takes place. $1 - cov(\sigma)$ thus denotes how big percentage of the state space has not been covered. It is assumed that this part has to be targeted in the coverage enhancement phase. Multiplying this number with the total number of transitions and assertions, $(1 - cov(\sigma)) \cdot |T \cup A|$, yields the number of transitions and assertions which are not yet covered by the simulation phase and need to be targeted by the coverage enhancement phase. Given that t_{ver} represents the average time spent in one coverage enhancement phase, it is straightforward to conclude that the expression in Equation 6.24 approximates the total time spent in the coverage enhancement phase.

$$t_{enh}(\sigma) = (1 - cov(\sigma)) \cdot |T \cup A| \cdot t_{ver} \tag{6.24}$$

The time spent in the simulation phase depends mainly on the assertion checking time. The assertion checking procedure is invoked after each transition firing. The total time spent in the simulation phase is hence linear to the number of transitions fired. The number of transitions fired can be approximated with the simulation length, in particular for large values. Equation 6.25 shows the corresponding expression, where t_{fir} is the average time it takes to fire a transition, including the subsequent assertion check.

$$t_{sim}(\sigma) \approx t_{fir} \cdot \sigma \tag{6.25}$$

The approximate total validation time is the sum of $t_{enh}(\sigma)$ and $t_{sim}(\sigma)$, as shown in Equation 6.26.

$$\begin{aligned} t_{tot}(\sigma) &= t_{enh}(\sigma) + t_{sim}(\sigma) = \\ &(1 - cov(\sigma)) \cdot |T \cup A| \cdot t_{ver} + t_{fir} \cdot \sigma \end{aligned} \tag{6.26}$$

The goal is to find a suitable simulation length σ, such that $t_{tot}(\sigma)$ is minimised. In Equation 6.26, there are three parameters whose values are unknown prior to validation, $cov(\sigma)$, t_{ver} and t_{fir}. The latter two can relatively straightforwardly be obtained by measuring the time it takes to run the model checker or fire and assertion check a transition respectively. Those parameters do not depend on the simulation length, but remain fairly constant throughout the entire process.

Before the first simulation phase, t_{ver} must be assigned an initial estimated value since it is too computationally expensive to invoke the model checker in order to obtain an authentic value. Not until after the first coverage enhancement phase, an authentic value has been obtained which can be used in the subsequent simulation phases. This initial value should depend on the size of the MUV.

Since it is relatively inexpensive to fire a transition and assertion check the result, t_{fir} does not need to be assigned an initial default value. After a few iterations in the simulation phase, an authentic value can quickly be obtained.

In order to determine the function $cov(\sigma)$, the following experiment is performed. First run the simulation phase with simulation length 1, and note down the obtained coverage in a diagram. Next, do the same with simulation lengths 2, 3, 4, etc. In practice, it is not necessary to run the simulation separately for each simulation length, but the coverage for each simulation length can be obtained in one single run.

The experiment will show that the longer the simulation length is, the higher is the obtained coverage. For short simulation lengths, many uncovered transitions and assertions are encountered on each simulation length, resulting in a rapid rise in coverage. The longer the simulation length, the less additional uncovered transitions and assertions are encountered compared the previous simulation length. This function is, thus, exponential.

It is, however, practically infeasible to empirically derive this function by extensive simulation. By doing that, a large part of the system has already been verified and the use of knowing the coverage function, in order to minimise total validation time, is severely diminished.

Since we cannot know the exact shape of the $cov(\sigma)$ function in advance, it has to be estimated. As the simulation phase progresses, more and more data about the coverage function can be collected, in the same way as the experiment described previously. The data collected for short simulation lengths obtained early during simulation, can be used to predict the coverage for longer simulation lengths.

Figure 6.15 shows a diagram in which the coverage obtained at each simulation length is marked (Sample) for a certain run. The graph only shows the results referring to one single simulation phase. Such diagrams follow in general an exponential curve of the form $cov(\sigma) = Ce^{D\sigma} + E$, where both C and D are

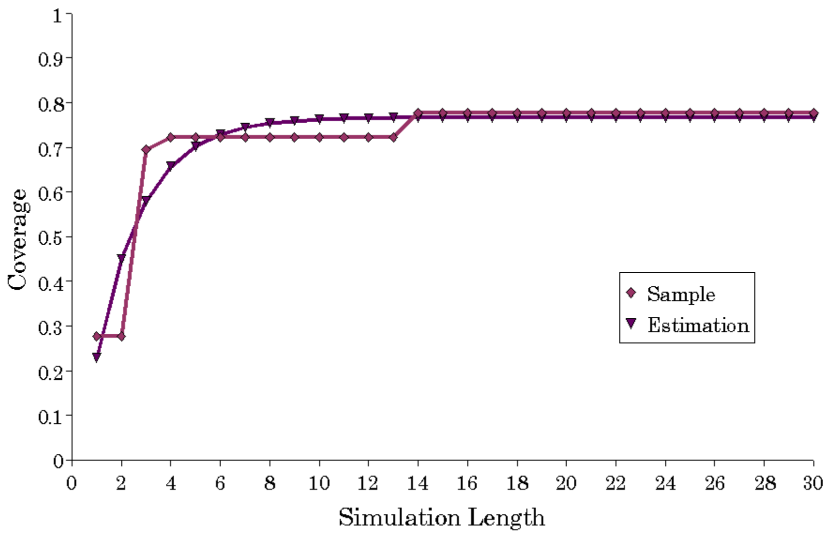


Figure 6.15: Relation between simulation length and coverage

negative. In order to obtain an estimation of the coverage function for instant simulation lengths longer than the one currently reached, these three parameters (C, D and E) have to be extracted from the points obtained from the simulation so far. This extraction is performed using the *Least Square Method*, under the constraint that $E \leq 1$. That method minimises the distance between the sample points (measured coverage) and the estimated curve.

Given the points in Figure 6.15, the least square method gives us that $C = -0.9147$, $D = -0.53$ and $E = 0.7668$. The resulting exponential curve is shown in the same figure. Having obtained this function, Equation 6.26 can be rewritten as Equation 6.27.

$$t_{tot}(\sigma) = (1 - Ce^{D\sigma} - E) \cdot |T \cup A| \cdot t_{ver} + t_{fir} \cdot \sigma \qquad (6.27)$$

Provided the expression of total validation time in Equation 6.27, it is straightforward to find the simulation length corresponding to the shortest validation time with analytical methods. The resulting expression is presented in Equation 6.28.

$$\sigma = \frac{\ln \dfrac{t_{fir}}{C \cdot D \cdot |T \cup A| t_{ver}}}{D} \qquad (6.28)$$

In the example of Figure 6.14 and Figure 6.15, $t_{fir} = 0.1255$, $t_{ver} = 33.36$ and $|T \cup A| = 36$. Using the formula in Equation 6.28 gives us that the optimal simulation length is 16, as computed in Equation 6.29.

$$\sigma = \frac{\ln \dfrac{0.1255}{(-0.9147) \cdot (-0.53) \cdot 36 \cdot 33.36}}{-0.53} \approx 16 \qquad (6.29)$$

For each new simulation phase (Line 3 in Figure 6.4), a new simulation length is calculated according to Equation 6.28. The new value will be more accurate than the previous ones, as more data, on which the calculation is based, have been collected.

The only reason for not reaching 100% coverage with this simulation technique, is that obtaining a coverage enhancement plan took too long time and timed out. In such cases, the parameter t_{ver} will be large. Analysing Equation 6.28 gives that the larger t_{ver} is, the larger simulation length σ should be chosen (considering that both C and D are negative, and that the quotient inside the logarithm generally is less than 1). As a consequence, when more and more coverage enhancement attempts fail, the simulation phase becomes longer. Spending more time in the simulation phase results in a bigger collection of encountered markings (states), which increases the probability of finding a transition-marking pair with a small distance (Definition 6.6), and thereby influences coverage in a positive way.

6.2.9 EXPERIMENTAL RESULTS

The proposed formal method-aided simulation technique has been tested on a variety of models and assertions. The results are presented in Table 6.5. The table compares the time needed and the coverage obtained using the static and dynamic stop criterion respectively. The values given are the average values of several runs on the same model and assertions. The complexity of the models is given in terms of number of transitions. Although this number is generally not enough to characterise the complexity of a model, it still provides a hint about the size of the model and is used due to the lack of a more accurate, but still concise, metrics. Examples 1 through 27 consist of randomly generated models. Examples 28 through 33 model a control unit for a mobile telephone, traffic light controller and a multiplier respectively. Each model is verified for two different properties.

Table 6.5: Experimental results

Ex.	Trans.	Time (s) Dyn.	Time (s) Static	Time Diff. (%)	Coverage (%) Dyn.	Coverage (%) Static	Cov. Diff (%)
1	28	22.67	24.54	-7.62	100	100	0.00
2	28	51.75	38.30	35.12	100	100	0.00
3	35	42.40	39.74	6.69	100	100	0.00
4	35	62.55	60.78	2.91	100	100	0.00
5	42	55.38	58.77	-5.77	100	100	0.00
6	42	82.67	78.71	5.03	100	100	0.00
7	49	70.45	80.42	-12.40	100	100	0.00
8	49	101.64	105.16	-3.35	100	100	0.00
9	56	93.60	378.28	-75.26	100	99	1.01
10	56	143.36	120.73	18.74	100	100	0.00
11	63	137.14	289.89	-52.69	100	99	1.01
12	63	157.23	161.75	-2.79	100	100	0.00
13	70	298.81	151.43	97.33	99.5	100	-0.50
14	70	345.21	196.22	75.93	99.5	100	-0.50
15	7	6.29	4.43	41.99	100	100	0.00
16	14	261.98	399.91	-34.49	95	95	0.00
17	14	270.23	277.01	-2.45	95	95	0.00
18	21	627.27	550.78	13.89	95	94	1.06
19	21	891.39	821.83	8.46	89	90	-1.11
20	7	7.57	4.66	62.45	100	100	0.00
21	7	16.41	10.49	56.43	100	100	0.00
22	14	253.19	240.65	5.21	98	95	3.16
23	14	265.08	388.45	-31.76	93	95	-2.04
24	30	15.27	10.42	46.55	100	100	0.00
25	75	119.37	93.06	28.27	100	100	0.00
26	150	564.54	504.37	11.93	100	100	0.00
27	225	1768.35	1604.84	10.19	100	100	0.00
28	31	1043.97	935.68	11.57	98	99	-1.01
29	31	599.19	417.30	43.59	95	100	-5.00
30	36	216.41	157.01	37.83	100	100	0.00
31	36	279.46	250.10	11.74	100	100	0.00
32	8	13.12	10.21	28.50	100	100	0.00
33	8	330.47	316.21	4.51	100	100	0.00

It should be emphasised that the simulation length used for the static stop criterion was obtained by empirically evaluating several different values, finally choosing the one giving the shortest validation time for comparison. The performance given by this method can consequently not be achieved in practice, and only serves as a reference.

As can be seen, in most cases using the dynamic stop criterion results in validation times close to those for the static stop criterion. There exist however cases where there is a big difference. This situation occurs if one method did not reach as high coverage as the other and, thus, had more timeouts in its coverage enhancement phase, thus adding to total time.

It can be deduced from the figures that, in average, the dynamic approach is 15% slower than using the static stop criterion (if that one could be used in practice, which is not the case). However, in 30% of the cases, the dynamic approach was actually faster. This situation may occur, since choosing the simulation length for the static stop criterion is not a very accurate process. It could happen that the dynamic stop criterion finds a simulation length closer to the actual optimum. The loss in coverage is, on the other hand, very small. In average, coverage is 0.12% lower using the dynamic approach.

Although the dynamic approach performs slightly worse on both aspects, it should be remembered that, in practice, it is impossible to reach the values listed in the table with the static approach. As mentioned previously, the values were obtained by trying several alternatives for the static simulation length, thus validating the system multiple times. It cannot be known in advance which simulation length results in the shortest validation time.

All models and all assertions have in addition been validated with pure model checking. The model checker found a solution, but in one order of magnitude longer time than the proposed approach. In some of the cases the model checker ran out of memory (2GB) before any result could be delivered.

The models were moreover also verified with pure simulation. The simulation went on for as long time as was used by the dynamic stop criterion. The total coverage obtained by the simulation process after this time, was always less than or, in some cases, equal to our proposed mixed approach.

PART III
Integration Verification

Chapter 7 Integration Verification Methodology

THIS CHAPTER PROVIDES an overview of the proposed integration verification methodology. It emphasises the particular problems related to such verification, also briefly mentioned in Chapter 4. The details are explained thoroughly in Chapter 8 and Chapter 9.

The chapter begins with introducing an example used to explain the verification process and its challenges.

7.1 Explanatory Example

To illustrate the methodology, an example of a military aircraft, built on the General Avionics Platform (GAP) [Loc91] model, is presented.

The system is centred around one single component, the MCC (Mission Control Computer). All other components communicate only with the MCC which then coordinates all requests and responses. Besides the MCC, the system consists of the following components: Radar, Display & Controls, Tracker and Weapon.

The Radar component repeatedly sends signals, with a regular time interval, concerning the current situation in the sky to keep other components updated. The Display & Controls component displays the information it receives from the radar, via the MCC, on a screen. It also notifies the MCC about the status of the controls, for instance if a "fire" command is issued. The Tracker component, when activated, traces one single enemy plane and issues orders to Weapon to aim at it. The Weapon component receives aiming and firing instructions.

The whole setting is illustrated in Figure 7.1 at a high level of abstraction, not taking communication details into account. Messages sent by one component are delivered to the recipient without loss. However, we would also like to specify and model the communication mechanism through which the components interact. A single segment LAN is chosen for this purpose. The selected protocol is connection based. This yields the situation in Figure 7.2, where the LAN is placed in the centre between the components and the protocol adapters. Note that from a formal and methodological point of view, all boxes in the figure including the LAN and the protocol handlers are also components.

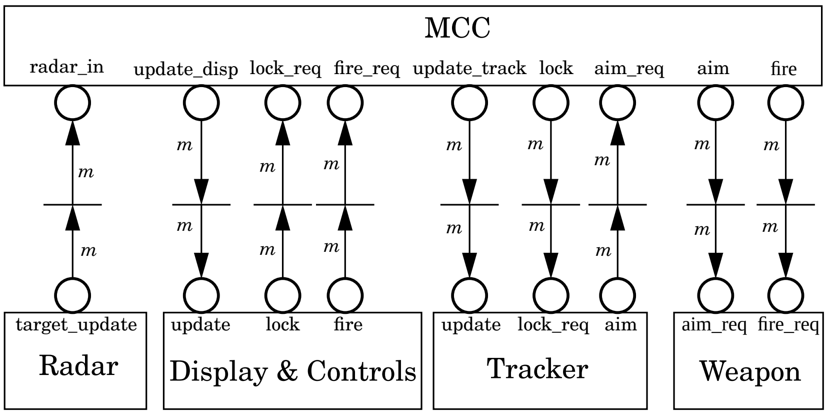


Figure 7.1: A high level model of the GAP example

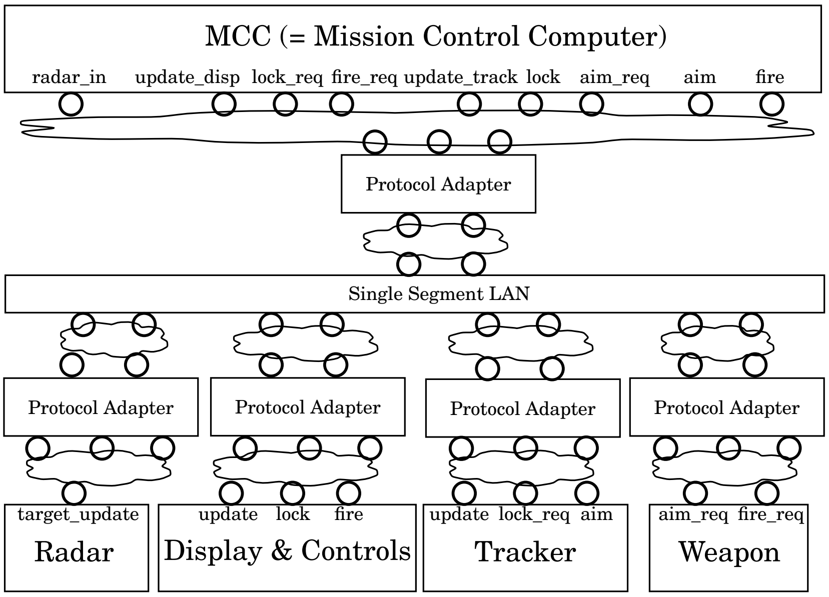


Figure 7.2: Refined GAP model

What remains to be added is the glue logics, represented by the clouds between the components.

As mentioned previously, a connection based protocol is used in the design. However, the components in the high-level model in Figure 7.1 are not designed to communicate over such a protocol. Thus, the functionality of establishing and maintaining a connection must be added in the glue logic. The same glue logic also has to handle errors in case the connection was refused. The model of such a glue logic between the Radar and its Protocol adapter can be seen in Figure 7.3 (the time delay intervals on the transitions are not shown in the figure for the sake of readability).

Before Radar can send any message, the glue logic must connect to the MCC. This is reflected in the figure by the fact that transition t_2 is not enabled until the protocol reported that it has successfully been connected and a token appears in p_2. To

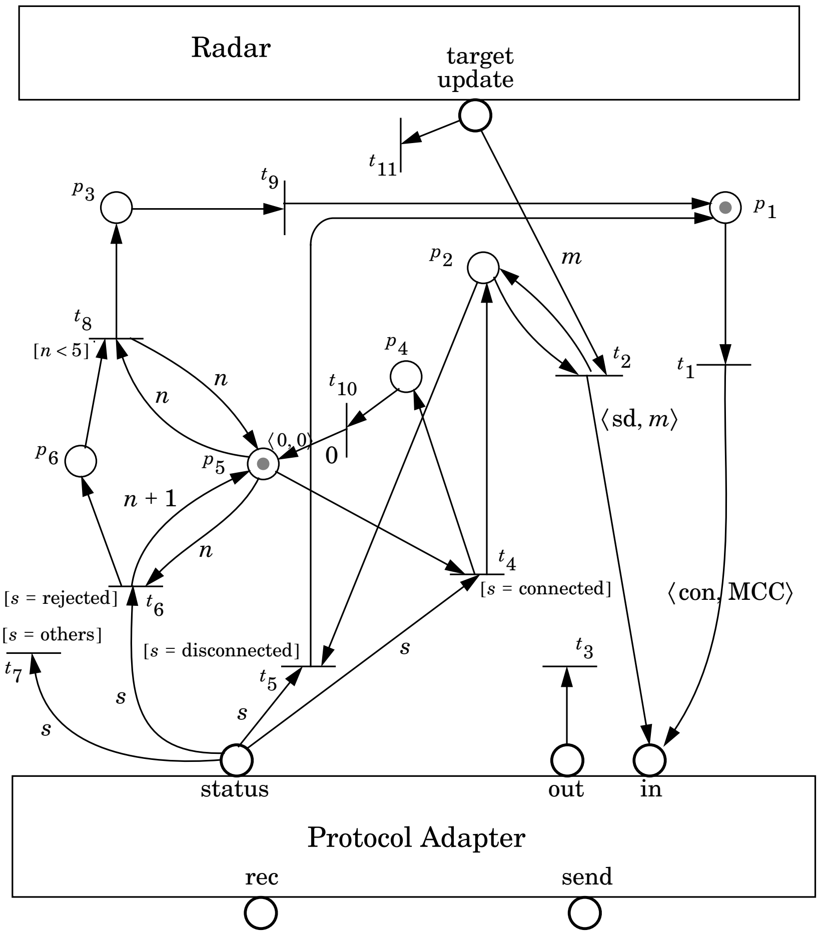


Figure 7.3: The glue logic between Radar and its Protocol

achieve this, a token with value $\langle \text{con}, \text{MCC} \rangle$ is passed to the Protocol adapter (t_1), indicating that a connection to component MCC is requested. When the connection is established, $\langle \text{sd}, m \rangle$ will be passed to the Protocol adapter (t_2). The first element of the tuple is a command to the protocol ("sd" is a shorthand for "send") and the second element is an argument to the command.

Here the argument is a tuple of the destination of the message and the message itself.

If, however, the connection failed (t_6), the glue logic will continue to attempt to connect, at most five times (t_8). It has been decided by the designer that it is always the peripheral components (not the MCC) which initiate any connection requests. The MCC, on the other hand, must always listen for connection requests from the other components.

7.2 Objective and Assumptions

The objective of the proposed methodology is to verify the interface properties of the components.

The methodology is based on the following three assumptions:

- The components themselves are already verified.
- The components have some requirements on their interfaces expressed in a formal notation.
- A particular model of the component is provided which is used in the verification process to represent the component.

The first assumption states that the components themselves are already verified by their providers, so they are considered to be correct. What remains to be verified is the interaction between the components through the glue logic, i.e. the integration.

According to the second assumption, the components impose certain requirements on their interfaces, which the environment must satisfy in order for the component to function correctly. The requirements are expressed with (T)CTL formulas, described in Section 3.3, in terms of the ports in the specific interface. It is important to note that these formulas do not describe the behaviour of the component itself, but rather describe how the component requires the rest of the system (its environment) to behave in order to work correctly.

Review the example introduced in Section 7.1. The communication protocol chosen in the example was connection based (Figure 7.2). A Protocol adapter implementing the chosen protocol was supplied and verified by a provider. (T)CTL formulas describing the expected input on each interface of the component were also supplied.

Two of the formulas provided together with the Protocol adapter component are:

$$\begin{array}{l}\mathbf{AG}\ ((status = \text{disconnected} \vee \text{init}) \rightarrow \\ \mathbf{A}\ [status = \text{connected}\ \mathbf{R}\ \neg in = \langle \text{send}, \_ \rangle])\end{array} \quad (7.1)$$

$$\begin{array}{l}\mathbf{AG}\ (status = \text{connected} \rightarrow \\ \mathbf{A}\ [status = \text{disconnected}\ \mathbf{R} \\ (\neg in = \langle \text{connect}, \_ \rangle \wedge \neg in = \langle \text{listen}, \text{-} \rangle)])\end{array} \quad (7.2)$$

Equation 7.1 states that the protocol can never receive a send command when it is disconnected. Equation 7.2 requires that as long as the protocol is already connected, it is prohibited to connect again. Note that all formulas are expressed only using values on the ports of one interface. In this example, the interface is considered to be formed by all ports of the Protocol adapter connected to the Radar through the glue logic.

The third assumption states that in order to verify the interface, models of the components connected to the interface are needed. Such models are called *stubs* and are formally defined in Chapter 8.

Consider the Protocol adapter component in Figure 7.3. The glue logic is connected to the interface $I = \{in, out, status\}$, but the component has more ports than those in this interface, namely the ports $send$ and rec. The behaviour of the ports in I depends actually also on the token exchange through these other ports. Consequently, a mechanism to abstract away unattached ports, in this case $send$ and rec, is needed.

Figure 7.4 shows how a simple stub for interface I of the Protocol adapter might look like. When the Protocol receives a connect (con) or listen (lis) command in port in, transition s_1

becomes enabled. In the real component, the response to such a request is the result of token exchange on the ignored ports. However, since those ports are abstracted away in the stub, the result of this exchange is considered non-deterministic from the point of view of interface I. This non-determinism is modelled in Figure 7.4 with the conflicting transitions s_4 and s_5. The

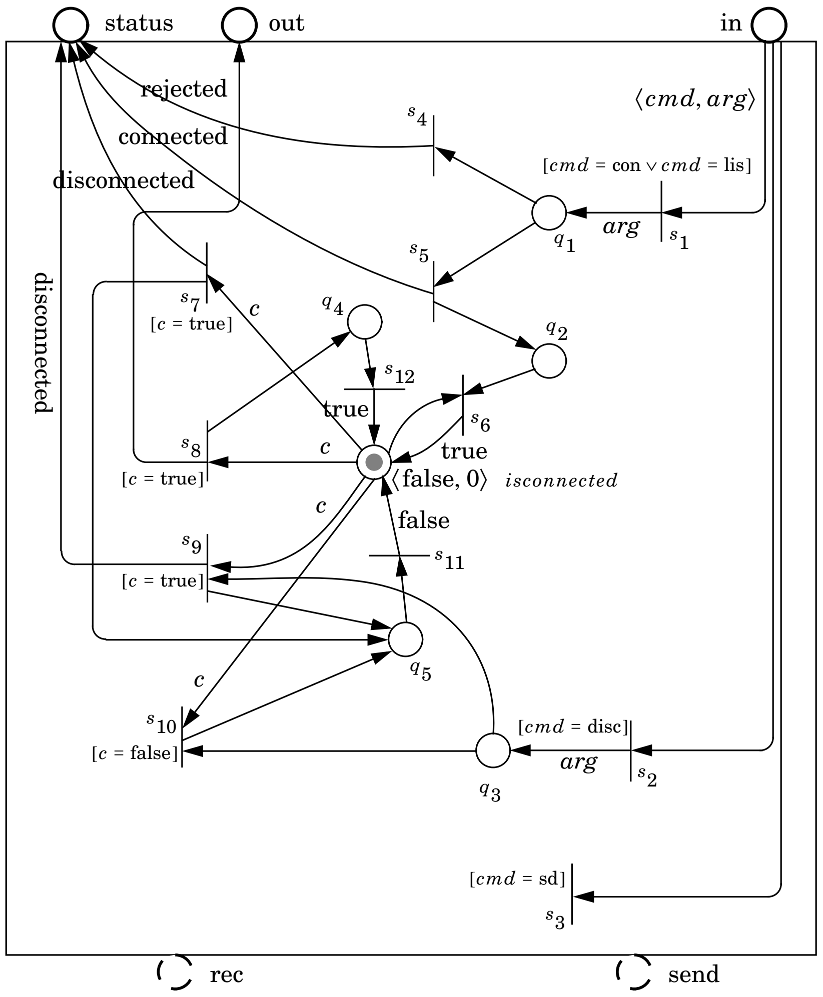


Figure 7.4: A simple stub of the Protocol adapter

response can either be ”rejected” or ”connected”. When connected, messages can be received from the party to which the component is currently connected. Transition s_8 models the receive behaviour, by emitting tokens to port *out* . It is, however, only able to do so when the component is connected. Analogously, send commands (sd) are simply consumed (transition s_3). Disconnect commands (disc) are taken care of similarly by transitions s_9 and s_{10} , depending on whether the Protocol was previously connected or not. Transition s_7 takes care of the case where the other party disconnects.

7.3 The Impact on Verification Using Different Stubs

Since a component has several interfaces, it has naturally also several stubs. This fact can be exploited by the verification process in order to reduce verification time.

Consider the situation in Figure 7.5. The system consists of two components, Doubler and Strange, and there is a glue logic connecting them. Doubler accepts a token with an integer value at in-port *arg*. In response, it will issue a token at out-port *output* with the value two times the value it received. Component Strange will issue one token on out-port *action* as an answer to

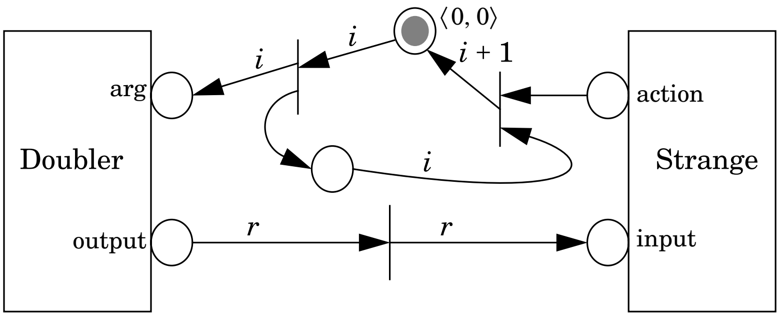


Figure 7.5: Example for stub demonstration

each token it receives on in-port *input*. The glue logic will provide the Doubler with an argument, starting with value 0 and increasing each time by one. The reply of the Doubler is given to Strange which will acknowledge by issuing a token on out-port action, which in turn will cause a new integer to eventually be provided to the Doubler.

Figure 7.6 lists a set of stubs corresponding to the example in Figure 7.5. The stub for interface $\{arg\}$ simply consumes any token which arrives, and the stub for $\{output\}$ produces tokens

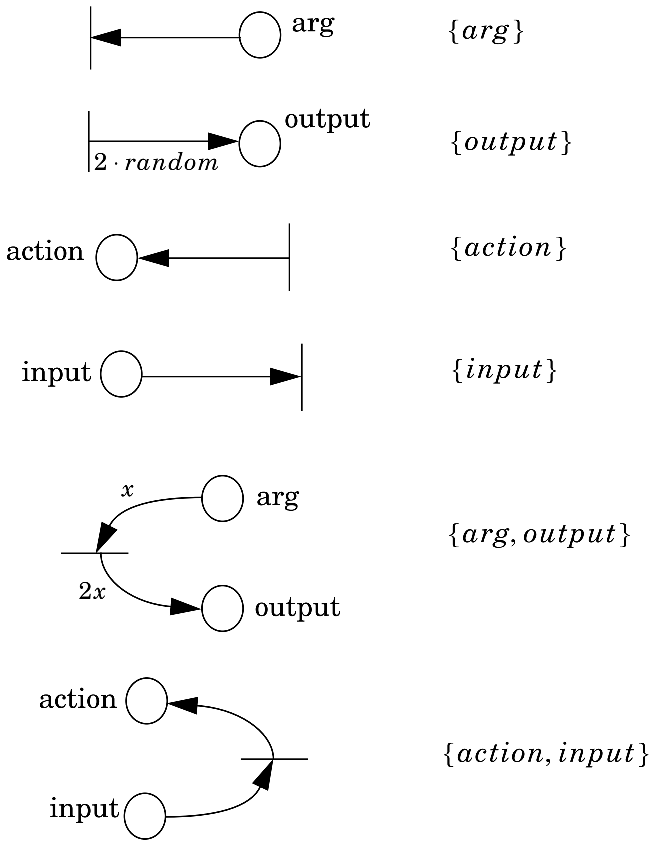


Figure 7.6: Stubs used in the example in Figure 7.5

with only even token values since Doubler only produces even values as a result of its input. $\{action\}$ and $\{input\}$ consumes and produces tokens respectively. No other behaviour can be observed by only looking at one individual port of Strange. The stubs for interfaces $\{arg, output\}$ and $\{action, input\}$ contain all ports of their respective components. Consequently, their stubs model the full component.

Let us elaborate on how this variety of stubs can be exploited for verification considering the following formulas:

$$\mathbf{AG}\ (input \rightarrow even(input)) \tag{7.3}$$

$$\mathbf{AG}\ (arg \rightarrow \mathbf{A}\ [arg\ \mathbf{U}\ \mathbf{A}\ [output\ \mathbf{R}\ \neg arg]]) \tag{7.4}$$

$$\mathbf{AG}\ (arg \rightarrow arg \geq 0) \tag{7.5}$$

$$\mathbf{AGEF}\ input < 0 \tag{7.6}$$

To check Equation 7.3 (if there is a token in place *input*, then the value of that token must be an even number), only the stubs for the interfaces $\{output\}$ and $\{input\}$ are needed. $\{input\}$ is needed because tokens must be consumed in order to obtain a deadlock-free system. $\{output\}$ is enough to produce tokens with only even numbers. The satisfiability of the property does not depend on the input on port arg. More complicated stubs like $\{arg, output\}$ and $\{action, input\}$ can also be used to obtain a correct result. However, as will be discussed in Chapter 8, using fewer and smaller stubs may reduce the verification time.

Equation 7.4 (if one argument is received by Doubler, another argument may not arrive until the result of the first one is produced), requires all ports to be included in the stubs since the causality between the ports is important for the property. Hence, stubs corresponding to the interfaces $\{arg, output\}$ and $\{action, input\}$ must be used. Equation 7.5 (if there is a token in place *arg*, then the value of that token is non-negative) can be checked using any set of stubs, since this property is satisfied due to a mechanism in the glue logic which does not depend on any component.

Let us look at Equation 7.6 (there is always a possibility that a negative value may arrive at port *input*) which obviously is not satisfied. However, if stubs with interfaces containing only a single port are used, the verification will indicate that the formula is satisfied, since the stub corresponding to interface $\{output\}$ may produce negative numbers. But if the stub corresponding to $\{arg, output\}$ is used, the verification will point out that the property is not true, which is the correct conclusion. Using simple stubs on this formula results in the property being satisfied whereas it is unsatisfied in reality, which is proven using more complex stubs. The situation for the other properties is that the properties are unsatisfied using simple stubs, whereas they are satisfied in reality, which is proven using more complex stubs. The reason for this difference is that Equation 7.6 is not an ACTL formula as opposed to the other formulas.

It is obvious that using the stubs covering all ports connected to the glue logic (called *top-level stubs*) for all components, we will get a correct verification for properties specified by any formula. However, we have many different stubs for each component. Thus, the following question has to be answered: Do we always have to use the top-level stubs in order to verify a certain formula? If the answer is "no", then which stub or combination of stubs to use for verification? These questions are of both theoretical and practical importance. From the practical point of view, selecting a certain combination of stubs can reduce the complexity of the verification process and, by this, the verification time. On the other hand, it can happen that certain stubs, possibly the top-level ones, are not available. Thus, it is important to provide a theoretical platform which allows designers to decide if it is possible to perform a correct verification with a certain combination of available stubs. This theoretical framework will be described in Chapter 8.

It could be the case, though, that the property being verified depends on a specific feature of the environment of the component, so that the behaviour described by the stubs is too general. We assume that these additional features are described as logic formulas capturing constraints related to ports not in the interface under verification. In such cases, it is possible to construct a model corresponding to these logic formulas. These models are then included in the verfication process together with the components. An algorithm to construct such a model for PRES+ is presented in Chapter 10.

7.4 Verification Methodology Roadmap

In order to support the designer, it is necessary to introduce some structure to the verification process, so that the designer clearly knows the sequence of steps to follow and if the results obtained at a certain moment are valid or not. If the results turn out not to be valid, the verification process should suggest what should be done in order to obtain a valid result. For this purpose, a roadmap has been developed. It should work as a guideline which the designer can follow to obtain good results in reasonable time.

The roadmap will be presented in the rest of the thesis as the particular aspects of the integration verification process are discussed in more detail.

The integration verification methodology consists of two main parts, presented in Chapter 8 and Chapter 9 respectively. The first part assumes that stubs are already given by the component providers. The problem is then to find the most appropriate set of them. The second part assumes that a model of the whole component is provided and that appropriate stubs can be automatically generated given this model.

Since the methodology includes these two distinct parts, the first question, in the roadmap, to be answered by the designer is

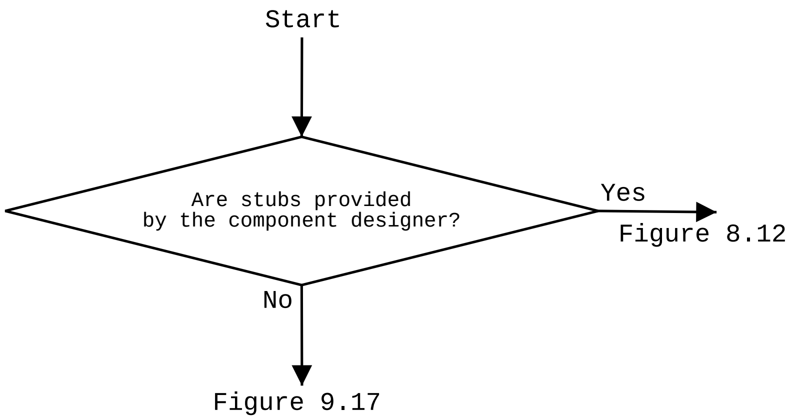


Figure 7.7: The start of the roadmap

intended to guide the verification into either part. Figure 7.7 presents the first question.

In the roadmap presented in this thesis, diamonds denote Yes and No questions to be answered by the designer. Depending on the answer for a particular case, different paths are taken as indicated on the edge of the diamond. Squares denote activities which have to be performed. Rounded squares (ovals) denote terminals where a verification result is reached.

Since there are, in general, several components connected to the glue logic under verification, stubs must be selected or created for each of them. Consequently, there is one instance of the roadmap for each stub or component. The instances are followed independently of each other, with synchronisation points where the actual verification itself takes place.

For example, one component already has stubs provided together with it, and another component does not, so they have to be created given the model of that component. A (set of) stub to represent each connected component must have been selected or created when the actual verification of the interface is performed.

The following chapters will now go into the details of verification methodology.

Chapter 8 Verification of Component-based Designs

IN THIS CHAPTER the theoretical framework underlying the integration verification methodology is presented. It gives formal definitions and presents important properties and relations. Experiments have also been performed. The chapter ends with a continuation of the roadmap introduced in Section 7.4.

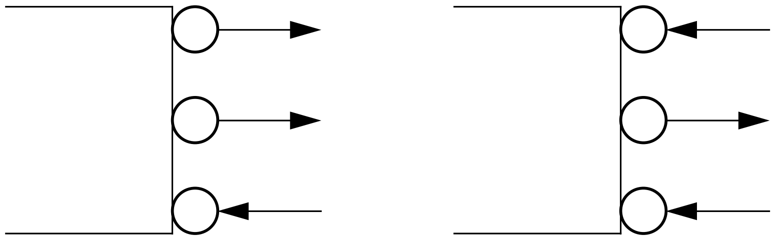
8.1 Definitions

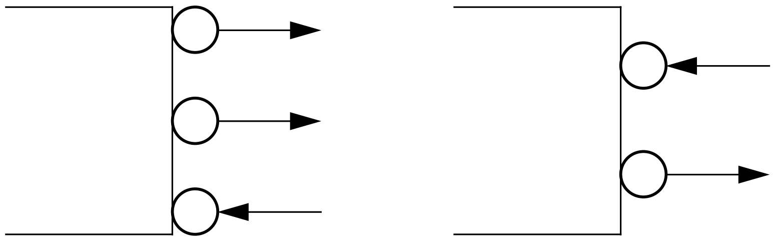
In Section 7.2 it was concluded that some representation of the components is necessary in the integration verification process. We have previously called such a representation describing the behaviour at an interface of a component "stub". In this section, a mathematical definition of a stub be given. Before defining a stub, some auxiliary concepts have to be defined.

Definition 8.1: Interface compatibility. Interfaces I_1 and I_2 are compatible if and only if there exists a bijection $f{:}I_1 \rightarrow I_2$ such that if $f(p) = q$, then p and q are both either in-ports or out-ports in their respective interface.

■

Remembering that interfaces are sets of ports (Definition 3.5), it is intuitive to see that two interfaces are compatible if they have equally many in-ports and equally many out-ports. Figure 8.1


(a) Incompatible interfaces


(b) Incompatible interfaces

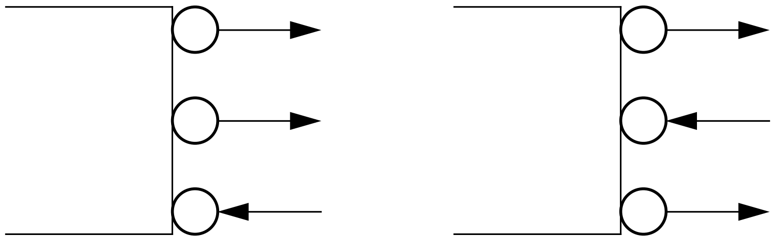

(c) Compatible interfaces

Figure 8.1: Illustration of interface compatibility

illustrates this concept further. The interfaces in Figure 8.1(a) are not compatible since the left-hand component has two out-ports and one in-port, whereas the situation in the right-hand component is the reverse. The interfaces in Figure 8.1(b) contain a different number of ports and are thus not compatible either. Only the interfaces in Figure 8.1(c) are compatible, since they have an equal number of in-ports and out-ports respectively.

Definition 8.2: Event. An *appearing event* is a tuple $e^+ = \langle p, k \rangle$, where p is a place and $k = \langle v_k, r_k \rangle$ is a token. An appearing event represents the fact that a token k with value v_k is put in place p at time moment r_k. A *disappearing event* is a tuple $e^- = \langle p, r \rangle$ where p is a place and r is a timestamp. A disappearing event represents the fact that a token in place p is removed at time r. Observe that for disappearing events we are not interested in the token value. An *event* e is either an appearing event or a disappearing event.

■

Definition 8.3: Observation. An observation o is a set of events $o = \{e_1, e_2, \ldots\}$. Given observation o and an interface I, the *restricted* observation $o|_I = \{\langle p, k \rangle \in o | p \in I\} \cup \{\langle p, r \rangle \in o | p \in I\}$. An *input* observation in is an observation which only contains appearing events defined on in-ports and disappearing events defined on out-ports. An *output* observation out is an observation which only contains appearing events defined on out-ports and disappearing events defined on in-ports.

■

Figure 8.2 illustrates the concept of observations according to Definition 8.3. The figure shows the flow of events as described by observation o, defined in the figure. Initially, at time $t = 0$, the ports do not contain any token. The observation states that a token with value 2 appears in port p at time moment 1. At time

$o = \{\langle p, \langle 2, 1\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle, \langle p, 4\rangle, \langle p, \langle 3, 8\rangle\rangle, \langle p, 9\rangle, \langle q, 9\rangle\}$

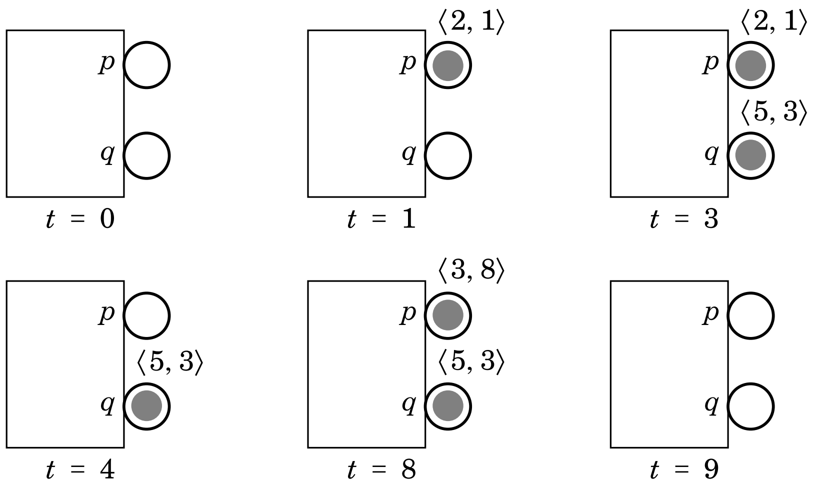


Figure 8.2: Illustration of observations

$t = 3$ another token appears in q and at time $t = 4$ p disappears. A token with value 3 then appears in port p at $t = 8$ and at time $t = 9$ both tokens in p and q disappear.

The restricted operation of o with respect to interface $\{p\}$ is $o|_{\{p\}} = \{\langle p, \langle 2, 1\rangle\rangle, \langle p, 4\rangle, \langle p, \langle 3, 8\rangle\rangle, \langle p, 9\rangle\}$ and the one restricted with respect to $\{q\}$ is $o|_{\{q\}} = \{\langle q, \langle 5, 3\rangle\rangle, \langle q, 9\rangle\}$. Moreover, in this particular case, $o|_{\{p, q\}} = o$.

Assuming that p is an in-port and q is an out-port, then the observation $in = \{\langle p, \langle 2, 1\rangle\rangle, \langle p, \langle 3, 8\rangle\rangle, \langle q, 9\rangle\}$ is an input observation and $out = \{\langle q, \langle 5, 3\rangle\rangle, \langle p, 4\rangle, \langle p, 9\rangle\}$ is an output observation.

The concept of input and output observations is defined from the point of view of the component. Events caused by the environment of the component are considered to be inputs and events caused by the component itself are considered to be outputs. Since consuming tokens from in-ports is an action performed by the component, such disappearing events are considered to be outputs, and vice versa regarding consuming tokens from an out-port.

Definition 8.4: Operation. Consider an arbitrary input observation in of component C. If events occur in the way described by in, we can obtain the output observation out by executing the PRES+ model of C. For each in, several different observations out are possible due to non-determinism. The set of all possible output observations out of C being the result of applying the input observation in to component C, is called the operation of component C from in and is labelled $Op_C(in)$. Given an operation $Op_C(in) = \{o_1, o_{2, \ldots}\}$ and an interface I of component C, the *restricted* operation $Op_C(in)|_I = \{o_1|_I, o_2|_I, \ldots\}$. ■

Intuitively, the operation of a component describes all possible behaviours (outputs) of that component, given a certain input pattern.

We are now ready to define stubs. In Chapter 7, stubs were described as a piece of PRES+ modelling the behaviour of a component with respect to a specific interface. Ports belonging to other interfaces should be abstracted away by introducing non-determinism.

Definition 8.5: Stub. Let us consider two components, S and C. I_S is the interface of S containing *all* ports of S. I_C is *any* interface of C. S is a stub of C with respect to interface I_C iff:

1. Interface I_S is compatible with interface I_C.
2. For any input observation in of component C, satisfying all requirements on ports not in I_C, $Op_C(in)|_{I_C} = Op_S(in|_{I_S})$.

■

Since the left-hand side is restricted to interface I_C, it is clear that events on ports not belonging to this interface are not considered. All possible inputs to C are considered, though. The meaning of the expression on the left-hand side is thus, the set of all possible output behaviours occurring in ports of I_C obtained by firing the PRES+ model of C given any possible input.

The set on the right-hand side denotes the set of all possible behaviours obtained by firing the PRES+ model of S given the same input as C, but only those events belonging to a port in I_S (implicitly applying the bijective function defined by the interface compatibility in Definition 8.1). The output does not need to be restricted since only output compatible with I_C is produced. However, the input must be restricted so that only events corresponding to ports existing in I_S are considered. The other events are left to non-determinism as discussed previously.

8.2 Relations between Stubs

As the concept of stubs has now been formally defined, we can investigate how stubs belonging to different interfaces of a component relate to each other. This will later be used to improve the verification process.

Definition 8.6: Top-level interface. The top-level interface of a component C, with respect to a glue logic G, is the set of all ports of the component to which the glue logic is connected, $I_{max}^{C,G} = C \cap G$. We will use the simple notation I_{max}, if it is either not important or it is clear from the context, to which component and glue logic we refer.

■

Returning to the example in Figure 7.3, which shows a glue logic between the two components Radar and Protocol adapter, $I_{max}^{Protocol,G} = \{in, out, status\}$ and $I_{max}^{Radar,G} = \{targetupdate\}$. For the sake of understanding the rest of this chapter, it should be noted that the involved components do have other interfaces connected to G than the top-level one. For instance, $\{in, out\}$, $\{in, status\}$ and $\{out\}$ are all examples of such interfaces of the Protocol adapter component, but none of them is a top-level interface with respect to the glue logic. Each of these interfaces has an associated stub as defined by Definition 8.5. Top-level interfaces are unique and they always exist if the glue logic is connected to the component.

The ports of a component C, can be divided into interfaces in many different ways. More precisely, every subset of I_{max} can be considered an interface for which a stub can be constructed. Figure 8.3 presents a partial order (it is actually even a lattice) of interfaces, and hence also of stubs, of a component connected to a glue logic through two in-ports (I_1 and I_2) and two out-ports (O_1 and O_2), based on the subset relation. The lattice induces

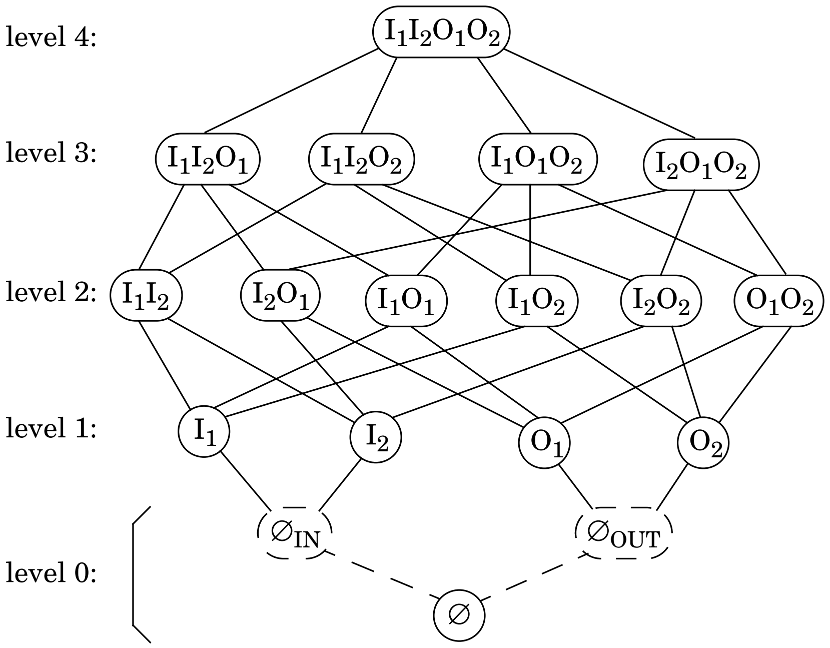


Figure 8.3: A partial order of interfaces

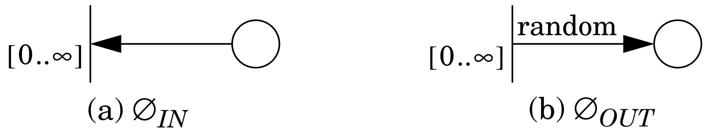


Figure 8.4: The models of the empty stubs

distinct levels of generality of the stubs. The top-level stub (the stub for the top-level interface), with interface $I_{max} = \{I_1, I_2, O_1, O_2\}$, exhibits exactly the same behaviour as its corresponding component, from the point of view of the glue logic connected to that interface. In the bottom of the lattice, we have the empty interface, for which there does not exist any stub and which is only of theoretical interest. If, for a certain verification, no stubs situated at level 1 or higher are applied at a certain port, then a so called empty stub is connected to that port. In the case of in-ports, the empty stub, $\varnothing_{IN}$, denotes the stub that consumes any token at any point in time. Similarly, the empty stub, $\varnothing_{OUT}$, denotes the stub that generates tokens with random values at any point in time. The models of these stubs are presented in Figure 8.4. It is useful to introduce the notation $\varnothing_p$ to denote the empty stub at port p. Whether $\varnothing_p$ is equal to $\varnothing_{IN}$ or to $\varnothing_{OUT}$ depends on whether p is an in-port or an out-port. We further elaborate on the use of empty stubs in Section 8.3.

Between I_{max} and $\varnothing$, stubs of different levels of generality can be found. For each level up in the lattice as more and more ports are included in the interfaces, the stubs become more and more specialised, which is manifested by the introduced causality between in-ports and out-ports of the respective interfaces.

On level 1, stubs for one-port interfaces are situated. If the interface only contains an in-port, the functionality of the stub is to consume the token at random times which, however, correspond to times when the full component could be able to consume the token, if it would be consumed at all. If it only contains an out-port, the functionality is to issue a new token with random value at random occasions. The value and time are random to the extent that the issued values could, in some circumstance, be issued by the full component at the time in question. Note the difference between these stubs and $\varnothing_{IN}$ and $\varnothing_{OUT}$, respectively. The empty stubs produce/consume tokens with random values and times with no regard to the component.

If higher level (level > 1) stubs contain both in-ports and out-ports, a certain degree of causality is introduced. The out-ports can no longer produce any arbitrary value on the tokens, but rather any value consistent with the token values arriving at the in-ports given the behaviour of the full component. Hence, for instance, in Figure 7.4 no token on port *out* can be issued unless the stub has received a connection or listen request at port *in* and accepted it. If there are other in-ports of the component, not represented in the interface of the stub, the output is considered non-deterministic from the point of view of the absent in-port, as in the case with the non-deterministic issuing of *rej* and *con* as an answer to a *connect* request described previously in Figure 7.4.

8.3 Verification Environment

In Section 7.3, the impact of using different sets of stubs was briefly discussed. It was concluded that it is enough to use simple stubs, from here on called *low-level stubs* referring to the lattice in Figure 8.3, in order to verify some properties. Other properties still required complicated, or *high-level*, stubs, where the causality between ports is still kept. This section tries to bring some order into that discussion and proposes a methodology which takes advantage of the variety of stubs to reduce verification time. First, the mathematical foundation must be set.

Definition 8.7: Interface partition. An interface partition P is a set of non-empty interfaces $P = \{P_1, P_2, \ldots\}$ such that $P_i \cap P_j = \varnothing$ for any i and j, $i \neq j$.

■

It should be pointed out that each port can, at most, belong to one interface in every partition. As a consequence of Definition 3.5, all ports in the same interface must belong to the same component. By convenience, the set of all ports belonging to the interfaces in partition P is denoted $Ports(P) = \bigcup_{i \in P} i$.

In the example of Figure 7.6, $P = \{\{arg\}, \{output\}\}$, $Q = \{\{arg\}, \{output\}, \{action, input\}\}$ and $R = \{\{arg, output\}, \{action, input\}\}$ are all interface partitions. $Ports(P) = \{arg, output\}$ and $Ports(Q) = Ports(R) = \{arg, output, action, input\}$. However, $S = \{\{\ \}, \{input\}\}$, $T = \{\{arg, action\}\}$ and $U = \{\{action\}, \{action, input\}\}$ are all examples of sets which are not interface partitions since S contains the empty set, T contains a set which in turn contains ports from different components and the interfaces of U are not disjoint.

Definition 8.8: Partition precedence. Partition P precedes partition Q, $P \propto Q$, if and only if $\forall p \in P \exists q \in Q : p \subseteq q$.

■

For every $p \in P$, there exists at most one $q \in Q$ that satisfies the subset relation. This is due to the fact that every port can at most belong to one interface in the partition.

Using P, Q and R as defined above, $P \propto Q$, since all interfaces in P are subsets of an interface in Q. It is also true that $P \propto R$ and $Q \propto R$. However, it is *not* the case that $R \propto Q$ since $\{arg, output\}$ is not a subset of any set in Q. Intuitively, the stubs corresponding to interfaces in R are more specialised than those in P or Q, since they capture more of the causalities and dependencies between their ports.

Theorem 8.1: The partition precedence relation is a partial order.

■

Proof: *Reflexivity:* $P \propto P \Leftrightarrow \forall p_1 \in P\ \exists p_2 \in P . p_1 \subseteq p_2$ which is trivially true since every set is a subset of itself.
Antisymmetry: Assume $P \propto Q$ and $Q \propto P$. The given assumption is equivalent to $(\forall p \in P \exists q \in Q : p \subseteq q) \wedge (\forall q \in Q \exists p \in P : q \subseteq p)$ according to Definition 8.8. Due to the observation that the existentially quantified p and q are uniquely determined, it is valid that $p \subseteq q \wedge q \subseteq p \Rightarrow p = q$. Since all elements of P and Q are equal, then $P = Q$.
Transitivity: Assume $P \propto Q$ and $Q \propto R$. $(\forall p \in P \exists q \in Q : p \subseteq q) \wedge (\forall q \in Q \exists r \in R : q \subseteq r) \Rightarrow$
$\forall p \in P \exists r \in R : p \subseteq r$, since the existentially quantified q in the first clause of the formula is included among the universally quantified q's in the second clause.

■

Theorem 8.2: The partition precedence relation has a top element P_{max}, including the top-level interfaces of all connected components, and bottom element $P_{min} = \varnothing$.

■

Proof: Assume P_{max} which contains only top-level interfaces and the empty partition $P_{min} = \varnothing$. Consider an arbitrary partition P. $P \propto P_{max}$ by definition since P_{max} only contains top-level interfaces and all interfaces of P must be a subset of one of the top-level interfaces due to the interface subset relation (Figure 8.3). $P_{min} = \varnothing \propto P$ is trivial.

■

In fact, the precedence relation does not only have top and bottom elements, but also this partial order is a lattice.

Definition 8.9: Environment. The environment corresponding to a partition $P = \{I_1, I_2, \ldots\}$ with respect to a set of ports J where $Ports(P) \subseteq J$, is defined as

$Env(P, J) = (\bigcup_{i \in P} S_i) \cup (\bigcup_{p \in J - Ports(P)} \varnothing_p)$ where each S_i is the stub for interface i, and $\varnothing_p$ is the empty stub attached to port p.

■

Let us consider the example in Figure 7.5 with the stubs of the components in Figure 7.6. With $J = \{arg, output, action, input\}$, Figure 8.5(a) shows the environment $Env(\{\{arg\}, \{action, input\}\}, J)$. Since port *output* is not included in the partition, the empty stub $\varnothing_{output}$ (see Figure 8.4) has been added. Figure 8.5(b) shows a similar example for $Env(\{\{arg\}, \{output\}\}, J)$. In Figure 8.5(c), no empty

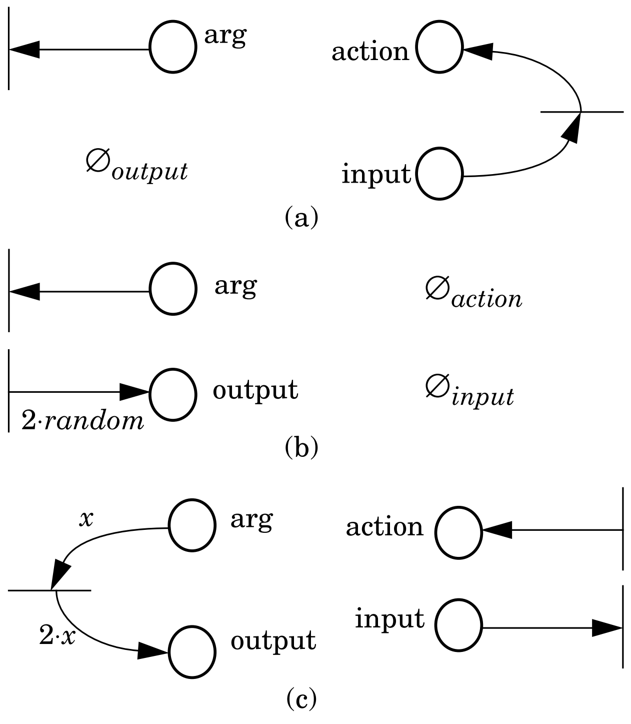
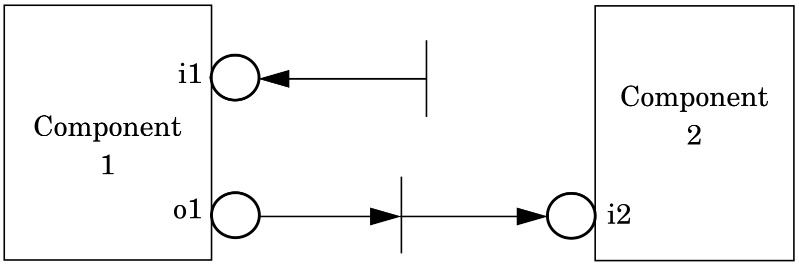


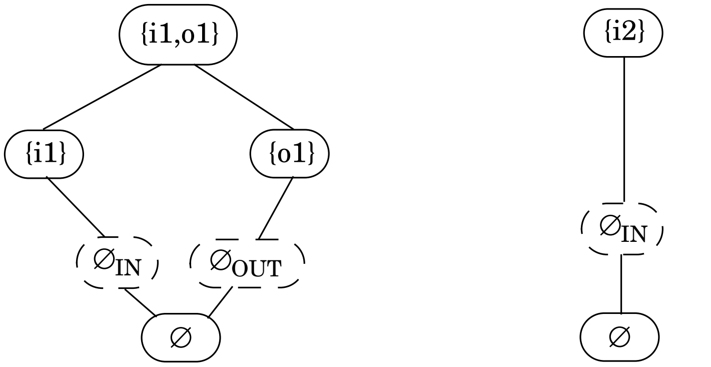
Figure 8.5: A few environments for the example in Figure 7.5

stub needs to be added for $Env(\{\{arg, output\}, \{action\}, \{input\}\}, J)$, since all ports in J are included in the partition.

If all the individual stubs in $Env(P, J)$ together are viewed as one single component, we obtain the environment corresponding to partition P with respect to the set of ports J. The name stems from the fact that such a component acts as the environment of the glue logic, connected to the ports in J, in the verification process. A synonymous name is *Verification Bench*. Based on Theorem 8.1 and Theorem 8.2, it is possible to construct a partial order (lattice) of partitions, i.e. environments, similar to that



(a) Components and glue logic



(b) Interface lattices

Figure 8.6: Components and corresponding interfaces

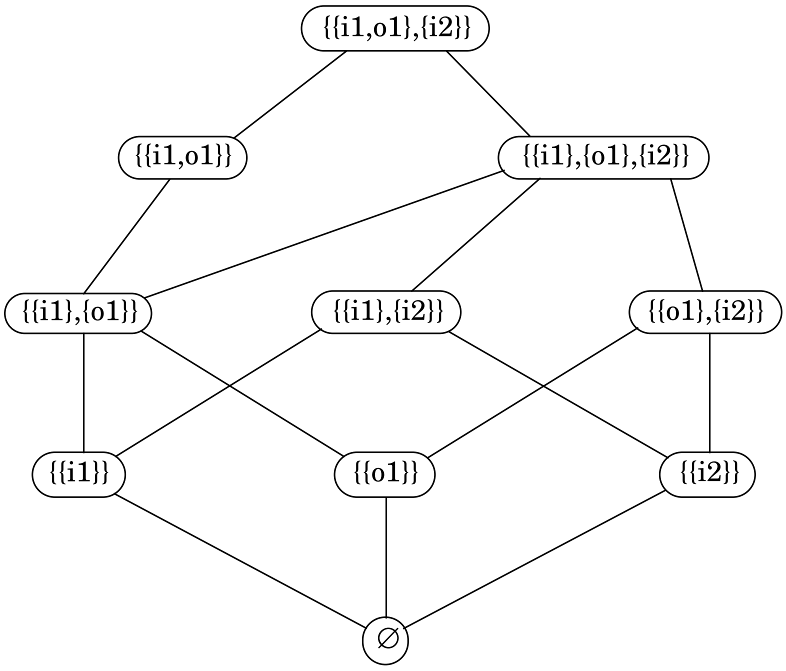


Figure 8.7: Partition (environment) lattice of the situation in Figure 8.6

done with individual stubs and their interfaces (Figure 8.3). Figure 8.6 introduces a very simple example consisting of two interconnected components. Figure 8.6(b) shows the interface (stub) lattice corresponding to each of the components. Figure 8.7 depicts the corresponding partition (environment) lattice.

Definition 8.10: Surrounding. The surrounding of a glue logic G, $Sur(G)$, is the part of the design Γ not including G or any component C connected to G, $C \cap G \neq \varnothing$. $Sur(G) = \Gamma - (G \cup \bigcup_{C \in \{C' | (C' \text{ is a comp. in } \Gamma) \wedge C' \cap G \neq \varnothing\}} C)$ ■

Figure 7.3 shows a glue logic G and its connected components Radar and Protocol adapter. These three entities are only a part of the design of the whole system shown in Figure 7.2. The whole system except G, Radar and Protocol adapter, is said to be the surrounding of G, $Sur(G)$. The glue logic G' in Figure 7.5 does not have any surrounding, $Sur(G') = \varnothing$.

8.4 Formal Verification with Stubs

Having shown that there are many possibilities in choosing the proper stubs, i.e. choosing the verification environment, for the verification problem at hand, a mechanism for helping the designer making this choice is needed. Such an approach is presented in this section through the following definitions and theorems.

Theorem 8.3: Given an input observation in, two partitions P_1 and P_2, $P_1 \propto P_2$, and a set of ports J where $Ports(P_1), Ports(P_2) \subseteq J$, then $Op_{Env(P_1, J)}(in) \supseteq Op_{Env(P_2, J)}(in)$.

■

Proof: Assume an arbitrary observation $o \in Op_{Env(P_2, J)}(in)$. This means that o is a possible output observation given the input observation in. By definition of partition precedence, $\forall p_1 \in P_1 \exists p_2 \in P_2 : p_1 \subseteq p_2$. Hence the restriction operator in $Op_C(in)|_{I_C} = Op_S(in|_{I_S})$ (see Definition 8.5) filters out more elements from the unrestricted operation when $I_S = I_C = p_2$ than when $I_S = I_C = p_1$. Consequently o must also pass the filter of p_1 and can be an output of $Env(P_1, J)$, i.e. $o \in Op_{Env(P_1, J)}(in)$.

■

Definition 8.11: Generalised operation. The generalised operation Op_C for component C is the union of all operations for every possible input observation, $Op_C = \bigcup_{in} Op_C(in)$.

■

According to Definition 8.4, an operation is the set of all possible outputs given a certain input. The generalised operation is the set of all possible outputs no matter what the input is. The generalised operation allows us to generalise Theorem 8.3 into the following corollary.

Corollary 8.1: Given partitions P_1 and P_2, $P_1 \propto P_2$, and a set of ports J where $Ports(P_1), Ports(P_2) \subseteq J$, then $Op_{Env(P_1, J)} \supseteq Op_{Env(P_2, J)}$.

■

Proof: Follows directly from Theorem 8.3 and Definition 8.11.

■

Definition 8.12: State sequence generator. A state, in this context, is a marking of ports. A state sequence generator is a function $\sigma(o, M_0)$, where o is an observation and M_0 is an initial state. The observation o may only contain appearing events and disappearing events on ports. The result of the function is a sequence of states obtained by iteratively applying the events in o to the previously obtained state (initially M_0) in the order indicated by their timestamps.

■

Let r_e denote the timestamp of an event $e \in o$. Assume $e = \langle p, \langle v, r_e \rangle \rangle$ or $e = \langle p, r_e \rangle$, depending on whether it is an appearing or disappearing event, and $E = \{e \mid \neg\exists e' \in o:(r_{e'} < r_e)\}$, i.e. the set of events with the lowest timestamp in o. Then Definition 8.12 can be recursively reformulated as $\sigma(o, M_0) = [M_0 : \sigma(o - E, M_0(E))]$, where $[h:T]$ denotes the head, h, and the tail, T, of a sequence, and $M_0(E)$ denotes the resulting state (marking) after applying all events in E on the initial state (marking) M_0. The basis of the recursion is $\sigma(\emptyset, M_0) = [M_0]$.

Figure 8.2 has illustrated the result of applying the state sequence generator on the given observation with an empty initial marking. Describing the contents of the ports in each time step mathematically, Equation 8.1 gives the solution.

$$\sigma(o, \emptyset) = [\emptyset, \{\langle p, \langle 2, 1\rangle\rangle\}, \{\langle p, \langle 2, 1\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle\} \\ , \{\langle q, \langle 5, 3\rangle\rangle\}, \{\langle p, \langle 3, 8\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle\}, \emptyset] \qquad (8.1)$$

The definitions given so far provide the necessary means to express the semantics of CTL formulas in the context of the theoretical framework we have introduced. First, recall the classical definitions [Cla99] for the two example formulas **AF** φ and **EG** φ for any CTL formula φ ($s \models \varphi$ means that formula φ holds in state s, and $\varphi \Leftrightarrow \psi$ denotes equivalence between two formulas):

$$s \models \mathbf{AF}\ \varphi \Leftrightarrow \forall \sigma \in P_M(s) \exists j \geq 0 : \sigma[j] \models \varphi \qquad (8.2)$$

$$s \models \mathbf{EG}\ \varphi \Leftrightarrow \exists \sigma \in P_M(s) \forall j \geq 0 : \sigma[j] \models \varphi \qquad (8.3)$$

$P_M(s)$ denotes the set of all possible sequences of states in model M where the first state is s. It should be noted that σ in these equations does not refer to the state sequence generator introduced in Definition 8.12, but is a variable quantified over a set of sequences of states. From these sample equations it is possible to extract how the state path quantifiers (**A**, **E**) and the time quantifiers (**G**, **F**) translate into the semantics of our theoretical framework. The difference between this model and ours, is that all definitions in our model are based on events, not states. The link between these two views is based on the state sequence generator in Definition 8.12. Equation 8.4 and Equation 8.5, where IN is the set of all possible input observations of component C, express the same semantics as Equation 8.2 and Equation 8.3 in terms of observations and operations.

$$M_0 \models \mathbf{AF}\ \varphi \Leftrightarrow \forall o \in Op_C \forall i \in IN \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \varphi \qquad (8.4)$$

$$M_0 \models \mathbf{EG}\ \varphi \Leftrightarrow \exists o \in Op_C \exists i \in IN \forall j \geq 0 : \sigma(o \cup i, M_0)[j] \models \varphi \quad (8.5)$$

The union is taken of both all possible input observations, $i \in IN$, and all possible output observations, $o \in Op_C$, and is passed to the state sequence generator to be used as in the classical definitions. The observations are quantified in the same way as the state sequences would have been done in Equation 8.2 and Equation 8.3.

In [Alu90] equivalent formulas to Equation 8.2 and Equation 8.3 are given for TCTL. Based on the discussion above, they can be trivially extended to formulas similar to Equation 8.4 and Equation 8.5.

Theorem 8.4: Assume the partitions P_1 and P_2, $P_1 \propto P_2$, a set of ports J where $Ports(P_1), Ports(P_2) \subseteq J$, an initial marking M_0 on the ports in J and a (T)ACTL formula, e.g. $\mathbf{AF}\ \varphi$, also expressed only on the ports in J. If $M_0 \models \mathbf{AF}\ \varphi$ for component $Env(P_1, J)$, then it is also true that $M_0 \models \mathbf{AF}\ \varphi$ for component $Env(P_2, J)$.

■

Proof:
$M_0 \models \mathbf{AF}\ \varphi \Leftrightarrow$
$\forall o \in Op_{Env(P_1, J)} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \varphi$, where IN is the set of all input observations on ports in the partitions, according to Equation 8.4. As a consequence of Corollary 8.1 and the fact that o and i are universally quantified, it is possible to conclude $\forall o \in Op_{Env(P_2, J)} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \varphi$.

■

The key point in the proof is the universal quantifiers of the observations o and i. For this reason the theorem only applies to (T)ACTL formulas, since they are exactly those formulas which can guarantee the universal quantifier.

Figure 8.8 illustrates the theorem. The area inside the outer circle denotes the set of behaviours (observations), i.e. the oper-

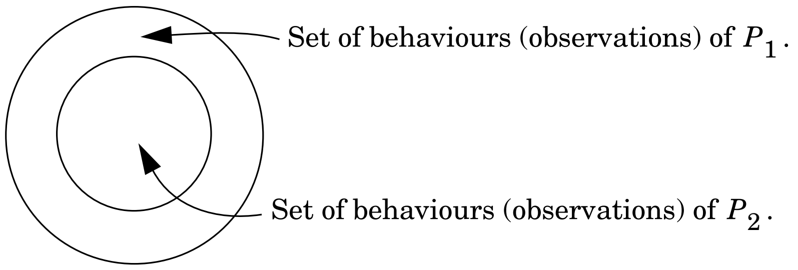

Set of behaviours (observations) of P_1.
Set of behaviours (observations) of P_2.

Figure 8.8: Illustration of Theorem 8.4

ation of $Env(P_1, J)$ $(Op_{Env(P_1, J)})$. According to Definition 8.9, this operation is produced by the union of all stubs corresponding to the interfaces in P_1. The area inside the inner circle denotes the set of behaviours of $Env(P_2, J)$ $(Op_{Env(P_2, J)})$. This set is a subset of the first one according to Corollary 8.1 since $P_1 \propto P_2$ by assumption. If a certain (T)ACTL formula holds for *all* behaviours in the bigger set, it does also hold for all behaviours in the subset. Seen in this way, the necessity of (T)ACTL formulas, as opposed to arbitrary (T)CTL formulas, becomes evident. An arbitrary (T)CTL formula cannot guarantee that the property holds for all behaviours, only that there is at least one behaviour satisfying it.

8.4.1 DISCUSSION

Theorem 8.4 provides the answers to the questions identified at the end of Section 7.3. Let us assume that we have a set C of two or more components which have been interconnected by a glue logic. A certain interface property, expressed as a (T)CTL formula φ, has to verified. The following situations can occur:

1. The verification is unmanageable in the context defined above. This is the case when formula φ is expressed in terms of ports which do not belong to any component in the set C or which, although they belong to a component in C, are not

part of the interface environment of the interface being verified.

2. If the verification is manageable, the following two situations can be identified:

   (a) Formula φ is not a (T)ACTL formula. In this case the verification has to be performed with top-level stubs for all connected components.

   (b) Formula φ is a (T)ACTL formula. In this case, if the formula is satisfied using stubs at any level, the property can be considered as satisfied (this is a direct consequence of Theorem 8.4).

Case 2(b) above is important, as it offers a certain degree of freedom in the case of verification with (T)ACTL formulas. If some top-level stubs are not available, but the property can be verified with lower-level stubs, this is sufficient for validation of the system. On the other hand, for reasons of complexity, the designer can choose to perform the verification with simpler low-level stubs. If the property is satisfied, such a verification is sufficient. If not, however, the verification using high-level stubs can still satisfy the property and thus demonstrate that the system is correct. Some experiments discussed in Section 8.5 illustrate this process.

8.5 Experimental Results

The following experiments concern the verification of systems resulted after the interconnection of components through a glue logic, according to the discussed methodology.

8.5.1 General Avionics Platform

In the first set of experiments, we have verified the interfaces connected to the glue logic in Figure 7.3. The glue logic interconnects the Radar and Protocol component as part of the General

Table 8.1: Experimental results for GAP example

	Partition		
Property	**1**	**2**	**3**
A	F 1.97	F 4.1	T 0.24
B	F 0.39	F 0.69	T 0.12
C	F 0.43	F 0.75	T 0.13
D	T 0.21	T 0.36	T 0.12

avionics platform (Figure 7.1 and Figure 7.2) [Loc91]. We illustrate the verification of four properties. Property A is **AGAF**$\neg update$ (the tokens in port "update" will always be consumed). Property D is **AGAF** $\neg out$ (the tokens in port "out" will always be consumed). Properties B and C are identical to Equation 7.1 and Equation 7.2. As can be seen, all formulas are ACTL. Three possible partitions were used whose relations are shown in the lattice in Figure 8.9. The results of the verification are shown in Table 8.1. The letters F and T in each cell of the table denote whether the property was satisfied (T) or not (F) with the corresponding environment. The numbers denote the verification time in seconds. It can be observed that all four properties imposed by the interconnected components are satisfied with the actual glue logic. For property D, the verification

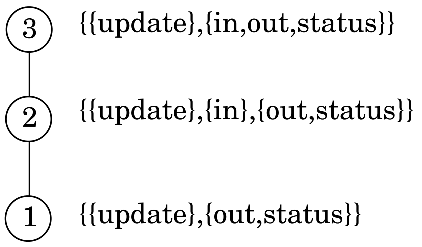


Figure 8.9: Partition lattice in the GAP example

can be done using the lowest level of the three interfaces (as the property is expressed by an ACTL formula, point 2(b) in Section 8.4.1 applies).

8.5.2 SPLIT TRANSACTION BUS

The second example refers to a split transaction bus (STB) in a multiprocessor DSP [Ack00]. An overview of the system is shown in Figure 8.10. The I/O interface and memory controller handles the interaction of the processing elements with the

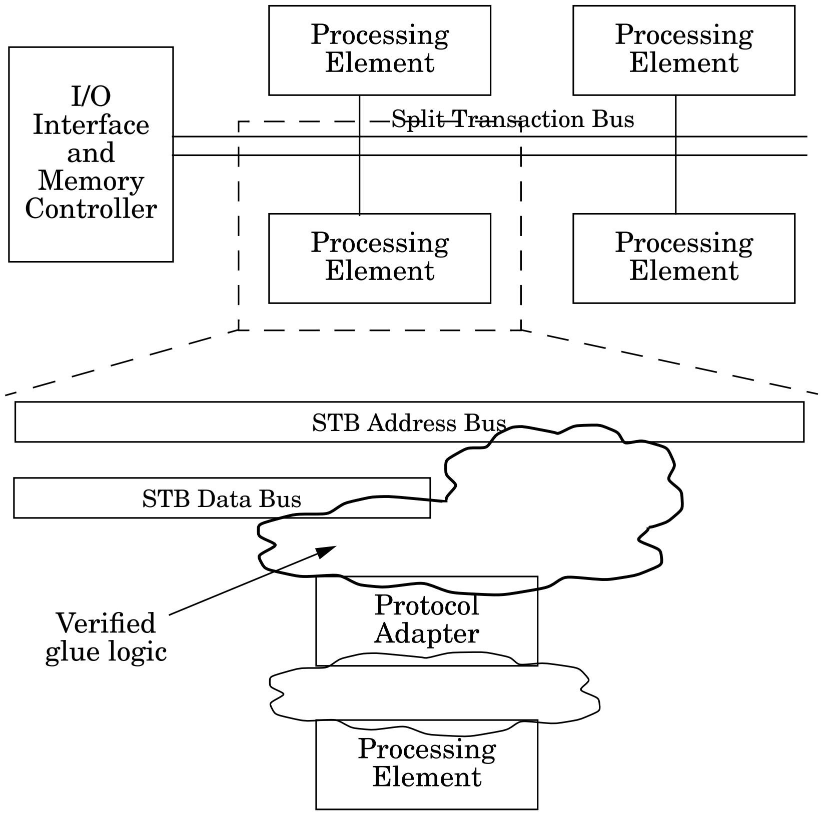


Figure 8.10: Schematic view of the STB example

memory system and the outside world, while the processing elements perform the real functionality. Each processing element contains one 32-b V8 SPARC RISC Core with a co-processor and reconfigurable L-1 cache memory. As suggested in the figure, the split transaction bus consists in fact of two buses, an address bus and a data bus. When the protocol adapter wants to send data, on request from the processing element, it must first request access to the address bus. After acknowledgement from the address bus, the protocol adapter suggests an identifier for the message transfer and associates it with the address of the recipient. This identifier is broadcast to all protocol adapters connected to the bus in order to notify all of them about used identifiers. The next step is to request access to the data bus. When the data bus has acknowledged the request, the identifier is sent followed by some portion (restricted in size by the bus) of the data. Then, the data bus is again requested and the same procedure continues until the whole block of data has been transmitted. The protocol adapter is now ready to service another request from the processing element. One functionality of the verified glue logic is to deliver messages from the protocol to the correct bus. Another aspect is to process the results and acknowledgements so that they can be correctly treated by the protocol adapter. For instance, the protocol component expects two different commands from an identifier broadcast (described above) of the address bus, depending on whether the protocol component currently in hold of the address bus is the component connected to this particular glue logic or the broadcast is the result of another component proposing an identifier.

Table 8.2 shows the verification results from the STB example. The high number of ports in the components yields a large lattice of environments. The one depicted in Figure 8.11 is not the full lattice. Only those environments which are involved in this particular experiment are included. Environment 12 consists of the top-level stubs for all three connected components. Environment 1 consists of only level 1 stubs on out-ports.

In order to give a better understanding of the properties, we will have a closer look at two of them. Property B, for instance, concerns with the fact that the glue logic must issue different commands to the protocol component when the address bus broadcasts the identifiers, depending on the source causing this event to happen. It is formulated as $\mathbf{AG}\ (rec \rightarrow rec \neq \langle \text{TRAN}, a\rangle \wedge a \neq \text{this\_component})$ where

Table 8.2: Experimental results for STB example

	Partition					
Property	**1**	**2**	**3**	**4**	**5**	**6**
A	F 0.41	F 3.28	F 0.34	F 162	T 156	F 345
B	T 0.14	T 0.41	T 0.16	T 17.6	T 24.8	T 16.9
C	F 0.23	F 0.74	F 0.23	F 19.7	F 29.7	F 18.6
D	F 0.38	F 0.89	F 0.37	F 129	F 45.9	F 97.7
E	T 0.20	T 0.58	T 0.21	T 28.1	T 54.2	T 29.2
F	F 0.34	F 0.68	F 0.31	T 18.7	T 26.2	T 16.5
G	F 0.41	T 0.43	F 0.44	T 18.5	T 26.3	T 17.0
H	T 0.21	T 1.30	T 0.22	F 167	F 438	F 344

	Partition					
Property	**7**	**8**	**9**	**10**	**11**	**12**
A	F 330	F 68.2	T 17.7	F 636	T 30.4	T 12.6
B	T 23.6	T 1.69	T 1.38	T 26.9	T 1.54	T 1.29
C	F 28.8	F 3.25	F 3.27	F 32.7	F 4.09	F 4.01
D	F 313	F 20.1	T 3.32	F 292	T 10.2	T 7.04
E	T 48.9	T 2.80	T 1.20	T 53.3	T 4.48	T 4.39
F	T 25.2	F 6.51	F 2.85	T 28.8	T 1.76	T 1.36
G	T 26.7	T 2.47	T 0.94	T 30.0	T 2.36	T 1.94
H	F 325	F 66.4	F 11.9	F 689	F 87.2	F 38.0

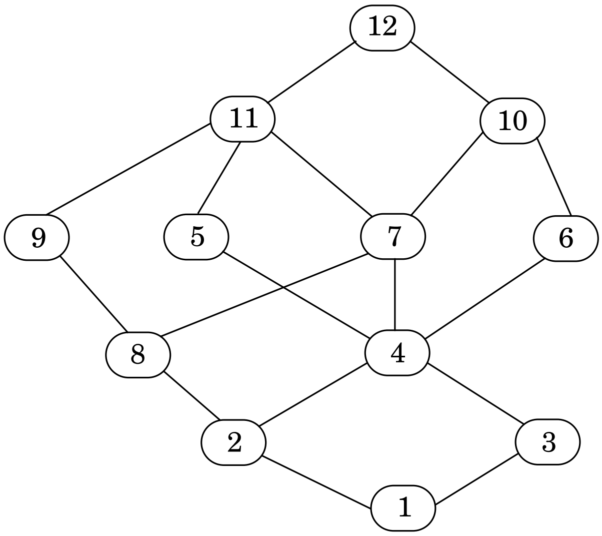


Figure 8.11: Partition lattice in the STB example

TRAN (transaction) is the command to be received by the protocol component when the source causing the event is the protocol adapter connected to the glue logic under verification. It should not be possible to receive such an event where the address is different from the one of the current component. Another property, D, $\mathbf{AG}\ ((addr.out = \text{ACK}) \rightarrow \mathbf{AF}\ addr.in = \text{drive\_addr})$, states a requirement according to which commands are to be given to the address bus: when the bus has acknowledged a request, it expects that the address and identifier are passed.

Properties A to G are expressed as ACTL formulas, while property H is not. It can be noticed that property C is not at all satisfied in the system. That is why the verification results for that property is false, no matter which environment is used. On the other extreme we find properties B and E which are satisfied even with the lowest level environment. Hence, being expressed as an ACTL formula, the property is satisfied with any environment. Property H is not an ACTL formula and can hence not be expected to behave according to the same pattern. Its behaviour can be described as the inverse of the behaviour of ACTL formulas, i.e. the property is satisfied when verified with low-level stubs, but is not satisfied with high-level stubs. Property G, also expressed as an ACTL formula, is also satisfied. This can be verified by using the top-level environment, but also by verifying with environment 2. According to point 2b in Section 8.4.1, the verification performed with environment 2 also guarantees that the property is satisfied with environments 4, 5, 6, 7, 8, 9, 10, 11 and 12, which means the complete system. This is, of course, not the case with property H which is expressed by a non-ACTL formula. Verification with environments 1 to 11 are not valid. The only verification which makes sense is using the top-level environment.

Let us have a look at verification times. For the two examples, taking each separately, the verification time with different environments is in the range 0.12-689 seconds. For a given property the verification times are small for the very low-level stubs and for the top-level stubs. This is due to the simplicity of the low-level stubs, on the one side, and the high degree of determinism of the top-level stubs (which reduces the state space) on the other side. Between these two extremes we can observe a, sometimes very sharp, increase of verification times for the stubs which are at a level close to the top. If a complete set of stubs is available, one can perform the verification using the top-level stubs. For non-(T)ACTL formulas, this is the only alternative. However, (T)ACTL formulas could be verified even if the top-level stubs are not at hand. In this case, a good strategy could be to start with the lowest level stubs, iteratively going upwards in the lattice of stubs until the property is satisfied.

8.6 Verification Methodology Roadmap

This section will continue the roadmap of Section 7.4 based on the work presented in this chapter.

The answer to the question leading the designer to this part of the roadmap (see Figure 7.7) gives us the assumption that stubs already exist and are provided by the designer of the components. The second question to be answered is shown in Figure 8.12. As the experimental results suggest, using top-level stubs, if they exist, gives a relatively short verification time and accurate results avoiding iterations. For this reason it is probably most efficient to immediately use top-level stubs.

If top-level stubs exist, the procedure is very simple as described in Figure 8.13. If the property is satisfied and it is ACTL, then it can be deduced according to Theorem 8.4 that the property really is satisfied. Otherwise if not ACTL, the property can only be proven satisfied to the extent given by the components, i.e. a particular behaviour of the surrounding is not taken into consideration (we will further elaborate on the aspects

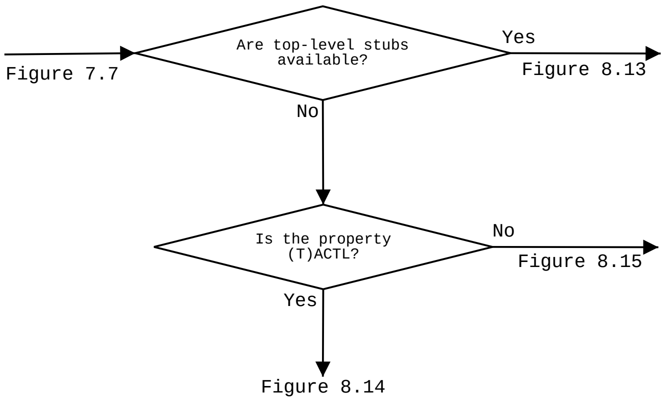


Figure 8.12: Continuation of the roadmap from Figure 7.7

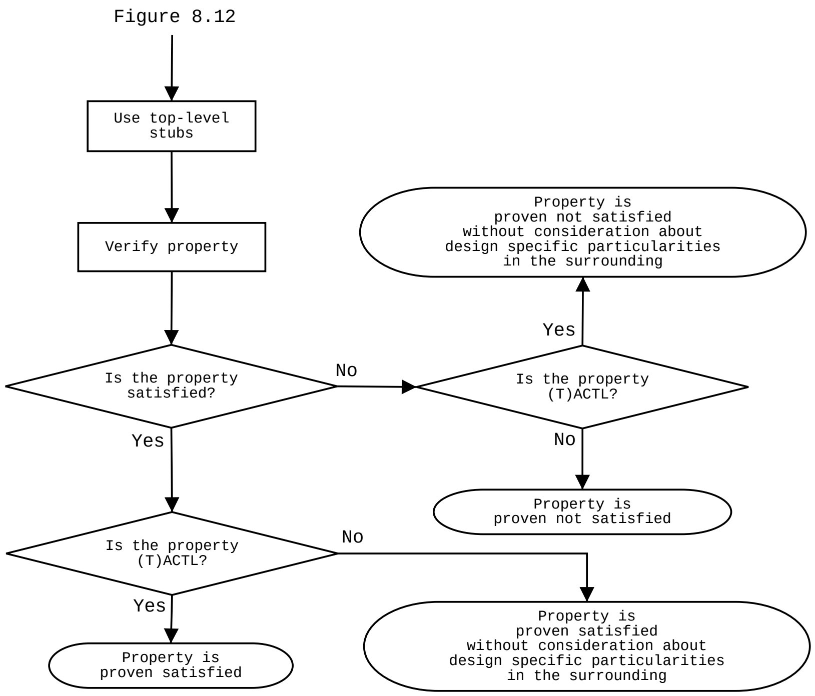


Figure 8.12
Use top-level stubs
Verify property
Is the property satisfied?
No
Yes
Is the property (T)ACTL?
Yes
Property is proven not satisfied without consideration about design specific particularities in the surrounding
No
Property is proven not satisfied
Is the property (T)ACTL?
No
Yes
Property is proven satisfied
Property is proven satisfied without consideration about design specific particularities in the surrounding

Figure 8.13: Roadmap when using top-level stubs, continuation from Figure 8.12

related to the surrounding in Chapter 10). The procedure is analogous when the property was not satisfied.

In the case top-level stubs do not exist, a choice between two similar procedures must be made depending on whether the property is ACTL or not. Figure 8.14 shows the procedure for ACTL formulas and Figure 8.15 for non-ACTL formulas. Start the iterative process by using stubs at the lowest level, since verification times are short when using such stubs. However, the experienced designer may directly use stubs at higher level if it is obvious that the property is not satisfied using the lowest level stubs. The verification result is evaluated as indicated by the

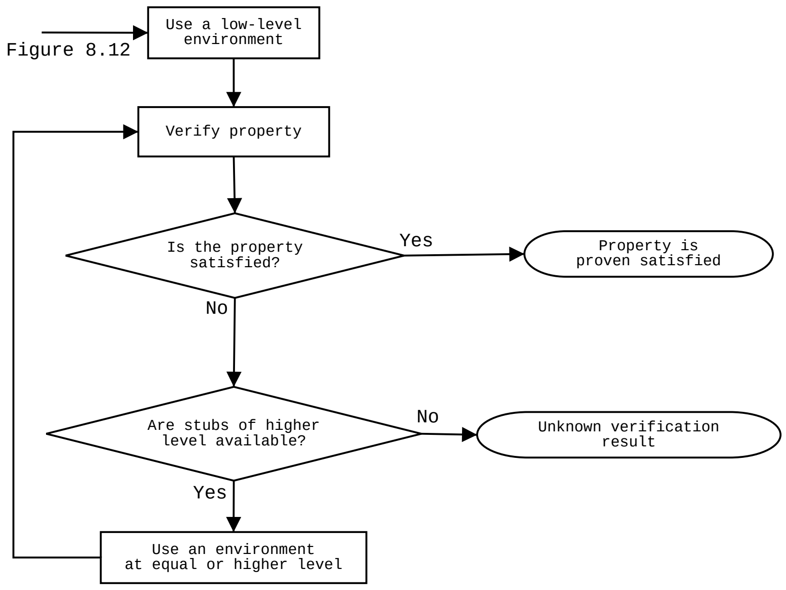


Figure 8.12
Use a low-level environment
Verify property
Is the property satisfied?
Yes
Property is proven satisfied
No
Are stubs of higher level available?
No
Unknown verification result
Yes
Use an environment at equal or higher level

Figure 8.14: Roadmap when using lower-level stubs on ACTL formulas, continuation from Figure 8.12

roadmap. When increasing the level of stubs, it is important that this is done by following a path in the stub lattice so that the assumptions in Theorem 8.4 are not violated. The diagnostic trace resulting from the model checking is very useful for guidance.

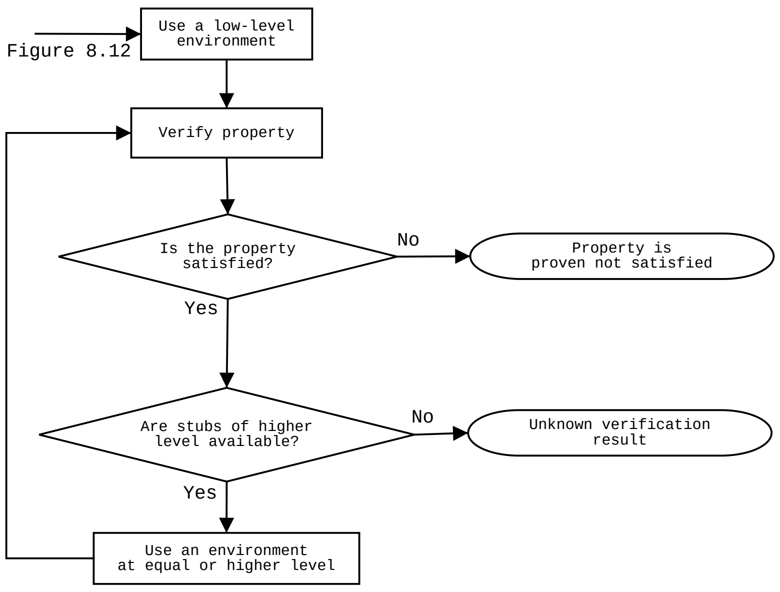


Figure 8.12
Use a low-level environment
Verify property
Is the property satisfied?
No
Property is proven not satisfied
Yes
Are stubs of higher level available?
No
Unknown verification result
Yes
Use an environment at equal or higher level

Figure 8.15: Roadmap when using lower-level stubs on non-ACTL formulas, continuation from Figure 8.12

Chapter 9 Automatic Stub Generation

CHAPTER 8 INTRODUCED a verification methodology where stubs are provided by the provider of the reusable components to represent the components in the integration verification process. If stubs of the desired level are not available, other stubs at lower level can be used instead. An alternative situation is that a PRES+ model of the system is available, but no particular stubs. In this chapter algorithms for automatically generating stubs, given the model of the component and the interface, are presented together with a methodology which explains how to use such stubs. Here, we assume that we do not know anything about the surrounding environment, as opposed to Chapter 10. Experimental results are also presented.

9.1 Pessimistic Stubs

The stub definition presented in Section 8.1 (Definition 8.5) is quite strict, requiring equality between the operations of the component and stub. That strictness makes it very difficult to automatically create stubs. The following definition relaxes the definition of a stub.

Definition 9.1: Pessimistic stub. Let us consider two components, S and C. I_S is the interface of S containing all ports of S. I_C is any interface of C. S is a pessimistic stub of C with respect to interface I_C iff:

1. I_C and I_S are compatible.
2. For any possible input in of component C, $Op_C(in)|_{I_C} \subseteq Op_S(in|_{I_S})$.

■

A pessimistic stub is consequently a stub which can generate more observations than its corresponding component and hence is more “pessimistic” about the set of possible observations. Of course, this might influence the accuracy of the verifications in which they are involved. However, for properties expressed as (T)ACTL formulas this does not necessarily lead to uncertain results. Stubs following Definition 8.5 are in this chapter called *exact stubs* in order to differentiate between the two types.

The following theorem helps us to evaluate the result of verification with pessimistic stubs.

Theorem 9.1: Assume two environments E_1 and E_2 of the same set of components and $Op_{E_1} \subseteq Op_{E_2}$, an initial marking M_0 and a (T)ACTL formula, e.g. $\mathbf{AF}\ \varphi$ expressed only on the ports of the stubs in E_1 and E_2. If $M_0 \models \mathbf{AF}\ \varphi$ for component E_2, then it is also true that $M_0 \models \mathbf{AF}\ \varphi$ for component E_1.

■

Proof:
$M_0 \models \mathbf{AF}\ \varphi \Leftrightarrow \forall o \in Op_{E_2} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \varphi$, where IN is the set of all input observations on ports in the partitions, according to Equation 8.4. As a consequence of the fact that o and i are universally quantified, it is straight-forward to conclude that $\forall o \in Op_{E_1} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \varphi$.
■

The intuition behind this theorem is the same as the intuition behind Theorem 8.4 illustrated in Figure 8.8. The set of behaviours of E_2 includes all the behaviours of E_1 according to the assumption. Hence, if a certain (T)ACTL property is true for all behaviours of E_2, it must also be true for all behaviours of E_1.

Theorem 9.1 allows us to use pessimistic stubs when verifying (T)ACTL formulas. The behaviours of the exact stub (see Definition 8.5) are also produced by the pessimistic one which, however, produces additional behaviours. This fulfils the assumptions of the theorem. So, if a property is satisfied using pessimistic stubs, we can confidently deduce that the property would also have held if exact stubs had been used instead. However, if the property is not satisfied, no conclusion can be drawn at all. In this case, the stubs must be made less pessimistic in order to exclude the undesired behaviour, which caused the property to be unsatisfied, from the operation of the stub.

9.2 The Naïve Approach

The straight-forward way to create a stub of a component, is to keep the original model of the component and add transitions with completely random time intervals and, in the case of an in-port, a random function, on all other ports than those given in the interface of the stub. This will clearly fulfill the requirements of a stub, according to Definition 9.1, since it is able to produce the same events as the component is able to. The difference between the naïve stub and the exact top-level stub is that the naïve stub assumes the most hostile surrounding possible whereas the exact stub complies with the assumptions on the other interfaces (see Definition 8.5).

The example component in Figure 9.1 will be used to explain and analyse the stub generation algorithms in this chapter. In all cases, a stub for the marked interface $\{p_1, p_2\}$ will be generated. The naïve stub for this component and interface is shown in Figure 9.2.

Figure 9.3 illustrates the difference between an exact stub and a naïve stub further. Figure 9.3(a) shows the model of a simple component. It is designed assuming input on ports p_3 and p_4 satisfying the formulas in Equation 9.1 (if there is a token in p_3, then there must arrive a token in p_4 in the future) and Equation 9.2 (no token may arrive in p_4 unless there was first a token in p_3).

$$\mathbf{AG}\ (p_3 \rightarrow \mathbf{AF}\ p_4) \tag{9.1}$$

$$\mathbf{AG}\ ((p_4 \vee \text{init}) \rightarrow \mathbf{A}[\mathbf{A}[p_3\ \mathbf{R}\ \neg p_4]\ \mathbf{R}\ p_4]) \tag{9.2}$$

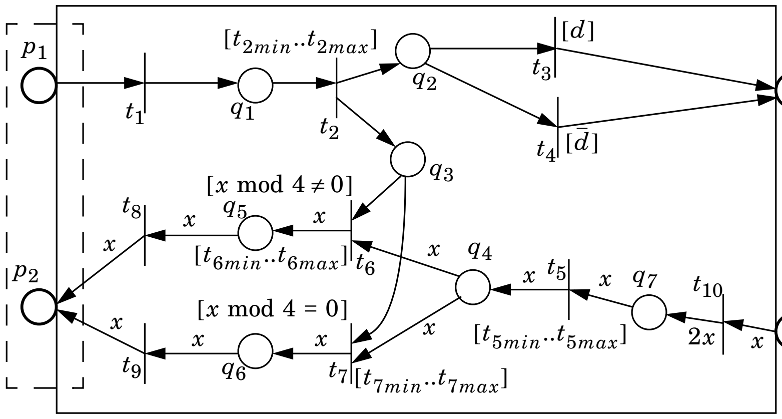


Figure 9.1: Example of a component for stub generation

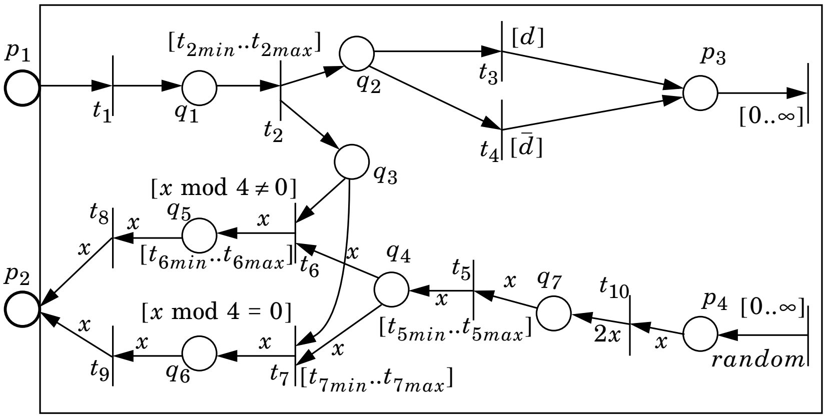


Figure 9.2: A naïve stub of the component in Figure 9.1

In Figure 9.3(b), the naïve stub for the interface $\{p_1, p_2\}$ is presented. Transitions are added to ports p_3 and p_4 as discussed previously. The transitions are added disregarding the assumptions captured in the formulas above. The exact stub is shown in Figure 9.3(c). This stub satisfies the assumptions.

To verify a design using naïve stubs is tremendously time consuming (see experimental results in Section 9.5). For this reason, an algorithm generating smaller stubs reducing verification time has been developed and is presented in the following sections.

9.3 Stub Generation Algorithm

The basic idea of the stub generation algorithm is to identify the parts of the given component which have an influence on the interface for which a stub should be generated. This is done by analysing the dataflow in the component. Once these parts have

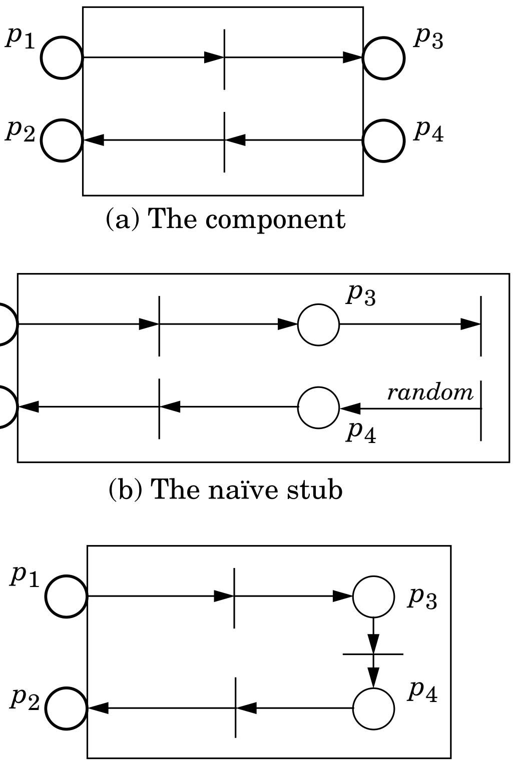


Figure 9.3: Comparison between exact and naïve stubs

been identified, the parts of the model which were excluded must be compensated for. This is the point where pessimism is introduced in the stub.

The stub generation algorithm consists of the three parts presented below. Each of them is explained separately in the following sections.

1. Dataflow analysis
2. Identification of stub nodes
3. Compensation for the excluded parts of the component

```
1  procedure traceBack(e: place or transition, p: port)
2      if not visited[e] then
3          visited[e] := true;
4          for each $d \in {}^{\circ}e$ do
5              DF[d, p] := DF[d, p] $\cup$ { e };
6              traceBack(d, p);
7
8  procedure traceForward(e: place or transition, p: port)
9      if not visited[e] then
10         visited[e] := true;
11         for each $d \in e^{\circ}$ do
12             DF[d, p] := DF[d, p] $\cup$ { e };
13             traceForward(d, p);
```

Figure 9.4: Algorithms for searching the dataflow

9.3.1 DATAFLOW ANALYSIS

The first step when identifying the parts to be included in the stub is to investigate the dataflow. This is a very simple procedure namely a graph search algorithm, as shown in Figure 9.4. These procedures are called once for each port in the interface of the stub. traceBack is called for out-ports and traceForward in the case of in-ports. visited is a mapping from places and transitions in the component to a boolean value. Initially, all places and transitions are mapped to false.

During the search through the graph, each node (place or transition) is moreover marked with the node that was immediately previously visited (Line 5 and Line 12) so that it is possible to obtain the path from an arbitrary node to a port in the interface. These markings represent the dataflow in the model. The dataflow marking must not only be able to distinguish the paths but also to which port each path leads. The dataflow marking is stored in a data structure (DF) for later use. The data structure associates a place or transition together with the original port to a set of neighbouring places or transitions which were immediately visited before the node just being visited. The algorithms in Figure 9.4 and the outlined data structure implement Definition 9.2.

Definition 9.2: Dataflow marking. A dataflow marking $df_I^C(n, p)$ is a set of nodes (places or transitions), which constitute the first step on a path from node n to port $p \in I$ in component C with an interface $I \subseteq C$. If C or I are evident from the context they may be omitted from the notation. As an extension we also define $df(n) = \bigcup_{p \in I} df_I(n, p)$.

■

Figure 9.5 reveals the dataflow marking for the example component. Every node is annotated with a set of arrows, solid and hollow. The type of the arrow reflects towards which port it points. In the figure, solid arrows point towards p_1 and hollow ones point towards p_2. Place q_3 is visited by the search algorithms both starting from p_1 (traceForward) and p_2 (traceBack). This means that both ports can be reached from q_3. As indicated in the figure, the path from q_3 to p_1 goes through t_2, and the path

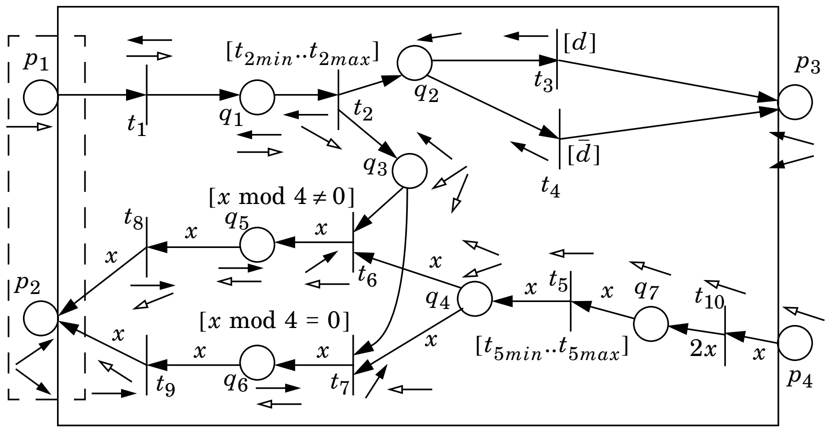


Figure 9.5: The dataflow marking of the component in Figure 9.1

from q_3 to p_2 through either t_6 or t_7. There is no path to p_1 from q_4, since q_4 was never reached in the dataflow search from p_1 (traceForward).

A dataflow marking is, intuitively, the set of arrows (maintaining their types) associated to a node obtained from the search. For future reference, it is also useful to introduce the following definitions based on the dataflow marking.

Definition 9.3: Divergence node. A node n is a divergence node if and only if $|df(n)| > 1$, i.e. there are several different paths leading to ports in the interface, or the arrows of n point in different directions.

■

Definition 9.4: Intersect node. A node n is an intersect node if and only if $\exists p \in I : df_I(n, p) \neq \emptyset \wedge df_I(n) \neq df_I(n, p)$, i.e. at least two arrows pointing in different directions are of different type (solid or hollow).

■

In Figure 9.5, amongst others, nodes t_1, q_1, q_4, p_3 and p_2 are divergence nodes. Nodes t_1, q_1, t_6 and t_7 are examples of intersect nodes. p_4, t_{10}, q_7 and t_4 are nodes which are neither divergence nor intersect nodes.

9.3.2 IDENTIFICATION OF STUB NODES

In order to describe the algorithm, the concept of separation point (SP) must first be defined.

Definition 9.5: Separation point. A separation point (SP) is a node (place or transition), which denotes the border between the parts of the component to be included in the stub and the part not to be included.

■

An SP can be situated at two different types of nodes:

1. Divergence node (e.g. p_3, q_3, q_4 and t_6 in Figure 9.5).

2. The node is a port in the interface (p_1 and p_2 in Figure 9.5).

The search for SPs starts in the ports not belonging to the specified interface and it must be repeated once for each such port. Figure 9.6 (traceNode) presents the algorithm.

Similar to traceForward and traceBack, traceNode is also a depth first search algorithm. During the search, three cases in particular can occur:

```
procedure traceNode(e: place or transition)
    if not tr_visited[e] then
        tr_visited[e] := true;
        if e is a port in the specified interf. or e is an intersect node then
            constructStub(e);
        else if e is a divergence node then
            tr_visited[e] := false;
            node spcand := traceCutEdge(e);
            if spcand = NULL then
                constructStub(e);
            else
                traceNode(spcand);
        else
            traceNode(the only element in DF[e]);

function traceCutedge(e: place or transition) returns place or transition
    if not tr_visited[e] and e is not an intersect node then
        tr_visited[e] := true;
        for each d∈DF[e] do
            if <e,d> is a cutedge then
                return d;
            else
                node cecand := traceCutedge(d);
                if cecand ≠ NULL then
                    return cecand;
    return NULL;
```

Figure 9.6: Algorithms for identifying which parts of a component to include in the stub

1. The node being visited is a port or an intersect node.
2. The node being visited is a divergence node.
3. The node being visited is neither of the above.

If node e being visited is a port, the stub is constructed using e as a separation point (Line 4). e is also used as a separation point if it is an intersect node. All nodes on the path between two ports in the specified interface, and only those nodes, are intersect nodes. For this reason, intersect nodes *must* be included in the resulting stub.

Otherwise, if node e being visited is a divergence node (Line 6), it is a *candidate* for being the separation point. However, there might be better separation point candidates if the search is continued (traceCutedge).

If node e being visited does not belong to either of the two categories above, the search continues following the dataflow as indicated by $df(e)$ (Line 14).

Let us return to the case where node e being visited is a divergence node. As mentioned, e is a candidate for being a separation point. The reason is that in divergence nodes, the dataflow is influenced from more than one direction and all influences in the dataflow should be kept in the stub. However, it might be the case that there is a cutedge[1] along the path between e and the ports in the specified interface (Line 8). The presence of a cutedge means that all data has to flow through the cutedge before reaching e, cancelling the importance of keeping e as a separation point since the dataflow between the ports in the specified interface will not be influenced by the divergence node e. If no cutedge was found, e is used as separation point (Line 10). Otherwise, the procedure starts all over from the cutedge (Line 12).

Searching for a cutedge (traceCutedge) is also a depth first search. If node e being visited is an intersect node, the search

1. An arc is a cutedge if the component becomes divided into two parts if the arc was to be removed from the graph.

```
1  procedure constructStub(e: place or transition)
2      if not visited[e] then
3          visited[e] := true;
4          res := res ∪ {e}; // including all arcs connecting e with res;
5          for each d ∈ DF[e] do
6              constructStub(d);
```

Figure 9.7: Algorithm for adding places and transitions to the resulting stub given a separation point

stops due to reasons already discussed (Line 17). Otherwise, all paths indicated by the dataflow marking are examined (Line 19). If, in that case, a cutedge is found, the neighbouring node of e is returned as being a new candidate for SP (Line 21). If a cutedge was not found, the search continues until one is found (Line 23). In case a cutedge was not found in the whole component, NULL is returned to indicate this situation (Line 26).

When a separation point is finally found, the stub is constructed originating from that point. As the other algorithms, this procedure is also a depth first search. Figure 9.7 shows the code of the algorithm.

All nodes visited by the algorithm are added to the resulting stub (Line 4). res is a global variable which will contain the generated stub when the stub generation algorithm has finished. The search progresses through the component by following the dataflow, i.e. the arrows created by traceForward and traceBack (Line 5).

Continuing the example in Figure 9.5, the search starts, for instance, from port p_3. p_3 is a divergence node (but not an intersect node) and according to the algorithm, a search for cutedges is started (Line 8 in Figure 9.6) while keeping in mind that p_3 might be chosen as a separation point in case traceCutedge fails. traceCutedge will eventually recognise that the arc between q_2 and t_2 is a cutedge and returns t_2 back to trace-

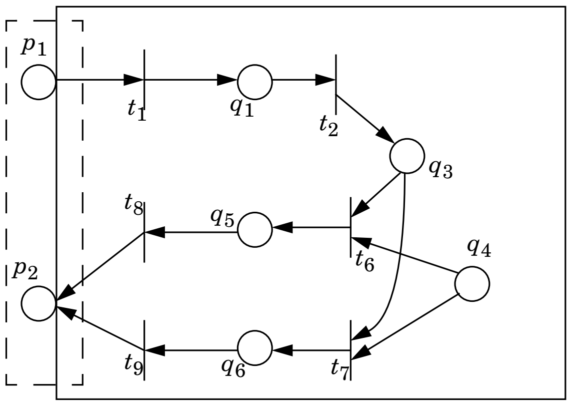


Figure 9.8: The places and transitions in the automatically generated stub

Node which assigns this value to spcand. The search continues as before now starting from t_2 (Line 12). However, since t_2 is an intersect node, it is chosen as a separation point and the stub is constructed starting from this point (Line 5). Figure 9.8 shows the resulting stub. At this point, everything is added except q_4. Time delay intervals, transition function and transition guards will be added later.

The procedure is repeated for port p_4. The first divergence node discovered in the search is q_4. According to the algorithm, a search for cutedge is started (Line 8 in Figure 9.6) while keeping q_4 as a candidate for being a separation point. traceCutedge discovers that both t_6 and t_7 are intersect nodes (Line 17) and returns NULL indicating that a cutedge was not found. As a result, traceNode concludes that q_4 must be chosen as a separation point (Line 10). At this point, q_4 is also added to the stub completing Figure 9.8.

9.3.3 COMPENSATION

All places and transitions on a path between two ports are included in the resulting stub as a result of the previous steps. However, there will be some nodes (either places or transitions) of which not all nodes in the postset or preset are also included in the stub. This means that they will not deliver or receive all needed output or input. These nodes are called *fork* or *join* nodes respectively and need additional treatment.

Definition 9.6: Fork node. Assume a component C and a stub $S \subseteq C$. A node $n \in V(S)$ is a fork node if and only if the corresponding node $n \in V(C)$ in the component has a node in its postset which is not in the stub, $\exists n' \in n^{\circ} : n' \notin S$.

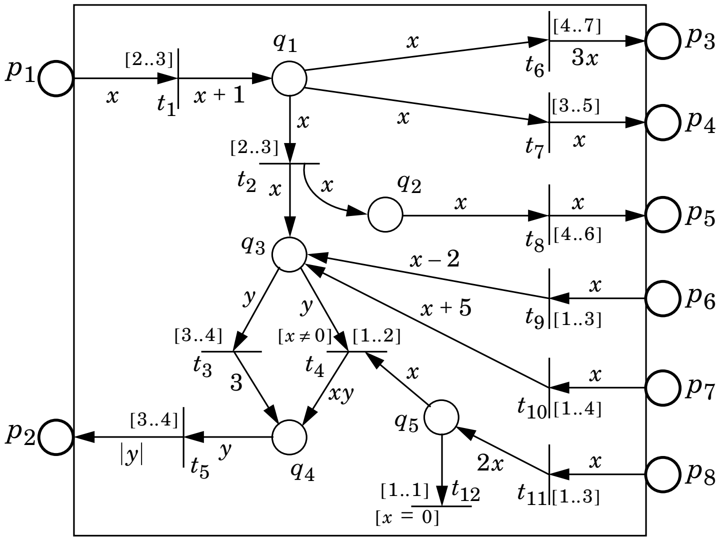
■

Definition 9.7: Join node. Assume a component C and a stub $S \subseteq C$. A node $n \in V(S)$ is a join node if and only if the corresponding node $n \in V(C)$ in the component has a node in its preset which is not in the stub $\exists n' \in {}^{\circ}n : n' \notin S$.

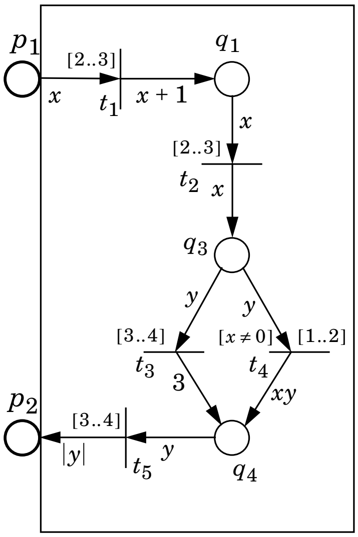
■

Figure 9.9(a) introduces an example of a component which will be used to explain how fork and join nodes are modified in the stub. Figure 9.9(b) shows the stub as generated by the algorithms in Figure 9.6, whereas Figure 9.9(c) presents the resulting stub after the compensation of the excluded parts. The exact procedure of compensation will be described shortly.

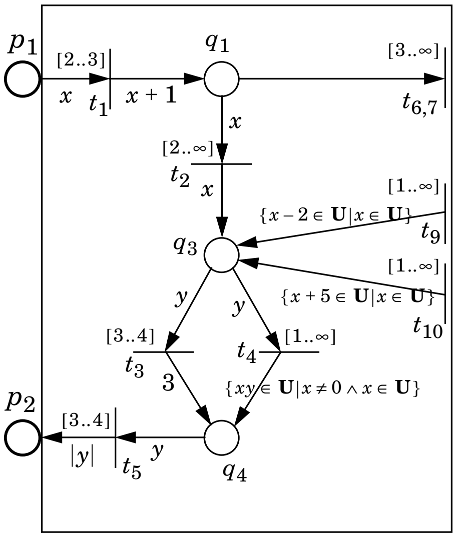
It is sometimes necessary to introduce randomness in the transition functions. This is denoted by a set of values, from which a value can be randomly chosen. The notation $f_t = \{2x \in \mathbf{Z} | x \in \mathbf{Z}\}$ consequently means that transition t may produce randomly any even integer number. The function does not have any arguments in this case, meaning that an even number is produced disregarding the token values in its input places. In the general case, the functions may have arguments,



(a) Component



(b) Stub as generated by the algorithms in Figure 9.6



(c) Stub

Figure 9.9: Example component and stub explaining the compensation of excluded parts

i.e. $f_t(y) = \{xy | x \in \mathbf{Z}\}$, where attention has also to be paid to the input token values.

Such transition functions are created with respect to a certain universe **U** containing all values possible in the design. In this chapter, for the sake of example, it is assumed that the universe consists of all integers, $\mathbf{U} = \mathbf{Z}$.

Case 1: fork place

If the fork node is a place (i.e. q_1 in Figure 9.9(a)), it means that tokens can disappear out of the stub, into the part of the net which is excluded. To model this, a new transition ($t_{6,7}$) is added to consume these tokens. The time interval of this transition is from the minimum delay of all postset transitions not included in the stub, to infinity ($[min(3, 4)..\infty] = [3..\infty]$).

The reason is that tokens can inherently not disappear before the stated lower limit, but, on the other extreme, the token might not be consumed at all.

Case 2: join place

If the join node is a place (q_3), it means that a token might appear in the place from outside the stub. This is modelled by adding all missing transitions in the preset of the place (t_9 and t_{10}). The newly added transitions are modified in the following manner:

The upper bound of the time interval is set to infinity. The lower bound is left unchanged. This models the fact that the corresponding transition in the full component might never be enabled.

The function of the added transition is the same as the function of the corresponding transition in the component except that all arguments contain random values conformant to a possible transition guard. Equation 9.3 expresses this formally.

$$f_t' = \{f_t(x_1, \ldots, x_n) \in \mathbf{U} | g_t(x_1, \ldots, x_n) \wedge x_1, \ldots, x_n \in \mathbf{U}\} \qquad (9.3)$$

The transition guard is set to $f_t' \neq \emptyset$.

The guards are not shown in Figure 9.9(c) for space reasons. Moreover, the guards are redundant in this case since they are always true, $f_{t_9}' = f_{t_{10}}' = \mathbf{Z} \neq \emptyset$.

$$f_{t_9}' = \{x - 2 \in \mathbf{Z} | x \in \mathbf{Z}\} = \mathbf{Z} \tag{9.4}$$

$$f_{t_{10}}' = \{x + 5 \in \mathbf{Z} | x \in \mathbf{Z}\} = \mathbf{Z} \tag{9.5}$$

Case 3: join transition

If the join node is a transition (t_4), the enabling of that transition depends on the part of the component excluded from the stub. The exact enabling times can therefore not be known. Accordingly, the maximum time delay is changed to infinity. Moreover, some parameters for the transition function lack a value.

The transition function f_t' is updated in a similar manner as for join places, with the exception that some parameters are fixed, as they come from preset places inside the stub. Equation 9.6 expresses this formally. x_i are parameters coming from places outside the stub, and y_i from places inside the stub.

$$\begin{aligned} f_t'(y_1, \ldots, y_m) = \{ & f_t(x_1, \ldots, x_n, y_1, \ldots, y_m) \in \mathbf{U} | \\ & g_t(x_1, \ldots, x_n, y_1, \ldots, y_m) \wedge x_1, \ldots, x_n \in \mathbf{U}\} \end{aligned} \tag{9.6}$$

The guard of the transition is set to $f_t'(y_1, \ldots, y_m) \neq \emptyset$.

The guard is not shown in Figure 9.9(c) for space reasons. Similar to the join place case, the guard is not necessary in the example. Equation 9.7 and Equation 9.8 explain why.

$$f_{t_4}'(y) = \{xy \in \mathbf{Z} | x \neq 0 \wedge x \in \mathbf{Z}\} \tag{9.7}$$

$$\forall y \in \mathbf{U} : f_{t_4}'(y) \neq \emptyset \tag{9.8}$$

Case 4: fork transition

If the fork node is a transition (t_2), a token in one of its excluded output places might disable the transition (forced safe PRES+). This fact is modelled by setting the maximum time delay of the fork transition to infinity.

To illustrate this situation, imagine the case where there are tokens in both p_5 and q_2 and the token in p_5 is never consumed by the glue logic connected to it. Transition t_2 will never become enabled.

■

Figure 9.10 shows the final result of applying this algorithm to the example in Figure 9.1 with respect to the interface $\{p_1, p_2\}$.

In all cases described above, some degree of pessimism is introduced. At some points, transition functions are randomised, as for t_5 in Figure 9.10. In the stub, this transition produces any value, since it assumes that any input is possible. In the full component, this is actually not the case since t_{10} (Figure 9.1), which provides input for t_5, only can produce even

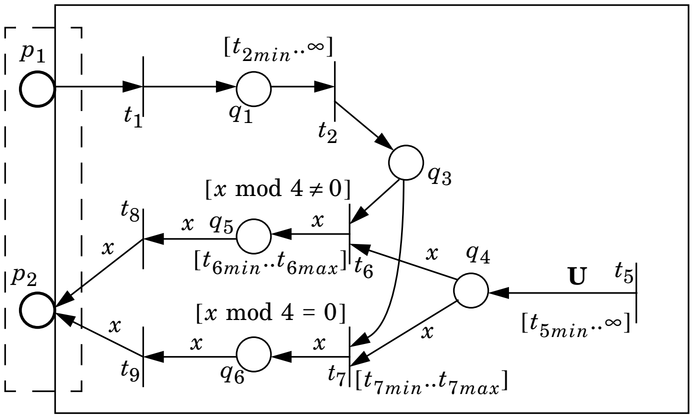


Figure 9.10: An automatically generated stub

numbers. Consequently, the stub is more pessimistic about possible values than an exact or naïve stub.

In particular cases, the algorithm may result in an empty model. This occurs when the data path from a certain port does not intersect that of another port in the interface. Obviously, as a special case, this occurs when the interface only contains one single port. Those ports are by definition either join or fork places and are modified accordingly.

In certain models, an SP may be situated at a port outside the specified interface. Such ports are neither join, nor fork places. Random transitions are in such cases added to the port, in the same way as random transitions were added for naïve stubs.

9.3.4 COMPLEXITY ANALYSIS

The algorithm is based on depth first search, which has time complexity $O(n + a)$, where n is the number of nodes and a the number of edges in a graph. Consequently, both traceForward and traceBack have this complexity.

Checking whether an edge is a cutedge or not is also a depth first search where you try to find another path from one node on the edge to the other, except through the particular cutedge candidate. The complexity is also $O(n + a)$.

Compensating for the excluded parts of the component is a scan through all nodes with a constant operation on each of them, leading to a complexity of $O(n)$.

In the worst case, every edge has to be checked whether it is a cutedge or not. The overall worst case complexity hence becomes $O(n + a(n + a)) = O(n + an + a^2)$. Assuming that there are more edges than nodes, the theoretical worst case complexity of the algorithm is quadratic in the number of edges, $O(a^2)$. However, it should be noted that, in practice, very few edges are checked for being cutedges. Consequently, execution time is practically close to linear.

9.4 Reducing Pessimism in Stubs

If a certain property was not satisfied using the generated stubs, it is necessary to consider the possibility that this is due to the pessimistic nature of the stub and not to a design error. The problem could be that the operation of the generated stub contains more observations than the corresponding component.

The operation of the stub must consequently be refined, i.e. the degree of pessimism must be reduced. The solution to this problem is to add some parts of the component, which were excluded in the stub generation, to the stub. However, in the general case, the designer does not have any detailed knowledge about the internals of the component and its stubs, so this procedure cannot be done manually. This leads to the necessity of automating the pessimism reduction procedure. Such an automatic procedure is possible assuming that all transition functions are invertible in the sense that, given a value, it is possible to obtain which set of arguments result in the given value.

What the designer must know in order to use the component is stated in the user documentation of the component, i.e. the events occurring on the ports. By following the diagnostic trace, obtained as a result from the verification, the designer can identify an unwanted behaviour on one of the ports of the component. If the unwanted behaviour is causal, i.e. the value itself is allowed at the particular port, but not at that particular ordering compared to other values, then it is not a matter of reducing pessimism, but it is a sign that the stub does not cover enough ports (compare with Section 8.2). Unwanted values and overestimation of the firing delay of transitions are, on the other hand, a matter of stub pessimism reduction. This fact is a consequence of the proposed stub generation algorithm and of the definition of pessimistic stubs.

Firing delays are overestimated with infinity in the stub generation algorithm. The reason for this was that there is no guarantee that the transitions will ever become enabled. However, assuming the most hostile surrounding possible, this can never be guaranteed in the full component either. Consequently, no pessimism reduction algorithm may ever be able to reduce this type of pessimism. Chapter 10 introduces a technique to also incorporate certain aspects of the surrounding into the verification process, and thereby solve this problem.

Thus, pessimism reduction of stubs is only applied when there is a value v in a port of the interface which cannot occur in that port in the full component. Pessimism can be reduced by iteratively adding transitions and places, which were previously removed from the component by the stub generation algorithm, until the unwanted value is eliminated. When adding a previously removed place or transition, all nodes in both the preset and postset of the place must also be modelled in accordance with the fork and join node cases of the stub generation algorithm on page 192. In the extreme case, the naïve stub is obtained when the stub is extended with all parts of the component. In order to automatically reduce the pessimism in a stub efficiently, in a way such that the possibility of value v to occur in a certain port is removed, the diagnostic trace resulting from the verification is helpful.

In order to explain the pessimism reduction algorithm, let us return to the previous example and the stub in Figure 9.10. In order to keep the example simple, it is assumed that the component is connected to a second component through a glue logic as depicted in Figure 9.11. The result of verifying the property **AG** $(r_1 \rightarrow even(r_1))$ (All tokens arriving in r_1 must have an

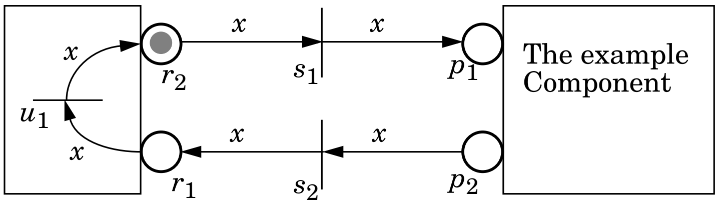


Figure 9.11: An example system

```
1  function pessRed(stub: PRES+; comp: PRES+; tr: trace) returns PRES+
2      for each n∈stub do
3          visited[n] := true;
4      oldStub := copy of stub;
5      newStub := oldStub;
6      repeat
7          Follow tr backwards until a join transition, t, is encountered;
8          u := the value resulting from t, also indicated by the trace;
9          visited[t] := false;
10         success := buildStub(newStub, t, u); // Defined in Figure 9.13
11         if not success then
12             newStub := oldStub;
13         else
14             oldStub := newStub;
15     until tr is finished;
16     return newStub;
```

Figure 9.12: The pessimism reduction algorithm

even value.) is clearly unsatisfied, since transition t_5 produces completely random values. A possible diagnostic trace given by the model checker is the following sequence of transitions (produced values in parenthesis, if any): s_1, t_1, t_2, $t_5(3)$, $t_6(3)$, $t_8(3)$, $s_2(3)$. Figure 9.12 outlines the pessimism reduction algorithm presented below.

By following the trace backwards from the end towards its beginning (Line 7), the possible nodes where the stub can be extended are discovered. The possible extension points are naturally those nodes where something was omitted in the stub generation, i.e. the join transitions. Join places do not exist in a generated stub since transitions in their presets are added due to case 2 on page 192. The first join transition encountered in the example sequence is t_5, which produced value $u = 3$ (Line 8).

```
function buildStub(stub: PRES+; t: transition; u: value) returns boolean
    if not visited[t] then
        visited[t] := true;
        stub := stub $\cup$ { t };
        if $f_t$ is constant then
            return $f_t \neq u$;
        else
            W := $f_t^{-1}(u)$;
            for each $w \in W$ do
                if $g_t(w)$ then
                    for each parameter $w_i$ of $f_t$ do
                        $p_i$ := the place corresponding to $w_i$;
                        if $p_i$ has an initial token with value $w_i$ then
                            return false;
                        stub := stub $\cup$ { $p_i$ };
                        if $°p_i = \emptyset$ then
                            return false;
                        else
                            for each $t_i \in °p_i$ do
                                success := buildStub(stub, $t_i$, $w_i$);
                                if not success then
                                    return false;
    return true;
```

Figure 9.13: Auxiliary function for the pessimism reduction algorithm

The part of the component not included in the stub is then examined backwards starting from the selected join transition (t_5) towards the ports (p_4), exploring the part of the component not included in the stub (Line 10 in Figure 9.12 and the function in Figure 9.13). The exploration is done in a depth first manner.

For each transition t visited, a value u to be eliminated is maintained. If the transition function of t, f_t, is constant, the algorithm fails if $f_t = u$ since it is impossible to avoid u in t

(Line 6 in Figure 9.13). Otherwise, the value u is avoided by having included the transition in the stub (Line 4).

If the function is not constant, it is needed to find out which set of function arguments can produce the unwanted value. This is done using the inverted function f_t^{-1}, as defined in Equation 9.9 (Line 8). In order to succeed, all arguments resulting in u must be in turn eliminated (Line 9) only taking into consideration those values which are satisfied by the guard (Line 10).

$$f_t^{-1}(x) = \{\langle x_1, \ldots, x_n \rangle \mid f_t(x_1, \ldots, x_n) = x \wedge x_1, \ldots, x_n \in \mathbf{U}\} \quad (9.9)$$

Each function argument may consist of several parameters, for instance transition t_4 in Figure 9.9(a). Each such parameter corresponds to a place. If the place has an initial token with a token value equal to u, it is impossible to eliminate the value, so the algorithm fails (Line 14). Otherwise, the place is added to the resulting stub (Line 15). It is also impossible to eliminate the value if the preset of the place is empty, i.e. the place is a port towards the surrounding (Line 17).

Otherwise, the search continues from the transitions in the preset of the place, trying to eliminate the value associated to the place (Line 20). If the algorithm fails for one transition, the total result will be a failure (Line 22).

Let us return to Figure 9.12. If buildStub failed, the modifications made on the stub are reverted, so that a new iteration can start with a fresh copy (Line 12). The algorithm then searches for the next join transition in the diagnostic trace (Line 7). This procedure continues until the whole trace has been examined (Line 15).

The first join transition encountered in the example in Figure 9.1 is t_5 with the value $u = 3$. $f_{t_5}(x) = x$ is not constant, so the transition must be further examined. The set of values resulting in 3 is $f_{t_5}^{-1}(u) = f_{t_5}^{-1}(3) = \{3\}$. The transition does not have any guard and has only one parameter corre-

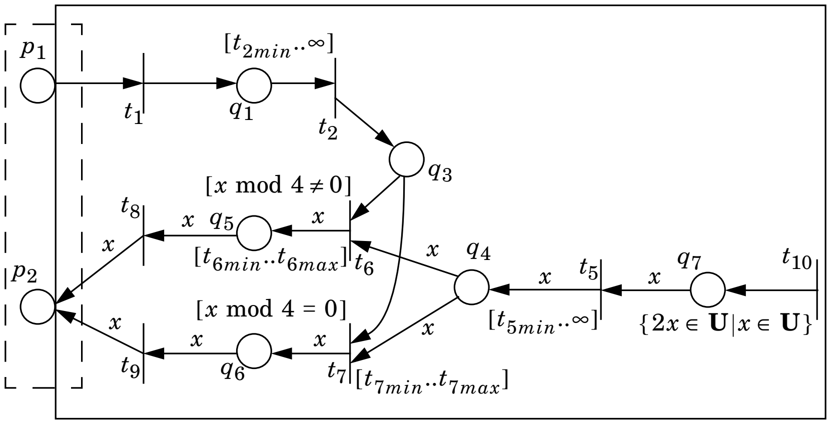


Figure 9.14: The resulting stub after pessimism reduction

sponding to place q_7. q_7 has in turn only one transition in its preset, t_{10}.

The function of t_{10}, $f_{t_{10}}(x) = 2x$, is not constant either. $f_{t_{10}}^{-1}(3) = \varnothing$, which means that buildStub stops and reports success. The unwanted value is eliminated.

Since there are no more join transitions in the diagnostic trace, pessRed also finishes by returning the stub in Figure 9.14. t_{10} only produces even values, so t_5 also only produces even values, which in turn causes the property to be satisfied.

9.4.1 COMPLEXITY ANALYSIS

buildStub is a depth first search with complexity $O(n + a)$, where n is the number of nodes and a is the number of edges in the graph. The main uncertainty in this analysis is the time it takes to invert a function. Assuming that the time for inverting and computing the functions in the graph takes $O(inv)$ in the worst case, the total time complexity of buildStub is $O(n \cdot inv + a)$.

buildStub is called once for each join transition in the diagnostic trace. The overall time complexity is consequently $O(t(n \cdot inv + a))$, where t is the number of join transitions in the trace. Assuming that the number of join transitions in the trace are few and inv is close to constant, we obtain a complexity close to linear.

9.5 Experimental Results

The proposed methodology is demonstrated on two examples: the General Avionics Platform (GAP), introduced in Section 7.1, and a cruise controller.

9.5.1 General Avionics Platform

The two components in the GAP example which were modelled and whose interconnection was especially verified were Tracker and Weapon (Figure 9.15). Tracker receives information from component Radar regarding the location of enemy aeroplanes. The pilot may point at a particular aeroplane on his screen and lock the weapons on it. Upon lock, Tracker repeatedly sends information to Weapon about the direction and distance of the

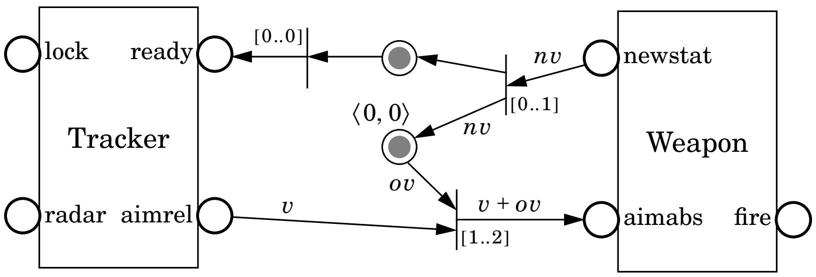


Figure 9.15: The verified glue logic in the GAP example

target aeroplane as long as the lock situation holds. Weapon continuously informs Tracker that it keeps up with the aiming instructions given by Tracker.

Three properties were checked in this setting:

1. Weapon must keep up with the aiming instructions given by Tracker.
   $\mathbf{AG}\ (aimrel \rightarrow \mathbf{AF}\ ready)$
2. Tracker must be able to send the aiming instructions at a certain rate.
   $\mathbf{AG}\ (aimrel \rightarrow \mathbf{AF}_{\leq 5} \neg aimrel)$
3. Tracker must only send aiming instructions within a certain direction (and distance) interval, e.g. it cannot aim backwards.
   $\mathbf{AG}\ (aimabs \rightarrow aimabs \in [min, max])$

The properties (all are (T)ACTL) were verified following the methodology described in this chapter. It was assumed that the only information given by the component provider was the model of the complete component. In particular, no predesigned stubs were provided.

Table 9.1 shows the verification results and times in seconds. T means that the property was satisfied in the corresponding verification environment and F means that it was unsatisfied.

First, stubs were obtained by running the models of Tracker and Weapon through the algorithm described in Section 9.3 (Env 0). In the case when the property was unsatisfied, the diag-

Table 9.1: Verification results and times for the GAP example

Prop.	Env 0	Pessimism Reduction	Env 1	Sum	Naïve
1	T 0.200	-	(T 15.104)	0.200	N/A
2	T 0.122	-	(T 4.159)	0.122	N/A
3	F 0.031	≈120	T 3.191	≈123	N/A

nostic trace was investigated and the proper stub had its pessimism reduced (Env 1). The properties were also verified using naïve stubs.

For properties 1 and 2 the verifications went very fast and using Env 0 was sufficient. Property 3 was however unsatisfied in Env 0, so the stub representing the Tracker component needed to have its pessimism reduced. The time accounted for pessimism reduction was spent on manual work like investigating the diagnostic trace and running the pessimism reduction algorithm. A low estimation of the time for pessimism reduction was 2 minutes, shown in the third column of Table 9.1.

Verifying with Env 1 took longer time due to its larger model complexity. For curiosity properties 1 and 2 were also verified using Env 1 although it would not have been necessary according to the methodology. Not surprisingly, it took substantially longer time than verifying them with Env 0. In either case, verifying the properties with stubs obtained by the algorithms was tremendously much faster than using naïve stubs.

Using naïve stubs took substantially longer time than using Env 0 or Env 1. The available verification equipment was not capable of efficiently handling the big amount of required memory leading to verification times of several weeks. For this reason, no results can be presented.

9.5.2 CRUISE CONTROLLER

The second set of experiments was done on a model of a car cruise controller (Figure 9.16). When the cruise controller is activated by the driver of the car, a signal is sent to the cruise controller module (CCM). The CCM immediately records the current speed (reference speed) which it will try to keep until the cruise controller is turned off. If it notices that the current speed of the car is lower than the reference value, it sends signals to the engine controller module (ECM) to increase the torque. If the

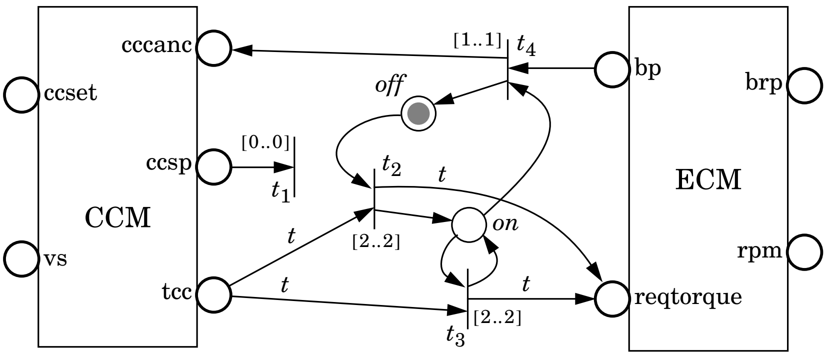


Figure 9.16: The verified glue logic in the cruise controller example

Table 9.2: Verification results and times for the CCM example

Prop	**Env 0**	**Pessimism Reduction**	**Env 1**	**Sum**	**Naïve**
1	F 0.147	≈120	-	≈120	N/A
2	T 151.3	-	(T 22905)	151.3	N/A
3	F 0.146	≈120	T 26095	≈26220	N/A

speed is higher than expected, the opposite command is issued. In case the driver pushes the brake pedal, a signal is sent to the CCM to turn off itself.

The properties to be verified are the following:

1. The brake signal must be processed sufficiently fast.
   $\mathbf{AG}\ (bp \rightarrow \mathbf{AF}_{<1} \neg bp)$
2. The requested torque is below 100%.
   $\mathbf{AG}\ (reqtorque \rightarrow reqtorque \leq 1)$
3. The reference value is positive.
   $\mathbf{AG}\ (ccsp \rightarrow ccsp \geq 0)$

All properties are (T)ACTL and it is assumed that no stubs were provided by the designer of the components.

Table 9.2 presents the verification results and times in seconds in the same style as the results given in Table 9.1. The same procedure was followed in these experiments as in the GAP example.

The verification of property 1 showed that it was unsatisfied with Env 0. Hence, the diagnostic trace was examined. It turned out that the error was located in the glue logic, not in any of the stubs. The reason was that the brake signal (token) is never consumed if the CCM was turned off as can be seen in Figure 9.16, transition t_4.

The difference of verification times between the GAP and CCM examples are several orders of magnitude. The reason is twofold:

1. Bigger interaction with inherently random system environment, e.g. turning on and off the system, braking or varying driving pattern (speed).
2. The generated stubs are nearly as big as the components themselves, due to their structure.

Although, the verification times are long, they are still far from the situation using naïve stubs.

9.6 Verification Methodology Roadmap

This section continues the verification roadmap introduced in Section 7.4 based on the work presented in this chapter.

The question answered in Figure 7.7 gives us the assumption that we must ourselves generate the stubs used in the verification. As indicated by the next question (Figure 9.17) it is necessary to have a model of the whole component in order to be able to proceed with the verification. If such a model exists, a stub is created for the interface in question using the algorithm in Section 9.3. In the next step, the property is verified using the generated stub. If the property was satisfied and (T)ACTL, it is

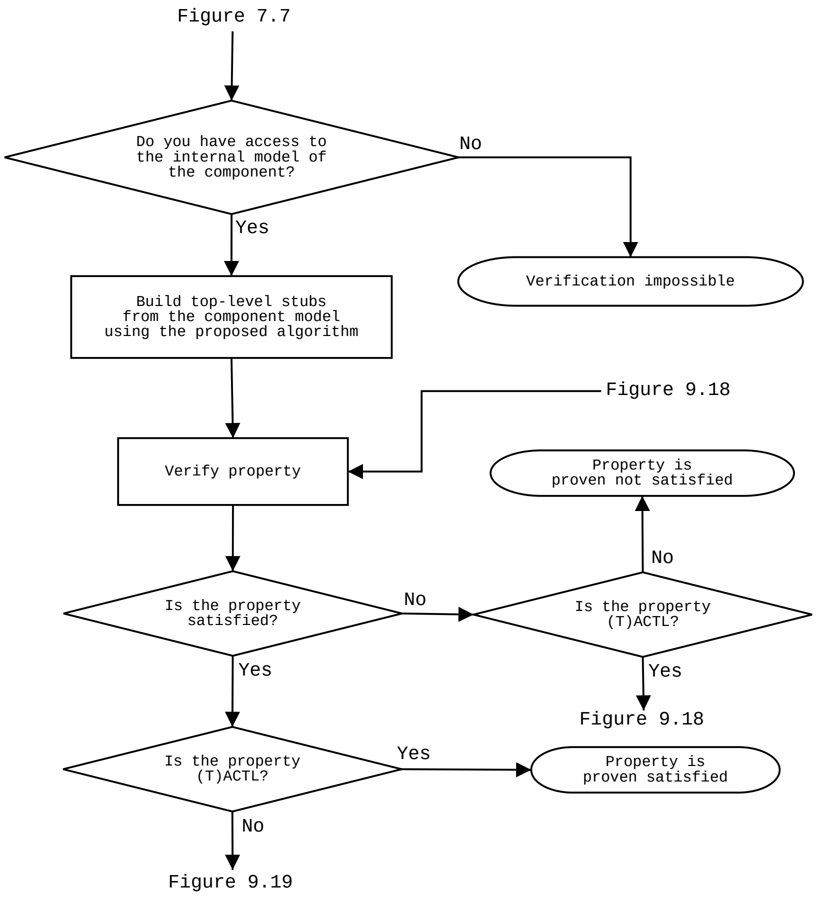


Figure 7.7
Do you have access to the internal model of the component?
No
Yes
Verification impossible
Build top-level stubs from the component model using the proposed algorithm
Figure 9.18
Verify property
Property is proven not satisfied
No
Is the property satisfied?
No
Is the property (T)ACTL?
Yes
Yes
Figure 9.18
Is the property (T)ACTL?
Yes
Property is proven satisfied
No
Figure 9.19

Figure 9.17: Continuation from Figure 7.7 when no stubs are provided by the designer

proven that the system satisfies it. If the property was satisfied, but not ACTL, naïve stubs might need to be used (Figure 9.19). Otherwise, if the property was not satisfied and not (T)ACTL, it is proven unsatisfied in the system. However, if the property is (T)ACTL, pessimism has to be reduced in the stub (Figure 9.18).

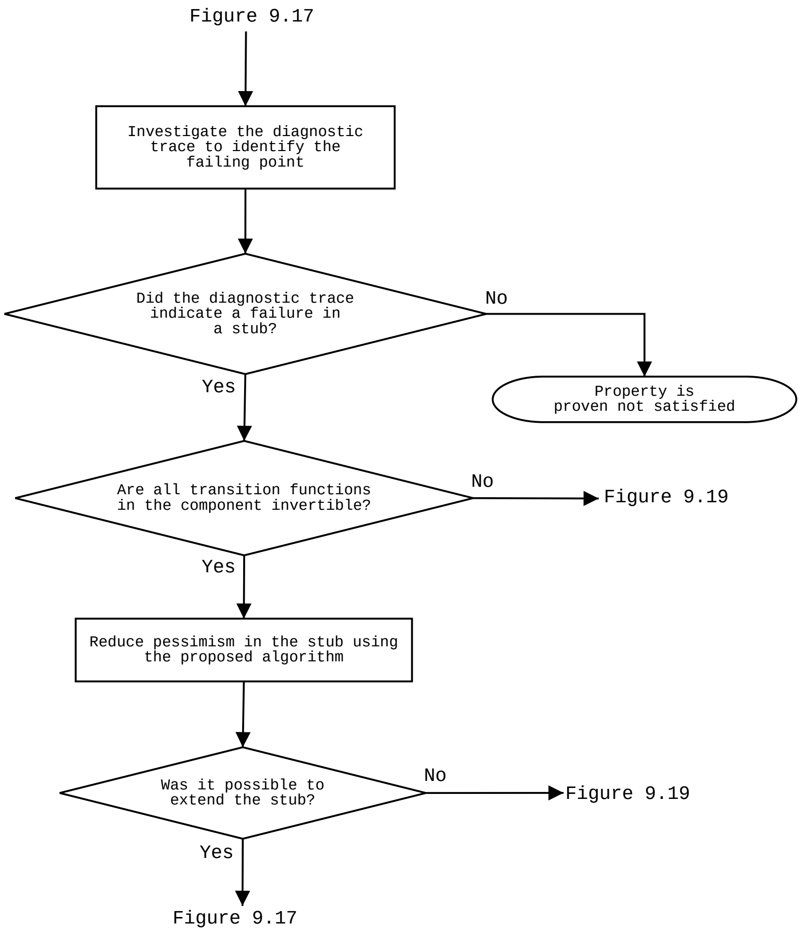


Figure 9.17
Investigate the diagnostic trace to identify the failing point
Did the diagnostic trace indicate a failure in a stub?
No
Property is proven not satisfied
Yes
Are all transition functions in the component invertible?
No
Figure 9.19
Yes
Reduce pessimism in the stub using the proposed algorithm
Was it possible to extend the stub?
No
Figure 9.19
Yes
Figure 9.17

Figure 9.18: Continuation of the roadmap from Figure 9.17

Pessimism is reduced by first investigating the diagnostic trace obtained from the verification. If the trace indicated a fault in the glue logic, i.e. not in a stub, the property is proven not satisfied. In case the fault was found in the stub and all functions of the component are invertible, pessimism is reduced according to

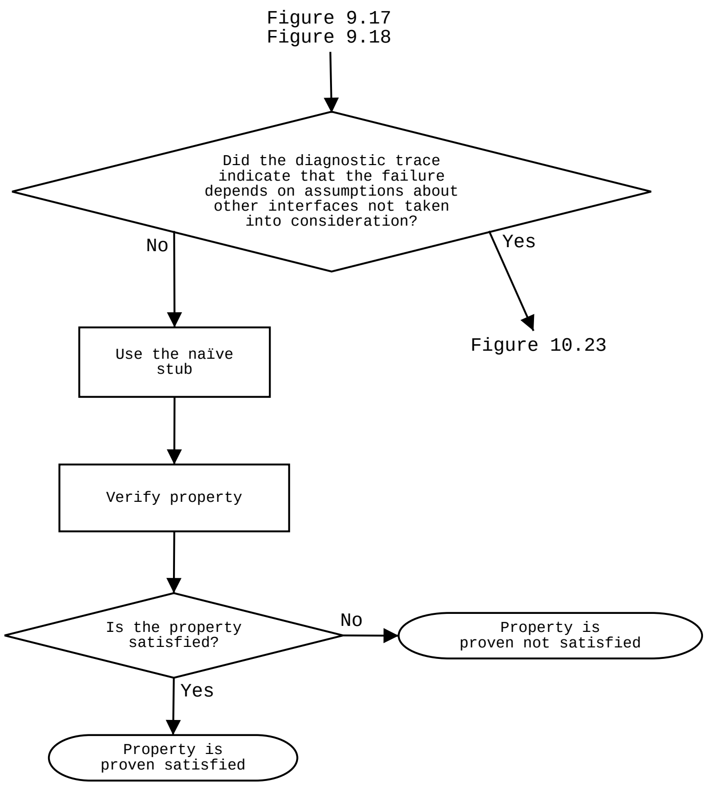


Figure 9.17
Figure 9.18
Did the diagnostic trace indicate that the failure depends on assumptions about other interfaces not taken into consideration?
No
Yes
Figure 10.23
Use the naïve stub
Verify property
Is the property satisfied?
No
Property is proven not satisfied
Yes
Property is proven satisfied

Figure 9.19: Continuation of the roadmap from Figure 9.17 and Figure 9.18

the algorithm presented in Section 9.4. If not all functions are invertible, naïve stubs have to be considered. The same happens if it was impossible to further reduce pessimism. When a new less pessimistic stub has been obtained, the property is verified again.

As mentioned above, if a verification result could not be concluded, naïve stubs have to be considered (Figure 9.19). However, if the diagnostic trace suggested that assumptions on the surrounding are violated, using naïve stubs will not solve the problem (see discussion around Figure 9.3 about the difference between naïve and exact stubs). The solution to that problem is presented in Chapter 10. Otherwise, the naïve stub is used straight-forwardly.

Chapter 10 Modelling the Surrounding

TOGETHER WITH EACH COMPONENT, a set of (T)CTL formulas is provided as requirements on the input on all interfaces of the component. However, stubs generated by the algorithms presented in Chapter 9 disregard from this fact and always assume the worst case surrounding. A less pessimistic verification result might be obtained if the information provided by the formulas on other interfaces than those being verified are incorporated into the verification process in the place of those interfaces. Moreover, system specific assumptions about the surrounding might also have to be made in order to obtain a good verification result. In this way, stubs no longer assume the worst case surrounding but a surrounding satisfying certain given requirements. Figure 10.1 illustrates this mechanism.

This chapter will present an algorithm which translates an arbitrary ACTL formula into a PRES+ model, such that this model can produce all possible observations (behaviours) still

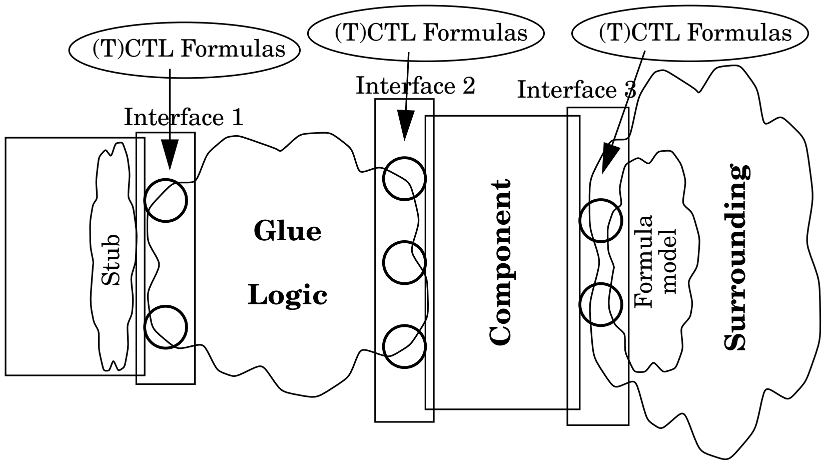


Figure 10.1: Overview of the methodology presented in this chapter

consistent with the formula. The resulting PRES+ model is then attached to the component on the interface on which the formula was expressed. The component with the attached formula model (Figure 10.1) is then treated as a stub in the subsequent verification.

Existing work has already approached this issue using finite automata on infinite words for LTL and ACTL [Gru94]. The work presented in this chapter is based on this translation method. In fact, many definitions presented in this chapter, in particular in Section 10.2.1, are based on similar definitions in [Gru94], although most of them are modified in order to fit the PRES+ representation and our interpretation of CTL formulas (see Section 3.3).

Other work tries to remove the restriction of ACTL and be able to derive automata for all CTL formulas [Kup96]. However, in this case the translation cannot be performed into normal automata on infinite words, but only into so called tree automata. Since there is no direct correspondence between tree automata and Petri-nets, this generalisation cannot be applied in our case. Consequently, the translation algorithm presented below assumes an ACTL formula, or a conjunction or disjunction of ACTL formulas. Conjunctions of formulas are of special interest since they allow to create one single PRES+ model from several formulas.

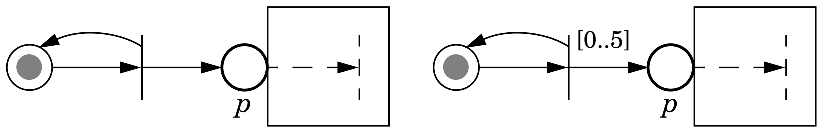
10.1 Preliminaries

10.1.1 INTRODUCTORY EXAMPLE

Consider the ACTL formula **AGAF** p . The formula states that p must repeatedly hold some time in the future. It is however not defined when this future must come, only that it must come eventually. Figure 10.2(a) shows an ad hoc construction of a PRES+ model representing this formula. It should be noted that all generated models are connected to a component. Therefore, it might happen that tokens disappear or appear in the ports, without an explicit transition firing in the generated PRES+ model.

Unfortunately, the model in Figure 10.2(a) does not fully correspond to the formula, since there is nothing which will ever force the transition to fire. As a result, it is not certain that p will be marked in the future.

In order to avoid this problem, the **F** and **U** operators in the ACTL formula must have an upper time bound, before which the subformula must hold, e.g. $\mathbf{AGAF}_{\leq 5}p$. The time bound is trans-



(a) (b)

Figure 10.2: Petri-nets constructed ad hoc for the formula **AGAF** p

ferred to the corresponding transition. The model now has a mechanism to force the transition to fire and p will be repeatedly marked in the future. From here on, in the context of modelling ACTL formulas in PRES+, it is assumed that such time bounds on **F** and **U** operators exist. A PRES+ model for the example formula is shown in Figure 10.2(b).

10.1.2 FORMULA NORMALISATION

In order to simplify the algorithm, the formula for which a PRES+ model should be generated must be written in a normal form according to the following rules:

1. Implications of the form $p \rightarrow q$ must be rewritten as $\neg p \vee q$, so that the only boolean operators in the formula are $\neg$, $\wedge$ and $\vee$.
2. Subformulas of the form $\neg(p \Re v)$, where p is a port, v is a value and $\Re$ is a relation, for example the equality relation =, must be rewritten as $\neg p \vee p \overline{\Re} v$, where $\overline{\Re}$ is the complementary relation of $\Re$, in this case the disequality relation $\neq$, in order to enforce the correct semantics.
3. $\mathbf{AG}\ \varphi$ is rewritten as $\mathbf{A}\ [\mathit{false}\ \mathbf{R}\ \varphi]$.
4. $\mathbf{AF}_{\leq j}\varphi$ is rewritten as $\mathbf{A}\ [\mathit{true}\ \mathbf{U}_{\leq j}\varphi]$.

Table 10.1 shows a few examples of ACTL formulas and their normalisation. *true* and *false* are abbreviated as t and f respectively.

Table 10.1: Examples of (T)ACTL formulas and their normalisation

Formula	**Normalisation**
$\mathbf{AG}\ (p \rightarrow p \leq 5)$	$\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \leq 5)]$
$\mathbf{AF}_{\leq 4}p$	$\mathbf{A}[t\ \mathbf{U}_{\leq 4}p]$
$\mathbf{AGAF}_{\leq 4}p$	$\mathbf{A}[f\ \mathbf{R}\ \mathbf{A}[t\ \mathbf{U}_{\leq 4}p]]$
$\mathbf{AG}\ (p > 10 \rightarrow \mathbf{AF}_{\leq 2}q \leq 5)$	$\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \leq 10 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 2}q \leq 5])]$

10.2 The ACTL to PRES+ Translation Algorithm

The algorithm consists of the following main steps:

1. Place generation.
2. Timer insertion for **U** operators
3. Transition generation
4. Insertion of initial tokens

Each of these four steps is explained in more detail in the following sections. The steps are in principle executed in sequence with minor exceptions. Section 10.2.5 gives a summary of these steps and a final overview of the algorithm is presented.

The basic idea of the algorithm is to identify a set of states (markings) satisfying the particular ACTL formula. Each state represents a particular future behaviour of the model. The PRES+ model changes state as a response to inputs received and outputs emitted, in such a way that the formula will stay satisfied. The resulting PRES+ model will have one place for each such state of future behaviour.

If a state represents a behaviour which includes that a certain event must occur within a certain time bound, it is necessary to insert a mechanism called timers into that state in order to guarantee that the specified event will occur in time.

Transitions are then inserted to represent all possible state changes satisfying the formula.

All states satisfying the formula can potentially be the initial state. The selection must be made dynamically so that all possibilities are accounted for in the verification. A mechanism for selecting the initial state is included at the last step of the algorithm.

The formula $\mathbf{AG}(p \rightarrow \mathbf{AF}_{\leq 3} q < 10)$ will be used as an example for explaining the algorithm presented in this chapter. This formula is rewritten as stated in Equation 10.1 according to the normalisation rules. It is further assumed that p is an out-port and q is an in-port of an attached component. Section 10.3 provides further examples.

$$\psi = \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])] \qquad (10.1)$$

10.2.1 PLACE GENERATION

The first step of the algorithm is to create places to the PRES+ model. Before describing the algorithm, a few definitions and concepts must be presented.

Definition 10.1: Set of elementary formulas. The set $el(\varphi)$ of elementary formulas of the formula φ is defined by the following equations ([Gru94] modified).

1. If $\varphi = true$ or $\varphi = false$, then $el(\varphi) = \varnothing$. If $\varphi = p$ (where p is a port of a component) or $\varphi = \neg p$, then $el(\varphi) = \{p\}$. If $\varphi = p\Re v$, then $el(\varphi) = \{p, p\Re v\}$.
2. If $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \varphi_1 \vee \varphi_2$, then $el(\varphi) = el(\varphi_1) \cup el(\varphi_2)$.
3. If $\varphi = \mathbf{A}[\varphi_1 \mathbf{U}_{\leq j} \varphi_2]$, then $el(\varphi) = \{\mathbf{AX}\,\mathbf{A}\,[\varphi_1 \mathbf{U}_{\leq j} \varphi_2]\} \cup el(\varphi_1) \cup el(\varphi_2)$.
   If $\varphi = \mathbf{A}[\varphi_1\ \mathbf{R}\ \varphi_2]$, then $el(\varphi) = \{\mathbf{AX}\,\mathbf{A}\,[\varphi_1\ \mathbf{R}\ \varphi_2]\} \cup el(\varphi_1) \cup el(\varphi_2)$.

■

Considering the example formula ψ, the set of elementary formulas is shown in Equation 10.2.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])]}_{1}, \underbrace{p}_{2}, \underbrace{\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]}_{4}, \underbrace{q}_{8}, \underbrace{q < 10}_{16} \right\} \qquad (10.2)$$

An elementary formula expresses a certain aspect about the model. **AX** formulas describe a certain future behaviour, whereas atomic propositions say something about the current state of the system.

In the rest of this chapter, large sets of subsets of $el(\varphi)$ will very often be referred. In order to achieve an acceptably condense representation, we will use a numerical notation for subsets of $el(\varphi)$.

Each subset will be labelled S_i, where i is a number according to the following scheme. Every elementary formula is assigned a power of 2, see Equation 10.2. i is the sum of the numbers corresponding to the formulas included in the desired set. Table 10.2 lists all subsets of $el(\psi)$ with their associated S_i annotation.

Definition 10.2: Subformula. The set $sub(\varphi)$ of subformulas of the formula φ is defined by the following equations ([Gru94] modified).

1. If $\varphi = true$ or $\varphi = false$ or $\varphi = p$ or $\varphi = p\Re v$ (an atomic proposition), then $sub(\varphi) = \{\varphi\}$. If $\varphi = \neg p$, then $sub(\varphi) = \{\varphi, p\}$.
2. If $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \varphi_1 \vee \varphi_2$ or $\varphi = \mathbf{A}[\varphi_1 \mathbf{U}_{\leq j} \varphi_2]$ or $\varphi = \mathbf{A}[\varphi_1 \mathbf{R}\ \varphi_2]$, then $sub(\varphi) = \{\varphi\} \cup sub(\varphi_1) \cup sub(\varphi_2)$.

■

Equation 10.3 presents the set of all subformulas of the example formula ψ.

$$sub(\psi) = \{\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])], f, \neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10], \neg p, p, \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10], t, q < 10\} \tag{10.3}$$

Definition 10.3: Atomic propositions. The set of atomic propositions in a formula φ is defined as $AP(\varphi) = el(\varphi) - \{\mathbf{AX}\ \varphi_1 \in el(\varphi)\}$. This function can also be lifted to sets of formulas: $AP(\Psi) = \bigcup_{\varphi \in \Psi} AP(\varphi)$. It is con-

Table 10.2: Listing of all subsets of $el(\psi)$

S_i	**Subset of** $el(\psi)$
0	$\varnothing$
1	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])]\}$
2	$\{p\}$
3	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p\}$
4	$\{\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10]\}$
5	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10]\}$
6	$\{p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10]\}$
7	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10]\}$
8	$\{q\}$
9	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], q\}$
10	$\{p, q\}$
11	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p, q\}$
12	$\{\mathbf{AX}\ (\mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10]), q\}$
13	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q\}$
14	$\{p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q\}$
15	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q\}$
16	$\{q < 10\}$
17	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], q < 10\}$
18	$\{p, q < 10\}$
19	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p, q < 10\}$
20	$\{\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q < 10\}$
21	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q < 10\}$
22	$\{p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q < 10\}$
23	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q < 10\}$
24	$\{q, q < 10\}$
25	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], q, q < 10\}$
26	$\{p, q, q < 10\}$
27	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], p, q, q < 10\}$
28	$\{\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q, q < 10\}$
29	$\{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10])], \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q, q < 10\}$
30	$\{p, \mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3}q < 10], q, q < 10\}$
31	$el(\psi)$

venient to additionally define $AP_{in}(\varphi)$ and $AP_{out}(\varphi)$ to mean the set of atomic propositions which denote in-ports and out-ports of a connected component, respectively. Furthermore, let $AP_{rel}(\varphi) = AP(\varphi) - AP_{in}(\varphi) - AP_{out}(\varphi)$ denote the set of atomic propositions with relations, and $AP_{rin}(\varphi)$ and $AP_{rout}(\varphi)$ those atomic propositions with relations which refer to an in-port or out-port respectively.

■

In the example formula, $AP(\psi) = \{p, q, q < 10\}$, $AP_{out}(\psi) = \{p\}$ and $AP_{in}(\psi) = \{q\}$. $AP_{rel}(\psi) = AP_{rout}(\psi) = \{q < 10\}$ and $AP_{rin}(\psi) = \emptyset$.

The atomic propositions with relation in a set of elementary formulas s impose certain restrictions on the token values in the ports corresponding to the atomic proposition. The universe from which values are chosen is denoted **U** (see also Section 9.3.3). **U** contains all values which could possibly occur in the design. In all examples of this chapter, it is assumed that the universe is the set of all integers, $\mathbf{U} = \mathbf{Z}$.

Definition 10.4: Port values. The set of in-port values of the set of elementary formulas s is defined as $PV_{in}(s) = \{k \in \mathbf{U} | \forall p \Re v \in s : (p \in AP_{in}(s) \rightarrow k \Re v)\}$. The set of out-port values is defined with respect to a particular out-port p as $PV_{out}(s, p) = \{k \in \mathbf{U} | \forall p \Re v \in s : k \Re v\}$, where $p \in AP_{out}(s)$.

■

The set of in-port values of a set of elementary formulas s is the set of values in the universe **U** which satisfy all relations, with an atomic proposition referring to an in-port, in s. The atomic propositions in the relations do not need to be the same. There must still exist a value which satisfies all in-port relations in the set. The reason that all in-port relations must be satisfied simultaneously disregarding the atomic proposition in the relation stems from the fact that transitions in PRES+ only can have one function. The values produced by that function must simultaneously satisfy all atomic propositions with relation corresponding to in-ports in s.

In the case of out-ports, each atomic proposition is examined separately, as opposed to in-ports, since transition functions may have several arguments.

In the example, $PV_{in}(S_9) = \{\dots, -1, 0, 1, 2, \dots\} = \mathbf{Z}$ and $PV_{in}(S_{25}) = \{k \in \mathbf{Z} | k < 10\} = \{\dots, -1, 0, 1, 2, \dots, 9\}$. $PV_{out}(S_{27}, p) = \mathbf{Z}$ and $PV_{out}(S_{27}, q)$ does not exist since q is not an out-port.

Having defined port values, it is possible to determine which sets of elementary formulas are legal.

Let ψ denote the formula for which a PRES+ model is to be constructed. Let $S(\psi)$ be defined as indicated by Equation 10.4, where $p, p\Re v \in AP(\psi)$. $S(\psi)$ is the power set of $el(\psi)$ but where subsets containing a contradictory set of elementary formulas are removed. A set of elementary formulas s can be contradictory for two reasons:

1. The set contains an atomic proposition with relation ($p\Re v$), but not the atomic proposition itself (p). Such a set is contradictory since the relation says that place p contains a token related in the particular way, but the absence of the atomic proposition p indicates that p, on the contrary, does not contain any token, in other words $\exists p\Re v \in s : p \notin s$. (E.g. $\{p\Re v, q, \mathbf{AX}\, \mathbf{A}[f\ \mathbf{R}\ q]\}$)
2. The set contains atomic propositions with relations, where there does not exist any value that can satisfy all relations corresponding to in-ports at the same time, $PV_{in}(s) = \varnothing$. The same holds for out-ports, but for each atomic proposition taken separately, in other words $\exists p \in AP_{out}(\psi) : PV_{out}(s, p) = \varnothing$. (E.g. $\{q, q < 10, q > 20\}$)

$$S(\psi) = 2^{el(\psi)} - \{s \in 2^{el(\psi)} | \exists p\Re v \in s : p \notin s\} - \\ \{s \in 2^{el(\psi)} | PV_{in}(s) = \varnothing \vee \exists p \in AP_{out}(\psi) : PV_{out}(s, p) = \varnothing\} \qquad (10.4)$$

Definition 10.5: Legal (Contradictory) set of elementary formulas. A set of elementary formulas, s, is legal if and only if $s \in S(\psi)$, and s is contradictory if and only if $s \notin S(\psi)$, where $S(\psi)$ is defined as in Equation 10.4.

■

The set $\{\mathbf{AX}\,\mathbf{A}[p\ \mathbf{R}\ q], p < 10\}$ is contradictory since the formula p is not a member of the set, but $p < 10$ is. $\{p < 10, p, q > 20, q\}$ is also a contradictory set assuming that both p and q are in-ports, since there does not exist any value which is both less than 10 and greater than 20. However, assuming than p is an out-port and q is an in-port makes the same set legal. $\{\mathbf{AX}\,\mathbf{A}[p\ \mathbf{R}\ q], p < 10, p, q > 5, q\}$ is a legal set of elementary formulas, even assuming that both p and q are in-ports.

Continuing with the example formula ψ in Equation 10.1, $S(\psi) = \{S_0..S_{15}, S_{24}..S_{31}\}$. Elements S_{16} to S_{23} are not included into S, since they contain $q < 10$ but not q and hence are contradictory.

Identifying $S(\psi)$ simplifies the rest of the algorithm, in the sense that it no longer needs to consider contradictory situations. From now on, only legal sets of elementary formulas are considered.

After having identified the legal sets of elementary formulas, it is needed to find out which legal sets of elementary formulas satisfy the formula ψ for which a PRES+ model should be generated. Definition 10.6 introduces a function, Φ, for this purpose.

Definition 10.6: $\Phi(\varphi)$. Formula mapping $\Phi(\varphi)$ from $el(\psi) \cup sub(\psi) \cup \{true, false\}$ to $S(\psi)$ is defined recursively as follows [Gru94]:

1. $\Phi(true) = S(\psi)$, $\Phi(false) = \emptyset$. If $\varphi \in el(\psi)$, then $\Phi(\varphi) = \{s \in S(\psi) | \varphi \in s\}$. If $\varphi = \neg\varphi_1$, then $\Phi(\varphi) = S(\psi) - \Phi(\varphi_1)$.

2. If $\varphi = \varphi_1 \wedge \varphi_2$, then $\Phi(\varphi) = \Phi(\varphi_1) \cap \Phi(\varphi_2)$. If $\varphi = \varphi_1 \vee \varphi_2$, then $\Phi(\varphi) = \Phi(\varphi_1) \cup \Phi(\varphi_2)$.
3. If $\varphi = \mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$, then $\Phi(\varphi) = \Phi(\varphi_2) \cup (\Phi(\varphi_1) \cap \Phi(\mathbf{AX}\ \varphi))$. If $\varphi = \mathbf{A}\,[\varphi_1\ \mathbf{R}\ \varphi_2]$, then $\Phi(\varphi) = \Phi(\varphi_2) \cap (\Phi(\varphi_1) \cup \Phi(\mathbf{AX}\ \varphi))$.

■

$\Phi(\varphi)$ denotes the maximal set of legal elementary formulas satisfying the formula φ. This intuitively means that the algorithm should generate a PRES+ model which realises $\Phi(\psi)$, i.e. can produce all events described by the sets in $\Phi(\psi)$.

The following results are useful for later illustration of the example and were obtained as partial results while computing $\Phi(\psi)$.

- $\Phi(\psi) = \Phi(\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])]) = \{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\}$
- $\Phi(\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])]) = \{S_1, S_3, S_5, S_7, S_9, S_{11}, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\}$
- $\Phi(\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]) = \{S_0, S_1, S_4, S_5, S_6, S_7, S_8, S_9, S_{12}, S_{13}, S_{14}, S_{15}, S_{24}, S_{25}, S_{26}, S_{27}, S_{28}, S_{29}, S_{30}, S_{31}\}$
- $\Phi(\neg p) = \{S_0, S_1, S_4, S_5, S_8, S_9, S_{12}, S_{13}, S_{24}, S_{25}, S_{28}, S_{29}\}$
- $\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]) = \{S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}, S_{24}, S_{25}, S_{26}, S_{27}, S_{28}, S_{29}, S_{30}, S_{31}\}$
- $\Phi(\mathbf{AX}\ (\mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])) = \{S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}, S_{28}, S_{29}, S_{30}, S_{31}\}$
- $\Phi(q < 10) = \{S_{24}, S_{25}, S_{26}, S_{27}, S_{28}, S_{29}, S_{30}, S_{31}\}$

Definition 10.7: Progress formulas. A progress formula is any elementary formula except atomic propositions. Assuming a set of elementary formulas Ψ, $PF(\Psi) = \Psi - AP(\Psi)$. This function can also be lifted to sets of sets of formulas, $PF(\Gamma) = \bigcup_{\Psi \in \Gamma} PF(\Psi)$.

■

For example, $PF(S_{31}) = S_5$. $PF(\Phi(\psi)) = \{S_1, S_5\}$ since $PF(S_1) = PF(S_9) = PF(S_{25}) = PF(S_{27}) = S_1$ and $PF(S_5) = PF(S_7) = PF(S_{13}) = PF(S_{15}) = PF(S_{29}) = PF(S_{31}) = S_5$.

Progress formulas express how the system should behave (progress) over a period of time and therefore tell us something about the future. Atomic propositions, on the other hand, only express the current state of the system. For this reason, we need to treat progress formulas and atomic propositions separately.

Each set of elementary formulas will correspond to a state (marking) in the final PRES+ model. $\Phi(\psi)$ consequently denotes the set of all states satisfying the property ψ. All these states, as well as transitions between them, must thus be captured by the final PRES+ model. Each set of possible progress formulas $pf \in PF(\Phi(\psi)) \cap \Phi(\psi)$ (The sets in $\Phi(\psi)$ which only contain progress formulas) will have a corresponding place. In this way, it is possible to express all states. The intersection is added to ensure that all sets of progress formulas for which a place is to be created are indeed satisfied by the property.

In our example introduced on page 215, the resulting PRES+ model will have two places corresponding to the two sets in $PF(\Phi(\psi)) \cap \Phi(\psi) = \{S_1, S_5\}$. A token in the place corresponding to the set of progress formulas S_5 and another token in port p reflects the state (set of elementary formulas) S_7. State S_{25} is modelled by a token in the place corresponding to progress formulas S_1 together with a token in q with a value less than 10. This state actually also corresponds to S_9, an observation which is important in the context of redundancy (Definition 10.12).

Figure 10.3 shows the algorithm (createInitialPlaces) for creating the places which initially have to be present in the resulting

```
1   procedure createInitialPlaces($\psi$: ACTL)
2       for each s $\in$ PF($\Phi(\psi)$) $\cap$ $\Phi(\psi)$ do
3           createPlace(s);
4       for each p $\in$ AP($\psi$) do
5           add a place $p_p$ to net;
6           P(p) := $p_p$;
7
8   procedure createPlace(s: set of ACTL)
9       add a place $p_i$ to net;
10      $\Psi(p_i)$ := s;
11      $P_{in}$(s) := { $p_i$ };
12      P(s) := $p_i$;
13      addTimers($p_i$); -- defined in Figure 10.4
```

Figure 10.3: The algorithm for creating the places in the resulting PRES+ model

PRES+ model. The variable net is a global variable of type PRES+ which in the end will contain the final resulting model. The procedure createPlace is taken out from the main procedure since it will be needed later for creating additional places to the model.

One place for each member set $s \in PF(\Phi(\psi)) \cap \Phi(\psi)$ (Line 2, Line 3 and Line 9) is created. Denote the set of progress formulas that a place p_i corresponds to as $\Psi(p_i)$ (Line 10). Dually, for each place p_i, a set of places associated to a set of elementary formulas s, $P_{in}(s)$, is maintained. It will record the places that have to be marked when the PRES+ model enters the state represented by s. Their function will become clear in Section 10.2.3, where transitions will be added to the model. Initially, $P_{in}(s) = \{p_i\}$ (Line 11). $P(s) = p_i$ maps a set of elementary formulas to the place. As opposed to $P_{in}(s)$, $P(s)$ will not be modified during the course of later steps of the algorithm. In the end, a mechanism called *timers* may have to be added to the place (Line 13). The purpose of timers, their functionality and how they are added will be described in Section 10.2.2.

Ports corresponding to the atomic propositions (without relations) occurring in the formula must also be added to the model (Line 5). They are moreover associated to their corresponding atomic proposition through the mapping $P(p)$ (Line 6).

In our example, the resulting model has two places (excluding timers), p_1 and p_5, corresponding to formulas in S_1 and S_5 respectively. This means that $\Psi(p_1) = S_1$, $\Psi(p_5) = S_5$, and $P_{in}(S_1) = \{p_1\}$ and $P_{in}(S_5) = \{p_5\}$. Moreover, the formula has two atomic propositions p and q, so two places p_p and p_q are created for this reason. $P(p) = p_p$, and $P(q) = p_q$.

10.2.2 Timer Insertion for **U** Operators

In Section 10.1.1, it was concluded that **F** and **U** operators are forced to have an associated upper time bound (deadline) before which a certain specified event has to occur. Since **F** operators are rewritten as **U** operators in the normalisation, only **U** operators need to be considered.

In order to make sure that the desired events eventually will happen and that they will happen in time, timers must be introduced.

In Definition 10.8, it should be remembered that $\Psi(p_i)$ denotes the set of progress formulas for which the place p_i was created. See also Line 3 in Figure 10.3.

Definition 10.8: Set of U formulas. The set of U formulas in place p_i is expressed as $U(p_i) = \{\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] | \mathbf{AX}\,\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \in \Psi(p_i)\}$.

■

Place p_1 in the example does not have any U formula, $U(p_1) = \emptyset$ and place p_5 has got one, $U(p_5) = \{\mathbf{A}\ [t\ \mathbf{U}_{\leq 3} q < 10]\}$.

```
procedure addTimers($p_i$: place)
    for each $\varphi \in U(p_i)$ do
        add places $p_{ix}$ and $p_{ix}$' as indicated by Figure 10.5 to net;
        add transition $t_{ix}$ asin indicated by Figure 10.5 to net;
        set time delay of $t_{ix}$ to [0..j] where j is the upper bound
            associated to the U operator in $\varphi$;
        $P_{in}(\Psi(p_i)) := P_{in}(\Psi(p_i)) \cup \{ p_{ix} \}$;
        Timerin($p_i$, $\varphi$) := $p_{ix}$;
        Timerout($p_i$, $\varphi$) := $p_{ix}$' ;
```

Figure 10.4: Algorithm for adding timers to a place

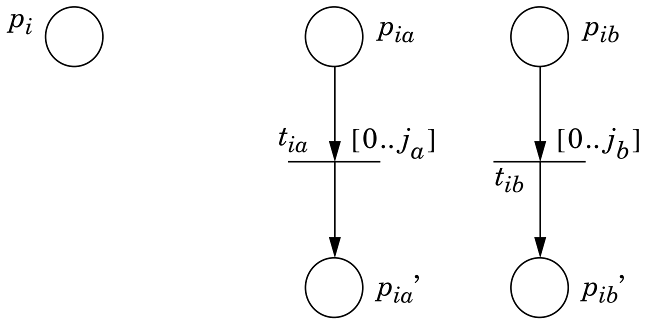


Figure 10.5: Adding timers to a place

One timer per U formula in a place must be added, so that the deadline of each U formula can be timed independently from each other. Figure 10.4 presents the algorithm for adding timers to a place p_i. The algorithm is also illustrated in Figure 10.5 for the case when p_i has two U formulas ($|U(p_i)| = 2$).

A timer is a piece of the PRES+ model consisting of two places (e.g. p_{ia} and p_{ia}') between which there exists a transition from p_{ia} to p_{ia}' with a time delay interval of the type $[0..j]$ for any non-negative number j (e.g t_{ia}). Place p_{ia} is called the start place of the timer, and p_{ia}' is called the end place. Places p_{ib} and p_{ib}' and transition t_{ib} together constitute another timer.

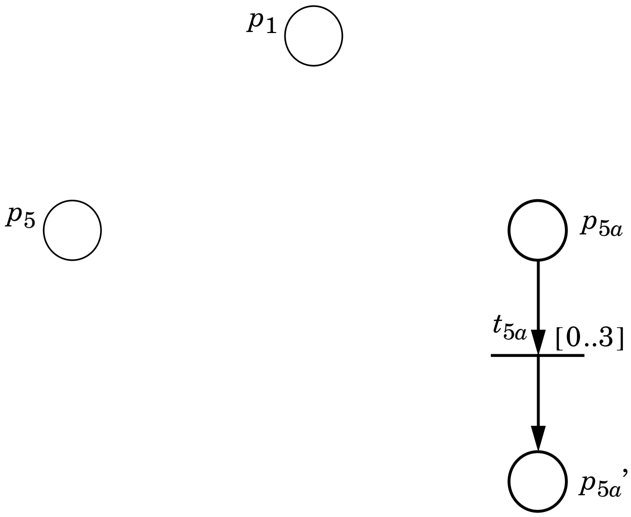


Figure 10.6: Adding timers to the example model

All timers must be simultaneously started when the PRES+ model enters the state represented by p_i. The mapping $P_{in}(\Psi(p_i))$ contains all places to be marked when the Petri-net should enter the particular state. Consequently, all start places p_{ix} are added to the mapping (Line 6).

The exact use of these mappings will become clear in Section 10.2.3. $Timerin(p_i, \varphi)$ and $Timerout(p_i, \varphi)$ record the start and end places respectively of the timer corresponding to U formula φ, for future reference (Line 7 and Line 8).

In the case of the example formula ψ (defined on page 215), only place p_5 has a timer. Figure 10.6 presents the result of adding the timer corresponding to the only U formula in that place. At this point, $P_{in}(\Psi(p_5)) = \{p_5, p_{5a}\}$. It still holds that $P_{in}(\Psi(p_1)) = \{p_1\}$.

Moreover, $Timerin(p_5, \mathbf{A}\ [t\ \mathbf{U}_{\leq 3} q < 10]) = p_{5a}$ and $Timerout(p_5, \mathbf{A}\ [t\ \mathbf{U}_{\leq 3} q < 10]) = p_{5a}'$.

10.2.3 TRANSITION GENERATION

The major remaining step, after adding places and timers, is to add the transitions to the PRES+ model.

Definition 10.9: Target formulas. The set of target formulas of a place p_i is defined as $TF(p_i) = \bigcap_{\mathbf{AX}\ \varphi \in \Psi(p_i)} \Phi(\varphi)$. ■

Definition 10.10: Target places. The set of target places of a place p_i is defined as $TP(p_i) = PF(TF(p_i))$. ■

The set of target formulas contains the sets of elementary formulas representing the events which can happen next, given that there is a token in p_i. Consequently, as a basic rule, one transition will be added for each set of elementary formulas in $TF(p_i)$ realising the particular event described by that set of elementary formulas. However, as will be seen later, no transitions will be added for sets of elementary formulas which do not contain any atomic proposition, since these sets do not contribute to any event on the ports.

The set of target places contains the sets of progress formulas representing places in the PRES+ model (see Section 10.2.1) to which there is a target formula. In other words, $TP(p_i)$ is the set of what is left when all atomic propositions have been removed from all sets in $TF(p_i)$.

In our example, $TF(p_1) = \Phi(\psi) = \{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\}$ and $TF(p_5) = \{S_5, S_7, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\}$ according to the more detailed computation in Equation 10.5. In addition, $TP(p_1) = TP(p_5) = \{S_1, S_5\}$.

$$\begin{aligned} TF(p_5) &= \Phi(\psi) \cap \Phi(\mathbf{A}\ [t\ \mathbf{U}_{\leq 3} q < 10]) = \\ &\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\} \cap \\ &\{S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}, S_{24}, S_{25}, S_{26}, S_{27}, \\ &S_{28}, S_{29}, S_{30}, S_{31}\} = \\ &\{S_5, S_7, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\} \end{aligned} \tag{10.5}$$

Let us assume that q is an in-port of a component, $q \in AP_{in}(el(\psi))$. Assume further that the PRES+ model is in a state $p_i \cup \{q\}$, i.e. a token in the place p_i and another token in q. Assume further that this state satisfies the formula φ, $\Psi(p_i) \cup \{q\} \in \Phi(\varphi)$. Since the PRES+ model is connected to a component at q, a transition in that component may consume the token, which forces the model to change state to p_i (just a token in p_i, and no token in q). However, it is possible that state p_i might not satisfy formula φ, $\Psi(p_j) \notin \Phi(\varphi)$, so the possibility of involuntarily ending up in this state must be eliminated. Remember that the model ended up in this state as a result of an act of the component connected at port q, not as a result of the model representing the ACTL property itself. The concept of validity is defined to identify such situations.

Definition 10.11: Valid elementary set. A set of elementary formulas, s, is a valid elementary set in place p_i, if for all $p \in AP_{in}(s)$,
$s - \{p\} - \{p \Re v \in s\} - AP_{out}(s) - AP_{rout}(s) \in TF(p_i)$ and $s - \{p\} - \{p \Re v \in s\}$ is recursively a valid elementary set.
■

In the example, S_{27} is a valid elementary set in place p_1, since $S_{27} - \{q\} - \{q < 10\} = S_1 \in TF(p_1)$ and q is the only atomic proposition corresponding to an out-port. An example of a situation where a set of elementary formulas is not valid is given in Section 10.3.2.

Although the resulting model will be correct, adding a transition for each element in $TF(p_i)$ might lead to a model with unnecessarily many transitions. That could lead to longer verification times than needed. Some transitions might namely be redundant.

Definition 10.12: Redundant elementary set. A set of elementary formulas s is redundant with respect to a set of sets of elementary formulas S, if and only if there exists a set

$s' \in S$, $s \neq s'$, with $PF(s) = PF(s')$, $AP_{in}(s) = AP_{in}(s')$, $AP_{out}(s) = AP_{out}(s')$ and $PV_{in}(s) \subseteq PV_{in}(s')$ and $\forall p \in AP_{out}(s){:}PV_{out}(s, p) \subseteq PV_{out}(s', p)$.

■

Intuitively, s is redundant with respect to S, if there is another set s' with the same progress formulas and the same atomic propositions but where s' can produce more values than s, and accept more values than s on each of their input places. The transition corresponding to s' can hence produce the very same events as s (and more). The conclusion is that the transition corresponding to s is redundant and does not need to be included in the resulting model.

It is, for example, not necessary to add a transition t for a set containing $q < 10$, if there already is a transition t' for a set containing q, but not $q < 10$, since t' anyway is able to produce all events produced by t.

In our example, it is evident that, in place p_1, a transition for $S_{25} = S_1 \cup \{q, q < 10\}$ is not needed since there exists a transition for $S_9 = S_1 \cup \{q\}$, $PF(S_{25}) = PF(S_9) = S_1$, $AP_{in}(S_{25}) = AP_{in}(S_9) = \{q\}$, $AP_{out}(S_{25}) = AP_{out}(S_9) = \varnothing$ and $PV_{in}(S_{25}) \subseteq PV_{in}(S_9)$. Hence S_{25} is redundant with respect to, for instance, $\{S_1, S_5, S_9, S_{23}, S_{25}, S_{31}\}$.

The detailed procedure of how to add transitions differs a bit depending on whether a timer has been added to a particular place or not.

No timer was added to the place

Given a place p_i, $TF(p_i)$ contains sets of elementary formulas representing events which may happen next, considering that the PRES+ model is in the state represented by place p_i. For each set of elementary formulas $s \in TF(p_i)$, a transition t must, consequently, be added to enable the event described by the set s to happen (see Figure 10.7, Line 3 to Line 5). No transition is, however, added if s does not contain any atomic proposition ($s \in TP(p_i)$), since such sets do not contribute with any events on the ports and therefore are useless. As a result, a transition is added only if $s \in TF(p_i) - TP(p_i)$. Similarly, the transition is not added either if it is redundant or not valid in the target place.

Realising the events described by s is performed by moving the token from the source place p_i to the place indicated by the progress formulas in the particular elementary set, $PF(s)$. At the same time, tokens must be placed in or consumed from the ports as indicated by the atomic propositions in s, $AP(s)$. Review the discussion about how a state is represented in the PRES+ model, on page 223. A state corresponding to s is a

```
procedure addTransitions(p_i: place)
    if $\Psi(p_i) \neq \emptyset$ then
        for each $s \in TF(p_i) - TP(p_i)$ do
            if s is not redundant with respect to TF(p_i) and
                    s is valid in P(PF(s)) then
                add transition t to net;
                $°t := \{ p_i \}$;
                if there is no place corresponding to PF(s) then
                    createPlace(PF(s));
                $t° := P_{in}(PF(s))$;
                connectToPorts(t, s);
                set time delay of t to findTimeDelay(s, p_i);
    else
        for each $f \in 2^{AP(\psi)} - \{ \emptyset \}$ do
            add transition t to net;
            $°t := \{ p_i \}$;
            $t° := \{ p_i \}$;
            connectToPorts(t, f);
            set time delay of t to [0..∞];
```

Figure 10.7: The standard algorithm for adding the transitions belonging to place p_i.

marking where there is a token in the place representing the progress formulas in s, $PF(s)$, and there are tokens in the ports occurring as atomic propositions in s, $AP(s)$, with token values consistent with all relations given in s.

The preset of the added transition t must consequently be $\{p_i\}$ (Line 6), since we are leaving p_i and no timers have been added to it. The postset must contain all places in $P_{in}(PF(s))$ (Line 9), since we are entering the place corresponding to $PF(s)$. Thereby, possible timers associated to that place are also started. However, if the target place has not previously been created in the model, it must first be created using the procedure described in Figure 10.3 (Line 8).

Besides this, the ports corresponding to the atomic propositions must be included in the preset and postset. Exactly how to do this, including assigning a transition function and guard (Line 10), as well as determining the time delay interval of the transition (Line 11), is explained next.

A special case is if the source place corresponds to the empty set of elementary formulas (Line 12). In this case, there is no restriction on what events are allowed to happen next, as specified by its target formulas $TF(p_0) = S(\psi)$, where $\Psi(p_0) = \varnothing$. In order to avoid unnecessary state space explosion, this behaviour can be modelled by adding transitions which together cover all possible scenarios (Line 13 and Line 14). One transition is thus added for each combination of atomic propositions in the formula. The model will stay in this state forever (Line 15 and Line 16), as there is no restriction on future behaviour.

Figure 10.8 presents the algorithm for connecting a transition to the ports according to a given set of elementary formulas. The transition t is connected to the ports as follows. Each atomic proposition in s corresponding to an out-port is incorporated into t's preset (Line 2). Similarly, each atomic proposition in s corresponding to an in-port is incorporated into t's postset (Line 3).

Next, the atomic propositions with relations must be taken care of. If the atomic proposition refers to an out-port, the relation is added in conjunction with the other such relations to form the guard of the transition (Line 6). If there are no atomic propositions with relation referring to out-ports, the transition does not have any guard, i.e. the guard is always true (Line 4).

The transition function is set to return randomly any value from $PV_{in}(s)$ (Line 8).

What remains to be determined is the time delay of the transition (Line 11 in Figure 10.7). Figure 10.9 shows how this delay is computed. Normally, there is no requirement on the time when a certain event has to be performed. This means that the transition should be able to fire after 0 time units and before

```
1  **procedure** connectToPorts(t: transition, s: set of elementary formulas)
2      $°t := °t \cup P(AP_{out}(s))$;
3      $t° := t° \cup P(AP_{in}(s))$;
4      g := true;
5      **for each** $p\Re v \in AP_{rout}(s)$ **do**
6          $g := g \wedge p\Re v$;
7      set guard of t to g;
8      set function of t to return a random value from $PV_{in}(s)$;
```

Figure 10.8: The algorithm for adding interaction with the ports to transition t as specified by the set s.

```
**function** findTimeDelay(s: set of elementary formulas, $p_i$: place) **returns**
        time interval
    **if** $\Psi(p_i) \cup AP_{out}(s) \cup AP_{rout}(s) \notin TF(p_i)$ **then**
        return [0..0];
    **else**
        return [0..$\infty$];
```

Figure 10.9: Algorithm for finding the correct time delay interval of a transition

infinity inclusive, i.e. $[0..\infty]$. However, there are circumstances when the transition must be taken immediately, i.e. have the time delay interval $[0..0]$. This situation may occur when a token arrives at an out-port. The arrival of this token means that the model changed state from $\Psi(p_i)$ to $\Psi(p_i) \cup AP_{out}(s) \cup AP_{rout}(s)$, where s is the set of elementary formulas corresponding to the transition in question and p_i is the current place. It might be the case that this new state is not satisfied by the property in the current state, i.e. $\Psi(p_i) \cup AP_{out}(s) \cup AP_{rout}(s) \notin TF(p_i)$ (Line 2). The PRES+ model must immediately move to a state where it does hold by firing the transition at hand. Therefore, the time delay has to be $[0..0]$.

Place p_1 in the example does not have any timer, so it follows the procedure above for creating its transitions. Figure 10.10 shows the result of adding the transitions to p_1. Some arcs on transitions are not attached to any place in the figure, but they are associated to an atomic proposition. This is a short-hand meaning that they are attached to the port representing the atomic proposition. The transition has, in such cases, a function which produces random values from **U**. If the atomic proposition is denoted with a relation, the function produces random values which still satisfy all relations involved, i.e. its port values.

For example, t_2 has an output arc labelled q. This means that the arc is connected to port q, and that the function associated to transition t_2 generates random values from **U**. Transition t_5 has an output arc with the associated relation $q < 10$. That arc is also connected to q, but the transition has an associated function which only produces random values less than 10.

Remember from the previous discussion that $TF(p_1) = \{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\}$. No transitions are created for S_1 and S_5 since they do not contain any atomic propositions. $S_7 = S_5 \cup \{p\}$, which means that the target places are $P_{in}(PF(S_7)) = P_{in}(S_5)$ and a token should also be consumed from port p. The token should be consumed, i.e. belong to the preset, as opposed to produced, since p is an out-port of an attached component. The transition t_1 is added to represent S_7. It has no guard and its function produces a completely random value since there is no atomic proposition with relation involved. The time delay interval is set to $[0..0]$ since $S_1 \cup \{p\} = S_3 \notin TF(p_1)$. Since p_5 has a timer associated to it, $P_{in}(S_5) = \{p_5, p_{5a}\}$. Moreover, all transitions have source place p_1.

Next, a transition corresponding to $S_9 = S_1 \cup \{q\}$ should be considered. As can be seen, the target place is equal to the source place, i.e. p_1. Since S_9 contains an atomic proposition corresponding to an out-port, the validity of this set of elemen-

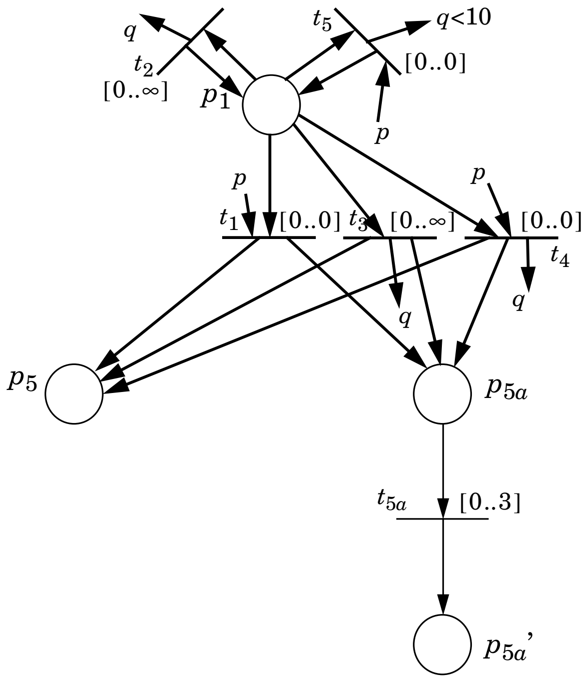


Figure 10.10: The result of adding the transitions of place p_1 to the example formula

tary formulas must be checked. In this case, as with all other sets of elementary formulas in this example, S_9 is valid due to $S_9 - \{q\} = S_1 \in TF(p_1)$. Neither is the set redundant. It is concluded that the transition, t_2, should indeed be added. In addition, a token with a completely random value is placed in the port corresponding to q when fired. Since S_9 does not contain an atomic proposition corresponding to an out-port, and hence $S_1 \cup \varnothing = S_1 \in TF(p_1)$, the time delay is set to $[0..\infty]$. The procedure progresses similarly for the sets S_{13}, S_{15} and S_{27}. The remaining sets of elementary formulas among the target formulas are redundant with respect to $TF(p_1)$, and consequently no transitions are added for these sets.

The place has a timer

For places with timers, it is necessary to identify whether a certain set of elementary formulas among the target formulas has to occur before the deadline stipulated by the timers or not.

Definition 10.13: Requiring U formulas. The set of requiring U formulas of a place p_i and a set of elementary formulas s is defined as $RUF(p_i, s) = \{\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \in U(p_i) | s \in \Phi(\varphi_2)\}$.

■

Intuitively, the set of requiring U formulas is the set of U formulas in p_i which require s to occur before its associated upper time bound.

Definition 10.14: Timer triggered formulas. The set of timer triggered formulas of a set of U formulas, U, is defined as $TTF(U) = \bigcap_{\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \in U} \Phi(\varphi_2)$.

■

The timer triggered formulas denote the set of sets of elementary formulas corresponding to events of which one has to be performed before the deadlines of all the U formulas in U.

In the example, $RUF(p_5, S_{25}) = \{\mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]\}$, but $RUF(p_5, S_7) = \emptyset$. The timer triggered formulas of $U(p_5)$ are $TTF(U(p_5)) = \Phi(q < 10)$ and more explicitly $TTF(U(p_5)) = \{S_{24}, S_{25}, S_{26}, S_{27}, S_{28}, S_{29}, S_{30}, S_{31}\}$.

Figure 10.11 presents the algorithm for adding transitions to a place with timers. Similar to the non-timer case, transitions are added for each non-redundant and valid (Line 3) set $s \in TF(p_i) - TP(p_i)$ (Line 2). There are basically two cases. Either $PF(s) = \Psi(p_i)$ or $PF(s) \neq \Psi(p_i)$, i.e. either the target place is the same as p_i or it is not.

In the first case (Line 4 in Figure 10.11), all U formulas requiring the event s are examined (Line 5). If s is not redundant with respect to the set of the timer triggered formulas, a transition corresponding to s is added (Line 7). The end places of all timers requiring s are connected to the transition as input (Line 9). Since s stipulates that the model should stay in the same state, the timers must be restarted, as a result of firing the transition, in order to guarantee a new occurrence of a timer triggered event. The output of the transition is therefore the start places of the timers (Line 10). The time delay interval is set to $[0..0]$ as no further time may elapse after the timer has fired (Line 12). That might exceed the deadline imposed by the timer.

If s does not have any requiring U formulas, the timers need not to be involved in the transition. Place p_i is both an input and an output place of the added transition (Line 14 to Line 18).

In order to leave a state with timers (Line 19), s must be triggered by all timers. Otherwise, the event prescribed by some timer will not be fulfilled. The input of the added transition is p_i together with the end places of all timers (Line 21 to Line 23), and output is the places corresponding to the new state (Line 26). The time delay interval is set to $[0..0]$ for the same reasons as on Line 12.

In the example, place p_5 contains one U formula. Remember that $TF(p_5) = \{S_5, S_7, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\}$, and $TTF(U(p_5)) = \{S_{24}, S_{25}, S_{26}, S_{27}, S_{28}, S_{29}, S_{30}, S_{31}\}$. Figure 10.12 shows the result of the procedure.

```
procedure addTransitionsForTimers($p_i$: place)
    for each s $\in$ TF($p_i$) - TP($p_i$) do
        if s is valid in P(PF(s)) then
            if PF(s) = $\Psi$($p_i$) then
                if RUF($p_i$, s) $\neq \emptyset$ then
                    if s is not redundant w.r.t TTF(RUF($p_i$, s)) then
                        add transition t to net;
                        for each $\varphi \in$ RUF($p_i$, s) do
                            °t := °t $\cup$ { Timerout($p_i$, $\varphi$) };
                            t° := t° $\cup$ { Timerin($p_i$, $\varphi$) };
                        connectToPorts(t, s);
                        set time delay of t to [0..0];
                else if s is not redundant w.r.t TF($p_i$) then
                    add transition t to net;
                    °t := { $p_i$ };
                    t° := { pi };
                    connectToPorts(t, s);
                    set time delay of t to findTimeDelay(s, $p_i$);
            else if s $\in$ TTF(U($p_i$)) and s is not red. w.r.t. TTF(U($p_i$)) then
                add transition t to net;
                °t := { $p_i$ };
                for each $\varphi \in$ RUF($p_i$, s) do
                    °t := °t $\cup$ { Timerout($p_i$, $\varphi$) };
                if there is no place corresponding to PF(s) then
                    createPlace(PF(s));
                t° := $P_{in}$(PF(s));
                connectToPorts(t, s);
                set time delay of t to [0..0];
```

Figure 10.11: The algorithm for adding transitions to a place with timers

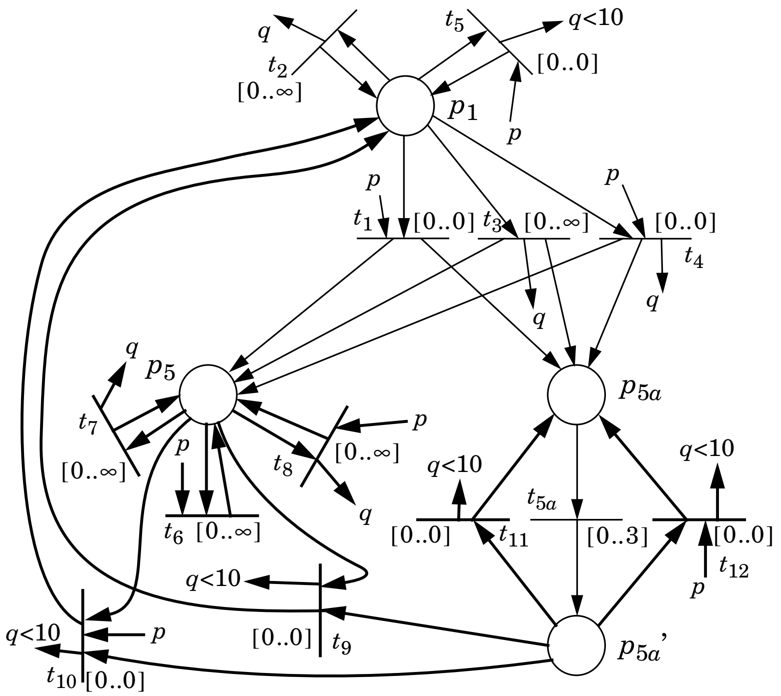


Figure 10.12: The result of adding the transitions of place p_5 to the example formula

As usual, no transition is added for S_5, since that set does not contain any atomic propositions. $PF(S_7) = S_5 = \Psi(p_5)$ (Line 4 in Figure 10.11), but $RUF(p_5, S_7) = \varnothing$. Since S_7 is not required by any timer and is not redundant, the transition is added as a loop around place p_5 (Line 13), see transition t_6 in the figure. The same goes for S_{13} and S_{15}, which result in transitions t_7 and t_8 respectively. $PF(S_{25}) = S_1$ and $S_{25} \in TTF(U(p_5))$, implying that the transition, t_9, is added between $\{p_5, p_{5a}'\}$ and $P_{in}(S_1)$ (Line 23 and Line 26). S_{27} is handled similarly, resulting in transition t_{10}. The target place of both S_{29} and S_{31} is p_5, and $RUF(p_5, S_{29}) = RUF(p_5, S_{31}) \neq \varnothing$ which means that the timer must be restarted (Line 9 and Line 10) as illustrated by transitions t_{11} and t_{12} in the figure.

10.2.4 INSERTION OF INITIAL TOKENS

The last step of the algorithm is to insert initial tokens in the PRES+ model. Figure 10.13 presents the algorithm for this purpose.

In Section 10.2.1, one place was created for each member in $PF(\Phi(\psi)) \cap \Phi(\psi)$. Any of these places (or sets of places if timers were introduced) are randomly and dynamically chosen for the initial token. This selection mechanism is in practice modelled with non-determinism. A place $start$ is added to the model (Line 7). For each candidate for the initial place, i.e. $PF(\Phi(\psi)) \cap \Phi(\psi)$, a transition is added from $start$ to that candidate (Line 8 to Line 12). The transition has time delay $[0..0]$ since this choice must be performed instantly before any other time consuming action is taken in the system.

In case there is only one candidate for the initial place, i.e. $|PF(\Phi(\psi)) \cap \Phi(\psi)| = 1$, there is naturally no need for adding

```
1   **procedure** insertInitialToken
2       **if** $| PF(\Phi(\psi)) \cap \Phi(\psi) | = 1$ **then**
3           add token in $P_{in}$(the only elt of $PF(\Phi(\psi))$) with value <0, 0>;
4       **else if** $\exists\ p_i$ such that $\Psi(p_i) = \{ AX\ \psi \}$ **then**
5           add token in $P_{in}(\Psi(\psi))$ with value <0, 0>;
6       **else**
7           add place start to net with an initial token with value <0, 0>;
8           **for each** $s \in PF(\Phi(\psi)) \cap \Phi(\psi)$ **do**
9               add transition t to net;
10              $°t := \{ start \}$;
11              $t° := P_{in}(s)$;
12              set time delay of t to [0..0];
```

Figure 10.13: The algorithm for adding an initial token

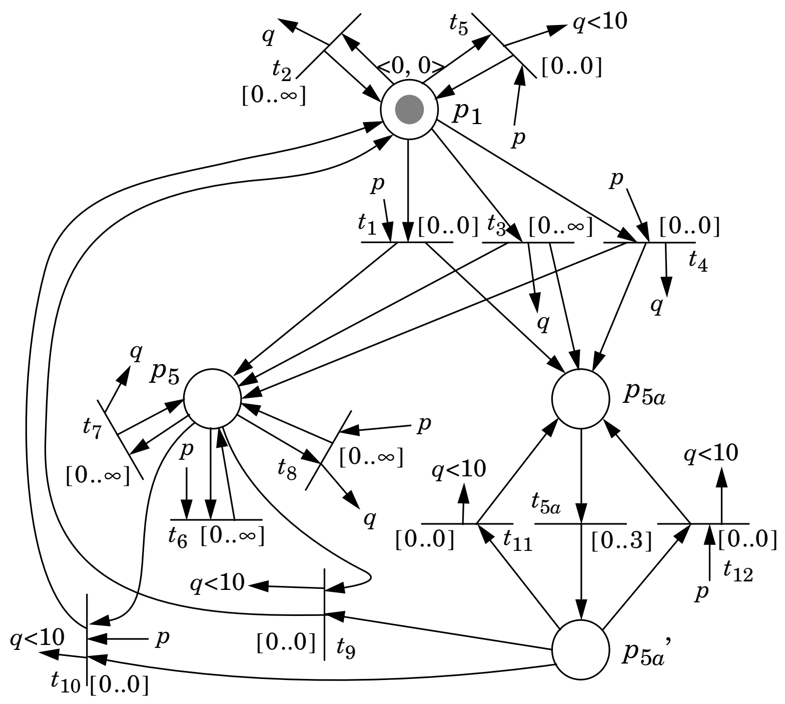


Figure 10.14: The resulting PRES+ model of the example formula

this mechanism for choosing the initial place. The only existing place is directly chosen to be the initial one (Line 3).

If one of the candidate places corresponds to a set of elementary formulas consisting of only the one formula expressing that the whole property ψ shall continue to hold, $\mathbf{AX}\psi$, then that place can safely be chosen as the initial state (Line 5). The reason is that it by its nature will guarantee that the future behaviour of the model conforms to the property ψ.

In the example, there are two candidates for the initial token, S_1 and S_5. However, $S_1 = \{\mathbf{AX}\psi\}$ and consequently the place corresponding to that set, p_1, is chosen as the initial place

```
1 function generateFormulaStub($\psi$: ACTL) returns PRES+
2     createInitialPlaces($\psi$);
3     for each place $p_i$ in net corr. to a set of elementary formulas do
4         if | U($\Psi(p_i)$) | > 0 then
5             addTransitionsForTimers($p_i$);
6         else
7             addTransitions($p_i$);
8     insertInitialTokens;
9     return net;
```

Figure 10.15: The algorithm for generating a PRES+ model given an ACTL formula

(Line 5 in Figure 10.13). The final PRES+ model representing the example property is shown in Figure 10.14.

10.2.5 Summary

Section 10.2 has so far presented all steps of the PRES+ generation procedure from an arbitrary ACTL formula. Figure 10.15 presents the overall algorithm where the previously presented steps are put into context.

The first step is to create the places corresponding to a state representing different requirements on the future behaviour in the PRES+ model. Places corresponding to atomic propositions, i.e. ports are also created (Line 2). Timers are moreover added as part of the place creation process. After that, transitions corresponding to events as given by sets of elementary formulas are added (Line 3). This is performed in different ways depending on whether timers were added to the place (Line 5 or Line 7 respectively). The last step is to insert the initial tokens (Line 8).

10.3 Examples

In order to highlight and emphasise certain aspects of the algorithms presented in Section 10.2, a few examples are presented in this section. The reader is encouraged to follow the algorithms presented previously as the examples are explained.

10.3.1 PLACE WITH EMPTY CORRESPONDING ELEMENTARY SET

This section provides an example leading to a PRES+ model with a place of which the corresponding set of elementary formulas is empty. The formula which will be used is $\mathbf{AF}_{\leq 3}p < 10$. p is in this example an in-port.

The first task when generating the PRES+ model corresponding to a formula is to normalise it, i.e. $\psi = \mathbf{A}[t\ \mathbf{U}_{\leq 3}p < 10]$.

The next step is to find out its elementary formulas $el(\psi)$ and $\Phi(\psi)$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 3}p < 10]}_{1}, \underbrace{p}_{2}, \underbrace{p < 10}_{4} \right\} \tag{10.6}$$

$S(\psi) = \{S_0, S_1, S_2, S_3, S_6, S_7\}$, because S_4 and S_5 are contradictory since they contain $p < 10$ but not p.

- $\Phi(\psi) = \Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 3}p < 10]) = \{S_1, S_3, S_6, S_7\}$
- $\Phi(\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 3}p < 10]) = \{S_1, S_3, S_7\}$
- $\Phi(p < 10) = \{S_6, S_7\}$

$PF(\Phi(\psi)) \cap \Phi(\psi) = \{S_0, S_1\} \cap \Phi(\psi) = \{S_1\}$, which means that the resulting model will only have one state place, p_1. However, since there is one U formula in p_1, a timer is added.

Figure 10.16 shows the resulting PRES+ model after transitions have been added and initial tokens have been inserted.

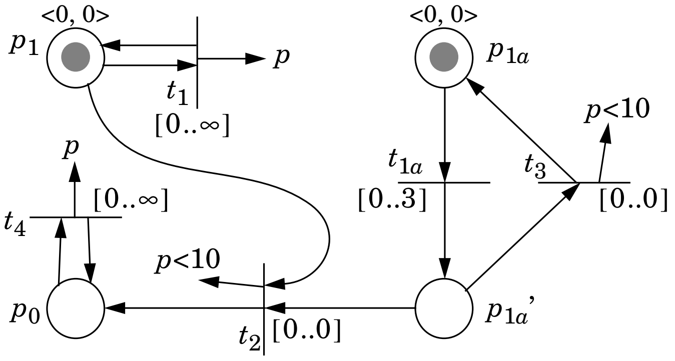


Figure 10.16: The resulting PRES+ model of the formula $\mathbf{AF}_{\leq 3}p < 10$

$TF(p_1) = \{S_1, S_3, S_6, S_7\}$ and $TTF(U(p_1)) = \{S_6, S_7\}$. No transition is added for S_1 since it does not contain any atomic propositions. As for S_3, it is valid since $S_3 - \{p\} = S_1 \in TF(p_1)$, so a transition corresponding to that set must be added. $PF(S_3) = S_1 = \Psi(p_1)$, but no timer requires it, $RUF(p_1, S_3) = \varnothing$, so the transition is added as a loop around p_1, see transition t_1 in the figure.

Since $PF(S_6) = \varnothing$ for which there does not yet exist a place in the model, such a place is created, p_0 in the figure. S_6 is furthermore valid in its target place due to the fact that $S_6 - \{p\} - \{p < 10\} = \varnothing \in TF(p_0)$. A transition, t_2, corresponding to S_6 is added between $\{p_1, p_{1a}'\}$ and p_0.

Due to the fact that $PF(S_7) = S_1 = \Psi(p_1)$ and the timer requires it, $RUF(p_1, S_7) = \{\mathbf{A}[t\ \mathbf{U}_{\leq 3}p < 10]\}$, a transition t_3 is added between the end and start places of the timer.

Place p_0 is associated with an empty set of elementary formulas. From that place, all possible behaviours can occur. Since the property only contains one atomic proposition, p, only one transition ($2^1 - 1$), t_4, which can produce all possible output to the port representing p, is added to the model.

The model only has one start place since $|PF(\Phi(\psi)) \cap \Phi(\psi)| = 1$. This implies that the initial tokens are directly inserted to $P_{in}(S_1)$.

Note that $p < 10$ only has to be satisfied once, both according to the PRES+ model and according to the formula.

10.3.2 PLACE WITH MORE THAN ONE TIMER

This example aims to highlight what happens with a place with more than one timer. The formula used to illustrate this is $\mathbf{AG}(\mathbf{AF}_{\leq 2}p \vee \mathbf{AF}_{\leq 5}q)$. After normalisation, $\psi = \mathbf{A}[f\ \mathbf{R}\ (\mathbf{A}[t\ \mathbf{U}_{\leq 2}p] \vee \mathbf{A}[t\ \mathbf{U}_{\leq 5}q])]$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\,\mathbf{A}[f\ \mathbf{R}\ (\mathbf{A}[t\ \mathbf{U}_{\leq 2}p] \vee \mathbf{A}[t\ \mathbf{U}_{\leq 5}q])]}_{1}, \right.$$
$$\left. \underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]}_{2}, \underbrace{p}_{4}, \underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 5}q]}_{8}, \underbrace{q}_{16} \right\} \tag{10.7}$$

$S(\psi) = \{S_0..S_{31}\}$. No sets have to be removed, since the formula does not contain any atomic propositions with relation.

- $\Phi(\psi) = \{S_3, S_5, S_7, S_9, S_{11}, S_{13}, S_{15}, S_{17}, S_{19}, S_{21}, S_{23}, S_{25}, S_{27}, S_{29}, S_{31}\}$
- $\Phi(\mathbf{AX}\ \psi) = \{S_1, S_3, S_5, S_7, S_9, S_{11}, S_{13}, S_{15}, S_{17}, S_{19}, S_{21}, S_{23}, S_{25}, S_{27}, S_{29}, S_{31}\}$
- $\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 2}p] \vee \mathbf{A}[t\ \mathbf{U}_{\leq 5}q]) = \{S_2..S_{31}\}$
- $\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]) = \{S_2..S_7, S_{10}..S_{15}, S_{18}..S_{23}, S_{26}..S_{31}\}$
- $\Phi(\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]) = \{S_2, S_3, S_6, S_7, S_{10}, S_{11}, S_{14}, S_{15}, S_{18}, S_{19}, S_{22}, S_{23}, S_{26}, S_{27}, S_{30}, S_{31}\}$
- $\Phi(p) = \{S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}, S_{20}, S_{21}, S_{22}, S_{23}, S_{28}, S_{29}, S_{30}, S_{31}\}$

- $\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 5} q]) = \{S_8..S_{31}\}$
- $\Phi(\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 5} q]) = \{S_8..S_{15}, S_{24}..S_{31}\}$
- $\Phi(q) = \{S_{16}..S_{31}\}$

Three places are created corresponding to each set in $PF(\Phi(\psi)) \cap \Phi(\psi) = \{S_3, S_9, S_{11}\}$. S_3 and S_9 only have one U formula and therefore will only have one timer each. Transitions are added to them in a similar way as in Figure 10.14. In this section, we will concentrate on place p_{11}, corresponding to set S_{11}, which contains two U formulas. Figure 10.17 shows the part of the resulting PRES+ model corresponding to this set of progress formulas.

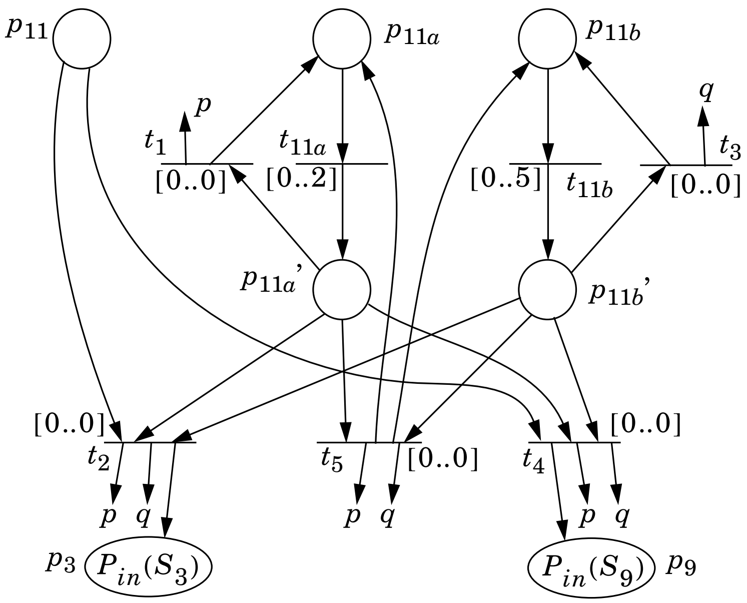


Figure 10.17: The resulting PRES+ model corresponding to progress formulas S_{11} of the formula $\mathbf{AG}(\mathbf{AF}_{\leq 2} p \vee \mathbf{AF}_{\leq 5} q)$

$TF(p_{11}) = \{S_{11}, S_{13}, S_{15}, S_{19}, S_{21}, S_{23}, S_{27}, S_{29}, S_{31}\}$.
$TTF(U(p_{11})) = \{S_{20}, S_{21}, S_{22}, S_{23}, S_{28}, S_{29}, S_{30}, S_{31}\}$.

As usual, each set of elementary formulas in $TF(p_{11})$ is examined. S_{11} is not added since it does not contain any atomic propositions. S_{13} is not added either because $PF(S_{13}) = S_9 \neq S_{11} = \Psi(p_{11})$ and $S_{13} \notin TTF(U(p_{11}))$. $PF(S_{15}) = S_{11} = \Psi(p_{11})$, on the other hand, and $RUF(p_{11}, S_{15}) = \{\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]\}$, so the transition is added so that it restarts the corresponding timer, see transition t_1. No transition is added for S_{19} for the same reason as S_{13}. No transition is added for S_{21} either as it is not valid in its target place, $S_{21} - \{p, q\} = S_1 \notin TF(S_1) = \Phi(\psi)$. S_{23} does not have the target place p_{11}, but $S_{23} \in TTF(U(p_{11}))$, so transition t_2 is added with $\{p_{11}, p_{11a}', p_{11b}'\} = {}^{\circ}t_2$. S_{27} causes transition t_3 to be created for similar reasons as S_{15}. S_{29} is analogous to S_{23}, resulting in transition t_4. The target place of S_{31} is p_{11}, and both timers require it. Transition t_5 is added to the model in a similar fashion as t_1 and t_3 with the difference that it is connected to both timers.

10.3.3 GUARDS ON TRANSITIONS

Formulas containing out-port atomic propositions with relation lead, in general, to a PRES+ model with guards on certain transitions. This phenomenon will be demonstrated using the formula $\mathbf{AG}((p = 2) \to \mathbf{AF}_{\leq 7}q)$, where p is an out-port and q is an in-port of a connected component. Normalisation gives the formula $\psi = \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7}q])]$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\,\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7}q])]}_{1}, \underbrace{p}_{2}, \right. \quad (10.8)$$
$$\left. \underbrace{p \neq 2}_{4}, \underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 7}q]}_{8}, \underbrace{q}_{16} \right\}$$

$S(\psi) = \{S_0..S_3, S_6..S_{11}, S_{14}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$, since in all other sets s, $p \neq 2 \in s$ but $p \notin s$. Hence, those sets are contradictory.

- $\Phi(\psi) = \{S_1, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{19}, S_{23}, S_{25}, S_{27}, S_{31}\}$
- $\Phi(\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7} q])]) = \{S_1, S_3, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{19}, S_{23}, S_{25}, S_{27}, S_{31}\}$
- $\Phi(\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7} q]) = \{S_0, S_1, S_6..S_{11}, S_{14}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$
- $\Phi(\neg p) = \{S_0, S_1, S_8, S_9, S_{16}, S_{17}, S_{24}, S_{25}\}$
- $\Phi(p \neq 2) = \{S_6, S_7, S_{14}, S_{15}, S_{22}, S_{23}, S_{30}, S_{31}\}$
- $\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 7} q]) = \{S_8..S_{11}, S_{14}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$
- $\Phi(\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 7} q]) = \{S_8, S_9, S_{10}, S_{11}, S_{14}, S_{15}, S_{24}, S_{25}, S_{26}, S_{27}, S_{30}, S_{31}\}$
- $\Phi(q) = \{S_{16}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$

$PF(\Phi(\psi)) \cap \Phi(\psi) = \{S_1, S_9\}$, so two places p_1 and p_9 are created in the resulting PRES+ model. Place p_9 contains a U formula, so a timer is added for that place. Figure 10.18 shows the resulting PRES+ model.

Starting with adding transitions for place p_1, $TF(p_1) = \{S_1, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{19}, S_{23}, S_{25}, S_{27}, S_{31}\}$. No transitions are added for S_1 and S_9 since they do not contain any atomic proposition. The set S_7 results in transition t_1. It contains the atomic proposition with relation $p \neq 2$. Due to the fact that p is an out-port, that relation will become a guard of t_1. Transition t_2 is added for the set S_{11}, it contains p, but not $p \neq 2$. The time delay is set to $[0..0]$, since $S_1 \cup \{p\} = S_3 \notin TF(p_1)$. No transition is added for S_{15} due to redundancy with S_{11}. $PV_{out}(S_{11}, p) = \mathbf{Z}$ and $PV_{out}(S_{15}, p) = \mathbf{Z} - \{2\}$, so $PV_{out}(S_{15}, p) \subseteq PV_{out}(S_{11}, p)$.

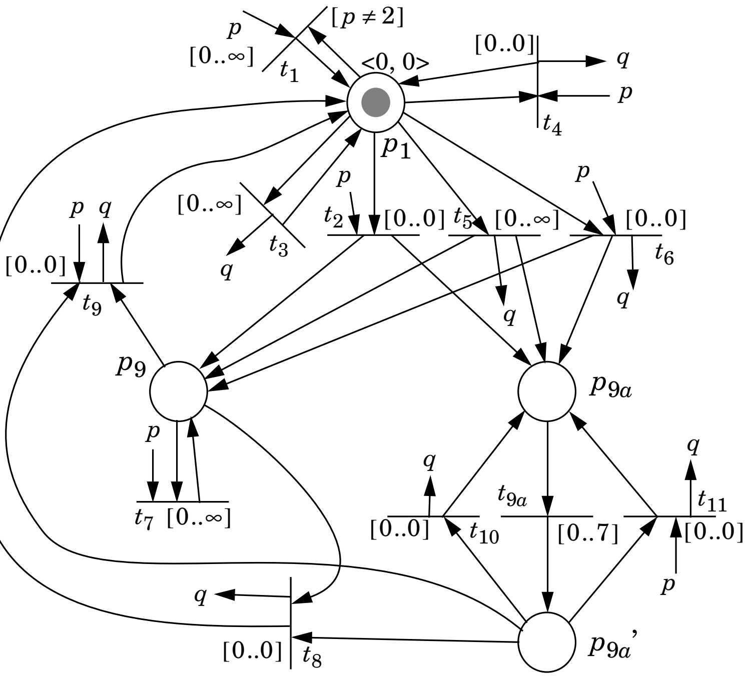


Figure 10.18: The resulting PRES+ model of the formula $\mathbf{AG}((p = 2) \to \mathbf{AF}_{\leq 7}q)$

S_{17}, S_{19}, S_{25} and S_{27} causes t_3, t_4, t_5 and t_6 to be added respectively. No transitions are added for S_{23} and S_{31} due to redundancy with S_{19} and S_{27} respectively.

Let us continue with place p_9 containing a timer. $TF(p_9) = \{S_9, S_{11}, S_{15}, S_{17}, S_{19}, S_{23}, S_{25}, S_{27}, S_{31}\}$ and $TTF(U(p_9)) = \{S_{16}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$. No transition is added for S_9 as usual. $PF(S_{11}) = S_9 = \Psi(p_9)$, but $RUF(p_9, S_{11}) = \varnothing$, hence it must be added as a loop around p_9, see transition t_7. S_{15} is redundant with S_7 and not added. Transition t_8 is added due to S_{17}. $S_{17} \in TTF(U(p_9))$ so it must originate from the timer with the delay interval $[0..0]$. S_{19} results in t_9 for similar reasons. For S_{25} and S_{27}, transitions t_{10} and t_{11} are added respectively. Both $S_{25}, S_{27} \in TTF(U(p_9))$ and their target place is p_9, meaning that the timer is restarted. S_{23} and S_{31} are redundant.

Finally, since $PF(\Phi(\psi)) \cap \Phi(\psi) = \{S_1, S_9\}$ and $S_1 = \mathbf{AX}\,\psi$, the place corresponding to S_1, p_1, is determined to be the initial place.

10.4 Verification Methodology Roadmap

This section continues the verification methodology roadmap from Section 9.6 and in particular Figure 9.19. Figure 10.19 shows the continuation of the roadmap.

The first question to answer is if the diagnostic trace, obtained from the previous verification, indicates that the verification result is due to an unwanted input from the surrounding. Such an input has its origin in a random transition attached to at least one component port that is connected to the surrounding of the glue logic (Section 6.2). In this case, the verification outcome could be the result of the fact that the surrounding does not satisfy the requirements of the component on the other interfaces than the one connected to the glue logic on which the verification is performed. If that is *not* the case, then the property is proven not satisfied. Otherwise, a PRES+ model is generated corresponding to that property, as described in this chapter.

The generated PRES+ model is then connected to the stub and the system is verified. If the property is satisfied, it is proven satisfied in the whole system. Otherwise, the newly obtained diagnostic trace is examined again in order to find out if it violates another requirement on the surrounding. If it does, a new PRES+ model is generated given both the previous formulas and the new one as a conjunction. The iteration continues until a final verification result is obtained.

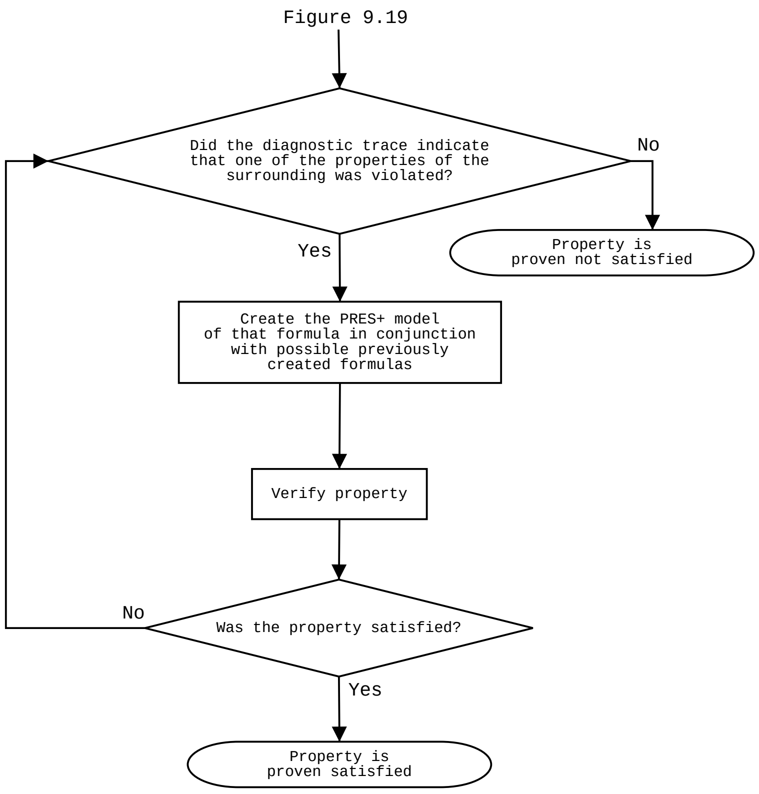


Figure 9.19
Did the diagnostic trace indicate that one of the properties of the surrounding was violated?
No
Property is proven not satisfied
Yes
Create the PRES+ model of that formula in conjunction with possible previously created formulas
Verify property
Was the property satisfied?
No
Yes
Property is proven satisfied

Figure 10.19: Continuation of the roadmap in Figure 9.19, useFormulas

Chapter 11 Case Study: A Mobile Telephone Design

THE PRESENTED INTEGRATION verification methodology provides a powerful means to verify large systems. This chapter describes a case study to demonstrate how a real-life system can be verified using the methodology. The verified system is a mobile telephone design.

11.1 The Mobile Telephone System

Figure 11.1 shows an overview picture of the model and how the components forming the model are connected. It consists of the following seven components communicating via an AMBA bus.

1. Microphone. The microphone sends voice data to the transmitter.
2. Buttons. When dialling, the buttons component sends information about which buttons were pressed to the controller.
3. Speaker. The speaker receives voice signals from the receiver and converts them to sound.

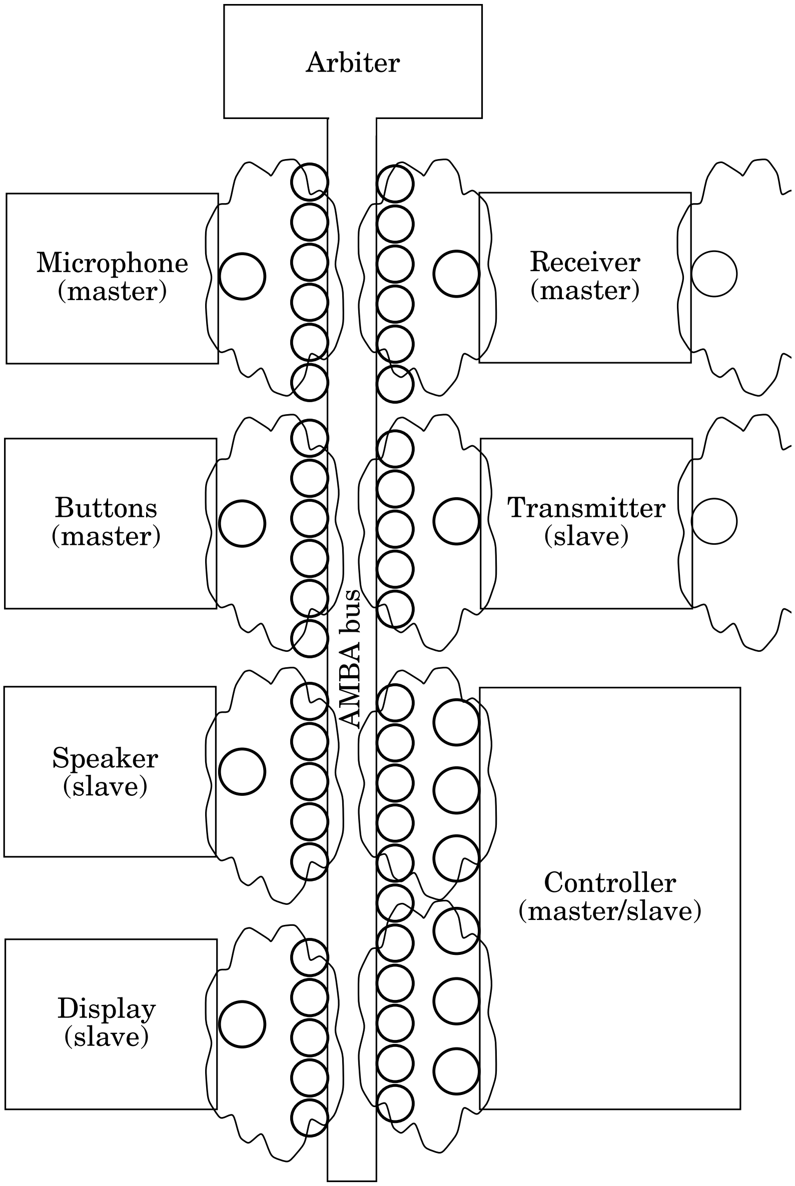


Figure 11.1: Overview model of the case study system, a mobile telephone design

4. Display. The display shows on a small screen information sent to it by the controller.
5. Receiver. The receiver receives data from the base-station of the mobile telephone network and passes it on to the designated component.
6. Transmitter. The transmitter receives data from other components in the telephone and passes it on to the base-station.
7. Controller. The controller coordinates the tasks of the other components.

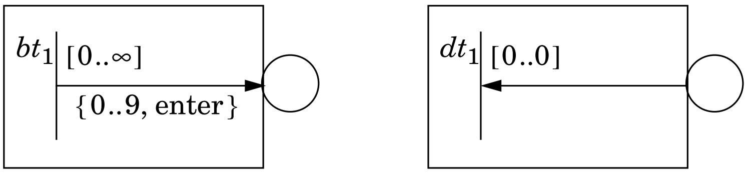
As mentioned previously, these components are supposed to communicate over an AMBA bus. However, since the AMBA bus imposes a certain protocol and the components are not designed for that protocol, glue logics adapting the components to this protocol are inserted.

A few of the components which are directly involved in the example are explained in more detail in the following sections.

11.1.1 Buttons and Display

The peripheral components, such as Buttons and Display, which are used to interact with the end user, are modelled in a simplistic way as shown in Figure 11.2.

In this case study, we assume that the telephone has eleven buttons: the numbers 0 to 9 plus the button “enter”. When the end user wants to dial a number, he enters the number, presses the button “enter”, after which the telephone tries to satisfy the request. From the point of view of the component Buttons, the



(a) Buttons (b) Display

Figure 11.2: Models of components Buttons and Display

buttons can be pressed in any order at any time. This is modelled by a transition with time delay interval $[0..\infty]$ and the function "random value from the set $\{0..9, \text{enter}\}$". The Buttons component has no idea about the semantics of each button being pressed. It is the task of the controller to determine what should happen when a particular button is pressed.

The situation is similar but reverse for Display. Display receives commands about what to show on its screen. In PRES+ terms, this means that tokens in its port are consumed as they appear. The time delay interval depends on how fast the information is processed by the component. In this example, it is assumed that the information is immediately taken care of, i.e. the time delay interval is $[0..0]$.

11.1.2 CONTROLLER

The controller component keeps track of what is happening in the system and acts accordingly. Figure 11.3 shows a model of the component.

Places *accbutton* and *noaccbutton* are marked when the controller is able or is not able to process button data respectively. The data is simply discarded if it is not immediately accepted. Transitions ct_1 to ct_4 take care of this functionality. The transitions have guards so that different actions can be taken depending on which button was pressed. This model only makes a difference between if a number was pressed, $b \in \{0..9\}$, or if "enter" was pressed, $b = \text{enter}$. When dialling a number, signals (tokens) are also sent in order to update the display. Having pressed "enter" the telephone number is sent to the transmitter.

Places *calling* and *nocall* record whether a phone call is taking place or not. Transition ct_5 therefore updates these places when a phone call is to be made. Transition ct_7 takes care of incoming phone calls and ct_8 and ct_9 handle the end of a call.

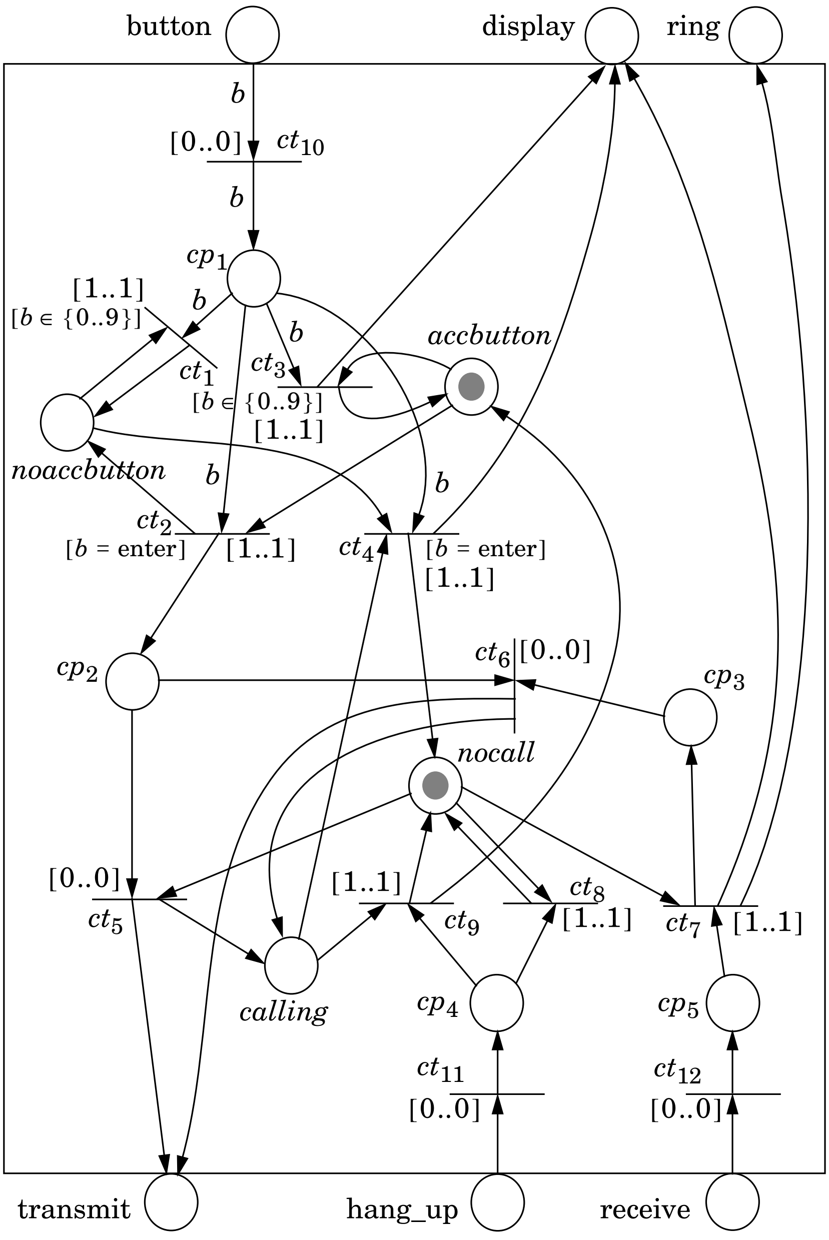


Figure 11.3: Model of the Controller component

11.1.3 AMBA BUS

All components communicate through an AMBA bus [Roy03]. The AMBA bus was previously introduced in Section 6.1.2. Here, however, it will be described in more detail and adapted to the PRES+ model used in this case study.

The AMBA bus consists of two parts, Arbiter and Bus. Figure 11.4 and Figure 11.5 introduce PRES+ models of these two parts, respectively. The components communicating over the bus are furthermore divided into two categories, master and slave. Figure 11.1 indicates to which category each component in the example belongs. Components sending messages are masters and components receiving messages are slaves.

Any master wanting to send data on the bus must first request access to it from the arbiter by emitting the signal HREQBUS. The arbiter will eventually grant access (HGRANT) to any master requesting it, and at the same time, avoid starvation. Once a master is granted access, it may send one bunch of data every clock cycle (time unit, in terms of PRES+). All bunches of data do not necessarily have to be addressed to the same slave. When sending the last bunch, the master notifies this by emitting the signal HTRANS.

However, if a slave is not ready to receive, it is able to put the transaction on hold, or in AMBA bus terms *split* (HRESP), until it eventually becomes ready (HREADY). During the time period when it is not yet ready to receive, the arbiter might give the access to the bus to another requesting master. When the slave declares itself ready to receive the split data, the master on hold is automatically granted access to the bus again.

The AMBA bus actually consists of two buses, one address bus and one data bus. When a master sends a bunch of data on the bus, it sends the address of the receiving slave on the address bus and the data on the data bus.

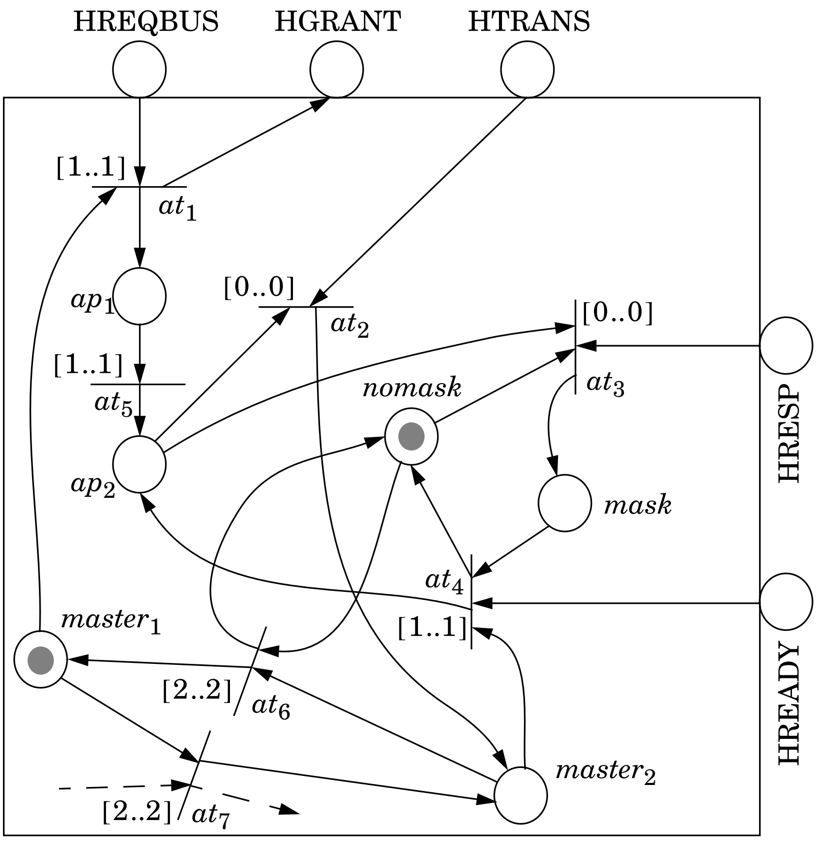


Figure 11.4: Model of the Arbiter component

Figure 11.4 shows a part of the model of the arbiter corresponding to one particular master. The part in the figure is copied once for each master. Places $master_x$ represent which master currently holds the token in a round-robin schedule. The master holding the token has the opportunity to get access to the bus. If a request has not arrived from that particular master, the token moves to the place corresponding to the next master, at_6 (in the case of master 2) or at_7 (in the case of master 1). The dashed arcs on transition at_7 are connected to place *nomask* of

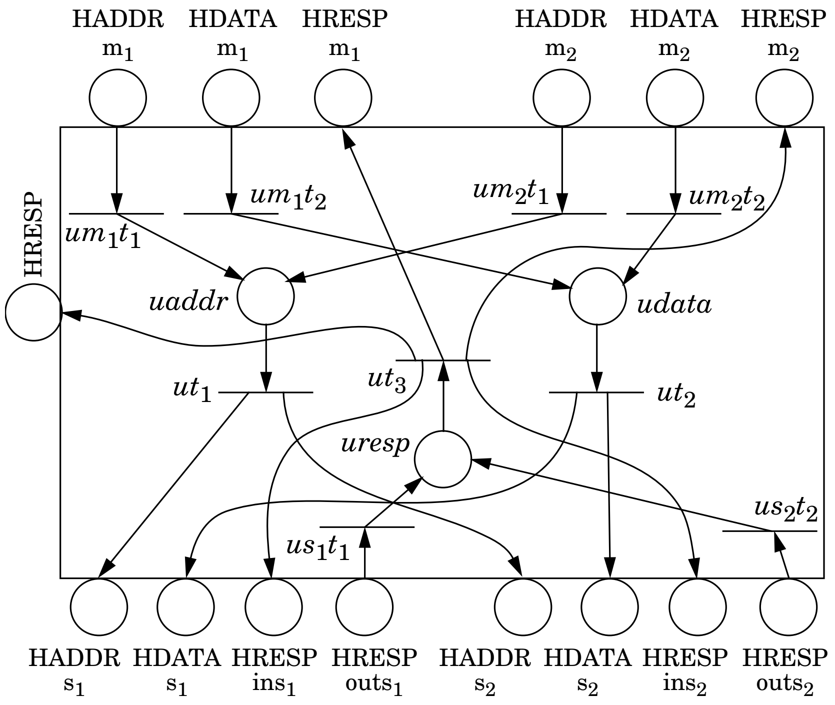


Figure 11.5: Model of the Bus component

master 2 in a similar way as at_6. Place *mask* is marked when a slave has split the transaction of that master. *nomask* is marked otherwise.

The bus itself distributes tokens sent to it to all components connected to it. Figure 11.5 shows a model of the Bus component. All transitions have time delay interval $[0..0]$ and transition function identity. Consequently, the bus distributes exactly the same token as received to the rest of the components in zero time.

Port HRESP is directly connected to the arbiter through the port with the same name.

11.1.4 GLUE LOGICS

As has been shown, the components do not contain any functionality to communicate with and over the bus. For this reason, it is necessary to adapt the components and insert a glue logic between the component and the bus.

This design principally contains two types of glue logic, one for handling the master functionality and one for handling the slave functionality for each type of component respectively. Consequently, the glue logics which are situated between the bus and a slave component (see Figure 11.1) is a slave functionality glue logic, whereas the glue logics situated between the bus and a master component is a master functionality glue logic.

Master functionality

A model of the glue logic which is active when the controller serves as a master, is shown in Figure 11.6. The main problem to be solved by the glue logic is in case of a slave splitting a transaction initiated by the current master. For this purpose, the glue logic must always remember the last transaction. The address is stored during one clock cycle in cmp_7 and the data in cmp_9. After one clock cycle the stored items are removed by cmt_{11} and cmt_{14} respectively. When the master is regranted access to the bus, transitions cmt_{12} and cmt_{15} become enabled and resend the data. The tokens, however, stay in their respective places in case the resent data is again split.

Meanwhile a transaction is split, no new data can be sent by the component. Presence or absence of tokens in $cmanosplit_1$ and $cmdnosplit_1$ regulate this behaviour.

The glue logic may receive tokens from HRESP even though its master did not send anything. This can be the result of a transaction of another master being split. Remember that the bus distributes split requests to all connected components. In

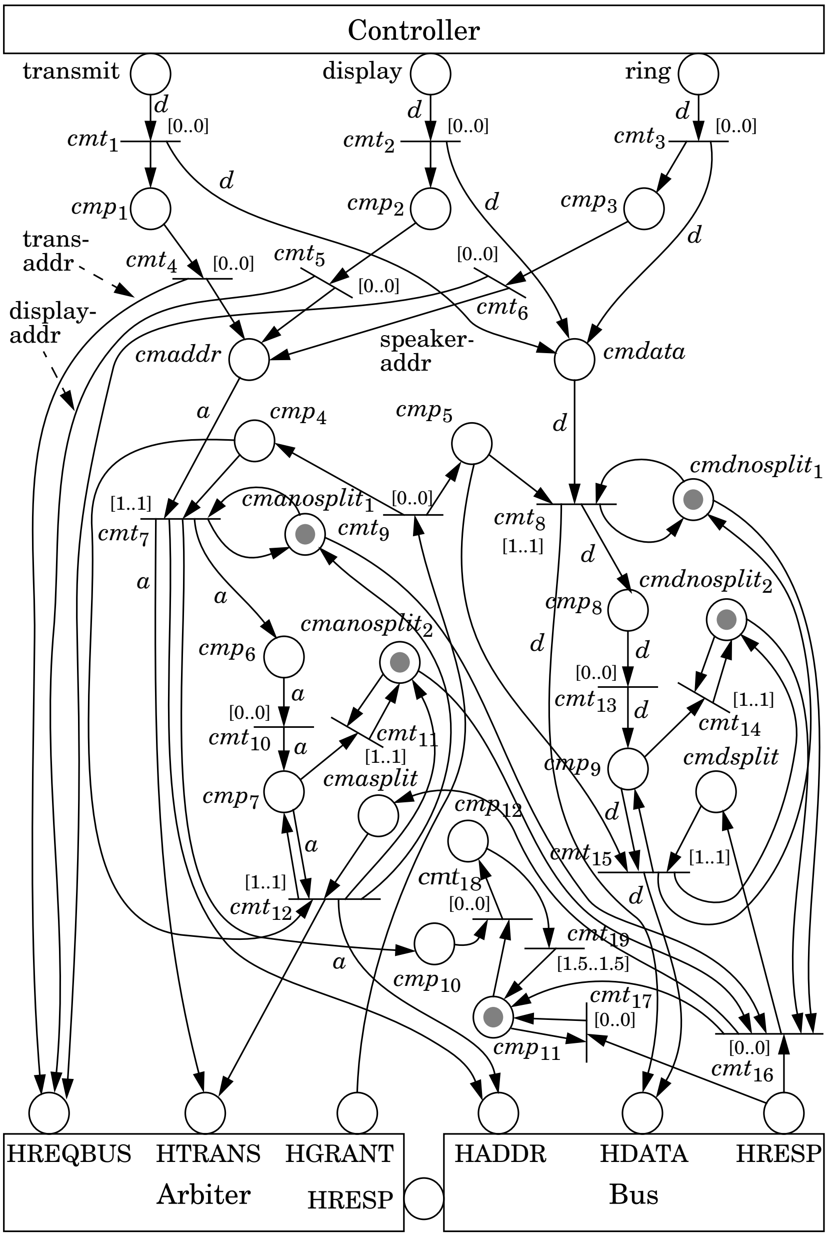


Figure 11.6: A model of the glue logic for the master functionality of the controller

order to keep track of whether such a split request is intended for the current master or not the structure consisting of places cmp_{10} to cmp_{12} is created. A token in cmp_{12} means that the current master has just sent and a possible split request is consequently intended for itself. Before the next clock cycle the token is however moved back to cmp_{11} through transition cmt_{19}. With a token in cmp_{11} incoming split requests are immediately consumed leading to no further action since they are not intended for this master.

Slave functionality

The main function of the glue logics handling the slave functionality is to split a transaction in case the component is not ready to receive. Afterwards, when the component is ready to receive a message again, the glue logic must notify the arbiter by placing a token in the port HREADY. A model of the glue logic handling the slave functionality of the controller is shown in Figure 11.7.

When the slave is ready to receive data, a token is located in place $ready$. Otherwise, there are tokens in both places $notready_1$ and $notready_2$. It is necessary to have two places $notready$ since transition cst_7 might disturb transition cst_5 in case of incoming data from the bus. The intended behaviour is that cst_5 shall fire 2 time units after enabling. With only one place $notready$, firing cst_7 would instantly first disable and then enable cst_5 again, erroneously resulting in a new enabling time for cst_5.

Meanwhile, if the slave is not ready and data is sent to it, a token is placed in HRESPout to indicate that the transaction is split, (transition cst_7). Furthermore, the address and data being sent at the time must also be removed. This is also true when the transaction of another master was split, (transition cst_{10}).

In this case study, it is assumed that the controller is ready to receive data again 2 clock cycles after a previous reception

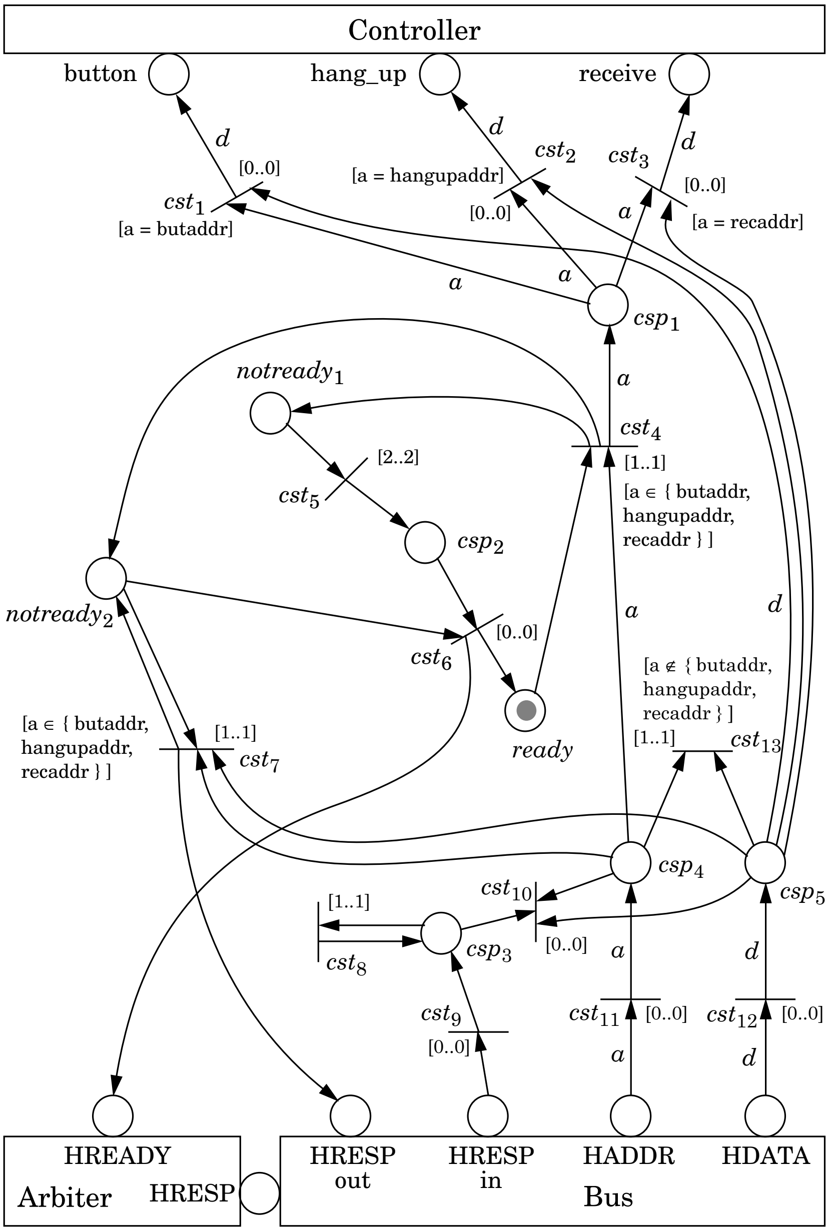


Figure 11.7: A model of the glue logic for the slave functionality of the controller

(cst_5). After these two cycles the slave indicates to the arbiter that it is ready again (cst_6).

11.2 Verification of the Model

The integration verification process is illustrated with three properties:

1. The controller only receives legal values for button.
   AG ($button \rightarrow button \in \{0..9, \text{enter}\}$)
2. When a slave has split a transaction, it will be ready again in the future.
   AG (HRESP → **AF** HREADY)
3. When a master has been granted access to the bus, it must eventually close the transaction.
   AG (HGRANT → **AF** HTRANS)

The verification is conducted following the roadmap presented in previous chapters.

11.2.1 PROPERTY 1

The first property to be verified states that the controller must only receive legal values for button. The components included in the verification of this property were the controller, arbiter, bus

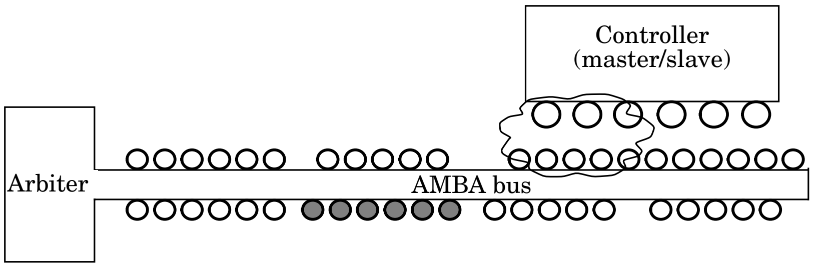


Figure 11.8: The part of the system used to verify property 1 and property 2

and the slave functionality glue logic as illustrated in Figure 11.8. Table 11.1 presents the result of the different stages in the verification process.

The property was first verified using empty stubs on all components, except the bus for which a stub was generated. The property was not satisfied using this environment since any data could arrive on the HDATA port of the bus, as indicated by the diagnostic trace. It took about 1 second to obtain this result.

Since the property was not satisfied, the diagnostic trace must be examined. According to the diagnostic trace, the bus produced a value on port HDATAs_x which is not allowed. In order to do the verification, it was necessary to make an assumption about the surrounding. In this case, it has to be assumed that only data in the set $\{0..9, \text{enter}\}$ can occur in port HDATAm_x. The property is formally given in Equation 11.1.

$$\mathbf{AG}\ (\text{HDATA}m_x \rightarrow \text{HDATA}m_x \in \{0..9, \text{enter}\}) \qquad (11.1)$$

A PRES+ model for this formula was created together with a new version of the bus stub, now also including port HDATAm_x, indicated by the shaded interface in Figure 11.8. Using this new stub, the property was satisfied using approximately 2 minutes verification time.

The positive verification result was obtained by making an assumption about the surrounding. In order to finally conclude

Table 11.1: Verification results of property 1

Step	**Environment**	**Res**	**Time**
Initial	All empty stubs, except bus generated	false	1.32s
Add assumption on HDATA	All empty stubs, except bus, assumption	true	125.33s
Verify assumption	Buttons top-level stub, other stubs empty	true	7.58s

the positive result, the correctness of the assumption in Equation 11.1 must first be established.

The components involved in verifying the assumption were the buttons, arbiter, bus and master functionality glue logic (Figure 11.9). A top-level stub for buttons and empty stubs for the other components was enough for obtaining a result within 7.58 seconds.

11.2.2 PROPERTY 2

The second property states that when a slave has split a transaction, it must become ready again in the future. The components included in the verification of this property were controller, arbiter, bus and the slave functionality glue logic, as also illustrated in Figure 11.8. Table 11.2 presents the result of the different stages in the verification process.

This verification has been started with a faulty glue logic. The fault is due to that the slave functionality glue logic did not emit HRESP in time (Section 11.1.3). This fault was finally fixed after

Table 11.2: Verification results of property 2

Step	**Environment**	**Res**	**Time**
Initial	All empty stubs, except {HRESP*in*, HRESP*out*, HRESP}	false	2.47s
Use higher level stubs	Initial except Level 1 stubs for HADDR, HDATA	false	28.39s
Correct design error	Initial except Level 1 stubs for HADDR, HDATA	false	87.57s
Use higher level stubs	Empty stubs for controller and top-level stub for the bus	true	246.14s

detection (see the third verification step below) by changing the time delay interval in one transition in the glue logic.

At first, the property was verified using empty stubs on all components, except that the bus had one generated stub corresponding to interface $\{\mathrm{HRESP}ins_x, \mathrm{HRESP}outs_x, \mathrm{HRESP}\}$. The property was however not satisfied in this environment. The diagnostic trace indicated that messages were sent too quickly on port HADDR and HDATA. In other words, an infinite amount of data was sent in the same clock cycle. In the real system, only one bunch of data can be sent in the same clock cycle. The problem was solved by increasing the level of the stubs on ports HADDR and HDATA from empty to level one stubs. These stubs were given (created manually).

The property was again verified in the updated environment, but it was still not satisfied. The diagnostic trace led to the design error in the glue logic as described previously. After fixing the error, the property was reverified using the very same environment, but still with a negative verification result.

The problem this time was a too pessimistic stub for the bus component. This caused the fact that no signal would ever be emitted on port HREADY. Due to the pessimism in the generated stub, it was exchanged with a given one[1]. After additional 4 minutes, the property was finally satisfied.

11.2.3 PROPERTY 3

The third property states that when a master has been granted access to the bus, it must eventually close the transaction. The components included in the verification of this property were the buttons, arbiter, bus and master functionality glue logic, as illustrated in Figure 11.9. Table 11.3 presents the result of the different stages in the verification process.

1. Another way to continue the verification would have been to continue with less pessimistic stubs generated automatically and, if needed, with added models corresponding to assumptions on the surrounding.

This verification was also started with a faulty glue logic. The fault consisted in that the glue logic could not differentiate whether a particular split request was a result of its own attempts to send or not. The fault was fixed, after detection (see the fourth verification step below) during verification, by adding a structure to keep track of the necessary information.

As with the verification of the previous properties, the first environment used consisted of empty stubs. In this environment, Arbiter may grant access to the bus without it even being requested. Consequently, after such an unrequested grant, data

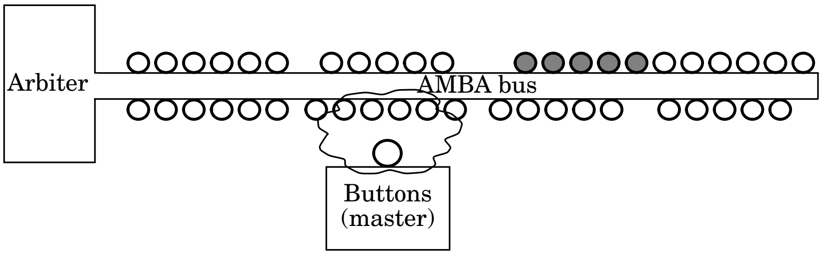


Figure 11.9: The part of the system used to verify property 3 and the additional assumption of property 1

Table 11.3: Verification results of property 3

Step	Environment	Res	Time
Initial	All empty stubs	false	0.14s
Use higher level stubs	Initial except arbiter stub	false	0.52s
Add property 2 as assumption	Arbiter stub given, button stub empty, bus with assumption	false	2.58s
Correct design error	Arbiter stub given, button stub empty, bus with assumption	true	2467.42s

will not be sent and in particular the transaction will not be closed. Thus, the property is not satisfied.

To avoid this problem revealed by the diagnostic trace, the empty stubs of the arbiter were replaced with a given stub. After half a second's verification time, the property proved again unsatisfied. The diagnostic trace shows that the reason was that a transaction can be split, but the slave will never signal after a while that it is ready to receive data again. It is, however, a requirement on the slaves to eventually signal that they are again ready after a split. Therefore, a PRES+ model corresponding to the formula $\mathbf{AG}\,(\text{HRESP} \rightarrow \mathbf{AF}_{\leq 5}\text{HREADY})$ was generated and attached to the bus on the shaded interface in Figure 11.9. Note that it is not necessary to verify this assumption as it is a requirement of the arbiter and the bus in order to work properly. Besides, the property was already verified in the previous section. Even with this extra assumption the property proved unsatisfied.

The diagnostic trace indicated an error in the glue logic. It did not record whether the split requests were a result of its own attempts to send or not. A mechanism for this was added and the property was reverified with the same environment. After 41 minutes a positive result was obtained.

11.3 Discussion

This chapter has tried to demonstrate how to use the verification methodology presented in this thesis, in practice.

The successive steps through the methodology are guided by the diagnostic trace, which all the time gives feedback to the user what to do next. It might indicate that too pessimistic stubs were used, that there is an error in the glue logic, or that assumptions regarding the surrounding have to be introduced.

PART IV
Conclusions and Future Work

Chapter 12 Conclusions and Future Work

THIS THESIS HAS PRESENTED verification techniques related both to component verification and integration verification of component-based embedded system designs. This chapter summarises these techniques and points out interesting issues for future work.

12.1 Conclusions

Embedded systems are becoming increasingly common in our everyday lives. These systems are also becoming increasingly complex. In order to reduce the design complexity, designers resort to using predesigned components, and utilise an IP-based design methodology.

Due to the high complexity, the task of building such systems correctly becomes increasingly challenging. In order to meet this challenge, verification is introduced as an integrated part of the embedded systems design flow so that errors are found early in the design process. In this process, both the components themselves and the integration of the components have to be verified.

As for component verification, two techniques are proposed in the thesis. The first technique enables formal verification of SystemC designs at several levels of abstraction. In order to be able to perform such a verification, SystemC designs are translated into a formal Petri-net based representation.

For larger designs, the feasibility of formal verification might be impeded due to state space explosion. In such cases, the designer has to resort to simulation. The second component verification technique, proposed in this thesis, tries to enhance the performance and coverage of the simulation process by injecting formal methods. After some time of simulation, an uncovered (with respect to a certain coverage metrics) part of the state space is identified. A model checker is then invoked to obtain a coverage enhancement plan which is used to guide the simulation into the uncovered part. The coverage enhancement plan consists of the diagnostic trace when checking the model for a certain property. The invocation frequency of the model checker is dynamically controlled with the aim of minimising total validation time.

The proposed integration verification technique takes advantage of the fact that the individual components in the system are already verified, for instance by using the component verification techniques described above. The focus of the technique lies on the interfaces of the components and on the glue logics interconnecting them. Every component interface has a number of properties associated to it which must be satisfied by the rest of the system in order to work correctly.

The interfaces of the components are verified one at a time for the properties associated to them. In order to actually perform such verification of the interfaces, high-level models of their components must also be included in the verification, so that the connected glue logic can interact with an environment through the interface. For this reason, so called *stubs* are introduced into the verification process. Stubs are models of the components with respect to a certain interface. From the point of view of this interface, it is not possible to distinguish between the stub and the full component.

An interface is a set of ports. Since a component has several different interfaces, and stubs are defined with respect to an interface, there also exist several stubs to choose from. This fact can be exploited for properties expressed in (T)ACTL, in order to reduce verification time. Using stubs with interfaces containing few ports, i.e. lower-level stubs, generally leads to shorter verification times. However, using low-level stubs might lead to a false verification result. If the property turns out to be unsatisfied using a certain set of stubs, another set consisting of stubs at a higher level must be used. This search for a new higher-level stub is guided by the diagnostic trace obtained from the verification. The methodology iterates in this manner until the property is proven satisfied or top-level stubs are used.

Until this point, it has been assumed that the stubs are given by the designer of each component. In case no stubs have been provided by the designer of the component, it is possible to generate the stub automatically given a model of the component and an interface. The proposed algorithms generate stubs which actually produce more events than the full component does. We say that such stubs are *pessimistic*. This enforces an iterative approach where the pessimism in the stubs is reduced as long as the ACTL properties are not satisfied. An algorithm for such a pessimism reduction has also been presented in the thesis.

The generated stubs might be too pessimistic to be used in verification, due to the fact that they assume that their surrounding is as hostile as possible. They assume that tokens may appear at those ports of the component not belonging to the stub interface at any time with any value. This assumption about the surrounding is sometimes too pessimistic. There is consequently a need to be able to express properties about the surrounding and incorporate them into the verification process.

Properties regarding the surrounding can also be expressed as ACTL formulas. An algorithm to generate a PRES+ model which produces all possible events still satisfying an ACTL formula has been presented. The generated model can then be attached to the components involved in the current verification.

A roadmap has also been developed to guide the designer throughout the integration verification methodology. A case study has, moreover, been presented in order to demonstrate the feasibility of applying the approach to realistic designs.

12.2 Future Work

This section presents a few issues which could be investigated further in order to improve, refine and extend the techniques presented in the thesis.

- The translation approach from SystemC to PRES+ (Chapter 5) would be strengthened if the result from the proposed translation procedure is proven correct based on a formal semantics of SystemC. Such a proof would not completely solve the problem of legitimacy of the translation approach by itself, although certain question marks would be straightened out. The optimal scenario, although difficult, would be to directly prove equivalence with the reference implementation of SystemC itself.
- In the formal method-aided simulation technique (Section 6.2), many parameters have to be considered. A more thorough investigation of the optimality of these, including heuristics to optimally determine them, is desired. A few examples of such parameters include the distance measure between a transition and a marking, stop criterion, time-out of the coverage enhancement plan obtainment, etc.
- The proposed stub generation algorithm generates pessimistic stubs (Chapter 9), which must later be refined in case the verified ACTL formula was unsatisfied. A heuristic approach

could be developed in which also the property to be verified is an input parameter, beside the interface and the model of the component, to the stub generation algorithm. The property might give additional hints on which parts of the component are absolutely necessary to include in the stub, and without which the property is deemed to be unsatisfied. The property to be verified might also be useful for the pessimism reduction algorithm.

- At a certain point in the stub generation algorithm, edges (arcs) are checked if they are cutedges or not (Section 9.3.2). An edge is a cutedge if the graph (PRES+ model) becomes disconnected if removed. If such cutedges exist in a graph, it is said that it has connectivity 1. The stub generation algorithm could be generalised and let the connectivity be given by the designer. If a connectivity of 2 is given, the stub would be cut at a point where two arcs, if removed, result in a disconnected model instead of 1. The higher connectivity value is chosen, the more pessimistic will the resulting stub be, since bigger parts of the component will be abstracted away. This feature could be useful in order to obtain smaller stubs in highly connected components. Heuristics to choose good connectivity values should also be developed.
- The algorithm to generate a PRES+ model from an ACTL formula (Chapter 10) imposes certain restrictions on the class of formulas it can handle, especially when it comes to the timing aspects. At present, timing requirements are only allowed on the **F** and **U** operators, and only as an upper bound. The possiblity to relax these constraints, allowing timing information on any operator, and with arbitrary relations on the time bounds, should be investigated.

References

[Ack00] B. Ackland, A. Anesko, D. Brinthaupt et al, “A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP”, in *Journal of Solid-State Circuits*, vol 35 no 3, 2000

[Alb01] K. Albin, “Nuts and Bolts of Core and SoC Verification”, in *Proc. DAC*, pp. 249-252, 2001

[Alu90] R. Alur, C. Courcoubetis, D.L. Dill, “Model Checking for Real-Time Systems”, in *Proc. Symposium on Logic in Computer Science*, pp. 414-425, 1990

[Alu94] R. Alur and D.L. Dill, “A theory of timed automata”, in *Theoretical Computer Science*, pp. 126:183-235, 1994

[And02a] T.L. Anderson, “Verification reuse enables design reuse”, *EETimes*, 19 Dec. 2002

[And02b] T.L. Anderson, “Verification: Reuse It or Lose It”, in *Proc. DesignCon*, 2002

[Bai03] M. Baird, “SystemC 2.0.1 Language Reference Manual”, *Open SystemC Initiative*, 2003

[Bal96] F. Balarin, “Approximate reachability analysis of timed automata”, in *Proc. Real-Time Systems Symposium*, 1996, pp. 52-61

[Ber05] S. Bernsen, “Combating Design Complexity with Electronic System Level (ESL) Methodology”, *Information Quarterly*, Vol. 4 No. 2, pp. 36-40, 2005

[Bra93] D. Brand, “Verification of Large Synthesized Designs”, in *Proc. ICCAD*, pp. 534-537, 1993

[Bry86] R.E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, in *Transactions on Computers*, Vol. C-35, No 8, pp. 677-691, 1986

[Bur90] J.R. Burch, E.M. Clarke, K.L. McMillan, “Symbolic Model Checking: 10^{20} States and Beyond”, in *Proc. LICS*, pp. 428-439, 1990

[Cam96] R. Camposano, J. Wilberg, “Embedded System Design”, in *Design Automation for Embedded Systems*, vol. 1, pp. 5-50, Jan 1996

[Cha02] A. Chakrabarti, P. Dasgupta, P.P. Chakrabarti et al, “Formal Verification of Module Interfaces against Real Time Specifications”, in *Proc. DAC*, pp. 141-145, 2002

[Cla86] E.M. Clarke, E.A. Emerson, A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”, in *Transactions on Programming Languages and Systems*, pp. 8(2):244- 263, 1986

[Cla99] E.M. Clarke, O. Grumberg, D.A. Peled, “Model Checking”, *The MIT Press*, 1999

[Cor00] L.A. Cortés, P. Eles, Z. Peng, “Verification of Embedded Systems using a Petri Net based Representation”, in *Proc. ISSS*, pp. 149-155, 2000

[Cor03] L.A. Cortés, P. Eles, Z. Peng, "Modeling and Formal Verification of Embedded Systems based on a Petri Net Representation", in *Journal of Systems Architecture*, pp. 49(12-15):571-598, 2003

[Cou90] O. Coudert, J.C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits", in *Proc. ICCAD*, pp. 126-129, 1990

[Cyr94] D. Cyrluk, S. Rajan, N. Shankar et al, "Effective Theorem Proving for Hardware Verification", in Proc. Int. Conf. on Theorem Provers in Circuit Design, Theory, Practice and Experience, pp. 203-222, 1994

[Daw96] C. Daws, S. Yovine, "Reducing the number of clock variables of timed automata", in *Proc. Real-Time Systems Symposium*, 1996, pp. 73-81

[Dre02] R. Drechsler, D. Große, "Reachability Analysis for Formal Verification of SystemC", in *Proc. Euromicro DSD*, 2002, pp. 337-340

[Dru03] L. Druckner, "SystemC Verification Library speeds transaction-based verification", in *EETimes Online*, http://www.eetimes.com/, 24 Feb. 2003

[Gaj00] D. Gajski, A C.-H. Wu, V. Chaiyakul et al, "Essential Issues for IP Reuse", in *Proc. ASP-DAC*, pp. 37-42, 2000

[Gir93] E. Girczyc, S. Carlson, "Increasing Design Quality and Engineering Productivity through Design Reuse", in *Proc. DAC*, pp. 48-53, 1993

[Gra97] S. Graf, H. Saidi, "Construction of abstract state graphs with PVS", in *Lecture Notes in Computer Science*, Vol. 1254, pp. 72-83, 1997

[Gro03] D. Große, R. Drechsler, "Formal Verification of LTL Formulas for SystemC Designs", in *Proc. ISCAS*, 2003, pp. 245-248

[Gro05] D. Große, R. Drechsler, "CheckSyC: An Efficient Property Checker for RTL SystemC Designs", in *Proc. ISCAS*, 2005, pp. 4167-4170

[Gru94] O. Grumberg, D.E. Long, "Model Checking and Modular Verification", in *ACM-TOPLAS*, Vol 16 No 3, pp. 843-871, 1994

[Haa99] J. Haase, "Design Methodology for IP Providers", in *Proc. DATE*, pp. 728-732, 1999

[Hab05] A. Habibi, S. Tahar, "Design for Verification of SystemC Transaction Level Models", in *Proc. DATE*, 2005, pp. 560-565

[Hab06] A. Habibi, S. Tahar, "Design and Verification of SystemC Transaction-Level Models", in *Trans. on VLSI Systems*, Vol 14(1), pp. 57-68, 2006

[Hen02] T.A. Henzinger, S. Qadeer, S.K. Rajamani et al, "An assume-guarantee rule for checking simulation", in *Trans. on Programming Languages and Systems*, Vol 24(1), pp. 51-64, 2002

[Hes03] A. Hessel, K.G. Larsen, B. Nielsen et al., "Time-optimal Real-Time Test Case Generation using UPPAAL", in *Proc. FATES*, pp. 114-130, 2003

[Kar01] D. Karlsson, P. Eles, Z. Peng, "A Front End to a Java Based Environment for the Design of Embedded Systems", in *Proc. DDECS*, pp. 71-78, 2001

[Kar02] D. Karlsson, P. Eles, Z. Peng, "Formal Verification in a Component-based Reuse Methodology", in *Proc. ISSS*, pp. 156-161, 2002

[Kar03] D. Karlsson, “Towards Formal Verification in a Component-based Reuse Methodology”, Licentiate Thesis No 1058, Linköping Institute of Technology, 2003, http://www.ep.liu.se/lic/science_technology/10/58/

[Kar04a] D. Karlsson, P. Eles, Z. Peng, “A Formal Verification Methodology for IP-based Designs”, in Proc. Euromicro DSD, pp. 372-379, 2004

[Kar04b] D. Karlsson, P. Eles, Z. Peng, “A Formal Verification Approach for IP-based Designs”, in *Proc. FDL*, pp. 556-567, 2004

[Kar05] D. Karlsson, P. Eles, Z. Peng, “Validation of Embedded Systems using Formal Method aided Verification”, in *Proc. Euromicro DSD*, pp. 196-199, 2005

[Kar06] D. Karlsson, P. Eles, Z. Peng, “Formal Verification of SystemC Designs Using a Petri-Net Based Representation”, in *Proc. DATE*, 2006

[Kea98] M. Keating, P. Bricaud, “Reuse Methodology Manual for System-on-a-Chip Designs”, *Kluwer Academic Publishers*, 1998

[Kro05] D. Kroening, N. Sharygina, “Formal Verification of SystemC by Automatic Hardware/Software Partitioning”, in *Proc. MEMOCODE*, 2005, pp. 101-110

[Kup96] O. Kupferman, O. Grumberg, “Branching Time Temporal Logic and Tree Automata”, *Information and Computation*, pp. 125(1):62-69, 1996

[Lo98] K.C. Lo, “Design for Reuse”, in *Proc. Colloquium on Systems on a Chip*, pp. 11/1-11/6, 1998

[Loc91] C.D. Locke, D.R. Vogel, T.J. Mesler, “Building a Predictable Avionics Platform in Ada: A Case Study”, in *Proc. RTSS*, pp. 181-189, 1991

[McM97] K.L. McMillan, “A compositional rule for hardware design refinement”, in *Proc. Computer Aided Verification*, LNCS 1254, pp. 207-218, 1997

[Mis81] J. Misra, K.M. Chandy, “Proofs of networks of processes”, in *Trans. on Software Engineering*, Vol 7(4), pp. 417-426, 1981

[Piz04] A. Piziali, “Functional Verification Coverage Measurement and Analysis”, *Kluwer Academic Publishers*, 2004

[Ros05] A. Rose, S. Swan, J. Pierce et al., “Transaction Level Modeling in SystemC”, *Open SystemC Initiative*, 2005

[Roy03] A. Roychoudhury, T. Mitra, S.R. Karri, “Using formal techniques to Debug the AMBA System-on-Chip Bus Protocol”, in *Proc. DATE*, pp. 828-833, 2003

[Rus01] J. Rushby, “Theorem Proving for Verification”, in *Lecture Notes in Computer Science*, Vol. 2067, pp. 39-57, 2001

[Sav00] W. Savage, J. Chilton and R. Camposano, “IP Reuse in the System on a Chip Era”, in *Proc. ISSS*, pp. 2-7, 2000

[See02] R. Seepold, N.M. Madrid, A. Vörg et al, “A Qualification Platform for Design Reuse”, in *Proc. ISQED*, pp. 75-80, 2002

[Spi03] B. Spitznagel, D. Garlan, “A Compositional Formalization of Connector Wrappers”, in *Proc. ICSE*, pp. 374-384, 2003

[Swa97] G. Swamy, “Formal Verification of Digital Systems”, in *Proc. International Conference on VLSI Design*, pp. 213-217, 1997

[Syn03] Synopsys whitepaper, “Hybrid RTL Formal Verification Ensures Early Detection of Corner-Case Bugs”, 2003

[Tas04] S. Tasiran, Y. Yu, B. Batson, “Linking Simulation with Formal Verification at a Higher Level”, *IEEE Design & Test of Computers*, Vol. 21:6, Nov-Dec 2004

[UPP] UPPAAL, http://www.uppaal.com/

[Var01] M. Varea, B. Al-Hashimi, “Dual Transitions Petri Net based Modelling Techniques for Embedded Systems Specification”, in *Proc. DATE*, pp. 566-571, 2001

[Wan93] F. Wang, A. Mok, E. A. Emerson, “Symbolic model-checking for distributed real-time systems”, *Lecture Notes in Computer Science*, Vol. 670, 1993

Abbreviations

AMBA	Advanced microprocessor bus architecture
ASIC	Application specific integrated circuit
ACTL	A (universal path quantifier) CTL
BDD	Binary decision diagram
CCM	Cruise controller module
CTL	Computation tree logic
DSP	Digital signal processor
ECM	Engine controller module
GAP	General avionics platform
HDL	Hardware description language
IP	Intellectual property
LAN	Local area network
LTL	Linear temporal logic
MCC	Mission control computer
MUV	Model under verification
PRES+	Petri-net based representation of embedded systems
RISC	Reduced instruction set computer
RTL	Register-Transfer level
SL	Signal level

SP	Separation point
SPARC	Scalable processor architecture
STB	Split transaction bus
TA	Timed automata
TACTL	Timed ACTL
TCTL	Timed CTL
TL	Transaction level
TLM	Transaction level modelling

Notations

PRES+

Notation	*Description*
C , C_i	Component
d_t^-	Lower bound of the time delay interval of transition t
d_t^+	Upper bound of the time delay interval of transition t
f_t	Transition function of transition t
g_t	Transition guard of transition t
Γ	An arbitrary PRES+ model
I	Set of input arcs
I	Interface
k	Token
M	Marking
M_0	Initial marking
$M(p)$	Marking of place p
$M(p)_v$	Value of the token in p
O	Set of output arcs
p , p_i	Place

$^\circ p$	Set of input transitions of place p
p°	Set of output transitions of place p
P	Set of places
$P(\Gamma)$	Set of places in Γ
r	Token timestamp
t , t_i	Transition
$^\circ t$	Set of input places of transition t
t°	Set of output places of transition t
T	Set of transitions
$T(\Gamma)$	Set of transitions in Γ
v	Token value
$V(\Gamma)$	Set of nodes (places and transitions) in Γ

Computation Tree Logic (CTL)

Notation	*Description*
$\mathbf{A}$	Universal path quantifier
$\mathbf{E}$	Existential path quantifier
$\mathbf{F}$	Temporal operator *future*
$\mathbf{F}_{\leq x}$	Temporal operator *future* with upper time bound x
φ	CTL formula
$\mathbf{G}$	Temporal operator *globally*
$\mathbf{Q}$	Arbitrary path quantifier
$\mathbf{R}$	Temporal operator *releases*
$\Re$	Arbitrary relation on token values
$\overline{\Re}$	Complementary relation of $\Re$
$\mathbf{U}$	Temporal operator *until*
$\mathbf{U}_{\leq x}$	Temporal operator *until* with upper time bound x
$\mathbf{X}$	Temporal operator *next step*

Formal Method Aided Simulation

Notation	*Description*
$A(\varphi)$	Set of activation sequences corresponding to φ
a_{act}	Number of activated assertions
a_{tot}	Total number of assertions
C	Parameter in the function $cov(\sigma)$
cov	Total coverage
cov_a	Assertion coverage
cov_{tr}	Transition coverage
$cov(\sigma)$	Coverage obtained after simulation length σ
d	Order number in an activation sequence
D	Parameter in the function $cov(\sigma)$
$dist(t, V, M)$	Distance from t to M while respecting V
E	Parameter in the function $cov(\sigma)$ designating the anticipated maximum possible coverage
K	CTL atomic proposition representing a set of markings
σ	Simulation length
$t_{enh}(\sigma)$	Total expected time in the coverage enhancement phase
t_{fir}	Average time to fire a transition, including the subsequent assertion checking
$t_{sim}(\sigma)$	Total expected time in the simulation phase
$t_{tot}(\sigma)$	Total expected verification time
t_{ver}	Average time spent in the coverage enhancement phase
tr_{fir}	Number of fired distinct transtions
tr_{tot}	Total number of transitions
$\mathbf{U}$	Universe, set of all possible token values in a design
V	Set of token values

Notation	*Description*
e	Event
e^+	Appearing event
e^-	Disappearing event
$Env(P, I)$	Environment corresponding to interface partition P with respect to the interface I
G	Glue logic
I_C	Interface of a component
I_S	Interface of a stub
I_{max}	Top-level interface
$I_{max}^{C,G}$	Top-level interface of C with respect to G
in	Input observation
IN	Set of all possible input observations
o	Observation
$o\|_I$	Observation restricted to interface I
Op_C	Generalised operation of C
$Op_C(in)$	Operation of C from in
$Op_C(in)\|_I$	Operation of C from in restricted to I
out	Output observation
P, P_i	Interface partition
P_{max}	Partition containing only top-level interfaces
P_{min}	The empty interface partition
$Ports(P)$	Set of ports occurring in an interface partition P
S	Stub
$Sur(G)$	Surrounding of glue logic G
$\sigma(o, M_0)$	State sequence generator
$\varnothing$	Empty stub
$\varnothing_{IN}$	Empty in-port stub
$\varnothing_{OUT}$	Empty out-port stub
$\varnothing_p$	Empty stub at port p
$\propto$	Partition precedence relation

Automatic Stub Generation

Notation	*Description*
$df(n)$	Dataflow marking at node n
$df_I(n, p)$	Dataflow marking at node n to port p in interface I
$df_I^C(n, p)$	Dataflow marking at node n to port p in interface I of component C
E, E_i	Environment
f_t^{-1}	Inverted transition function of transition t
$\sigma[i]$	The i^{th} element of sequence σ
$\mathbf{U}$	Universe, set of all possible token values in the design
$\mathbf{Z}$	Set of all integers

Modelling the Surrounding

Notation	*Description*
$AP(\varphi)$	Set of atomic propositions in φ
$AP(\Psi)$	Set of atomic propositions occurring in at least one formula in the set Ψ
$AP_{in}(\varphi)$	Set of atomic propositions in φ which denote in-ports in the design
$AP_{out}(\varphi)$	Set of atomic propositions in φ which denote out-ports in the design
$AP_{rel}(\varphi)$	Set of atomic propositions with relation in φ
$AP_{rin}(\varphi)$	Set of atomic propositions with relation in φ which refer to in-ports in the design
$AP_{rout}(\varphi)$	Set of atomic propositions with relation in φ which refer to out-ports in the design
$el(\varphi)$	Set of elementary formulas of φ
$\Phi(\varphi)$	Formula mapping of φ
$P(p)$	Port corresponding to the atomic proposition p

$P(s)$	Place corresponding to the set of progress formulas s
$PF(\Psi)$	Set of progress formulas in a set of elementary formulas Ψ
$P_{in}(s)$	Set of places to be marked when entering the state represented by the set of progress formulas s
ψ	CTL formula for which a PRES+ shall be generated
$\Psi(p_i)$	Set of progress formulas corresponding to place p_i
$PV_{in}(s)$	Port values of s related to in-ports
$PV_{out}(s, p)$	Port values of s related to out-port p
$RUF(p_i, s)$	Set of requiring U formulas of place p_i and set of elementary formulas s
s	Arbitrary set of elementary formulas
$S(\psi)$	Set of legal sets of elementary formulas of ψ
S_i	A short-hand for a particular, explicitly defined, set of elementary formulas
$sub(\varphi)$	Set of subformulas of φ
$TF(p_i)$	Set of target formulas of place p_i
$Timerin(p_i, \varphi)$	Start place of the timer in p_i corresponding to U formula φ
$Timerout(p_i, \varphi)$	End place of the timer in p_i corresponding to U formula φ
$TP(p_i)$	Set of target places of place p_i
$TTF(U)$	Set of timer triggered formulas of the set of U formulas U
$\mathbf{U}$	Universe, set of all possible token values in the design
$U(p_i)$	Set of U formulas in p_i
$\mathbf{Z}$	Set of all integers