Towards Formal Verification in a Component-based Reuse Methodology

Daniel Karlsson

INSTITUTE OF TECHNOLOGY
LINKÖPING UNIVERSITY

ISBN 91-7373-787-9, ISSN 0280-7971
PRINTED IN LINKÖPING, SWEDEN
BY LINKÖPING UNIVERSITY

Abstract

EMBEDDED SYSTEMS are becoming increasingly common in our everyday lives. As techonology progresses, these systems become more and more complex. Designers handle this increasing complexity by reusing existing components (Intellectual Property blocks). At the same time, the systems must still fulfill strict requirements on reliability and correctness.

This thesis proposes a formal verification methodology which smoothly integrates with component-based system-level design using a divide and conquer approach. The methodology assumes that the system consists of several reusable components. Each of these components are already formally verified by their designers and are considered correct given that the environment satisfies certain properties imposed by the component. What remains to be verified is the glue logic inserted between the components. Each such glue logic is verified one at a time using model checking techniques.

The verification methodology as well as the underlying theoretical framework and algorithms are presented in the thesis.

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

Acknowledgements

SPECIAL THANKS go to my supervisors Petru Eles and Zebo Peng for giving me support and guidance even in the toughest moments of my graduate studies. They patiently put me back on the right track when I am about to drift away into a dead end or when I propose a relatively cumbersome solution to a relatively simple problem.

Thank you all colleagues at IDA in general and ESLAB in particular for creating a nice and friendly working atmosphere.

I would also like to thank the master thesis students Henrik Friman and Sriharsha Naidu for implementing algorithms related to or are part of this work. Their efforts have simplified or will simplify my work tremendously.

Dessutom vill jag tacka min familj, mamma, pappa, syster och bror, för att ni alltid vill det bästa för mig.

最后但并不最轻微，我也想感谢王智萍，我的未婚妻。她是我每天的香料。

Daniel Karlsson
Linköping, November 2003

Contents

Chapter 1
Introduction

THIS THESIS DEALS WITH issues related to formal verification, in particular model checking, of embedded systems designed with reusable components.

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

1.1 Motivation

It is a well-known fact that we increasingly often interact with electronic devices in our everyday lives. Such electronic devices are for instance cell phones, PDAs and portable music devices such as Mp3-players. Moreover, other, traditionally mechanical, devices are becoming more and more computerised. Examples of such devices are cars or washing machines. In fact, in 1999, 99% of all microprocessors were used in the type of systems mentioned above (embedded systems). Only the remaining 1% was used in general purpose computers [Tur99]. This situation indicates the big importance of embedded systems.

There is no single definition of an embedded system. Different people have their own opinions about what an embedded system really is. However, many people seem to agree that the following features are common to most embedded systems [Cam96]:

- They are part of a larger system with which they continuously or frequently interact.
- They have a dedicated functionality and are not intended to be reprogrammable. 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, while a desktop computer has a wide range of functionalities.
- 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.

It is quite evident that it is both very error-prone and time-consuming to design such complex systems. At the same time there is a strong economical incentive to decrease the time-to-market.

In order to manage the design complexity and to decrease the development time, 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 such predesigned IP blocks introduces the additional challenge that the exact behaviour of the block is unfamiliar to the designer, which can lead to design errors that are difficult to detect. Discovering such faults only after the fabrication of the chip can easily cause unexpected costs of US$500K - $1M per fault [Sav00]. This suggests the importance of a structured design methodology with a formal design representation, and in particular it suggests the need for formal verification. In highly safety-critical systems, such as airplanes or medical

equipment, it is even more evident that errors are not tolerable since it is not only for economic reasons, but human lives are jeopardised.

Formal 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.

1.2 Problem formulation

Since the trend is that systems are built more and more with reusable components, it becomes increasingly important to develop verification methodologies which can effectively cope with this situation and take advantage of it.

There are several aspects which make this task difficult. One is the complexity of the systems, which makes simulation based techniques very time consuming. On the other hand, formal verification of such systems suffers from state explosion. However, it can often be assumed that the design of each individual component has been verified [See02] and can be supposed to be correct. What remains to be verified is the interface logic and the interaction between components. Such an approach can handle both the complexity aspects (by a divide and conquer strategy) and the lack of information concerning the internals of predefined components.

In order to be able to formally verify the interface logic, it is also necessary to model the environment with which it is supposed to interact. In general, the part of the system with most influence on the interface logic to be verified is the components surrounding the logic. We assume that we have some high-level models of these components. However, due to several reasons, such as the fact that these components also need to interact with other interface logics, those high-level models cannot provide a complete specification of the environment. If this turns out to be

the case in a particular verification, there must exist a mechanism to detect this and to add the missing, but necessary, information.

The aim of this thesis is to develop a verification methodology which enables the designer to verify designs consisting of interconnected predefined components.

1.3 Contributions

In this thesis we propose a formal verification approach which smoothly integrates with a component based system-level design methodology for embedded systems. The approach is based on a timed Petri-net notation which is used to model the interface logic and the components. Once the model corresponding to the interface logic has been produced, the correctness of the system can be formally verified. The verification is based on the interface properties of the interconnected components and on abstract models of their functionality. Our approach represents a contribution towards increasing both design and verification efficiency in the context of a methodology based on component reuse.

The contributions providing a basis for the proposed verification methodology include:

- Theoretical framework. A theoretical framework underlying the verification methodology has been developed. It provides strict mathematical definitions about the high-level models of the components, called stubs, used in verification. Interesting results can be deduced from these definitions and used to improve verification efficiency.
- Automatic generation of stubs. An algorithm which, given a model of a component, generates a so called stub, has been developed. The algorithm builds on the theoretical frame-

work mentioned above. It furthermore removes the obligation of the IP provider to build appropriate stubs.
- 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) in the model to be verified, as assumptions about the rest of the system. In order to do so, they must be translated into the design representation used. We propose an algorithm for doing this.

All parts of the proposed verification methodology are demonstrated in experimental results, also presented in this thesis.

1.4 Thesis Overview

The rest of this thesis is structured as follows:

- Chapter 2 gives background information on major issues in the design of embedded system with emphasis on IP-based design. Other work related to verification in this context is also presented.
- Chapter 3 addresses existing theory and definitions needed in order to understand the thesis.
- Chapter 4 introduces the big picture in which context the rest of the chapters should be put. The main features of the proposed verification methodology are presented here.
- Chapter 5 presents the theoretical framework and fundamental properties of stubs.
- Chapter 6 describes the algorithms used for automatically generating stubs. Additional theory for this part is also given.
- Chapter 7 presents an algorithm for generating a stub which satisfies a given property. Such stubs are useful when making assumptions about system properties.

- Chapter 8 illustrates the whole verification methodology by following an example.
- Chapter 9 concludes the thesis and discusses possible directions for future work.

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. Important aspects of IP reuse in general are then presented, followed by a section introducing formal verification. In the end, related work concerning formal verification of IP-based designs is presented.

2.1 Design of Embedded Systems

Designing an embedded system successfully is a very complicated task. Therefore, it is necessary to break down this task into smaller ones. Figure 2.1 outlines the early stages of a typical embedded systems design flow, from the system specification until the final model where the system is mapped and scheduled. This is the part of the design flow, the system-level, to which the work presented in this thesis belongs.

The designer starts out with a specification written in a formal or informal language. The specification contains informa-

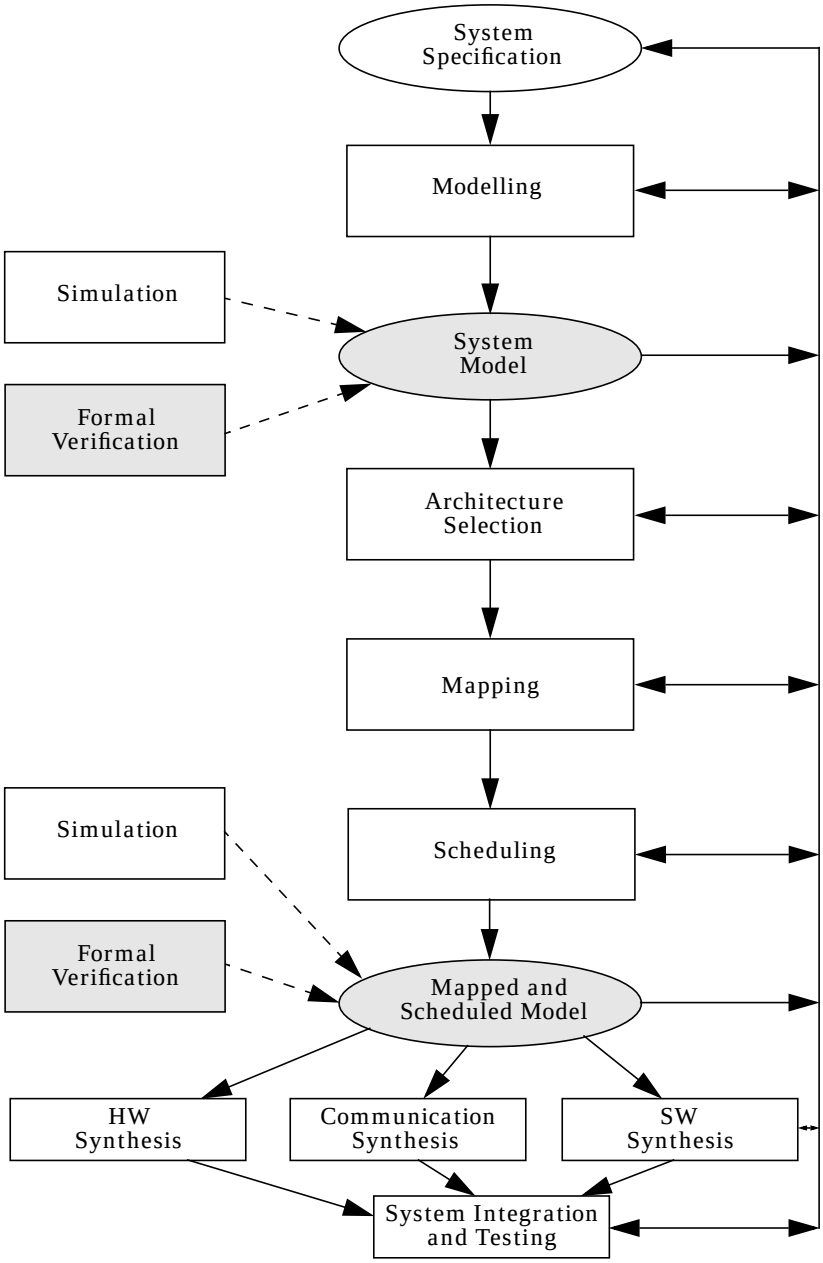


Figure 2.1: System-level design flow

tion about the system, such as its expected functionality, performance, cost, power etc. It does not specify how the system should be built, but only what system to build [Kar01].

The next step is to obtain a suitable system model. The model should capture all essential aspects of the design in order to facilitate later design steps. For this reason, it is important to choose suitable models of computation for the system model [Edw97], [Jan03].

When the system model is obtained, it should be validated to make sure that it really corresponds to the initial specification. This can be done either by simulation, formal verification or both.

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

The next step is to decide 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 collisions need either to be scheduled or a dynamic collision management mechanism has to be implemented. Constraints given in the original specification, e.g. response times, must still be satisfied. This can also be verified either by simulation, formal verification or both.

Later stages 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, a step backwards has to be performed. Such iterations are very costly, especially if errors are detected at later design steps, e.g. at prototyping when a physical model of the product has already been built.

Therefore, it is necessary 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. formal verification.

2.2 IP Reuse

By introducing reusable components, so called IP blocks (IP = intellectual property), several problems which would otherwise be absent, arise [Kea98], [Lo98]. On the other hand, using predesigned IP blocks is an efficient way of 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 twice, if it is worth this effort or not. Will the same functionality be used often enough in 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 IP user. The following subsections describe problems faced by the two categories respectively [Gaj00].

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 applied, in order to maximise the number of users. At the same time, efficiency, verifiability, testability etc must not suffer too much. 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.

Documentation is another issue which gains importance with IP reuse. The documentation must be thorough enough so that the IP user can decide whether the IP block at hand has the desired functionality and performance requirements. For this reason, data sheets, user manuals, simulation models, testbenches and more should be provided. If the provider fails to provide satisfactory documentation, it is necessary to give full customer support.

In order to facilitate reuse and to reduce the need for extensive documentation, standardisation is needed. Standardisation provides a uniform way of interconnecting components. In this way, the IP user does not need to spend time on learning how each different IP block communicates with its environment. The fact that all blocks communicate in the same way facilitates integration and makes it less error-prone. Another advantage of standardisation is that it becomes simpler to compare different

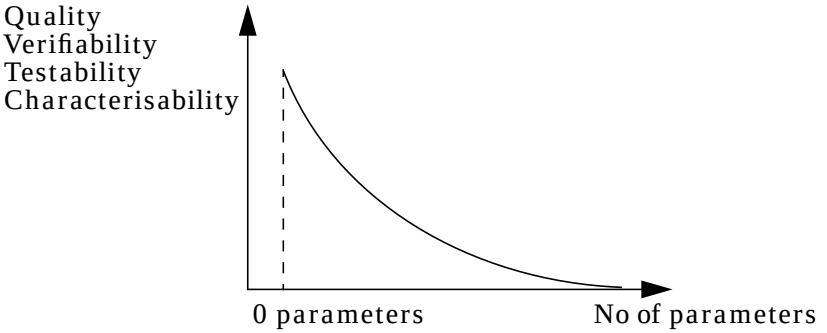


Figure 2.2: Impact of IP generality on various other parameters [Gaj00]

blocks with each other and the likelihood of choosing the best one is increased. Major standardisation efforts have been made by the VSI Alliance [VSI].

Quality assurance is also very important. Some IP users are still reluctant to adopt IP from third-party vendors because of quality assurance issues. Therefore, it is important to demonstrate that the IP delivers what it promises in its documentation. This is normally done by simulation and applying ”test vectors” to the model using a testbench [Gar98]. These methods should analyse both the functionality and the timing requirements of the IP [Yal99]. Other techniques include code coverage, prototyping and formal verification [Kea98].

Not only purely technical issues apply. There are also business and legal issues. One of the most important such issues is that of copyright. In order to protect the IP blocks from being copied by outsiders, the IP provider would like to mark the design with his signature. At the same time, it should be impossible, or as difficult as possible, to copy the design without also copying the signature. The signature must, of course, not alter the functionality of the block. One such technique is called *watermarking*. The IP provider encrypts a sentence which unambiguously ties the block to himself using a well-known encryption algorithm. The string obtained is seemlessly encoded into the design, only to appear when the block receives a certain, secret, input pattern. Revealing a copyright violation is done by applying the secret input pattern and decrypting the obtained output. Several watermarking techniques exist [Hon99], [Cal99].

2.2.2 IP USER

The IP user is the one who uses the IP blocks designed by the IP provider. This person faces similar problems as the IP provider but with a different twist.

The SoC (System on a Chip) market is literally overflowed with IP, so finding the most appropriate one for the design at

hand is not a trivial task. This imposes big requirements on the IP repository (library) [Koe98] and the search mechanisms [Reu99]. Once a seemingly suitable IP is found, it must be evaluated in the current design and possibly compared with another IP block. Hence, it is important to have an effective design exploration technique in order to produce an efficient system in short time [Pei99]. Simulation is one good method of evaluating a design and estimating its cost already at a high abstraction level early in the design process [Dal99].

Having found the proper components for the design, the next task of the IP user is to integrate them, so that they smoothly communicate with each other. However, it might be necessary to insert some logic (called *glue logic* in this thesis) between the components in order to achieve the required functionality. Figure 2.3 depicts this situation.

To facilitate the integration of components, approaches where communication related issues are separated from behaviour related issues have been developed [Row97], [Vah97].

Guaranteeing correctness of the interconnection of components and the communication is crucial but often neglected. If the interconnection does not work, the whole system will not work either, despite the fact that each individual component

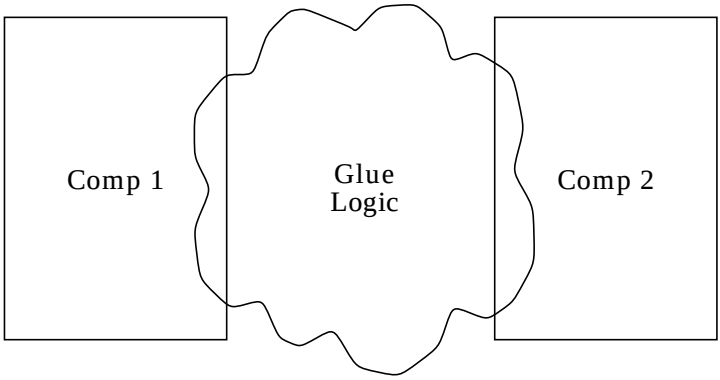


Figure 2.3: Two components and their glue logic

works according to its own specification. In this thesis, the glue logic is verified to make sure that it satisfies the requirements put by the components. If these requirements are not satisfied, the system will not work. In order to perform such verification, high-level models of the components must, in some way, be obtained [Kar02]. These models differ from the actual complete models of the components in the sense that they only have to capture the behaviour related to the interface of the glue logic under verification, while the other interfaces can, in principle, be ignored.

2.3 Formal Verification

The goal of formal verification is to find discrepancies between specification and design. As opposed to other verification techniques, such as simulation, formal verification techniques search exhaustively, but intelligently, the state space of the designed system. This means that all possible computation paths have been checked. In simulation, for example, the interaction with the system is done with an incomplete set of input vectors. Consequently, the obtained simulation results only have a certain degree of reliability.

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.

2.3.1 MODEL CHECKING

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 pos-

sibilities 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 3.2 will present more details about these logics.

The design, on the other hand, is usually given by a transition system. The exact formalism may vary between different model checking tools, but a common system, 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 larger subformulas are satisfied are marked based on the sets of states obtained for the smaller formulas. 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 gives a counter-example containing a sequence of transitions leading to a state which contradicts the 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 of 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 so called *state explosion problem*.

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 (or 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 in design refinement. When a new, refined, design is obtained, it is desired to check that it is equivalent with the old, less refined, version. 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 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.

$$background + environment + system \vdash specification \quad (2.1)$$

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.

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.

2.4 Formal Verification of IP Interconnection

Section 2.2 presented the aspects of IP reuse and Section 2.3 those of formal verification. In this section, work trying to combine these two areas, i.e. verifying the interconnection between two or more IP blocks is presented.

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 the correctness of 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. fulfills the assumptions of the other component, then their composition will satisfy the whole specification.

2.4.2 Modelling the Environment in the Property Formulas

A different approach 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.

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

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

2.4.3 Constructing Tableaux for Modelling the Environment

In order to be able to smoothly incorporate environments of a component expressed as property formulas, as in Section 2.4.1 and Section 2.4.2, into common model checking tools, they need to be translated into transition systems. When speaking of such translations, the transition systems are often referred to as *tab-*

leaux in literature. Unfortunately, not all formulas can be translated into tableaux. Only so called LTL formulas and ACTL formulas can be translated (see Section 3.2 for details) [Gru94].

The goal of the translation is to create a tableau which can produce all possible events, given the constraints of the formula. This means that the tableau must, in some sense, be maximal.

The technique of constructing tableaux from formulas can be used for basic model checking purposes. Having constructed the tableau T for the formula, T can be incorporated into a model M, obtaining a new model $M \parallel T$ behaving in the same way as M assuming the environment T. Verification can then be performed on $M \parallel T$ following normal procedures.

Chapter 3
Preliminaries

THIS CHAPTER PRESENTS the necessary background theory in order to understand the rest of this thesis. First, the design representation which will be used throughout the thesis will be introduced. Second, a brief introduction to Computation Tree Logic (CTL) follows. Finally, a brief overview of partial orders and lattices is presented.

3.1 The Design Representation: PRES+

As pointed out in Section 2.1, it is very important to choose a good design representation. In this work, we have chosen 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].

3.1.1 STANDARD PRES+

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

■

The following notions of classical Petri Nets and extensions typical to PRES+ are the most important in the context of this thesis (see 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.
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 $M(p) \neq \emptyset$.

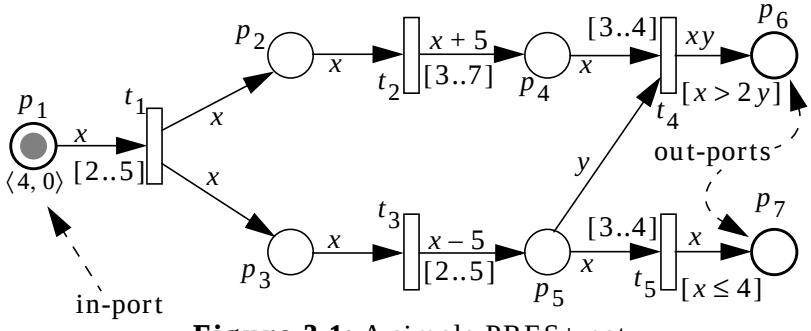
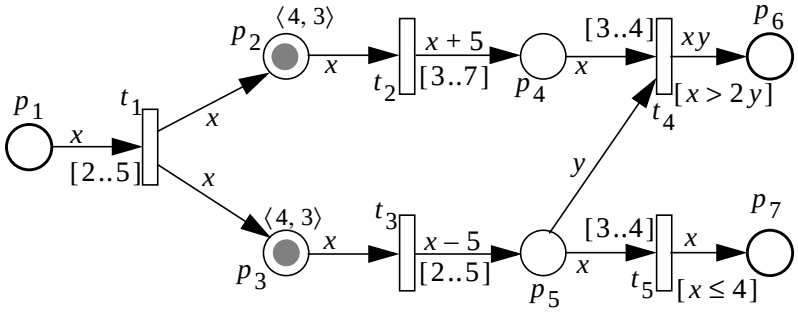


Figure 3.1: A simple PRES+ net

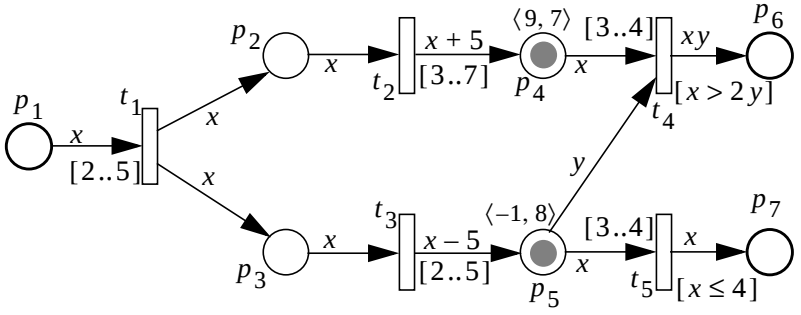
3. A transition has a function and a time delay interval associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp of the new tokens is the maximum timestamp of the enabling 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. 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. A transition can only be enabled if the value of its guard is true (see transitions t_4 and t_5).
5. The preset ${}^\circ t$ (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 t's guard is satisfied.

3.1.2 DYNAMIC BEHAVIOUR

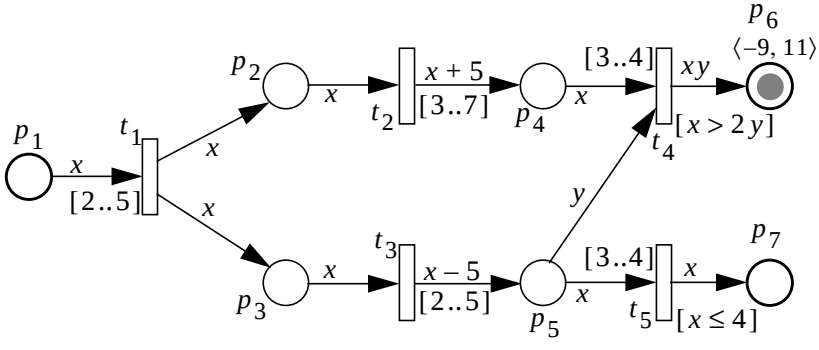
Figure 3.2 illustrates the dynamic behaviour of the example given in Figure 3.1. In the situation of Figure 3.1, transition t_1 can fire at any time between 2 and 5. Assuming that it fires at time 3, the situation in Figure 3.2(a) is reached. Both transitions t_2 and t_3 are now enabled. t_2 can fire 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 $3 + 5 = 8$.



(a)



(b)



(c)

Figure 3.2: Example of the dynamic behaviour of PRES+

Note that it is not necessary that all tokens have the same timestamp. 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 $max(7, 8) + 3 = 11$.

3.1.3 FORCED SAFE PRES+

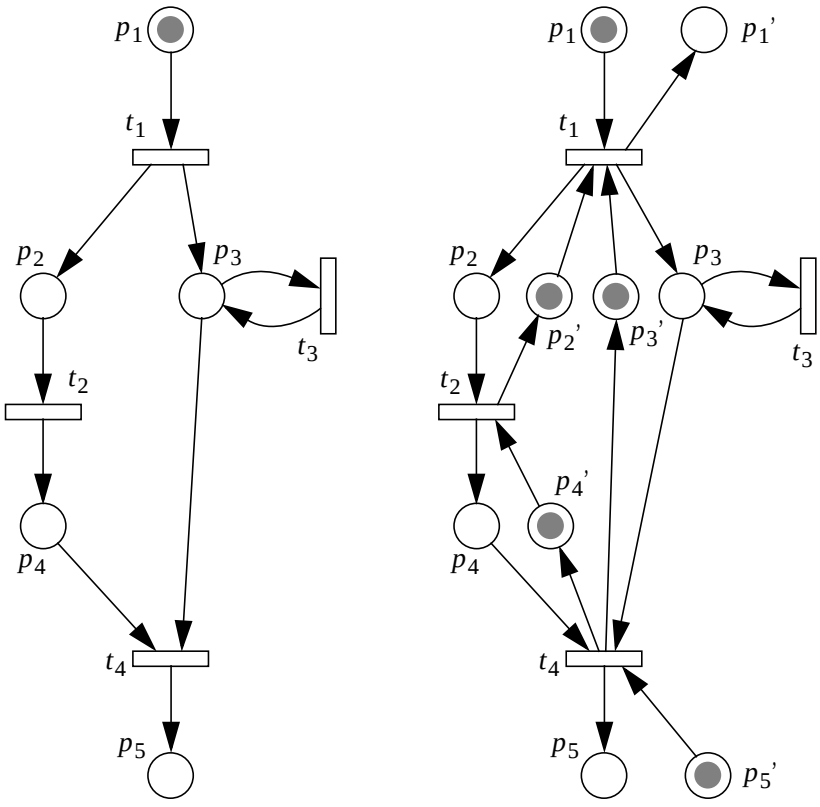
In this thesis, a modification to the enabling rule (item 6 in the list defining standard PRES+, Definition 3.1) of transitions is made.

- 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 guard is satisfied.

The rule is added to guarantee 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 relatively straight-forwardly 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 so that it has a shadow place 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. The 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 $\langle t_3, p_3 \rangle$.)



(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+

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

3.1.4 COMPONENT ASPECTS OF PRES+

We will now define three concepts critical to our methodology, in the context of the PRES+ notation.

Definition 3.2: Component. A component is a subgraph of the graph of the whole system $\Gamma = P \cup T$ (P is the set of places and T is the set of transitions) such that:

1. Two components $C_1, C_2 \subseteq \Gamma$, $C_1 \neq C_2$ may only overlap with their ports (Definition 3.3), $C_1 \cap C_2 = P_{con}$, where $P_{con} = \{p \in P | (p^\circ \subseteq C_2 \wedge {}^\circ p \subseteq C_1) \vee (p^\circ \subseteq C_1 \wedge {}^\circ p \subseteq C_2)\}$.
2. The pre- and postsets (${}^\circ t$ and t°) of all transitions t of a component C, must be entirely contained within the component, $t \in C \Rightarrow {}^\circ t, t^\circ \subseteq C$.

■

Definition 3.3: Port. A place p is an out-port of component C if $(p^\circ \cap C = \emptyset) \wedge ({}^\circ p \subseteq C)$. A place p is an in-port of C if $({}^\circ p \cap C = \emptyset) \wedge (p^\circ \subseteq C)$. 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.

Note that tokens can appear in in-ports at any time with any value. Dually, tokens can disappear from out-ports at any time. It should be imagined that the component is connected to other components placing and removing tokens from the in-ports and out-ports respectively.

Definition 3.4: Interface. An interface of component C is a set of ports $I = \{p_1, p_2, ..., p_n\}$ where $p_i \in C$.

■

Returning again 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 *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

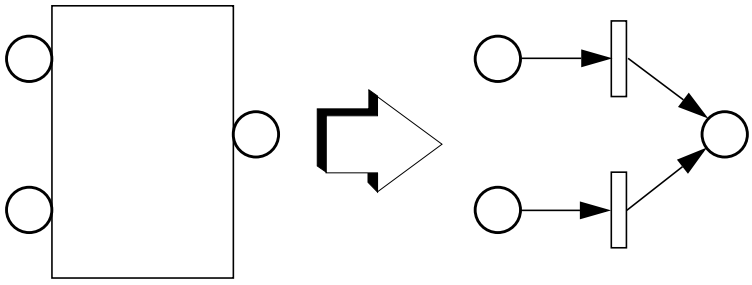


Figure 3.4: Component substitution

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).

3.2 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 possibilities of events happening in different futures.

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 (including the initial state). **F** states that the particular property will be true some time in the future (including the initial state), whereas **X** 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. **Q**[p **U** q] (for any path quantifier **Q**) means 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

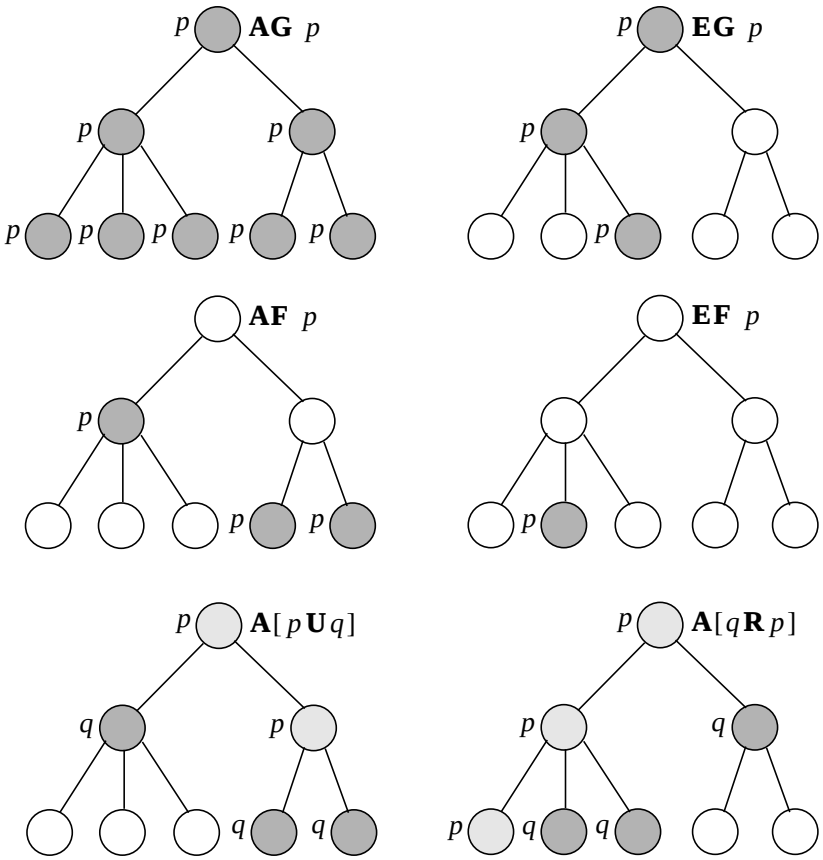


Figure 3.5: Illustration of different CTL formulas

including, q. It is not specified how far away the future when q is true is, but it must come eventually, hence **QF** q is also true.

$\mathbf{Q}[q\mathbf{R}p]$ 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, $\mathbf{AG}\,(p \rightarrow \mathbf{EF}\,q)$ means that once p is true, then it must be *possible* that q is true in the future. $\mathbf{AG}\,(p \rightarrow \mathbf{A}[p\mathbf{U}q])$ means that once p is true, it remains true until q becomes true and **AGAF** p states that p must be a reoccurring event, i.e. always be true in the future from any state no matter what happens.

The remaining question is how to interpret the atomic propositions in our design representation PRES+. Every place in the Petri-net has a label. A CTL formula consisting of an atomic proposition which is the label of a place, e.g. p, is true if there is a token in that place, p. A negated label, $\neg p$, is true if there does not exist any token in the corresponding place, p.

In order to be able to verify token values, labels can be compared to values using an appropriate relation $p\Re v$, where $\Re$ is a relation and v is any 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 formulas $p_4 = 9$ and $p_5 \leq 0$ are true.

The negation of an atomic proposition with relation, $\neg p\Re v$, means 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.

$\mathbf{AG}\,(p_1 \rightarrow \mathbf{AF}\,(p_6 \vee p_7))$ and $\mathbf{AG}\,(p_1 \rightarrow p_1 \leq 10)$ are both examples of CTL formulas for 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 does only occur 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 be able to set a time limit whithin 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 $\sim \in \{<, \leq, =, \geq, >\}$ intuitively indicates the relationship between time point x and the time point when p must be true. For instance, $\mathbf{AF}_{\leq 5}p$ 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 formulas augmented with time are called TACTL.

3.3 Partial Orders and Lattices

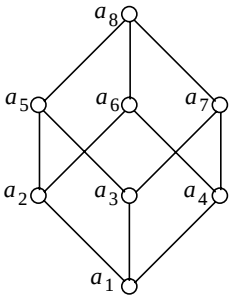
In Chapter 5, results based on partial orders and lattices are presented. This section will therefore give a brief introduction to this area. For more elaborate explanation, see [Grä78].

Definition 3.5: Partial order. A relation $\leq$ is a partial order if it is reflexive, antisymmetric and transitive.

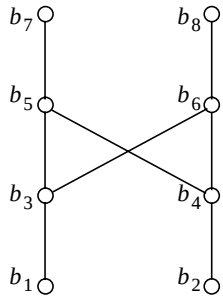
■

Definition 3.6: Poset. If $\leq \subseteq A \times A$ is a partial order, then the pair $\langle A, \leq \rangle$ is called a partially ordered set, or poset.

■



(a) Lattice



(b) Poset not a lattice

Figure 3.6: Examples of posets

Definition 3.7: Upper (Lower) bound. Let $\langle A, \leq \rangle$ be a poset and $B \subseteq A$. Then $x \in A$ is called an upper (lower) bound of B, $B \leq x$ ($x \leq B$), iff $y \leq x$ ($x \leq y$) for all $y \in B$.

■

Definition 3.8: Least upper (Greatest lower) bound. Let $\langle A, \leq \rangle$ be a poset and $B \subseteq A$. Then $x \in A$ is called a least upper (greatest lower) bound of B iff $B \leq x$ ($x \leq B$) and $x \leq y$ ($y \leq x$) whenever $B \leq y$ ($y \leq B$).

■

Least upper and greatest lower bounds are unique if they exist.

Definition 3.9: Lattice. A lattice is a poset $\langle A, \leq \rangle$ where every pair of elements $x, y \in A$ has a least upper bound, denoted $x \vee y$, and a greatest lower bound, denoted $x \wedge y$.

■

Figure 3.6 shows two examples of posets. The poset in Figure 3.6(a) is a lattice since every pair of nodes have both a least upper bound and a greatest lower bound. For instance, $a_2 \vee a_3 = a_5$ and $a_2 \wedge a_3 = a_1$. However, the poset in Figure 3.6(b) is not a lattice since several pairs do not have a least upper bound or greatest lower bound. Both

$b_3, b_4 \leq \{b_5, b_6\}$ (so they are both lower bounds), but neither $b_3 \leq b_4$ nor $b_4 \leq b_3$ (so none of them is the greatest lower bound).

Definition 3.10: Complete lattice. A complete lattice is a poset $\langle A, \leq \rangle$ where every subset $B \subseteq A$ (finite or infinite) has a least upper bound $\bigvee B$ and a greatest lower bound $\bigwedge B$. The element $\bigvee A$ is called the top element and $\bigwedge A$ is called the bottom element.

■

Theorem 3.1: Any finite lattice is a complete lattice.

■

According to Theorem 3.1, it is enough to prove that a finite poset is a lattice in order to prove that there exists a top or bottom element. The lattice in Figure 3.6(a) is consequently a complete lattice with top element a_8 and bottom element a_1.

Chapter 4
The Verification Methodology

THIS CHAPTER PROVIDES an overview of the proposed verification methodology. It is based on details explained thoroughly later in Chapter 5 and Chapter 6.

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

4.1 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 4.1 at a high level of abstraction. 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 4.2, 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 LAN and protocol handlers are also components. 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

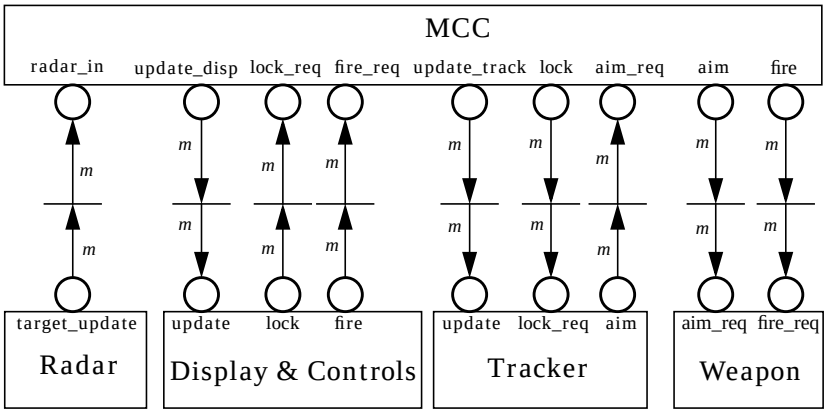


Figure 4.1: A high level model of the GAP example

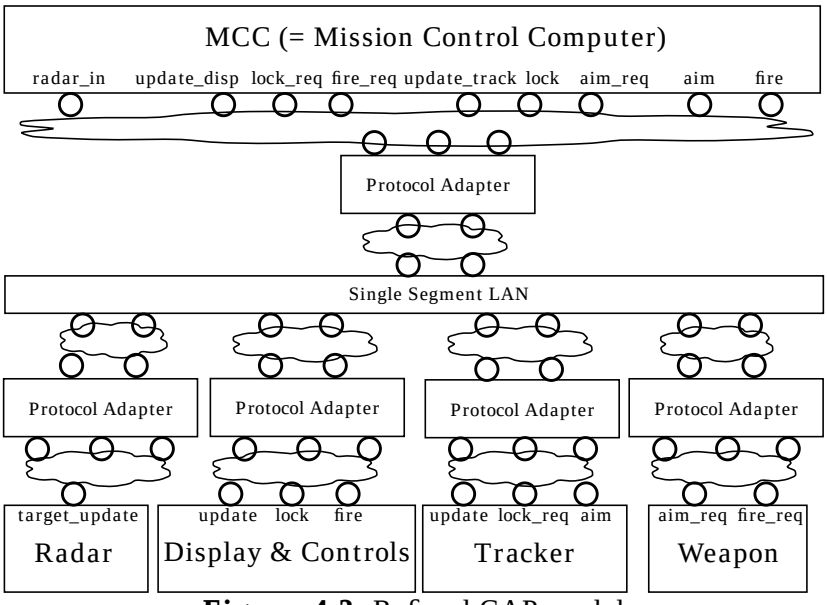


Figure 4.2: Refined GAP model

in Figure 4.1 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 4.3[1] (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

1. Inhibitor arcs are drawn with a small circle instead of an arrow in one end. The function of inhibitor arcs is to disable otherwise enabled transitions. In PRES+, inhibitor arcs are only syntactic sugar for a more complex structure which performs the same functionality.

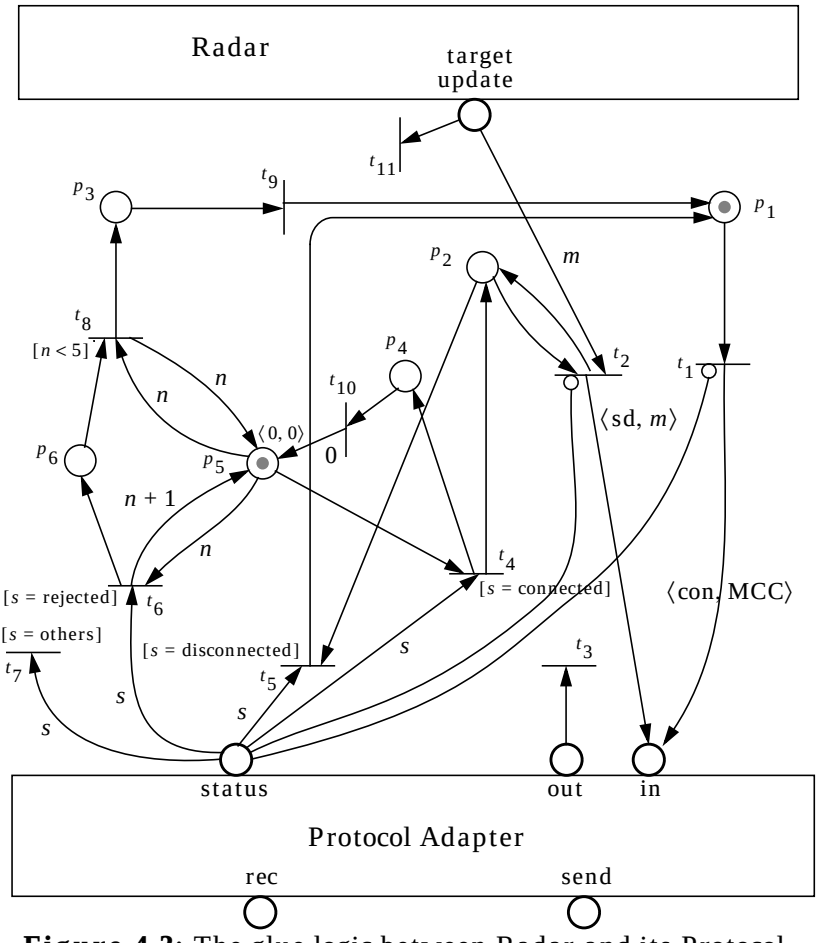


Figure 4.3: The glue logic between Radar and its Protocol

achieve this, a token with value $\langle \text{con}, \text{MCC} \rangle$ is passed to the Protocol adapter, meaning 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. 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, the glue logic will continue to attempt to connect, at most five times. 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.

4.2 Objective and Assumptions

The objective of the proposed methodology is the following: verify the glue logic so that it satisfies the requirements imposed by the connected components.

The methodology is based on the following three assumptions:

- The components themselves are already verified.
- The components have some requirements on their environment associated to them expressed in a formal notation.
- A model of the component is provided which is used together with the attached glue logic in the verification process.

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 glue logic and the interaction between the components through the glue logic.

According to the second assumption, the components impose certain requirements on their environment. These requirements have to be satisfied in order for the component to function correctly. The requirements are expressed with (T)CTL formulas, described in Section 3.2, in terms of the ports in a specific interface. It is important to note that these formulas do not describe the behaviour of the component itself, but they 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 4.1. The communication protocol chosen in the example was connection based

(Figure 4.2). A Protocol adapter implementing the chosen protocol was supplied and verified by a provider and (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:

$$\mathbf{AG}\ ((status = \text{disconnected} \vee \text{init}) \rightarrow \mathbf{A}\ [status = \text{connected}\ \mathbf{R}\ \neg in = \langle \text{send}, \_ \rangle]) \tag{4.1}$$

$$\mathbf{AG}\ (status = \text{connected} \rightarrow \mathbf{A}\ [status = \text{disconnected}\ \mathbf{R}\ (\neg in = \langle \text{connect}, \_ \rangle \wedge \neg in = \langle \text{listen}, \text{-} \rangle)]) \tag{4.2}$$

Equation 4.1 states that the protocol can never receive a send command when it is disconnected. Equation 4.2 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 be able to verify the glue logic, a model of the attached components is needed. Such models are called *stubs* and are formally defined in Chapter 5.

Consider the Protocol adapter component in Figure 4.3. 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 4.4 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 I. This non-determinism is modelled in Figure 4.4 with the conflicting transitions s_4 and s_5. The response can either be ”rejected” or ”connected”. When connected, messages can be received from the party to which the

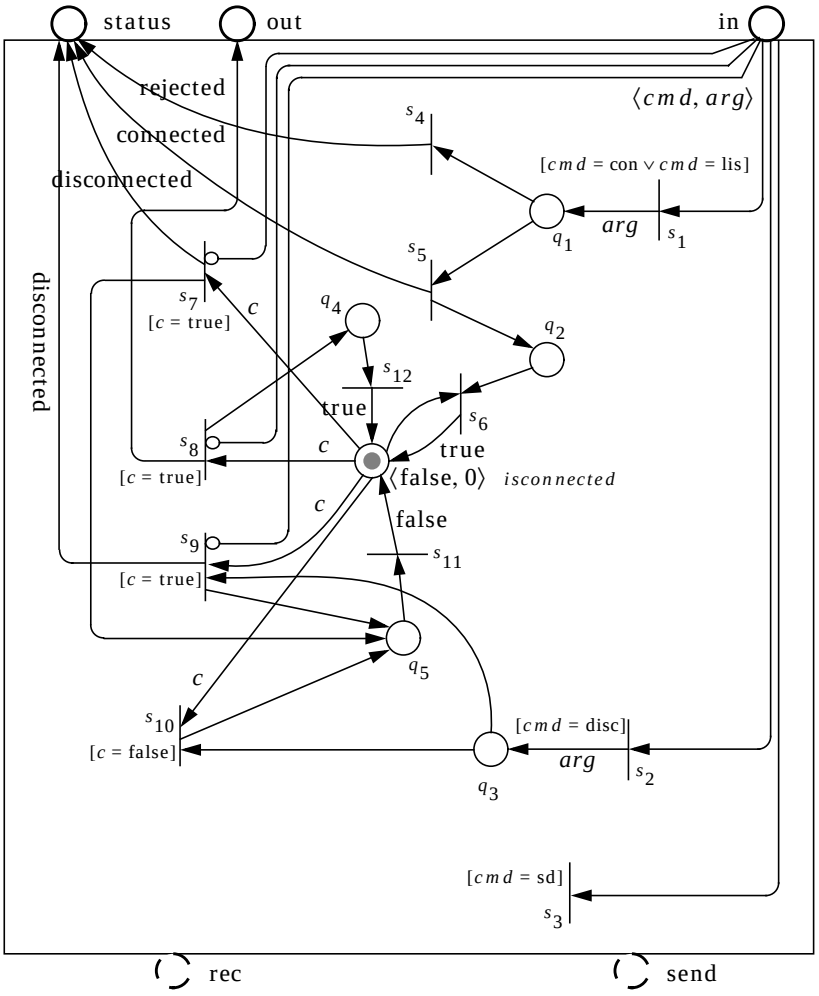


Figure 4.4: A simple stub of the Protocol adapter

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). Disconnection 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.

4.3 Methodology Overview

Our verification approach is illustrated in Figure 4.5. In order to verify the glue logic it is needed to integrate its model with stubs of the components it is connected to. These stubs capture the characteristics of the outputs produced by the components as a result of the given input and, by this, they provide the environment for the glue logic to be verified. The model composed of one or more stubs and the glue logic itself is then passed to the model checker together with the (T)CTL formulas associated to the involved interfaces of the components. The model checker then answers whether or not the given properties are satisfied.

Components, glue logics and stubs are all modelled in PRES+. It should be mentioned that, in order to perform the model checking, PRES+ has to be translated into the input language of the particular model checker used. For the work in this thesis, the models are translated into timed automata [Alu94] for the UPPAAL model checking environment [UPP], using the algorithms described in [Cor00]. In addition to the formulas provided together with the components, the designer can add formulas invented by himself which he wants to be verified. These additional formulas may be conditions on the ports of the components or they can refer to any place in the glue logic in order to verify the functionality of the glue logic.

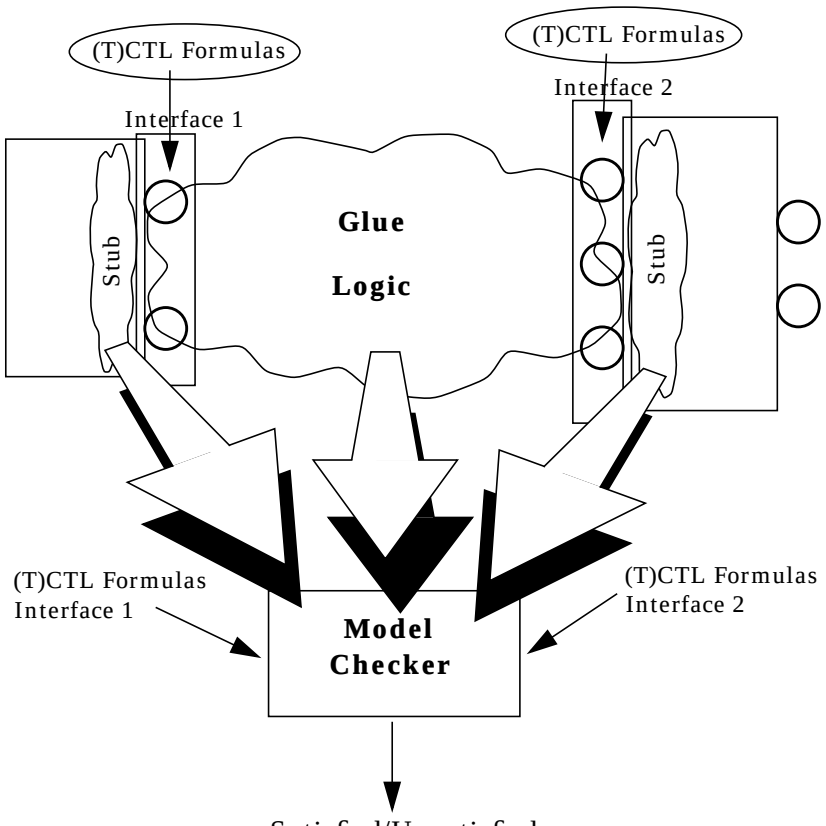


Figure 4.5: Overview of the proposed methodology

4.3.1 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 4.6. The system consists of two components, Doubler and Strange, and there is a glue logic in between 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 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 4.7 lists the stubs corresponding to the example in Figure 4.6. The stub for interface $\{arg\}$ simply consumes any token which arrives, and the stub for $\{output\}$ produces tokens 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:

$$\mathbf{AG}\ (input \rightarrow even(input)) \tag{4.3}$$

$$\mathbf{AG}\ (arg \rightarrow \mathbf{A}\ [arg\ \mathbf{U}\ \mathbf{A}\ [output\ \mathbf{R}\ \neg arg]]) \tag{4.4}$$

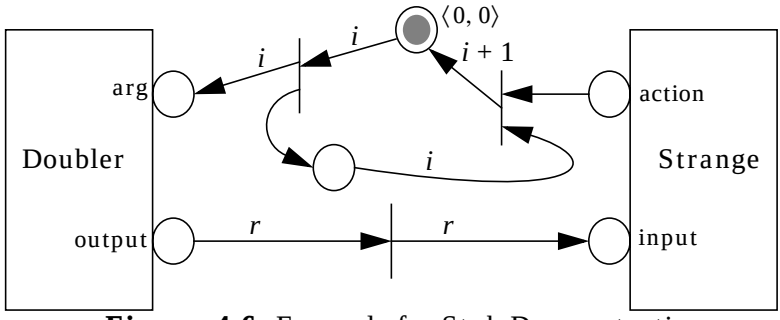


Figure 4.6: Example for Stub Demonstration

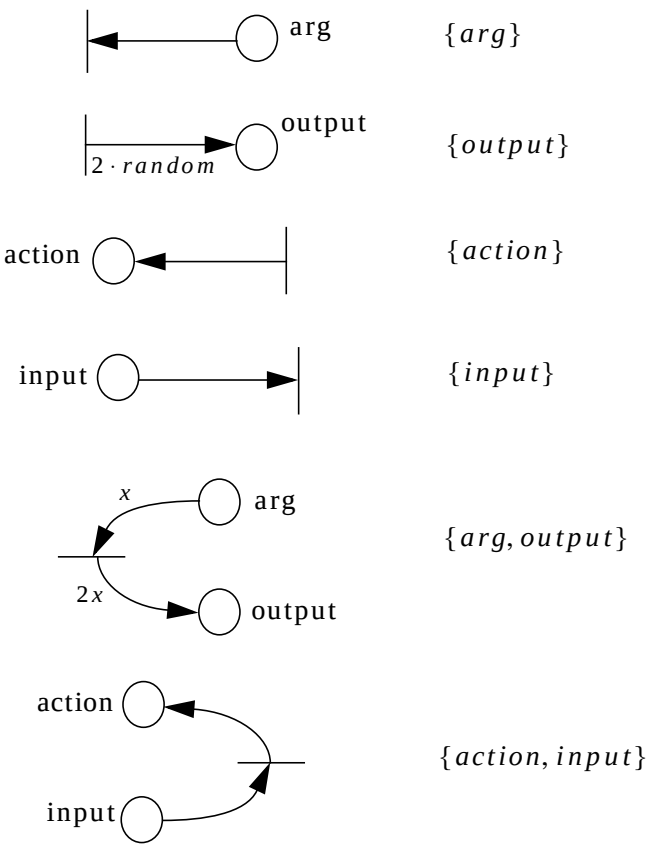


Figure 4.7: Stubs used in the example in Figure 4.6

$$\mathbf{AG}\ (arg \rightarrow arg \geq 0) \tag{4.5}$$

$$\mathbf{AGEF}\ input < 0 \tag{4.6}$$

To check formula 4.3 (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 5, using fewer and smaller stubs may reduce the verification time.

Formula 4.4 (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. Formula 4.5 (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.

Let us look at formula 4.6 (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 formula 4.6 is not an ACTL formula as opposed to the other formulas.

It is obvious that using the stub 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 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 5.

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 of the component not connected to the actual glue logic under verification. In such cases, it is possible to construct a model corresponding to these logic formulas, as was mentioned in Section 2.4.3. 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 7.

4.3.2 VERIFICATION METHODOLOGY ROADMAP

In order to support the designer, it is necessary to introduce some structure to the verification process, so that the verifyer 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 suggests 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 verifyer 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 verification process are discussed in more detail.

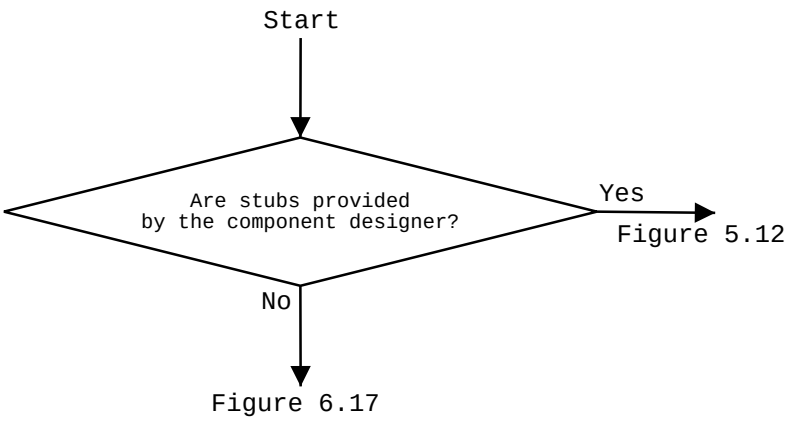


Figure 4.8: The start of the roadmap

The methodology consists of two main parts, presented in Chapter 5 and Chapter 6 respectively. The first part assumes that stubs are already given by the component providers. The problem is 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 created given this model.

Since the methodology includes these two distinct parts, the first question in the roadmap to be answered by the verifyer is intended to guide the verification into either part. Figure 4.8 presents the first question.

In the roadmap presented in this thesis, diamonds denote Yes and No questions to be answered by the verifyer. 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 verification (model checking) 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 the component. A (set of) stubs to represent each connected component must have been selected or created when the actual model checking of the glue logic is performed.

Chapter 5
Verification of Component-based Designs

IN THIS CHAPTER the theoretical framework underlying the 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 4.3.2.

5.1 Definitions

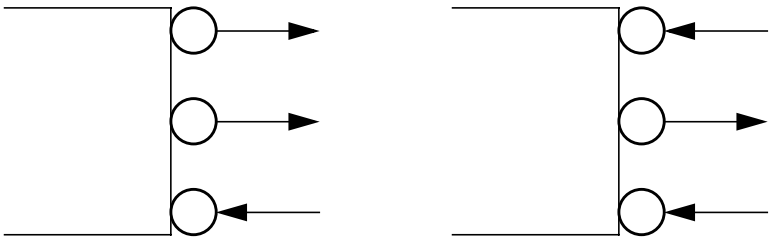
In Section 4.2 we concluded that some description mechanism of the components is necessary in the verification process. We have previously called components describing another component ”stub”. In this section, a mathematical definition of what a stub exactly is will be given. Before defining a stub, some auxiliary concepts have to be defined.

Definition 5.1: Interface compatibility. Interfaces I_1 and I_2 are compatible iff there exists a bijection $f: I_1 \rightarrow I_2$ such

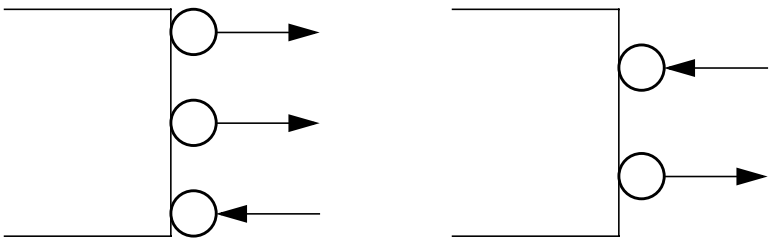
that if $f(p) = q$, then p and q are both either in-ports or out-ports in their respective interface.

■

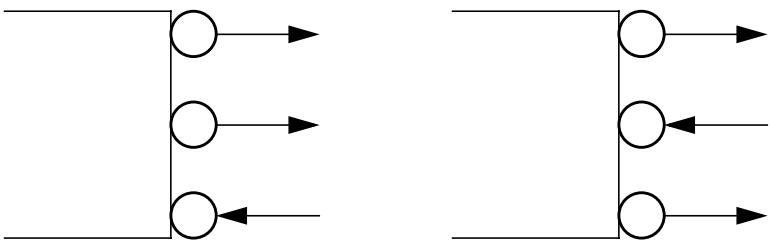
Remembering that interfaces are sets of ports, (Definition 3.4), it is inuitive to see that two interfaces are compatible if they have equally many in-ports and equally many out-ports. Figure 5.1 illustrates this concept further. The interfaces in Figure 5.1(a) are not compatible since the left-hand component



(a) Incompatible interfaces



(b) Incompatible interfaces



(c) Compatible interfaces

Figure 5.1: Illustration of interface compatibility

has two out-ports and one in-port, whereas the situation in the right-hand component is the reverse. The interfaces in Figure 5.1(b) contain a different number of ports and are thus not compatible either. Only the interfaces in Figure 5.1(c) are compatible, since they have an equal number of in-ports and out-ports respectively.

Definition 5.2: 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. Appearing events represent 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. Disappearing events represent 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 5.3: 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 $o|_I = \{\langle p, k \rangle \in o \mid p \in I\} \cup \{\langle p, r \rangle \in o \mid p \in I\}$. 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 5.2 illustrates the concept of observations according to Definition 5.3. 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

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

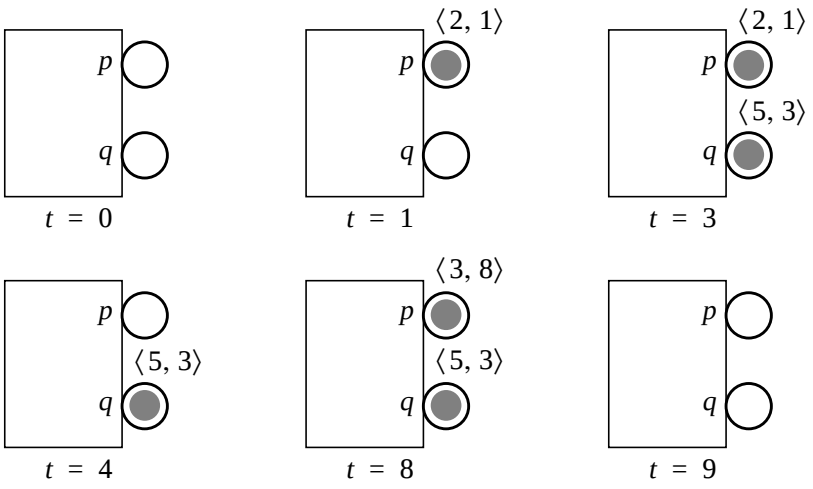


Figure 5.2: Illustration of observations

$t = 3$ 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.

When discussing about input and output observations, the interest is concentrated on what a user of a component inputs to it or receives as output from it. Since the user is also affected by the time when tokens are consumed by the component, the disappearing events on in-ports have also been included in output observations. A similar argument holds for disappearing events on out-ports in the case of input observations.

Definition 5.4: 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+ net 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 $Op_C(in)\big|_I = \{o_1\big|_I, o_2\big|_I, \ldots\}$. ■

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 what a stub is. In Chapter 4, stubs were described as a piece of PRES+ net 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 5.5: 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, $Op_C(in)\big|_{I_C} = Op_S(in\big|_{I_S})$. ■

Since the lefthand-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+ net 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+ net of S given the same input, but only those events of the input belonging to a port in I_S (implicitly applying the bijective function defined by the interface compatibility in Definition 5.1). 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.

5.2 Relations between Stubs

As the concept of stubs has now been formally defined, it is time to investigate how stubs belonging to different interfaces of a component relate to each other.

> **Definition 5.6:** Top-level interface. The top-level interface of a component C, with respect to a glue logic G, is the set of 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 4.3, 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, $\{in, out\}$, $\{in, status\}$, $\{out\}$ 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 5.5. 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 5.3 presents a partial order (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). The lattice induces 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. However, the implementation is not bound to be the same. In the bottom of the

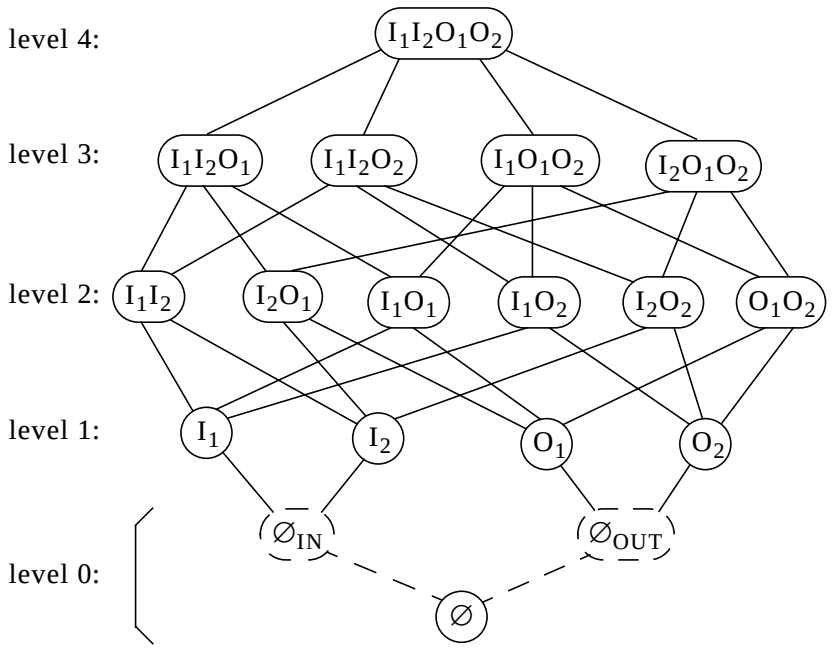


Figure 5.3: A partial order of interfaces

(a) $\varnothing_{IN}$ (b) $\varnothing_{OUT}$

Figure 5.4: The models of the empty stubs

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 5.4. 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 5.3.

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, more specialised stubs can be found which introduce 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 $\varnothing_{OUT}$, respec-

tively. 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 still consistent with the token values arriving at the in-ports given the behaviour of the full component. Hence, for instance, in Figure 4.4 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 4.4.

5.3 Verification Environment

In Section 4.3.1, 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 5.3, 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 5.7: Interface partition. An interface partition P is a set of non-empty interfaces $P = \{P_1, P_2, \ldots\}$ such that $P_i \cap P_j = \emptyset$ 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 3.4, 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 4.6, $P = \{\{arg\}, \{output\}\}$, $Q = \{\{arg\}, \{output\}, \{action, input\}\}$ 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 5.8: Partition precedence. Partition P precedes partition Q, $P \propto Q$, iff $\forall p \in P \exists q \in Q : p \subseteq q$.

■

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 accurate than those in P or Q, since they capture more of the causalities and dependencies between their ports.

Theorem 5.1: 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 5.8. 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 5.2: 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 5.3). $P_{min} = \varnothing \propto P$ is trivial.

■

In fact, the precedence relation does not only have top and bottom elements, but it is a lattice. Since it is a lattice and finite, Theorem 3.1 implies Theorem 5.2.

Definition 5.9: 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[1].

■

Let us consider the example in Figure 4.6 with the stubs of the components in Figure 4.7. With $J = \{arg, output, action, input\}$, Figure 5.5(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 5.4) has been added. Figure 5.5(b) shows a similar example for $Env(\{\{arg\}, \{output\}\}, J)$. In Figure 5.5(c), no empty

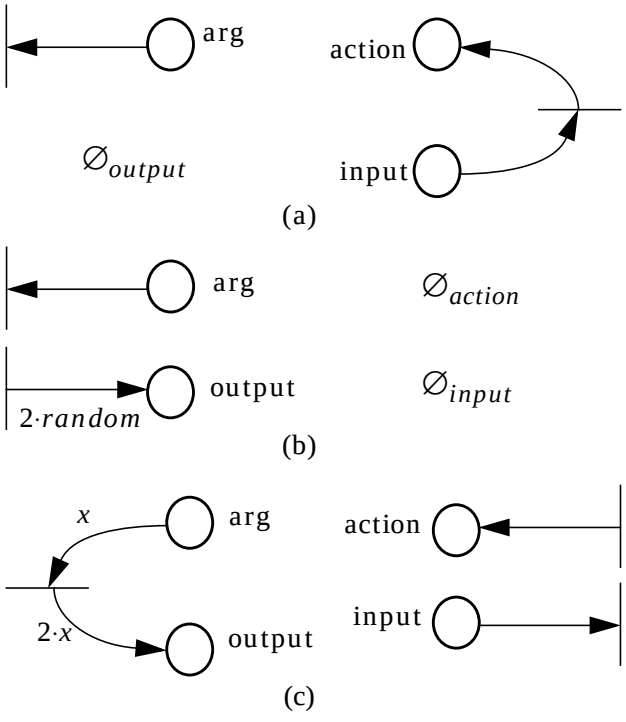
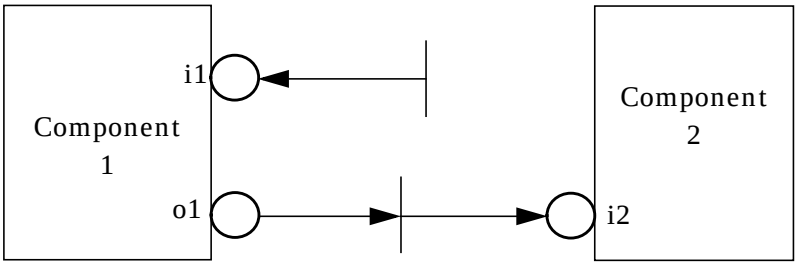


Figure 5.5: A few environments for the example in Figure 4.6

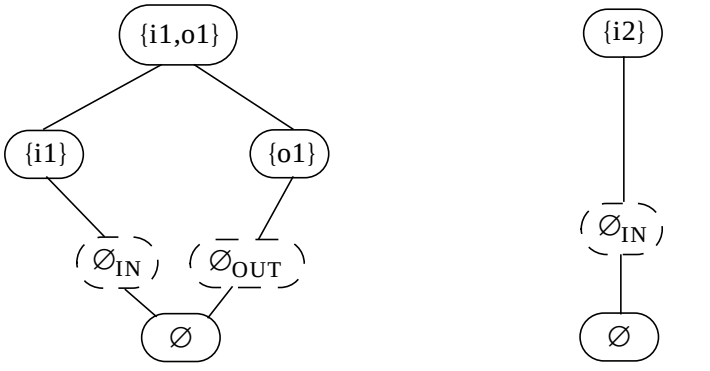
1. The union of two PRES+ nets can be reduced to the union of the places and transitions, respectively.

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 5.1 and Theorem 5.2, 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 5.6: Components and corresponding interfaces

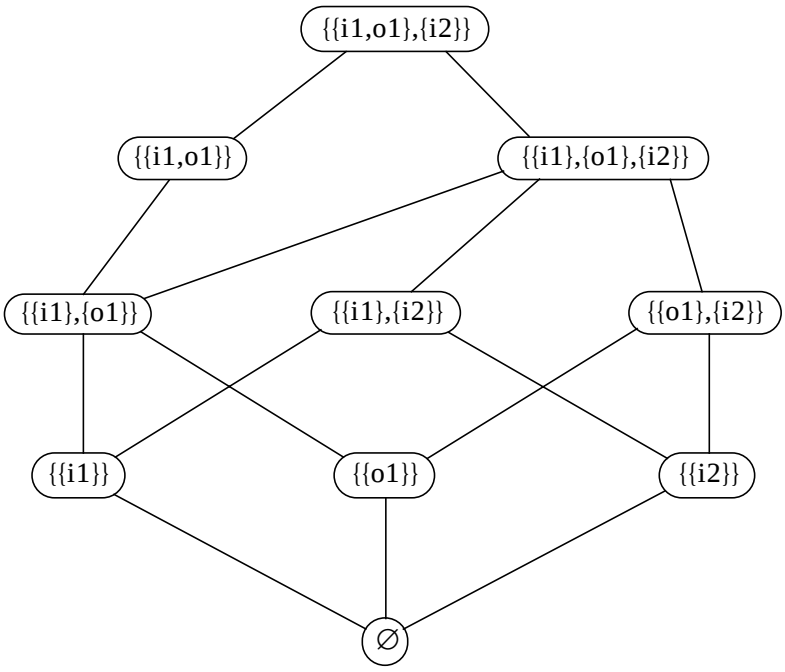


Figure 5.7: Partition (environment) lattice of the situation in Figure 5.6

done with individual stubs and their interfaces (Figure 5.3). Figure 5.6 introduces a very simple example consisting of two interconnected components. In Figure 5.6(b), we show the interface (stub) lattice corresponding to each of the components. Figure 5.7 depicts the corresponding partition (environment) lattice.

Definition 5.10: 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, $C \cap G \neq \emptyset$. $Sur(G) = \Gamma - (G \cup \bigcup_{C \in \{C' | (C' \text{ is a component in } \Gamma) \wedge C' \cap G \neq \emptyset\}} C)$.

■

Figure 4.3 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 4.2. 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 4.6 does not have any surrounding, $Sur(G') = \emptyset$.

5.4 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 verifier making this choice is presented through the following definitions and theorems.

Theorem 5.3: 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 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 5.5) filters out more elements from the unrestricted operation when $I_S = I_C = p_2$ than when $I_S = I_C = p_1$. Consequently p 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 5.11: Generalised operation. The generalised operation Op_C for component C is the union of all opera-

tions for every possible input observation, $Op_C = \bigcup_{in} Op_C(in)$.

■

According to Definition 5.4, 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 5.3 into the following corollary.

Corollary 5.1: 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 5.3 and Definition 5.11.

■

Definition 5.12: 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 $E = \{e | \neg\exists e' \in o : (r_{e'} < r_e)\}$, i.e. the set of events with the lowest timestamp in o. Then Definition 5.12 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 5.2 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 5.1 gives the solution.

$$\sigma(o, \varnothing) = [\varnothing, \{\langle p, \langle 2, 1\rangle\rangle\}, \{\langle p, \langle 2, 1\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle\}, \{\langle q, \langle 5, 3\rangle\rangle\}, \{\langle p, \langle 3, 8\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle\}, \varnothing] \quad (5.1)$$

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 ($s \models \phi$ means that formula ϕ holds in state s, and $\phi \Leftrightarrow \psi$ denotes equivalence between two formulas):

$$s \models \mathbf{AF}\ \phi \Leftrightarrow \forall \sigma \in P_M(s) \exists j \geq 0 : \sigma[j] \models \phi \quad (5.2)$$

$$s \models \mathbf{EG}\ \phi \Leftrightarrow \exists \sigma \in P_M(s) \forall j \geq 0 : \sigma[j] \models \phi \quad (5.3)$$

$P_M(s)$ 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 5.12, 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 of the world is based on the state sequence generator in Definition 5.12. Equation 5.4 and Equation 5.5, where IN is the set of all possible input observations of component C, express the same semantics as Equation 5.2 and Equation 5.3 in terms of observations and operations.

$$M_0 \models \mathbf{AF}\ \phi \Leftrightarrow \forall o \in Op_C \forall i \in IN \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \phi \quad (5.4)$$

$$M_0 \models \mathbf{EG}\ \phi \Leftrightarrow \exists o \in Op_C \exists i \in IN \forall j \geq 0 : \sigma(o \cup i, M_0)[j] \models \phi \quad (5.5)$$

The union is taken of both all possible input observations, $i \in IN$, and all possible output observations, $o \in Op_C$, and 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 5.2 and Equation 5.3.

In [Alu90] equivalent formulas to Equation 5.2 and Equation 5.3 are given for TCTL. Based on the discussion above, they can be trivially extended to formulas similar to Equation 5.4 and Equation 5.5.

Theorem 5.4: 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. $\mathbf{AF}\ \phi$, also expressed only on the ports in J. If $M_0 \models \mathbf{AF}\ \phi$ for component $Env(P_1, J)$, then it is also true that $M_0 \models \mathbf{AF}\ \phi$ for component $Env(P_2, J)$.

■

Proof:
$M_0 \models \mathbf{AF}\ \phi \Leftrightarrow \forall o \in Op_{Env(P_1, J)} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \phi$, where IN is the set of all input observations on ports in the partitions, according to Equation 5.4. As a consequence of Corollary 5.1 and the fact that o and i are universally quantified, it is possible to conclude $\forall o \in Op_{Env(P_2, J)} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \phi$.

■

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.

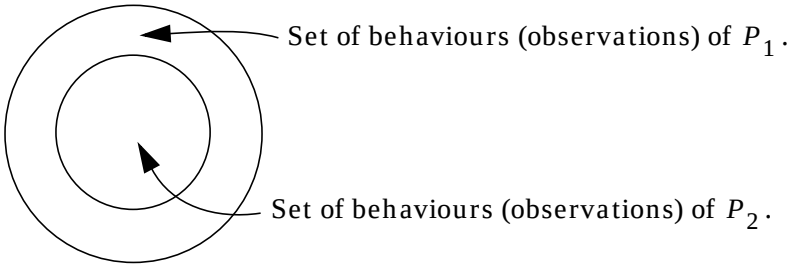


Figure 5.8: Illustration of Theorem 5.4

Figure 5.8 tries to illustrate the theorem. The area inside the outer circle denotes the set of behaviours (observations), i.e. the operation of $Env(P_1, J)$ ($Op_{Env(P_1, J)}$). According to Definition 5.9, 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 5.1 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 an arbitrary (T)CTL formula, becomes obvious. An arbitrary (T)CTL formula cannot guarantee that the property holds for all behaviours, only that there is at least one behaviour satisfying it.

5.4.1 DISCUSSION

Theorem 5.4 provides the answer to the questions identified at the end of Section 4.3.1. Let us assume that we have a set C of two or more components which have been interconnected by a glue logic. It has to be verified that a certain property, expressed as a (T)CTL formula ϕ, holds. 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 connected to the glue logic being verified.

2. If the verification is manageable, the following two situations can be indentified:
   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 5.4).

Case 2b above is important, as it offers a certain degree of liberty 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 5.5 illustrate this process.

5.5 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.

Table 5.1: 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

5.5.1 GENERAL AVIONICS PLATFORM

In the first set of experiments, we have verified the glue logic in Figure 4.3, which interconnects the Radar and Protocol component as part of the General avionics platform (Figure 4.1 and Figure 4.2) [Loc91]. We illustrate the verification of four properties. Property A is $\mathbf{AGAF}\neg update$ (the tokens in port "update" will always be consumed). Property D is $\mathbf{AGAF}\ \neg out$ (the tokens in port "out" will always be consumed). Properties B and C are identical to Equation 4.1 and Equation 4.2. Consequently, all formulas are ACTL. Three possible partitions were used whose relations are shown in the lattice in Figure 5.9. The results of the verification are shown in Table 5.1. The letters F and T in each cell of the table denote wether the property was satisfied (T) or not (F) with the corresponding environment. The numbers denote the verification time in seconds. It can be

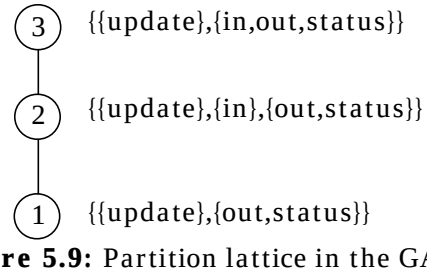


Figure 5.9: Partition lattice in the GAP example

observed that all four properties imposed by the interconnected components are satisfied with the actual glue logic. For property D, the verification can be done using the lowest level of the three interfaces (as the property is expressed by an ACTL formula, point 2b in Section 5.4.1 applies).

5.5.2 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 5.10. The I/O interface and memory controller handles the interaction of the processing element with the 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 STB consists in fact of two buses, the address bus and the data bus. When the protocol wants to send data, on request from the processing element, it must first request access to the address bus. After acknowledgement of the address bus, the protocol suggests an identifier for the message transfer and associates it with the address of the recipient. This identifier is broadcast to all protocol components 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 is now ready to service another request from the processing element. One functionality of the medium glue logic being verified 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. For instance, the

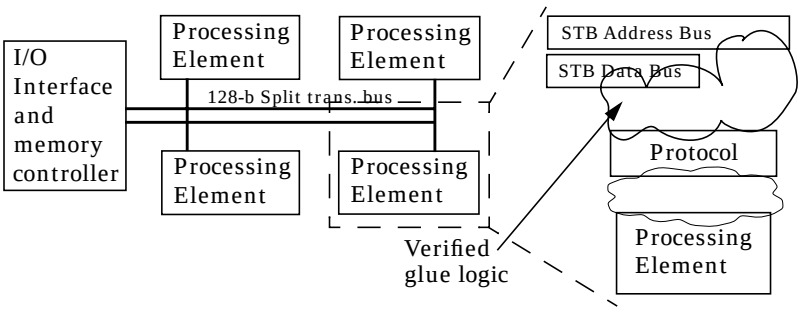


Figure 5.10: Schematic view of the STB example

protocol component expects two different commands from an identifier broadcast (described above) of the address bus, depending on wether 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 5.2 shows the verification results with the STB example. The high number of ports in the components yields a large lattice of environments. The one depicted in Figure 5.11 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 hence formulated as $\mathbf{AG}\ (rec \rightarrow rec \neq \langle \mathrm{TRAN}, a \rangle \wedge a \neq \mathrm{this\_component})$ where TRAN (transaction) is the command to be received by the protocol component when the source causing the event is the one 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, **AG** ((*addr.out* = ACK) → **AF** *addr.in* = 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

Table 5.2: 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

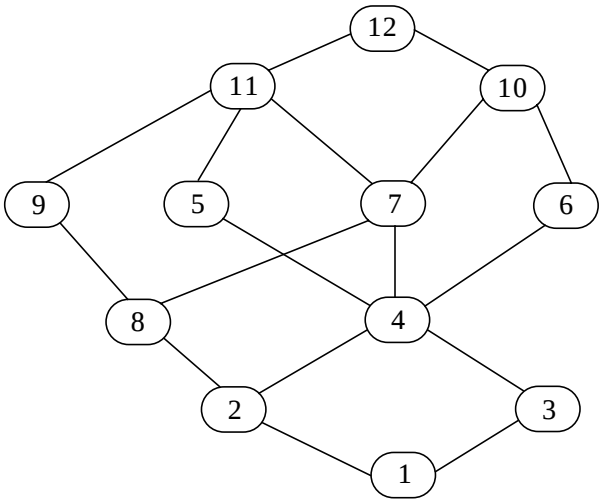


Figure 5.11: Partition lattice in the STB example

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 enironment 2. According to point 2b in Section 5.4.1, 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 limits 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, going upwards until the property is satisfied.

5.6 Verification Methodology Roadmap

This section will continue the roadmap of Section 4.3.2 based on the work presented in this chapter.

The answer to the question leading the verifyer to this part of the roadmap (see Figure 4.8) 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 5.12. 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 5.13. If the property is satisfied and it is ACTL, then it can be deduced according to Theorem 5.4 that the property really is satisfied. Otherwise if not ACTL, the property can only be proven satisfied to the extent given by the compo-

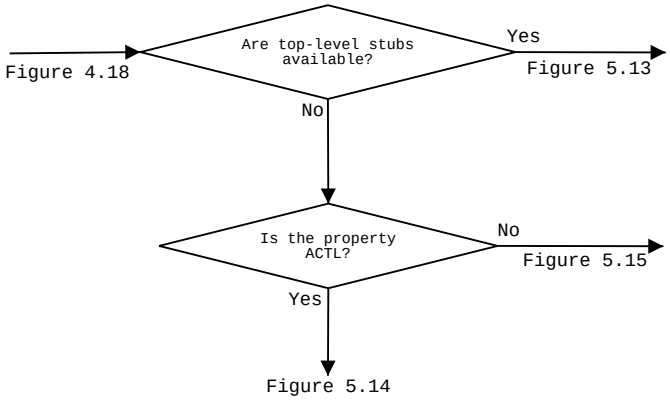


Figure 5.12: Continuation of the roadmap from Figure 4.8

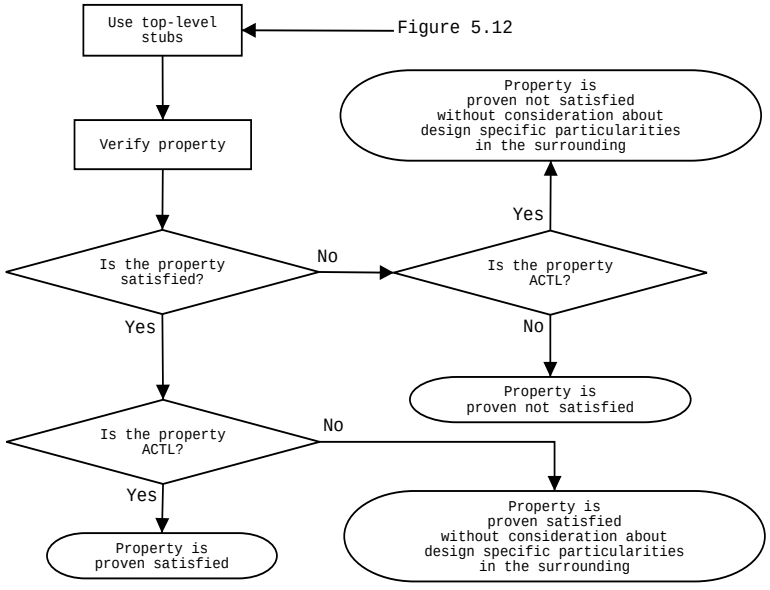


Figure 5.13: Roadmap when using top-level stubs, continuation from Figure 5.12

nents, i.e. a particular behaviour of the surrounding is not taken into consideration (we will further elaborate on the aspects

related to the surrounding in Chapter 7). 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 5.14 shows the procedure for ACTL formulas and Figure 5.15 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 verifyer 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 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 5.4 are not violated. The diagnostic

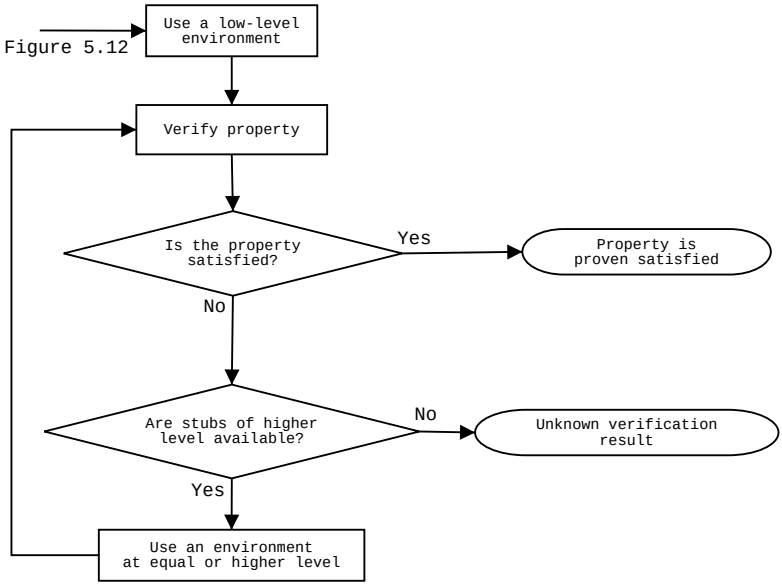


Figure 5.14: Roadmap when using lower-level stubs on ACTL formulas, continuation from Figure 5.12

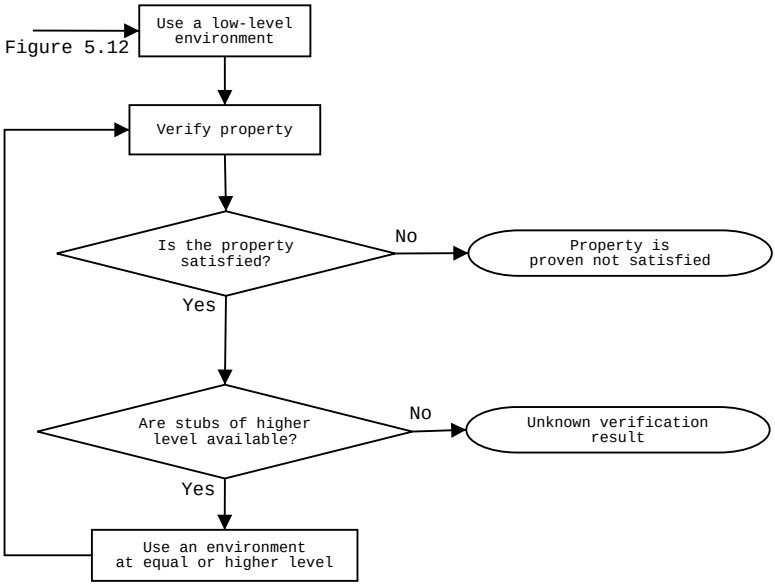


Figure 5.15: Roadmap when using lower-level stubs on non-ACTL formulas, continuation from Figure 5.12

trace resulting from the model checking is very useful for guidance.

Chapter 6 Automatic Generation of Stubs

CHAPTER 5 INTRODUCED A verification methodology where the stubs are provided by the designer of the reusable components. 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 7. Experimental results are also presented.

The example component in Figure 6.1 will be used to explain and analyse the stub generation algorithms in this chapter. In

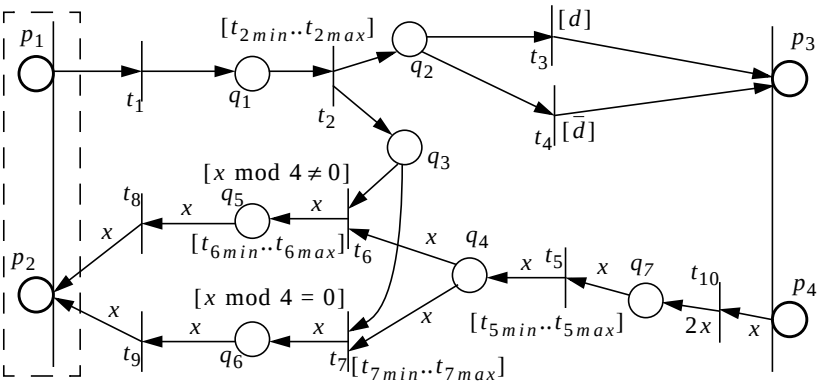


Figure 6.1: Example of a component for stub generation

all cases, a stub for the marked interface $\{p_1, p_2\}$ will be generated.

6.1 Pessimistic Stubs

The stub definition presented in Section 5.1 (Definition 5.5) is quite strict, requiring equality between the operations of the component and stub. That strictness makes it very difficult to automatically create stubs.

Definition 6.1: 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 events. 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 5.5 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 a verification with pessimistic stubs.

Theorem 6.1: 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. **AF** ϕ expressed only on the ports of the stubs in E_1 and E_2. If $M_0 \models$ **AF** ϕ for component E_2, then it is also true that $M_0 \models$ **AF** ϕ for component E_1.

■

Proof:
$M_0 \models$ **AF** $\phi \Leftrightarrow \forall o \in Op_{E_2} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \phi$,
where IN is the set of all input observations on ports in the partitions, according to Equation 5.4. As a consequence of the fact that o and i are universally quantified, it is straight-forward to conclude that $\forall o \in Op_{E_1} \forall i \in IN\ \exists j \geq 0 : \sigma(o \cup i, M_0)[j] \models \phi$.

■

The intuition behind this theorem is the same as the intuition behind Theorem 5.4 described on page 68. The set of behaviours of E_2 includes all the behaviours of E_1 according to the assumption. Hence, if a certain property is true for all behaviours of E_2, it must also be true for all behaviours of E_1.

Theorem 6.1 allows us to use pessimistic stubs when verifying (T)ACTL formulas. The behaviours of the exact stub (see Definition 5.5) are also produced by the pessimistic one which, however, produces additional behaviours. This fulfills the assumptions of the theorem. So, if a property is satisfied using the 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.

6.2 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 6.1, 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 5.5). The resulting stub is shown in Figure 6.2.

Figure 6.3 illustrates the difference between an exact stub and a naïve stub further. Figure 6.3(a) shows the model of a simple component. It is designed assuming input on ports p_3 and

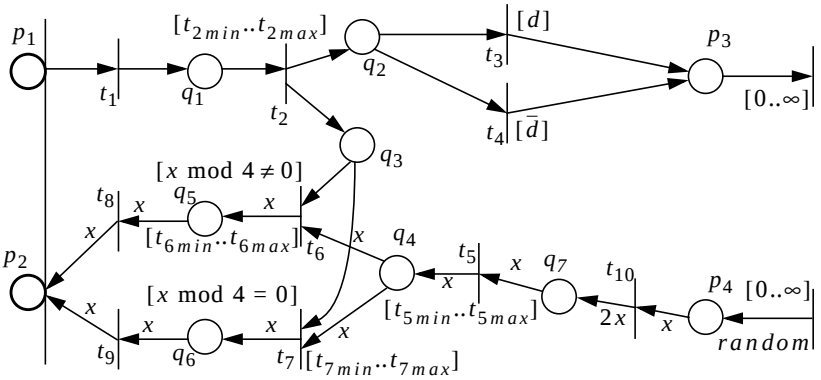
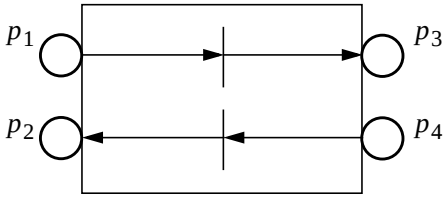
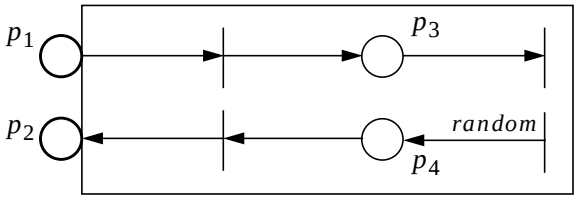


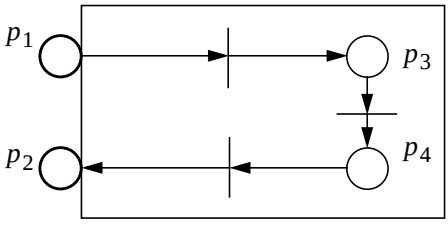
Figure 6.2: A naïve stub of the component in Figure 6.1



(a) The component



(b) The naïve stub



(c) The exact stub

Figure 6.3: Comparison between exact and naïve stubs

p_4 satisfying the formulas in Equation 6.1 (if there is a token in p_3, then there must arrive a token in p_4 in the future) and Equation 6.2 (no token may arrive in p_4 unless there was first a token in p_3).

$$\mathbf{AG}\ (p_3 \rightarrow \mathbf{AF}\ p_4) \tag{6.1}$$

$$\mathbf{AG}\ ((p_4 \vee \text{init}) \rightarrow \mathbf{A}[\mathbf{A}[p_3\ \mathbf{R}\ \neg p_4]\ \mathbf{R}\ p_4]) \tag{6.2}$$

In Figure 6.3(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 6.3(c). This stub satisfies the assumptions.

To verify a design using naïve stubs is tremendously time consuming (see experimental results in Section 6.5). For this reason, an algorithm generating smaller stubs reducing verification time has been developed and is presented in the following sections.

6.3 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 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.

Hence, 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

6.3.1 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 6.4. These procedures are called once for each port in the interface of the stub. traceBack is called for out-ports and traceForward in

```
1  procedure traceBack(e: place or transition, p: port)
2      if not visited[e] then
3          visited[e] := true;
4          for each d ∈ °e do
5              DF[d, p] := DF[d, p] ∪ { 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 ∈ e° do
12             DF[d, p] := DF[d, p] ∪ { e };
13             traceForward(d, p);
```

Figure 6.4: Algorithms for searching the dataflow

the case of in-ports. visited is initially false for all places and transitions. During the search through the graph, each node (place or transition) is marked with the node 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. The dataflow marking must consequently not only be able to distinguish the paths but also to which ports they lead. 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 6.4 and the outlined data structure implement Definition 6.2.

Definition 6.2: Dataflow marking. A dataflow marking $df_{if}^{C}(n, p)$ is a set of nodes (places or transitions), which constitute the first step on a path from node n to port $p \in if$ in component C with an interface $if \subseteq C$. If C or if are evident from the context they may be omitted from the notation. As an extension we also define $df(n) = \bigcup_{p \in if} df_{if}(n, p)$. ■

Figure 6.5 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 both starting from p_1 (traceForward) and p_2 (traceBackward). This means that both ports can be reached from q_3. As indicated by the figure, the path from q_3 to p_1 goes through t_2, and the path 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 6.3: Divergence node. A node n is a divergence node iff $|df(n)| > 1$, i.e. there are several different paths leading to ports in the interface, or the arrows of n point in different directions.

■

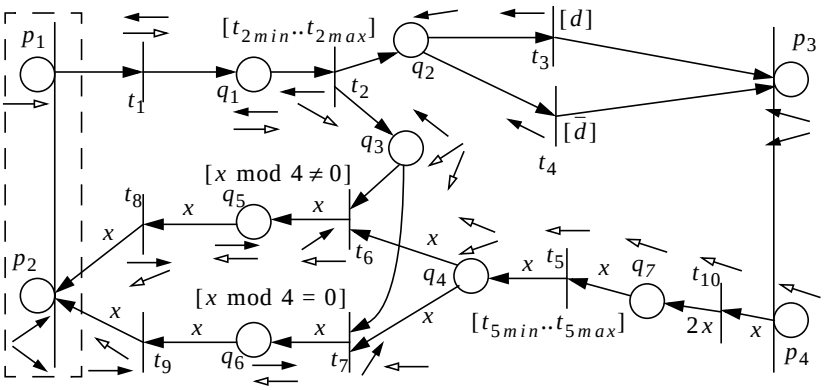


Figure 6.5: The dataflow marking of the component in Figure 6.1

Definition 6.4: Intersect node. A node n is an intersect node iff $\exists p \in if : df_{if}(n, p) \neq \emptyset \wedge df_{if}(n) \neq df_{if}(n, p)$, i.e. at least two arrows pointing in different directions are of different type (solid or hollow).

■

In Figure 6.5, 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} and q_7 are nodes which are neither divergence nor intersect nodes.

6.3.2 IDENTIFICATION OF STUB NODES

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

Definition 6.5: 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 6.5).
2. The node is a port in the interface (p_1 and p_2 in Figure 6.5).

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 6.6 (traceNode) presents the algorithm.

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

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 the node *e* being visisted 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.

```
1  procedure traceNode(e: place or transition)
2      if not tr_visited[e] then
3          tr_visited[e] := true;
4          if e is a port in the specified interf. or e is an intersect node then
5              constructStub(e);
6          else if e is a divergence node then
7              tr_visited[e] := false;
8              node spcand := traceCutEdge(e);
9              if spcand = NULL then
10                 constructStub(e);
11             else
12                 traceNode(spcand);
13         else
14             traceNode(the only element in DF[e]);
15
16 function traceCutedge(e: place or transition) returns place or transition
17     if not tr_visited[e] and e is not an intersect node then
18         tr_visited[e] := true;
19         for each d∈ DF[e] do
20             if <e,d> is a cutedge then
21                 return d;
22             else
23                 node cecand := traceCutedge(d);
24                 if cecand ≠ NULL then
25                     return cecand;
26     return NULL;
```

Figure 6.6: Algorithms for identifying which parts of a component to include in the stub

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

If the 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 when the 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 must 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 the node e being visited is an intersect node, the search must stop 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 no cutedge was found the search continues until one is found (Line 23). In case a cutedge was not found, 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,

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

this procedure is also a depth first search. Figure 6.7 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 traceBackward (Line 5).

Continuing the example in Figure 6.5, 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 6.6) 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 traceNode 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 6.8 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. Consequently, a search for cutedge is started (Line 8 in Figure 6.6) while keeping q_4 as a

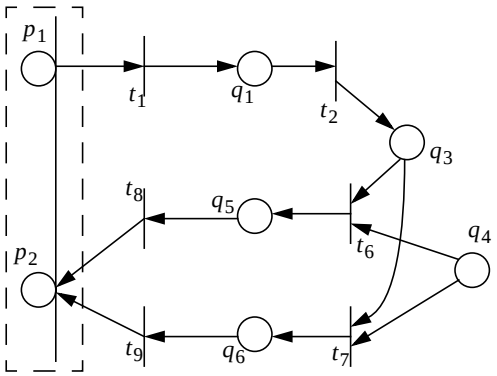


Figure 6.8: The places and transitions in the automatically generated stub

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 6.8.

6.3.3 COMPENSATION

All places and transitions on a path between two ports are included in the resulting stub as a result of 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 (receive) all needed output (input). These nodes are called *fork* (*join*) nodes and need additional treatment.

Definition 6.6: Fork node. Assume a component C and a stub $S \subseteq C$. A node $n \in S$ is a fork node iff the corresponding node $n \in C$ has a node in its postset which is not in the stub, $\exists n' \in n^{\circ} : n' \notin S$.

■

Definition 6.7: Join node. Assume a component C and a stub $S \subseteq C$. A node $n \in S$ is a join node iff the corresponding node $n \in C$ has a node in its preset which is not in the stub $\exists n' \in {}^\circ n : n' \notin S$.

■

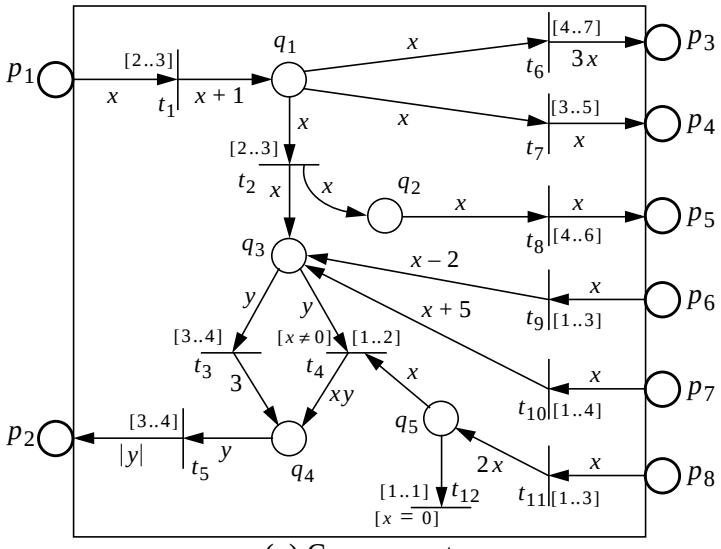
Figure 6.9(a) introduces an example of a component which will be used to explain how fork and join nodes are modified in the stub. Figure 6.9(b) shows the stub as generated by the algorithms in Figure 6.6, whereas Figure 6.9(c) presents the resulting stub after the excluded parts have been compensated for.

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. On the other hand, the functions may have arguments, i.e. $f_t(y) = \{xy | x \in \mathbf{Z}\}$, where attention has to be paid to the input token values.

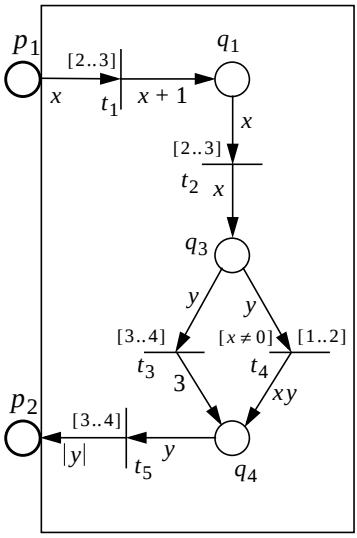
Such transition functions are created with respect to a certain universe **U** containing all values possible in the design. In this chapter, it is assumed that the universe consists of all integers, $\mathbf{U} = \mathbf{Z}$.

Case 1: fork place

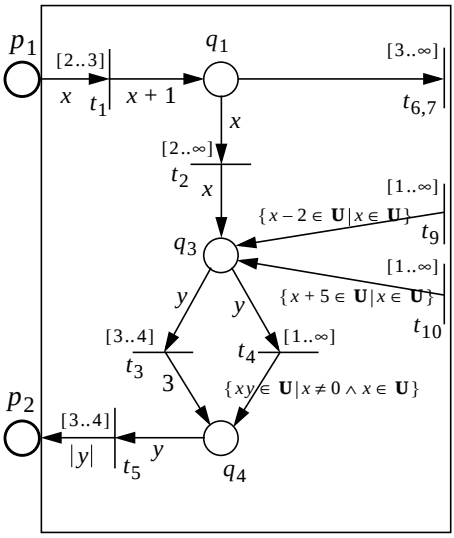
If the fork node is a place (i.e. q_1 in Figure 6.9(a)), it means that tokens can disappear out of the stub, into the part of the net which is excluded. To model this, a new transition ($t_{6,7}$) is added to consume these tokens. The time interval of this transition is from the minimum delay of all successor transitions not included in the stub, to infinity ($[min(3, 4)..\infty]$).



(a) Component



(b) Stub as generated by the algorithms in Figure 6.6



(c) Stub

Figure 6.9: Example component and stub explaining the compensation of excluded parts

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 one of which it replaces except that all arguments contain random values conformant to a possible transition guard. Equation 6.3 expresses this formally.

$$f_t' = \{f_t(x_1, \ldots, x_n) \in \mathbf{U} \mid g_t(x_1, \ldots, x_n) \wedge x_1, \ldots, x_n \in \mathbf{U}\} \qquad (6.3)$$

The guard is set to $f_t' \neq \varnothing$.

The guards are not shown in Figure 6.9(c) for space reasons. Moreover, the guards are redundant in this case since they are always true, $f_{t_9}' = f_{t_{10}}' = \mathbf{Z} \neq \varnothing$.

$$f_{t_9}' = \{x - 2 \in \mathbf{Z} \mid x \in \mathbf{Z}\} = \mathbf{Z} \qquad (6.4)$$

$$f_{t_{10}}' = \{x + 5 \in \mathbf{Z} \mid x \in \mathbf{Z}\} = \mathbf{Z} \qquad (6.5)$$

Case 3: join transition

If the join node is a transition (t_4), it means that it is not always the case that the transition will be enabled. Consequently the maximum time delay is changed to infinity. Moreover, some parameters for the transition function lack a value.

The new 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 places inside the stub. Equation 6.6 expresses this formally. x_i are parameters coming from places outside the stub, and y_i from places inside the stub.

$$f_t'(y_1, \ldots, y_m) = \{f_t(x_1, \ldots, x_n, y_1, \ldots, y_m) \in \mathbf{U} \mid g_t(x_1, \ldots, x_n, y_1, \ldots, y_m) \wedge x_1, \ldots, x_n \in \mathbf{U}\} \quad (6.6)$$

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

The guard is not shown in Figure 6.9(c) for space reasons. Similar to the join place case, the guard is not necessary in the example. Equation 6.7 and Equation 6.8 explain why.

$$f_{t_4}'(y) = \{xy \in \mathbf{Z} \mid x \neq 0 \wedge x \in \mathbf{Z}\} \quad (6.7)$$

$$\forall y \in \mathbf{U}: f_{t_4}'(y) \neq \emptyset \quad (6.8)$$

Case 4: fork transition

If the fork node is a transition (t_2), it means that the excluded net can disable it by not consuming the token in one of the output places. 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 6.10 shows the final result of applying this algorithm to the example in Figure 6.1 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 6.10. 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 6.1), which provides input for t_5, only can produce even numbers. Consequently, the stub is more pessimistic about possible values than an exact or naïve stub. On the other hand, it would not be feasible, in the general case, to incorporate the whole excluded part of the net into the function by composing functions. That would be the same as keeping the original net which would give a result similar to the naïve stub.

In particular cases, the algorithm may result in an empty model. This occurs when the datapath 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.

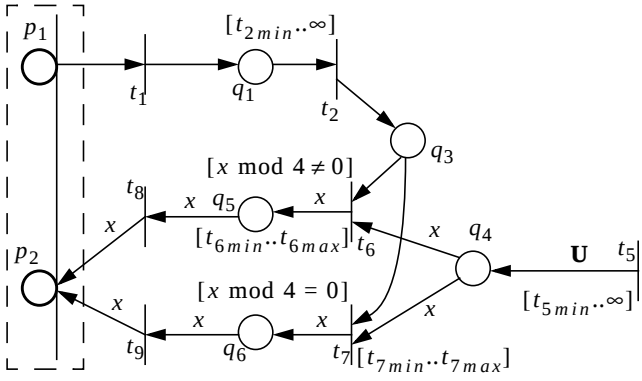


Figure 6.10: An automatically generated stub

6.3.4 COMPLEXITY ANALYSIS

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

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 $O(n + e(n + e)) = O(n + en + e^2)$. Assuming that there are more edges than nodes, the theoretical worst case complexity of the algorithm is quadratic in the number of arcs, $O(e^2)$. However, it should be noted that, in practice, very few edges are checked for being cutedges. Consequently, running time is practically close to linear.

6.4 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. pessimism must be reduced. The solution to this problem is to add some parts 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 by hand.

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. The inverted function can in itself be a (set of) function, which is the most general approach, or a stored table.

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 5.2). 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.

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 removed by the stub generation algorithm, until the unwanted value is eliminated. When adding a previously removed place, all transitions 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. 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 6.10. 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 6.11. The result of verifying the property **AG** $(r_1 \rightarrow even(r_1))$ (All tokens arriving in r_1 must have an 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, $t_5(3)$, $t_6(3)$, $t_9(3)$, $s_2(3)$. Figure 6.12 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 96. The first join transition encountered in the example sequence is t_5, which produced value $u = 3$ (Line 8).

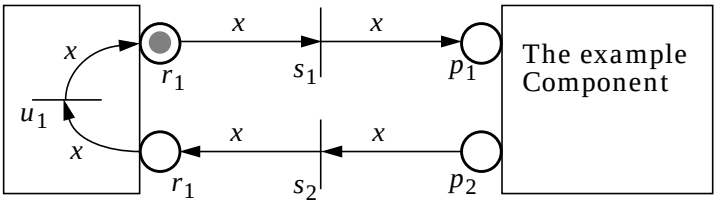


Figure 6.11: An example system

```
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 6.13
11         if not success then
12             newStub := oldStub;
13         else
14             oldStub := newStub;
15     until tr is finished;
16     return newStub;
```

Figure 6.12: The pessimism reduction algorithm

The part of the component not included in the stub is then examined backwards starting from the join transition (t_5) towards the ports (p_4), exploring the part of the component not included in the stub (Line 10 and Figure 6.13). 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 6.13). 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 6.9 (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).

```
1  function buildStub(stub: PRES+; t: transition; u: value) returns boolean
2      if not visited[t] then
3          visited[t] := true;
4          stub := stub ∪ { t };
5          if f_t is constant then
6              return f_t ≠ u;
7          else
8              W := f_t^-1(u);
9              for each w∈W do
10                 if g_t(w) then
11                     for each parameter w_i of f_t do
12                         p_i := the place corresponding to w_i;
13                         if p_i has an initial token with value w_i then
14                             return false;
15                         stub := stub ∪ { p_i };
16                         if °p_i = ∅ then
17                             return false;
18                         else
19                             for each t_i ∈ °p_i do
20                                 success := buildStub(stub, t_i, w_i);
21                                 if not success then
22                                     return false;
23     return true;
```

Figure 6.13: Auxiliary function for the pessimism reduction algorithm

$$f_t^{-1}(x) = \{\langle x_1, \ldots, x_n\rangle \mid f_t(x_1, \ldots, x_n) = x \wedge x_1, \ldots, x_n \in \mathbf{U}\} \quad (6.9)$$

Each function argument may consist of several parameters, for instance transition t_4 in Figure 6.9(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 6.12. 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 6.1 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 corresponding 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. $f_{t_{10}}^{-1}(3) = \emptyset$, 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 returning the stub in Figure 6.14.

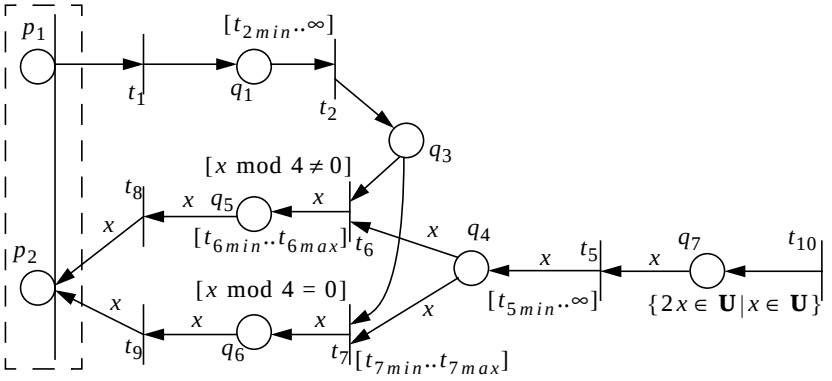


Figure 6.14: The resulting stub after pessimism reduction

t_{10} only produces even values, so t_5 also only produces even values, which in turn causes the property to be satisfied.

6.4.1 COMPLEXITY ANALYSIS

buildStub is a depth first search with complexity $O(n + e)$, where n is the number of nodes and e is the number of edges in the graph. The question mark 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 $O(n \cdot inv + e)$.

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

6.5 Experimental Results

The proposed methodology is demonstrated on two examples: the General Avionics Platform (GAP), introduced in Section 4.1, and a cruise controller.

6.5.1 GENERAL AVIONICS PLATFORM

The two components in the GAP example which were modelled and whose interconnection were especially verified were Tracker and Weapon (Figure 6.15). Tracker receives information from component Radar regarding the location of enemy airplanes. The pilot may point at a particular airplane on his screen and lock the weapons on it. Upon lock, Tracker repeatedly sends information to Weapon about the direction and distance of the target airplane as long as the lock situation holds. Weapon continuously informs Tracker that it keeps up with the aiming instructions given by Tracker.

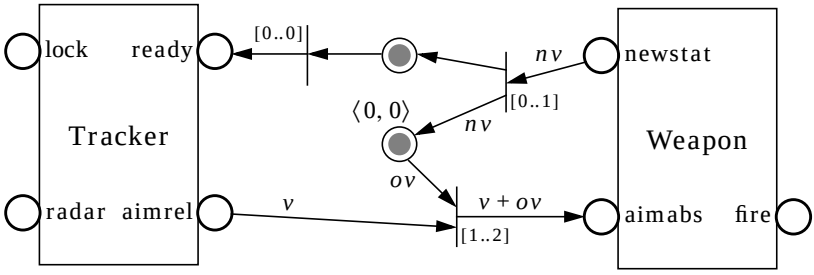


Figure 6.15: The verified glue logic in the GAP example

Three properties were checked in this setting:

1. Weapon must keep up with the aiming instructions given by Tracker.
   AG $(aimrel \rightarrow \mathbf{AF}\ ready)$
2. Tracker must be able to send the aiming instructions at a certain rate.
   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.
   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 6.1 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 6.3 (Env 0). In the case when the property was unsatisfied, the diagnostic trace was investigated and the proper stub had its pessi-

Table 6.1: Verification times for the GAP example

Prop.	Env 0	Pess. Red.	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

mism 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 6.1.

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.

6.5.2 CRUISE CONTROLLER

The second set of experiments was done on a model of a car cruise controller (Figure 6.16). When the cruise controller is activated by the driver of the car, a signal is sent to the cruise

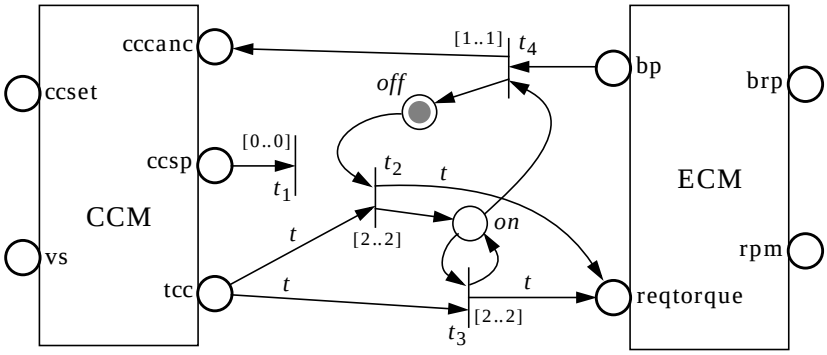


Figure 6.16: The verified glue logic in the cruise controller example

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 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.
   AG $(bp \rightarrow \mathbf{AF}_{<1} \neg bp)$
2. The requested torque is below 100%.
   AG $(reqtorque \rightarrow reqtorque \leq 1)$
3. The reference value is positive.
   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 6.2 presents the verification results and times in seconds in the same style as the results given in Table 6.1. The

Table 6.2: Verification times for the CCM example

Prop	Env 0	Pess. Red.	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

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 6.16, 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.

6.6 Verification Methodology Roadmap

This section continues the verification roadmap introduced in Section 4.3.2 based on the work presented in this chapter.

The question answered in Figure 4.8 gives us the assumption that we must ourselves generate the stubs used in the verifica-

tion. As indicated by the next question (Figure 6.17) 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 6.3. In the next step, the property is verified using the generated stub. If the property was satisfied and ACTL, it is proven that the system satisfies it. If the property was satisfied, but not

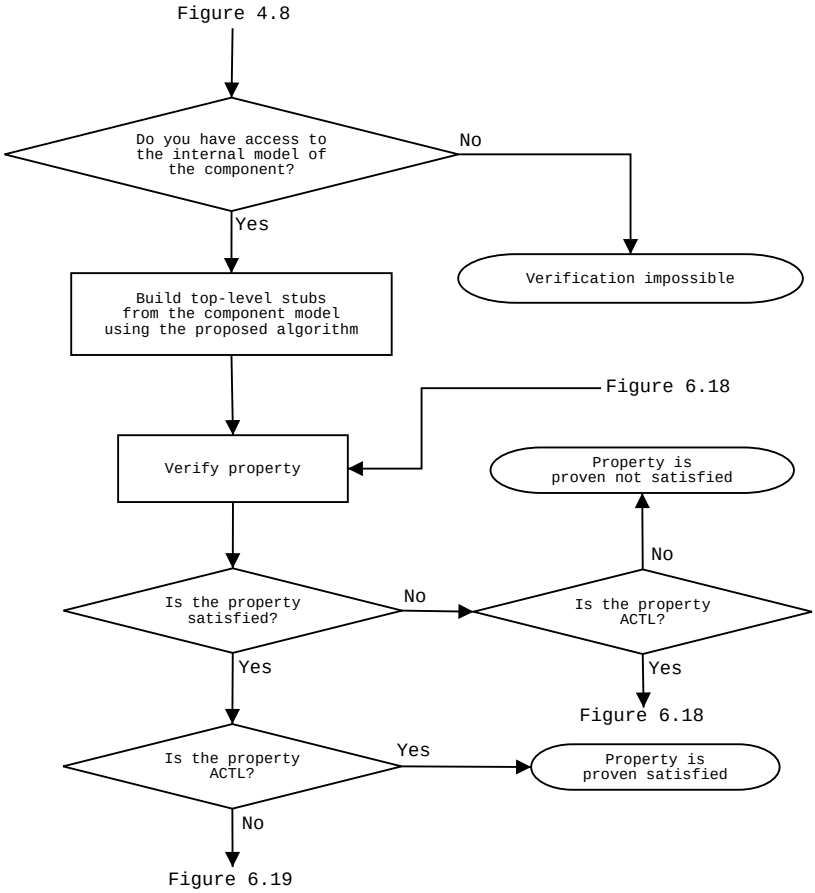


Figure 6.17: Continuation from Figure 4.8 when no stubs are provided by the designer

ACTL, naïve stubs might need to be used (Figure 6.19). Otherwise, if the property was not satisfied and not ACTL, it is proven unsatisfied in the system. However, if the property is ACTL, pessimism has to be reduced in the stub (Figure 6.18).

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

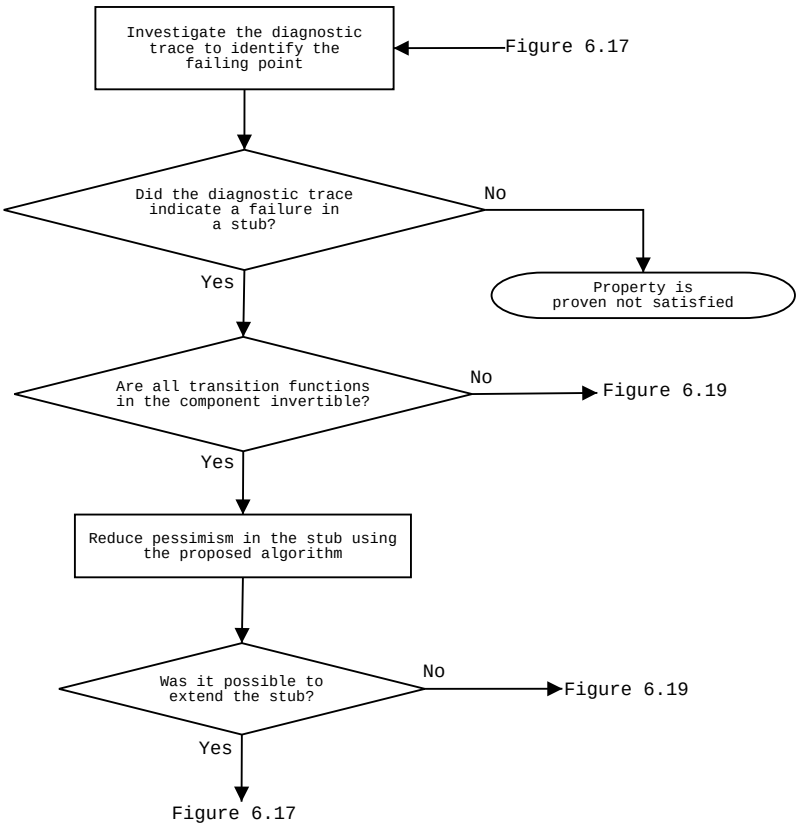


Figure 6.18: Continuation of the roadmap from Figure 6.17

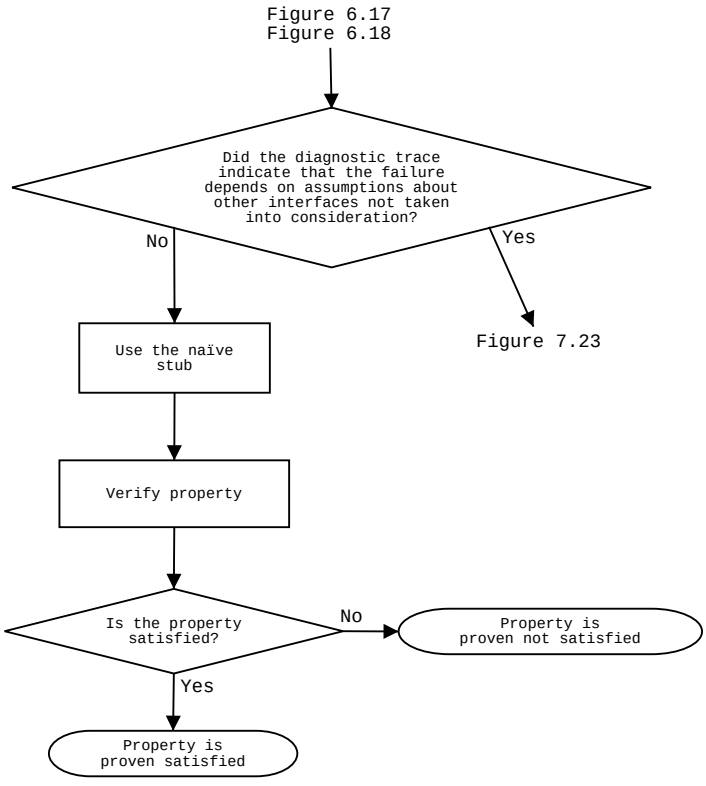


Figure 6.19: Continuation of the roadmap from Figure 6.17 and Figure 6.18

the algorithm presented in Section 6.4. If all functions are not 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 6.19). 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 6.3 about the difference

between naïve and exact stubs). The solution to that problem is presented in Chapter 7. Otherwise, the naïve stub is used straight-forwardly.

Chapter 7
Inclusion of the Surrounding into the Verification Process

TOGETHER WITH EACH COMPONENT, a set of (T)CTL formulas are provided as requirements on the input on all interfaces of the component. However, stubs generated by the algorithms presented in Chapter 6 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 connected to the glue logic are incorporated into the stubs themselves. 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 7.1 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

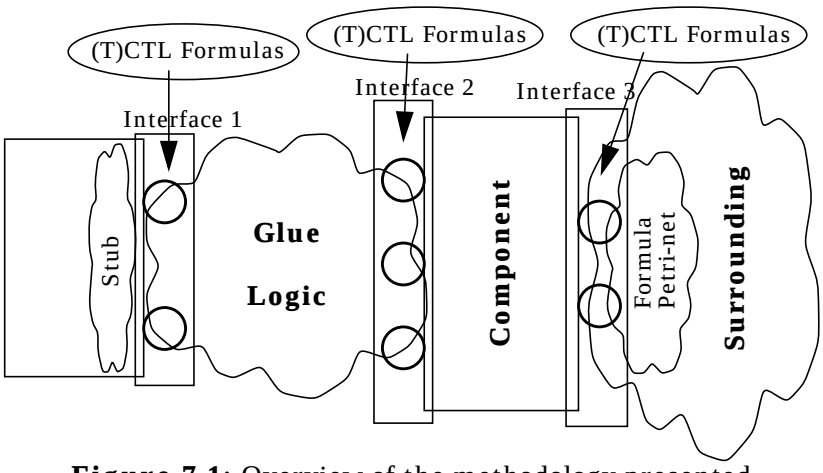


Figure 7.1: Overview of the methodology presented in this chapter

consistent with the formula. The resulting Petri-net is then attached to the component on the interface on which the formula was expressed. The component with the attached formula Petri-net (Figure 7.1) 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]. Our work is based on this translation method. In fact, many definitions presented in this chapter, in particular in Section 7.2.1, 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 better (see Section 3.2).

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 correspondance 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.

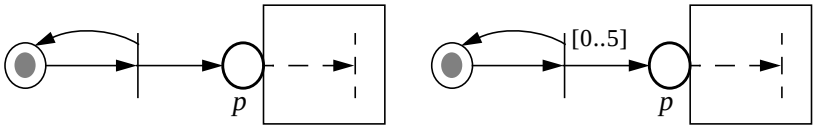
7.1 Preliminaries

7.1.1 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 7.2(a) shows an ad hoc construction of a PRES+ model simulating this formula. It should be noted that all generated models are connected to a component. Therefore it is possible that tokens disappear or appear in the ports corresponding to the atomic propositions in the formulas, without an explicit transition in the generated PRES+ model.

Unfortunately, the model in Figure 7.2(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 solve 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 transferred to the corresponding transition. The model now has a mechanism to force the transition to fire and p will be marked



(a) (b)

Figure 7.2: Petri-nets constructed ad hoc for the formula **AGAF** p

in the future. From here on, in the context of translating ACTL formulas to Petri-nets, it is assumed that such time bounds on **F** and **U** operators exist. A Petri-net model for this formula is shown in Figure 7.2(b).

7.1.2 FORMULA NORMALISATION

In order to simplify the algorithm, the formula for which a Petri-net 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. **AG** φ is rewritten as **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 7.1 shows a few examples of ACTL formulas and their normalisations. *true* and *false* are abbreviated as t and f respectively.

Table 7.1: Examples of (T)ACTL formulas and their normalisations

Formula	Normalisation
AG $(p \rightarrow p \leq 5)$	$\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \leq 5)]$
$\mathbf{AF}_{\leq 4} p$	$\mathbf{A}[t\ \mathbf{U}_{\leq 4} p]$
$\mathbf{AGAF}_{\leq 4} p$	$\mathbf{A}[f\ \mathbf{R}\ \mathbf{A}[t\ \mathbf{U}_{\leq 4} p]]$
AG $(p > 10 \rightarrow \mathbf{AF}_{\leq 2} q \leq 5)$	$\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \leq 10 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 2} q \leq 5])]$

7.2 The 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 executed in sequence. Section 7.2.5 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 satisfying the particular ACTL formula. Each state represents a particular behaviour of the net. The Petri-net changes state as a response to inputs received and outputs emitted, in a way that the formula will stay satisfied. The resulting Petri-net will have one place for each such state.

If the state represents a behaviour which includes that a certain U formula must hold, it is necessary to insert timers into that state in order to guarantee that the specified event will occur before the upper time bound of the U formula expires.

Transitions are 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 $\mathbf{AG}(p \rightarrow \mathbf{AF}_{\leq 3} q < 10)$ is used as an example for explaining the algorithm presented in this chapter. This formula is rewritten as $\psi = \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])]$ according to the normalisation rules. Section 7.3 provides further examples.

7.2.1 PLACE GENERATION

The first step of the algorithm is to create places to the Petri-net. Before describing the algorithm, a few definitions and concepts must be presented.

Definition 7.1: 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 $el(\varphi) = \emptyset$. 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 7.1.

$$el(\psi) = \Big\{ \underbrace{\mathbf{AX}\,\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])]}_{1}\ ,\ \underbrace{p}_{2}\ , \tag{7.1}$$
$$\underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]}_{4}\ ,\ \underbrace{q}_{8}\ ,\ \underbrace{q < 10}_{16}\ \Big\}$$

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, we will very often refer to large sets of subsets of $el(\varphi)$. In order to achieve an acceptably condence 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 7.1. i is the sum of the numbers corresponding to the formulas included in the desired set. Table 7.2 lists all subsets of $el(\psi)$ with their associated numbers.

Definition 7.2: 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 7.2 presents the set of all subformulas of the example formula ψ.

$$sub(\psi) = \{\mathbf{A}[f \mathbf{R} (\neg p \vee \mathbf{A}[t \mathbf{U}_{\leq 3} q < 10])], f, \neg p \vee \mathbf{A}[t \mathbf{U}_{\leq 3} q < 10], \neg p, p, \mathbf{A}[t \mathbf{U}_{\leq 3} q < 10], t, q < 10\} \quad (7.2)$$

Definition 7.3: 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 convenient 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.

■

Table 7.2: 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	$el(\psi)$

In the example formula, $AP(\psi) = \{p, q, q < 10\}$. It is further assumed in this example that p is an out-port and q is an in-port of a component, $AP_{out}(\psi) = \{p\}$ and $AP_{in}(\psi) = \{q\}$. $AP_{rel}(\psi) = AP_{rout}(\psi) = \{q < 10\}$ and $AP_{rin}(\psi) = \varnothing$.

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 6.3.3). **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 7.4: 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. It is defined 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 satifies all 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 consequently 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, $PV_{in}(S_9) = \{\ldots, -1, 0, 1, 2, \ldots\} = \mathbf{Z}$ and $PV_{in}(S_{25}) = \{k \in \mathbf{Z} | k < 10\} = \{\ldots, -1, 0, 1, 2, \ldots, 9\}$.
$PV_{out}(S_{27}, p) = \mathbf{Z}$ 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 Petri-net is to be constructed. Let $S(\psi)$ be defined as indicated by Equation 7.3, 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 which 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 separately, in other words $\forall p \in AP_{out}(\psi): PV_{out}(s) = \varnothing$. (E.g. $\{q, q < 10, q > 20\}$)

$$S(\psi) = 2^{el(\psi)} - \{s \in 2^{el(\psi)} | \exists p\Re v \in s: p \notin s\} - \{s \in 2^{el(\psi)} | PV_{in}(s) = \varnothing \wedge \forall p \in AP_{out}(\psi): PV_{out}(s) = \varnothing\} \quad (7.3)$$

Definition 7.5: Legal (Contradictory) set of elementary formulas. A set of elementary formulas, s, is legal iff $s \in S(\psi)$, and s is contradictory iff $s \notin S(\psi)$, where $S(\psi)$ is defined as in Equation 7.3.

■

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.

Continuing the example, $S(\psi) = \{S_0..S_{15}, S_{24}..S_{31}\}$. 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 Petri-net should be generated. Definition 7.6 introduces a function, Φ, for this purpose.

Definition 7.6: $\Phi(\varphi)$. Formula mapping $\Phi(\varphi)$ from $el(\psi)\cup sub(\psi)\cup\{true, false\}$ to $S(\psi)$ is defined recursively as follows [Gru94]:

1. $\Phi(true) = S(\psi)$, $\Phi(false) = \varnothing$. 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 7.7: 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 Petri-net. Each set of possible progress formulas $pf \in PF(\Phi(\psi))$ will have a corresponding place. In this way, it is possible to express all states.

In our example introduced on page 119, the resulting Petri-net will have two places corresponding to the two sets in $PF(\Phi(\psi)) = \{S_1, S_5\}$. 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 7.14).

Let us assume that q is an in-port of a component, $q \in AP_{in}(el(\psi))$. Assume further that the Petri-net is in state S_{27}, i.e. a token in the place S_1 and other tokens in p and q (with a value less than 10). This state satisfies the formula ψ, $S_{27} \in \Phi(\psi)$. Since the Petri-net is connected to a component at q, a transition in that component may consume that token which forces the Petri-net to change state to S_3. However, state S_3 does not satisfy formula ψ, $S_3 \notin \Phi(\psi)$, so the possibility of ending up in this state must be eliminated. A similar example can be shown for state S_{13}, which enters state S_5 when the token in q is consumed by the connected component. The difference is that S_5 does satisfy the formula, $S_5 \in \Phi(\psi)$.

The previous discussion has concluded that state S_{27} may lead to a state which does not satisfy the formula. Consequently, that state must be removed from the set of valid states.

Definition 7.8: Valid elementary set. A set of elementary formulas, s, is a valid elementary set in $\Phi(\psi)$, if for all $p \in AP_{in}(s)$, $s - \{p\} - \{p \Re v \in s\} \in \Phi(\psi)$ and $s - \{p\} - \{p \Re v \in s\}$ is recursively a valid elementary set. If $AP_{in}(s) = \emptyset$, then s is always a valid elementary set. Let $VES(\psi) = \{s' \in \Phi(\psi) | s' \text{ is a valid elementary set}\}$.

■

$VES(\psi)$ only contains the legal sets of elementary formulas s satisfying ψ, $s \in \Phi(\psi)$, which represent states which will still satisfy ψ even if one or more tokens are removed from the marked in-ports. From this point on, only valid elementary sets are considered, $s \in VES(\psi)$.

In our example, $VES(\psi) = \{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$, since for the set $S_{27} \in \Phi(\psi)$, $S_{27} - \{q\} - \{q < 10\} = S_3 \notin \Phi(\psi)$. Hence this set is not a valid elementary set. On the other hand, $S_{15} - \{q\} = S_7 \in \Phi(\psi)$, and S_7 does not have any in-port atomic propositions. The recursion ends and it is concluded that S_{15} is a valid elementary set.

Figure 7.3 shows the algorithm for creating the places in the resulting PRES+ model. The variable net is a global variable of type PRES+ which in the end will contain the final net.

Create one place for each member set $s \in PF(VES(\psi))$ (Line 3). Denote the set of progress formulas that a place p_i corresponds to as $\Psi(p_i)$ (Line 4). 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 which have to be marked when the Petri-net enters the state represented by s. Their function will become clear in Section 7.2.3, where transitions are added. Initially, $P_{in}(s) = \{p_i\}$ (Line 5). $P(s) = p_i$ maps a set of elementary formulas to the place. As opposed to $P_{in}(s)$, $P(s)$ will not change during the course of later steps of the algorithm. Ports corresponding to the atomic propositions (without relations) occurring in the formula must also be added to the model

```
1  procedure createPlaces(ψ: ACTL)
2      for each s ∈ PF(VES(ψ)) do
3          add a place p_i to net;
4          Ψ(p_i) := s;
5          P_in(s) := { p_i };
6          P(s) := p_i;
7      for each p ∈ AP(ψ) do
8          add a place p_p to net;
9          P(p) := p_p;
```

Figure 7.3: The algorithm for creating the places in the resulting Petri net

(Line 8). They are associated to their corresponding atomic proposition through the mapping $P(p)$ (Line 9).

In our example, the resulting net has two places, 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, it 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 $P(q) = p_q$.

7.2.2 TIMER INSERTION FOR **U** OPERATORS

In Section 7.1.1, 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 are 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 7.9, 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 4 in Figure 7.3.

Definition 7.9: Set of U formulas. The set of U formulas in place p_i is expressed as $U(p_i) = \{\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \mid \mathbf{AX}\,\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \in \Psi(p_i)\}$.

■

Place p_1 in our example does not have any U formula, $U(p_1) = \emptyset$ and place p_5 has got one, $U(p_5) = \{\mathbf{A}\,[t\ \mathbf{U}_{\leq 3} q < 10]\}$.

Definition 7.10: Top-level subformula. The set $sub_{top}(\varphi)$ of top-level subformulas of the formula φ is defined by the following equations.

1. If $\varphi = \neg\varphi_1$ or $\varphi = p$, where p is an atomic proposition, then $sub_{top}(\varphi) = \{\varphi\}$.
2. If $\varphi = \varphi_1 \vee \varphi_2$ or $\varphi = \varphi_1 \wedge \varphi_2$ then $sub_{top}(\varphi) = sub_{top}(\varphi_1) \cup sub_{top}(\varphi_2)$.
3. If $\varphi = \mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]$, then $sub_{top}(\varphi) = \{\mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]\} \cup sub_{top}(\varphi_1)$. If $\varphi = \mathbf{A}[\varphi_1\ \mathbf{R}\ \varphi_2]$, then $sub_{top}(\varphi) = \{\mathbf{A}[\varphi_1\ \mathbf{R}\ \varphi_2]\} \cup sub_{top}(\varphi_2)$.

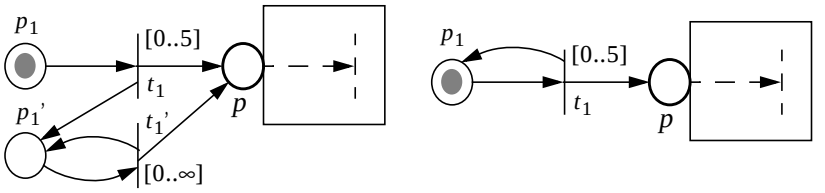
■

The $sub_{top}(\varphi)$ operator decomposes the formula φ into its clauses if it is a disjunction or conjunction. U and R formulas have their first or second subformula, respectively, decomposed. In these two cases, the formula itself is also a top-level subformula. Otherwise, the formula is left intact.

In our example from page 119, $sub_{top}(\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]) = \{\neg p, \mathbf{A}([t\ \mathbf{U}_{\leq 3} q < 10], t)\}$ and $sub_{top}(\mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])]) = \{\mathbf{A}([f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])], \neg p, \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10], t)\}$.

Definition 7.11: Timer repeating formula. A formula $\varphi \in U(p_i)$ is timer repeating in place p_i iff there exists a formula $\mathbf{AX}\,\mathbf{A}[\varphi_1\ \mathbf{R}\ \varphi_2] \in \Psi(p_i)$ such that $\varphi \in sub_{top}(\varphi_2)$.

■



(a) $\mathbf{AF}_{\leq 5} p$ (b) $\mathbf{AGAF}_{\leq 5} p$

Figure 7.4: Illustration of timer repeating

Formula $\mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10]$ is timer repeating in place p_5 in the example since $\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])] \in \Psi(p_5) = S_5$ and $\mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10] \in sub_{top}(\neg p \vee \mathbf{A}[t\ \mathbf{U}_{\leq 3} q < 10])$. The definition is not applicable to any formula in p_1 since it does not contain any U formula, $U(p_1) = \emptyset$.

The reason for handling formulas which are timer repeating differently from those which are not timer repeating is the semantics of the **R** operator. In $\mathbf{A}[\varphi_1\ \mathbf{R}\ \varphi_2]$, the second subforula φ_2 must hold as long as the first subformula φ_1 is not satisfied. Hence, if φ_2 contains a U formula, that U formula must continue to hold and the timer must consequently be restarted. If φ_2 does not contain a U formula, the timer *must not* be restarted, since the events in φ_2 do not *have to* occur several times.

Consider the formulas $\psi_1 = \mathbf{AF}_{\leq 5} p$ and $\psi_2 = \mathbf{AGAF}_{\leq 5} p$. ψ_1 states that p must be marked after at most 5 time units, but it does not say anything about what should happen after that deadline. Figure 7.4(a) illustrates this behaviour. p_1 is not timer repeating in this case. After having fired, the timer, transition t_1, is not restarted.

ψ_2 states that p must be marked after at most 5 time units, repeatedly. After being marked for the first time, it must be marked within 5 time units again. Figure 7.4(b) illustrates this behaviour. After having fired, the timer, transition t_1 is immedi-

```
1  procedure addTimers(p_i: place)
2      for each φ ∈ U(p_i) do
3          add places p_ix and p_ix' as indicated by Figure 7.6 to net;
4          add transition t_ix asin indicated by Figure 7.6 to net;
5          set time delay of t_ix to [0..j] where j is the upper bound
               associated to the U operator in φ;
6          P_in(Ψ(p_i)) := P_in(Ψ(p_i)) ∪ { p_ix };
7          Timerin(p_i, φ) := p_ix;
8          Timerout(p_i, φ) := p_ix' ;
9          if φ is non-timer repeating in p_i then
10             add place p_ix" as indicated by Figure 7.6 to net;
11             NontimerCopy(p_i, φ) := p_ix";
```

Figure 7.5: Algorithm for adding timers to a place

ately re-enabled in order to guarantee another firing. p_1 is in this case timer repeating.

Section 7.3.1 and Section 7.3.2 go into more detail regarding these two formulas.

One timer per U formula must be added, so that the deadline of each U formula can be timed independently from each other. Figure 7.5 presents the algorithm for adding timers to a place p_i. The algorithm is illustrated in Figure 7.6 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 real 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.

All timers must be simultaneously started when the Petri-net 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 places p_{ix} are added to the mapping (Line 6).

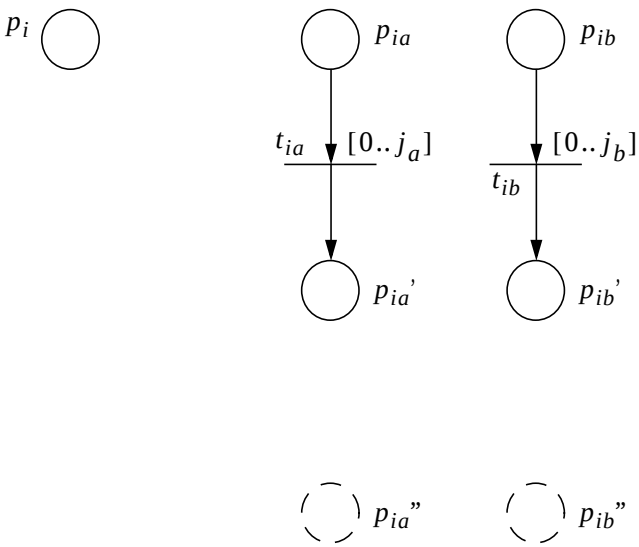


Figure 7.6: Adding timers to a place

The exact use of these mappings will become clear in Section 7.2.3. $Timerin(p_i, \varphi)$ and $Timerout(p_i, \varphi)$ record the start and end places respectively of the timers for future reference (Line 7 and Line 8).

In case the place is not timer repeating, one additional place per timer (p_{ix}'') must be added (Line 10) in order to capture the same state, but when the respective timer has already fired, and the events specified by the U formulas do not *have to* occur again. The mapping $NontimerCopy(p_i, \varphi)$ helps to refer to these places in later steps of the algorithm (Line 11).

In the case of the example formula ψ (defined on page 119), only place p_5 has a timer. Figure 7.7 presents the result of adding the timer corresponding to the only U formula in that place. At this point, $P_{in}(\Psi(p_5)) = \{p_5, p_{5a}\}$ and $P_{out}(\Psi(p_5)) = \{p_5, p_{5a}'\}$. It still holds that $P_{in}(\Psi(p_1)) = \{p_1\}$. Since the U formula in p_5 is timer repeating, no extra places p_{ix}'' are added.

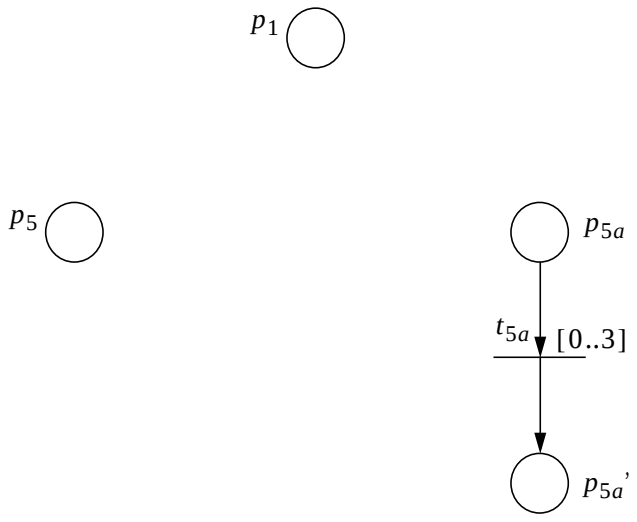


Figure 7.7: Adding timers to the example Petri net

7.2.3 TRANSITION GENERATION

At this point, all places and timers are added to the Petri-net. The major remaining step is to add the transitions to the PRES+ model. Transitions are added according to different procedures depending on whether a timer has been added to a particular place or not.

Definition 7.12: Target formulas. The set of target formulas of a place p_i is defined as $TF(p_i) = \bigcap_{\mathbf{AX}\ \varphi \in \Psi(p_i)} \Phi(\varphi) \cap VES(\psi)$.

■

Definition 7.13: 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.

The set of target places is the set of sets of progress formulas representing places in the Petri-net (see Section 7.2.1) 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 the sets in $TF(p_i)$.

In our example, $TF(p_1) = \{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$ and $TF(p_5) = \{S_5, S_7, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$ according to the more detailed computations in Equation 7.4 and Equation 7.5 respectively. In addition, $TP(p_1) = TP(p_5) = \{S_1, S_5\}$.

$$TF(p_1) = \Phi(\psi) \cap VES(\psi) = \quad (7.4)$$
$$\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\} \cap$$
$$\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\} =$$
$$\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$$

$$TF(p_5) = (\Phi(\psi) \cap \Phi(\mathbf{A}\,[t\ \mathbf{U}_{\leq 3} q < 10])) \cap VES(\psi) = \quad (7.5)$$
$$(\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{27}, S_{29}, S_{31}\} \cap$$
$$\{S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}, S_{24}, S_{25}, S_{26}, S_{27},$$
$$S_{28}, S_{29}, S_{30}, S_{31}\}) \cap$$
$$\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\} =$$
$$\{S_5, S_7, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$$

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 Petri-net 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 7.8, Line 2 to Line 4). 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 $s \in TF(p_i) - TP(p_i)$. Line 3 will be discussed later in connection with Definition 7.14.

Performing the events described by s is done by moving the token(s) 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 also as indicated by the atomic propositions in s, $AP(s)$. Review the discussion about how a state is represented in the Petri-net, on page 127. A state corresponding to s is a 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 5), 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 6), since we are entering the place corresponding to $PF(s)$. Thereby, possible timers in that place are also started.

```
1  procedure addTransitions(p_i: place)
2      for each s ∈ TF(p_i) - TP(p_i) do
3          if s is not redundant with respect to TF(p_i) then
4              add transition t to net;
5              °t := { p_i };
6              t° := P_in(PF(s));
7              connectToPorts(t, s);
8              set time delay of t to findTimeDelay(s, p_i);
```

Figure 7.8: The standard algorithm for adding the transitions belonging to place p_i.

```
1  procedure connectToPorts(t: transition, s: set of elementary formulas)
2      °t := °t ∪ P(AP_out(s));
3      t° := t° ∪ P(AP_in(s));
4      g := true;
5      for each pℜv ∈ AP_rout(s) do
6          g := g ∧ pℜv;
7      set guard of t to g;
8      set function of t to return a random value from PV_in(s);
```

Figure 7.9: The algorithm for adding interaction with the ports to transition t as specified by the set s.

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 7), is explained next, as well as determining the time delay interval of the transition (Line 8).

Figure 7.9 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 8 in Figure 7.8). Figure 7.10 shows how this delay is computed. Normally, there is no requirement on when a certain

```
1  function findTimeDelay(s: set of elementary formulas, p_i: place) returns
       time interval
2      if Ψ(p_i) ∪ AP_out(s) ∪ AP_rout(s) ∉ VES(ψ) then
3          return [0..0];
4      else
5          return [0..∞];
```

Figure 7.10: Algorithm for finding the correct time delay interval of a transition

event has to be performed. This means that the transition should be able to fire after 0 time units and before 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 Petri-net 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 valid, i.e. $\Psi(p_i) \cup AP_{out}(s) \cup AP_{rout}(s) \notin VES(\psi)$ (Line 2), in which case the Petri-net must immediately move to a valid state by firing the transition at hand. As a consequence, the time delay has to be $[0..0]$.

Previously, it has been assumed that a transition is added for every set of elementary formulas $s \in TF(p_i) - TP(p_i)$. Although the resulting Petri-net will be correct using this assumption, it will contain unnecessarily many transitions, leading to a longer verification time than needed. Some transitions are namely redundant.

Definition 7.14: Redundant elementary set. A set of elementary formulas s is redundant with respect to the set of sets of elementary formulas E, iff there exists a set $s' \in E$, $s \neq s'$, 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 E, 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 perform 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 net (Line 3 in Figure 7.8).

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 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, $\{S_1, S_5, S_9, S_{23}, S_{25}, S_{31}\}$.

Place p_1 in the example does not have any timer, so it follows the procedure above for creating its transitions. Figure 7.11 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 has a relation, the function produces random values which still satisfy all relations involved, i.e. its port values.

So, for example, t_2 has an output arc labelled q. This means that the arc is connected to port q, and that the function associ-

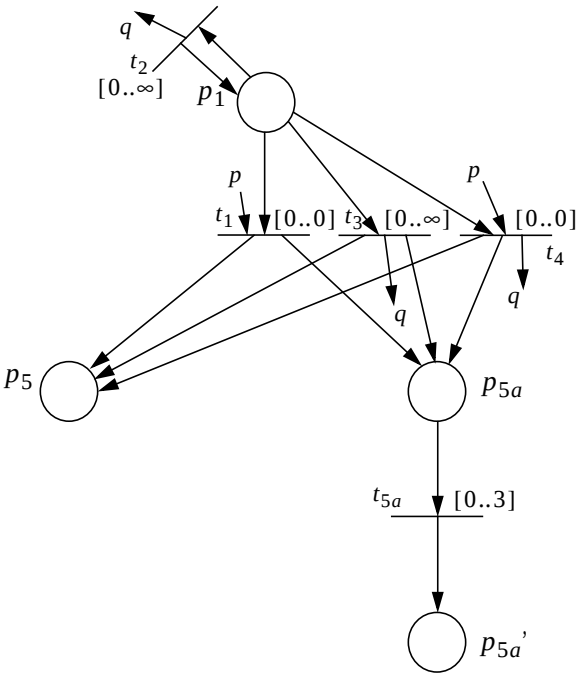


Figure 7.11: The result of adding the transitions of place p_1 to the example formula

ated to transition t_2 generates random values. Transition t_9 in Figure 7.14 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 Section 7.2.2 that $TF(p_1) = \{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$. 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 outport. 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 $S_1 \cup \{p\} = S_3 \notin VES(\psi)$. Since p_5 has a timer associated to it, $P_{in}(S_5) = \{p_5, p_{5a}\}$. All transitions have source place p_1.

Next, transition t_2 corresponding to S_9 must be added. $S_9 = S_1 \cup \{q\}$, so the target place is equal to the source place, i.e. p_1. In addition, a token with a completely random value is placed in the port corresponding to q when fired. Since $S_1 \cup \{q\} = S_9 \in VES(\psi)$, the time delay is set to $[0..\infty]$. The procedure progresses similarly for the sets S_{13} and S_{15}. The remaining sets are redundant with respect to $TF(p_1)$, and consequently no transitions are added for these sets.

The place has a timer

When timers are added to the place, it is necessary to identify whether a certain set of elementary formulas has to occur before a certain deadline or not.

Definition 7.15: Requiring U formulas. The set of requiring U formulas of a place p_i and a set of elementary formulas s is defined as $RUF(p_i, s) = \{\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \in U(p_i) \mid s \in \Phi(\varphi_2)\}$. ■

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 7.16: Timer triggered formulas. The set of timer triggered formulas of a a set of U formulas U is defined as $TTF(U) = \bigcap_{\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2] \in U} \Phi(\varphi_2) \cap VES(\psi)$. ■

The timer triggered formulas denote the set of sets of elementary formulas corresponding to events of which one has to be performed before the deadline of 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 $TTF(U(p_5)) = \{S_{25}, S_{29}, S_{31}\}$ according to Equation 7.6.

$$\begin{aligned} &TTF(U(p_5)) = \Phi(q < 10) \cap VES(\psi) = \\ &\{S_{24}, S_{25}, S_{26}, S_{27}, S_{28}, S_{29}, S_{30}, S_{31}\} \cap \\ &\{S_1, S_5, S_7, S_9, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\} = \\ &\{S_{25}, S_{29}, S_{31}\} \end{aligned} \tag{7.6}$$

Figure 7.12 and Figure 7.13 present the algorithm. Similar to the non-timer case, transitions are added for each non-redundant 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 not.

In the first case (Line 3 in Figure 7.12), all U formulas requiring the event s are examined. If there only are timer repeating formulas requiring s in p_i, $f = \emptyset$, then only one transition needs to be added. It is inserted between the end places of the timers corresponding to the requiring formulas, and the start places (Line 10 and Line 11). One transition is enough due to the fact that all involved timers may trigger at any time and put a token in their respective end places, and in that way enable the particular transition. This case should be compared with the one described in the next paragraph.

However, if there is at least one non-timer repeating formula requiring s, it is necessary to add one transition per combination of non-timer repeating formulas requiring s (Line 6). The reason is that all possibilities of determining which timers should be considered triggered with s must be covered. In the timer repeating case, this is not necessary since timers are always restarted, so the end place of the timers can always be reached. If there are no timer repeating formulas, a transition for $f = \emptyset$ must not be added, since such a transition would not be connected to any timer at all (Line 7).

```
1  procedure addTransitionsForTimers(p_i: place)
2      for each s ∈ TF(p_i) - TP(p_i) do
3          if PF(s) = Ψ(p_i) then
4              if RUF(p_i, s) ≠ ∅ then
5                  if s is not redundant w.r.t TTF(RUF(p_i, s)) then
6                      for each f ∈ (2^(the non-timer rep formulas of RUF(pi, s))) do
7                          if exists timer rep. formula φ ∈ RUF(p_i, s) or
                               f ≠ ∅ then
8                              add transition t to net;
9                              for each timer rep. formula φ ∈ RUF(p_i, s) do
10                                 °t := °t ∪ { Timerout(p_i, φ) };
11                                 t° := t° ∪ { Timerin(p_i, φ) };
12                             for each φ ∈ f do
13                                 °t := °t ∪ { Timerout(p_i, φ) };
14                                 t° := t° ∪ { NontimerCopy(p_i, φ) };
15                             connectToPorts(t, s);
16                             set time delay of t to [0..0];
17                     if s is not redundant w.r.t. TF(p_i) then
18                         for each non-timer rep. formula φ ∈ RUF(p_i, s) do
19                             add transition t' to net;
20                             °t' := { NontimerCopy(p_i, φ) };
21                             t'° := { NontimerCopy(p_i, φ) };
22                             connectToPorts(t', s);
23                             set time delay of t' to findTimeDelay(s, p_i);
24             else if s is not redundant w.r.t TF(p_i) then
25                 add transition t to net;
26                 °t := { p_i };
27                 t° := { pi };
28                 connectToPorts(t, s);
29                 set time delay of t to findTimeDelay(s, p_i);
30         else
31             addTransitionsForOtherDestinations(s, p_i); -- see Figure 7.13
```

Figure 7.12: Algorithm for adding transitions to a place with timers

```
1   procedure addTransitionsForOtherDestinations(s: set of CTL, p_i: place)
2       if s ∈ TTF(timer repeating formulas in U(p_i)) and
            s is not redundant w.r.t. TTF(timer repeating formulas in U(p_i)) then
3           for each f ∈ (2^(the non-timer rep formulas of RUF(pi, s))) do
4               add transition t to net;
5               °t := { p_i };
6               for each φ ∈ RUF(p_i, s) do
7                   if φ is timer repeating or φ ∉ f then
8                       °t := °t ∪ { Timerout(p_i, φ) };
9                   else
10                      °t := °t ∪ { NontimerCopy(p_i, φ) };
11              t° := P_in(PF(s));
12              connectToPorts(t, s);
13              if there exists a timer repeating formula in p_i or
                        f ≠ { all non-timer repeating formulas in U(p_i) } then
14                  set time delay of t to [0..0];
15              else
16                  set time delay of t to findTimeDelay(s, p_i);
```

Figure 7.13: Continuation of the algorithm for adding transitions to a place with timers

When a timer corresponding to a non-timer repeating formula is triggered, tokens are taken from the end places of those timers and new tokens are put in the non-timer copies (Line 13 and Line 14). An additional transition must also be added as a loop on the non-timer copies of each requiring timer in order to model the fact that the event *may* occur also after its first occurrence (Line 17 to Line 23).

The transitions added to the end place of a timer must all have time delay interval [0..0], since the event must occur immediately after the timers have fired. Otherwise, the deadline would be violated.

In case there is no timer requiring s, a transition is added as a loop around p_i (Line 24 to Line 29). The time delay interval follows the normal procedure, described in Figure 7.10, since no timers are involved.

Otherwise, if the target place is different from p_i, a transition can only be added if all timers corresponding to timer repeating formulas in the place require the event described by s (Line 2 in Figure 7.13). The reason is that, by definition, all such timers must have their requirements satisfied before their deadlines. This condition does not need to be applied on timers corresponding to non-timer repeating formulas, since tokens in the non-timer copies already indicate that they have been triggered.

The transition is added from p_i (Line 5) and the end places of timers corresponding to timer repeating formulas (Line 8) plus either the end places or the non-timer copies of the timers corresponding to non-timer repeating formulas (Line 8 and Line 10). All combinations of end places of the timers and their non-timer copies must be covered similar to the previous discussion. If the particular transition is added only from non-timer copies (apart from p_i), the time delay interval is set according to the normal rule (Line 16). Otherwise, at least one timer is involved, implying a time delay interval of $[0..0]$ (Line 14).

In the example, place p_5 contains one U formula, which is timer repeating. Remember that $TF(p_5) = \{S_5, S_7, S_{13}, S_{15}, S_{25}, S_{29}, S_{31}\}$, and $TTF(U(p_5)) = \{S_{25}, S_{29}, S_{31}\}$. Figure 7.14 shows the result of the procedure.

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 3 in Figure 7.12), but $RUF(p_5, S_7) = \emptyset$. Since S_7 is not triggered by any timer and is not redundant, the transition is added as a loop around place p_5 (Line 24), see transition t_5 in the figure. The same goes for S_{13} and S_{15}, which result in transitions t_6 and t_7 respectively. $PF(S_{25}) = S_1$ and $S_{25} \in TTF(U(p_5))$, meaning that the transition, t_8, is added between $\{p_5, p_{5a}'\}$ and $P_{in}(S_1)$ (Line 5 and Line 8 in Figure 7.13). The for loop on Line 3 in Figure 7.13 only iterates once for $f = \emptyset$ since p_5 does not contain any non-timer repeat-

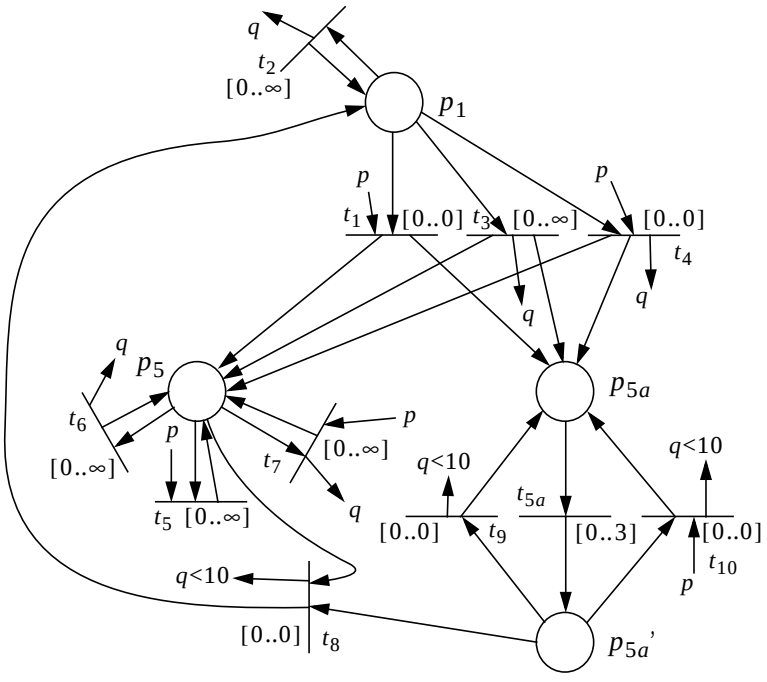


Figure 7.14: The result of adding the transitions of place p_5 to the example formula

ing formulas. The target place of S_{29} and S_{31} is p_5, and $RUF(p_5, S_{29}) = RUF(p_5, S_{31}) \neq \emptyset$ which means that the timer is restarted (Line 10 and Line 11 in Figure 7.12) as illustrated by transitions t_9 and t_{10} in the figure.

7.2.4 INSERTION OF INITIAL TOKENS

The last step of the algorithm is to insert initial tokens in the Petri-net. Figure 7.15 presents the algorithm for this purpose.

In Section 7.2.1, one place was created for each member in $PF(VES(\psi))$. Any of these places (or sets of places if timers were introduced) are randomly chosen for the initial token. This is in practice modelled with non-determinism. A place *start* is

```
1   procedure insertInitialToken
2       if | PF(VES(ψ)) | = 1 then
3           add tokens in P_in(the only elt of PF(VES(ψ))) with value <0, 0>;
4       else
5           add place start to net with an initial token with value <0, 0>;
6           for each s ∈ PF(VES(ψ)) do
7               add transition t to net;
8               °t := { start };
9               t° := P_in(s);
10              set time delay of t to [0..0];
```

Figure 7.15: Algorithm for add an initial token

added to the net (Line 5). For each candidate for the initial place, i.e. $PF(VES(\psi))$, a transition is added from *start* to that candidate (Line 6 to Line 10). The transition has time delay [0..0] since this choice must be performed instantly.

In case there is only one candidate for the initial place, i.e. $|PF(VES(\psi))| = 1$, there is naturally no need for adding this mechanism for choosing the initial place. The only existing place is directly chosen to be the initial one (Line 3).

In the example, there are two candidates for the initial token, S_1 and S_5. Consequently place *start* is created with two transitions, t_{11} and t_{12} in Figure 7.16. The figure shows the resulting Petri-net corresponding to the example formula as created by the proposed algorithm.

7.2.5 SUMMARY

Section 7.2 has so far presented all steps of the PRES+ generation procedure from an arbitrary ACTL formula. Figure 7.17 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 in the Petri-net. Places corresponding to atomic propositions, i.e. ports are also created (Line 2). Secondly, timers are added if

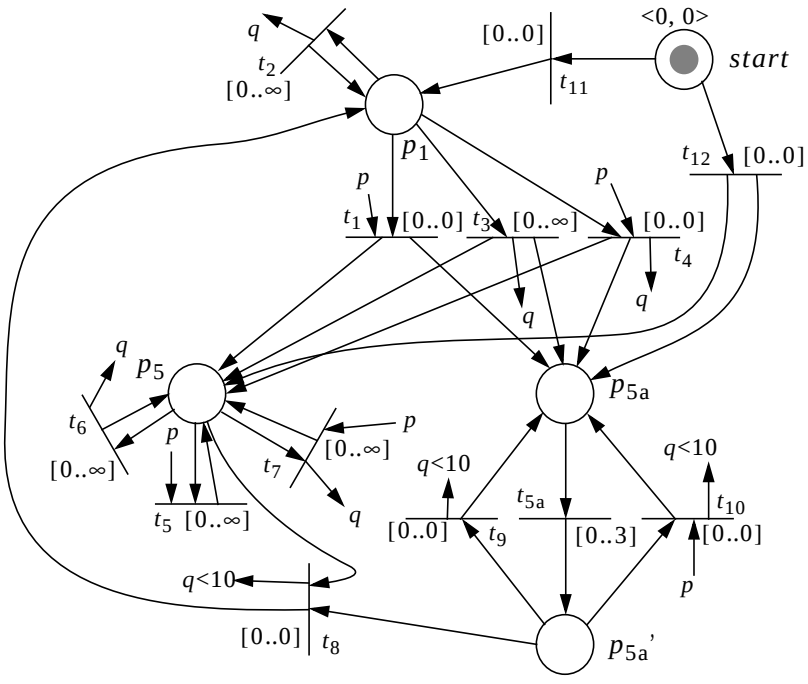


Figure 7.16: The resulting Petri net of the example formula

there is one or more U formulas among the progress formulas in the state (Line 4). The third step consists of adding transitions corresponding to events as given by sets of elementary formulas (Line 6). This is performed in different ways depending on the type of timers added to the place. The last step is to insert the initial tokens (Line 11).

7.3 Examples

In order to highlight and emphasise certain aspects of the algorithm presented in Section 7.2, a few examples are presented in this section. The reader is encouraged to follow the algorithms presented previously.

```
1   function generateFormulaStub(ψ: ACTL) returns PRES+
2       createPlaces(ψ);
3       for each s ∈ PF(VES) do
4           if | U(PF(s)) | > 0 then
5               addTimers(P(s));
6       for each s ∈ PF(VES(ψ)) do
7           if | U(PF(s)) | > 0 then
8               addTransitionsForTimers(P(s));
9           else
10              addTransitions(P(s));
11      insertInitialTokens;
12      return net;
```

Figure 7.17: The algorithm for generating a PRES+ model given an ACTL formula

7.3.1 PLACE WITH ONE NON-TIMER REPEATING U FORMULA

This section provides an example leading to a Petri-net with a place containing a non-timer repeating formula. The formula which will be used is $\mathbf{AF}_{\leq 3} p < 10$.

The first task when generating the Petri net corresponding 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 and $VES(\psi)$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3} p < 10]}_{1},\ \underbrace{p}_{2},\ \underbrace{p < 10}_{4} \right\} \quad (7.7)$$

$S(\psi) = \{S_0, S_1, S_2, S_3, S_6, S_7\}$, because S_4 and S_5 are contradictory since they contain $p < 10$ but not p.
$\Phi(\psi) = \Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 3} p < 10]) = \{S_1, S_3, S_6, S_7\}$
$\Phi(\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 3} p < 10]) = \{S_1, S_3, S_7\}$
$\Phi(p < 10) = \{S_6, S_7\}$

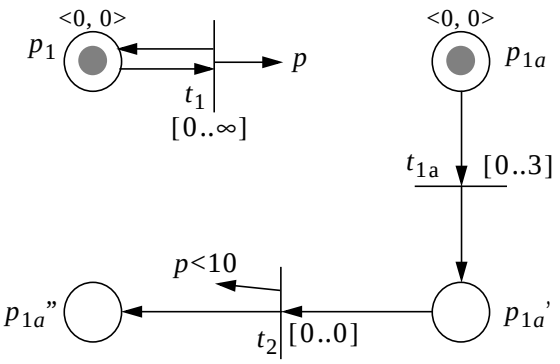


Figure 7.18: The resulting Petri net of the formula $\mathbf{AF}_{\leq 3} p < 10$

$VES(\psi) = \{S_1, S_3, S_7\}$. $S_6 \notin VES(\psi)$ since $S_6 - \{p, p < 10\} = S_0 \notin \Phi(\psi)$. $PF(VES(\psi)) = \{S_1\}$, which means that the resulting net will only have one state place, p_1. However, since there is one U formula in p_1 which is not timer repeating (there is no R formula at all), a timer is added as indicated by Figure 7.6. Consequently the non-timer copy p_{1a}'' is also added.

Figure 7.18 shows the resulting Petri-net after transitions have been added and initial tokens have been inserted.

$TF(p_1) = \{S_1, S_3, S_7\}$ and $TTF(U(p_1)) = \{S_7\}$. No transition is added for S_1 since it does not contain any atomic propositions. As for S_3, $PF(S_3) = S_1 = \Psi(p_1)$, 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 $PF(S_7) = S_1 = \Psi(p_1)$ and the timer requires it, $RUF(p_1, S_7) = \{\mathbf{A}[t\ \mathbf{U}_{\leq 3} p < 10]\}$, a transition t_2 is added between the timer and the corresponding non-timer copy. $\left|2^{RUF(p_1, S_7)}\right| = 2$, so there should actually be two such transitions. However, since there does not exist any timer repeating formula in p_1, no transition is added for $f = \varnothing$. Moreover, no

transition is added as a loop around p_{1a}" since S_7 is redundant with respect to $TF(p_1)$, $PV_{in}(S_7) \subseteq PV_{in}(S_3)$.

Because of the fact that $|PF(VES(\psi))| = 1$ a place *start* is not needed. 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 Petri-net and according to the formula.

7.3.2 PLACE WITH ONE TIMER REPEATING U FORMULA

This section provides an example of a formula leading to a Petri-net with a place containing a timer repeating formula. The formula which is used is $\mathbf{AGAF}_{\leq 3} p < 10$ It is similar to the formula discussed in Section 7.3.1, so a comparison with that example is encouraged.

The formula is normalised as $\psi = \mathbf{A}\,[f\ \mathbf{R}\ \mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]]$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\ \mathbf{A}\,[f\ \mathbf{R}\ \mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]]}_{1} \right. \tag{7.8}$$

$$\left. , \underbrace{\mathbf{AX}\ \mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]}_{2} , \underbrace{p}_{4} , \underbrace{p < 10}_{8} \right\}$$

$S(\psi) = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}\}$. The sets of formulas which only contain $p < 10$ but not p have been removed.

$\Phi(\psi) = \Phi(\mathbf{A}\,[f\ \mathbf{R}\ \mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]]) = \{S_3, S_7, S_{13}, S_{15}\}$
$\Phi(\mathbf{AX}\ \mathbf{A}\,[f\ \mathbf{R}\ \mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]]) = \{S_1, S_3, S_5, S_7, S_{13}, S_{15}\}$
$\Phi(\mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]) = \{S_2, S_3, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}\}$
$\Phi(\mathbf{AX}\ \mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]) = \{S_2, S_3, S_6, S_7, S_{14}, S_{15}\}$
$\Phi(p < 10) = \{S_{12}, S_{13}, S_{14}, S_{15}\}$

$VES(\psi) = \{S_3, S_7, S_{15}\}$. $S_{13} \notin VES(\psi)$ since $S_{13} - \{p, p < 10\} = S_1 \notin VES(\psi)$. Since $|PF(VES(\psi))| = S_1$, the resulting Petri net will only have one state place, p_3. However, the place has one U formula and it is timer repeating. A timer is consequently added as indicated by Figure 7.6. In this case, no non-timer copy will be created.

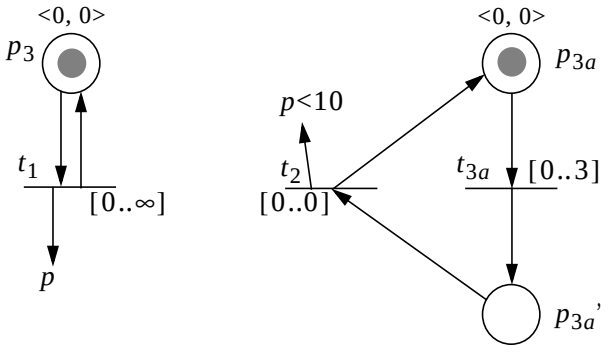


Figure 7.19: The resulting Petri net of the formula $\mathbf{AGAF}_{\leq 3} p < 10$

Figure 7.19 shows the resulting Petri net after having added transitions and inserted initial tokens.

$TF(p_3) = \{S_3, S_7, S_{15}\}$ and $TTF(U(p_3)) = \{S_{15}\}$. No transition is added for S_3 since it does not contain any atomic propositions. $PF(S_7) = S_3 = \Psi(p_3)$, a transitions connected to all requiring timers should be added. However, no timer requires S_7, $RUF(p_3, S_7) = \emptyset$, implying that a transition is added as a loop around p_3 instead, see transition t_1. $PF(S_{15}) = S_3 = \Psi(p_3)$ and $RUF(p_3, S_{15}) = \{\mathbf{A}\,[t\ \mathbf{U}_{\leq 3} p < 10]\}$, meaning that a transition t_2 is added from the end place of the timer to the start place, so that the timer is restarted when the transition is fired. Only one transition is added since no non-timer repeating formulas exist in p_3, so the only value on f is $f = \emptyset$.

In the resulting Petri-net, there is no non-timer copy of p_3, but the timer is restarted for every timer triggered event. This behaviour fits with the semantics of the formula which states that $p < 10$ must always be true within 3 time units, but it does not exclude the possibility that p sometimes may have other values.

7.3.3 PLACE WITH MORE THAN ONE TIMER REPEATING U FORMULA

This example aims to highlight what happens with a place with more than one timer repeating U formula. The formula used to illustrate this is $\mathbf{AG}(\mathbf{AF}_{\leq 2}p \vee \mathbf{AF}_{\leq 5}q)$. After normalisation, $\psi = \mathbf{A}[f\ \mathbf{R}\ (\mathbf{A}[t\ \mathbf{U}_{\leq 2}p] \vee \mathbf{A}[t\ \mathbf{U}_{\leq 5}q])]$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\,\mathbf{A}[f\ \mathbf{R}\ (\mathbf{A}[t\ \mathbf{U}_{\leq 2}p] \vee \mathbf{A}[t\ \mathbf{U}_{\leq 5}q])]}_{1}\ , \right. \tag{7.9}$$

$$\left. \underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]}_{2}\ , \underbrace{p}_{4}\ , \underbrace{\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 5}q]}_{8}\ , \underbrace{q}_{16} \right\}$$

$S(\psi) = \{S_0..S_{31}\}$. No sets have to be removed, since the formula does not contain any atomic propositions with relation.
$\Phi(\psi) = \{S_3, S_5, S_7, S_9, S_{11}, S_{13}, S_{15}, S_{17}, S_{19}, S_{21}, S_{23}, S_{25}, S_{27}, S_{29}, S_{31}\}$
$\Phi(\mathbf{AX}\ \psi) = \{S_1, S_3, S_5, S_7, S_9, S_{11}, S_{13}, S_{15}, S_{17}, S_{19}, S_{21}, S_{23}, S_{25}, S_{27}, S_{29}, S_{31}\}$
$\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 2}p] \vee \mathbf{A}[t\ \mathbf{U}_{\leq 5}q]) = \{S_2..S_{31}\}$
$\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]) = \{S_2..S_7, S_{10}..S_{15}, S_{18}..S_{23}, S_{26}..S_{31}\}$
$\Phi(\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 2}p]) = \{S_2, S_3, S_6, S_7, S_{10}, S_{11}, S_{14}, S_{15}, S_{18}, S_{19}, S_{22}, S_{23}, S_{26}, S_{27}, S_{30}, S_{31}\}$
$\Phi(p) = \{S_4, S_5, S_6, S_7, S_{12}, S_{13}, S_{14}, S_{15}, S_{20}, S_{21}, S_{22}, S_{23}, S_{28}, S_{29}, S_{30}, S_{31}\}$
$\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 5}q]) = \{S_8..S_{31}\}$
$\Phi(\mathbf{AX}\,\mathbf{A}[t\ \mathbf{U}_{\leq 5}q]) = \{S_8..S_{15}, S_{24}..S_{31}\}$
$\Phi(q) = \{S_{16}..S_{31}\}$

$VES(\psi) = \{S_3, S_7, S_9, S_{11}, S_{13}, S_{15}, S_{19}, S_{23}, S_{25}, S_{27}, S_{29}, S_{31}\}$ and $PF(VES(\psi)) = \{S_3, S_9, S_{11}\}$. Consequently, three places are created. S_3 and S_9 only have one U formula and therefore will only have one timer each. Both U formulas are moreover timer repeating in their respective place. Transitions are added to them in a similar way as in Section 7.3.2. In this section, we

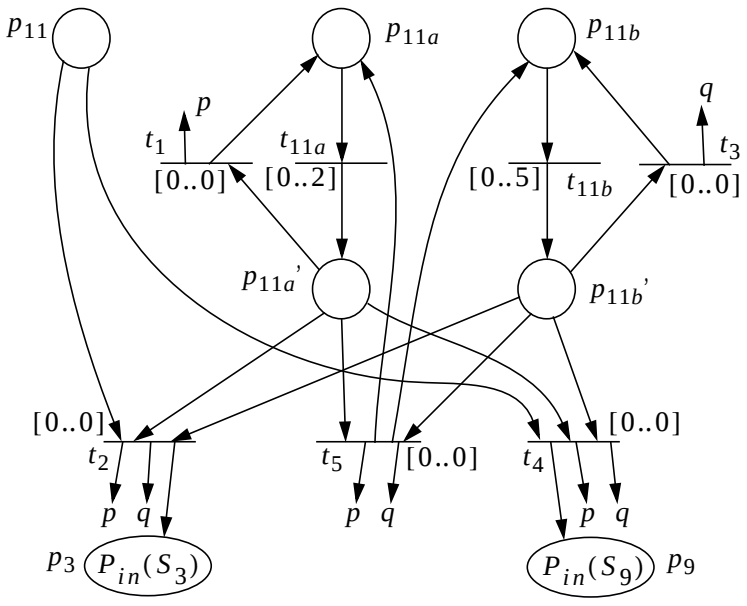


Figure 7.20: The resulting Petri net corresponding to progress formulas S_{11} of the formula $\mathbf{AG}(\mathbf{AF}_{\leq 2}p \vee \mathbf{AF}_{\leq 5}q)$

will concentrate on place p_{11} corresponding to set S_{11} which contains two U formulas. Figure 7.20 shows the part of the resulting Petri-net corresponding to this set of progress formulas.

$TF(p_{11}) = \{S_{11}, S_{13}, S_{15}, S_{19}, S_{23}, S_{27}, S_{29}, S_{31}\}$.
$TTF(U(p_{11})) = \{S_{20}, S_{21}, S_{22}, S_{23}, S_{28}, S_{29}, S_{30}, S_{31}\}$.

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 $S_{13} \notin TTF(U(p_{11}))$. $PF(S_{15}) = S_{11} = \Psi(p_{11})$ 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. Since

there are no non-timer repeating formulas throughout this example, at most one transition from a timer is added for $f = \emptyset$. No transition is added for S_{19} for the same reason as S_{13}. 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}. Both S_{29} and S_{31} are analogous to S_{23} resulting in transitions t_4 and t_5 respectively.

7.3.4 GUARDS ON TRANSITIONS

Formulas containing out-port atomic propositions with relation lead, in general, to a Petri-net with guards on certain transitions. This phenomenon will be demonstrated using the formula $\mathbf{AG}((p = 2) \rightarrow \mathbf{AF}_{\leq 7} q)$, where p is an out-port and q is an in-port of a connected component. Normalisation gives the formula $\psi = \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7} q])]$.

$$el(\psi) = \left\{ \underbrace{\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7} q])]}_{1}\ ,\ \underbrace{p}_{2}\ , \right. \tag{7.10}$$

$$\left. \underbrace{p \neq 2}_{4}\ ,\ \underbrace{\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 7} q]}_{8}\ ,\ \underbrace{q}_{16} \right\}$$

$S(\psi) = \{S_0..S_3, S_6..S_{11}, S_{14}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$, since in all other sets s, $p \neq 2 \in s$ but $p \notin s$. Hence, those sets are contradictory.

$\Phi(\psi) = \{S_1, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{19}, S_{23}, S_{25}, S_{27}, S_{31}\}$
$\Phi(\mathbf{AX}\ \mathbf{A}[f\ \mathbf{R}\ (\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7} q])]) =$
$\{S_1, S_3, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{19}, S_{23}, S_{25}, S_{27}, S_{31}\}$
$\Phi(\neg p \vee p \neq 2 \vee \mathbf{A}[t\ \mathbf{U}_{\leq 7} q]) = \{S_0, S_1, S_6..S_{11}, S_{14}..S_{19},$
$S_{22}..S_{27}, S_{30}, S_{31}\}$
$\Phi(\neg p) = \{S_0, S_1, S_8, S_9, S_{16}, S_{17}, S_{24}, S_{25}\}$
$\Phi(p \neq 2) = \{S_6, S_7, S_{14}, S_{15}, S_{22}, S_{23}, S_{30}, S_{31}\}$
$\Phi(\mathbf{A}[t\ \mathbf{U}_{\leq 7} q]) = \{S_8..S_{11}, S_{14}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$

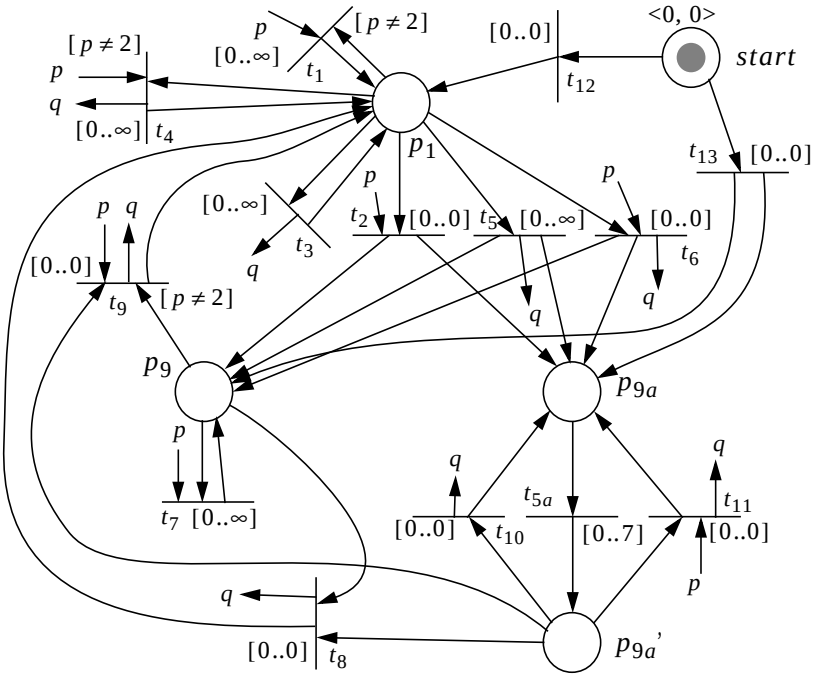


Figure 7.21: The resulting Petri net of the formula $\mathbf{AG}((p = 2) \rightarrow \mathbf{AF}_{\leq 7} q)$

$\Phi(\mathbf{AX}\ \mathbf{A}[t\ \mathbf{U}_{\leq 7} q]) = \{S_8, S_9, S_{10}, S_{11}, S_{14}, S_{15}, S_{24}, S_{25}, S_{26}, S_{27}, S_{30}, S_{31}\}$
$\Phi(q) = \{S_{16}..S_{19}, S_{22}..S_{27}, S_{30}, S_{31}\}$
$VES(\psi) = \{S_1, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{23}, S_{25}, S_{27}, S_{31}\}$, since $S_{19} - \{q\} = S_3 \notin \Phi(\psi)$. $PF(VES(\psi)) = \{S_1, S_9\}$, so two places p_1 and p_9 are created in the resulting Petri-net. Place p_9 contains a U formula which is timer repeating. Figure 7.21 shows the resulting Petri-net.

Starting with adding transitions for place p_1, $TF(p_1) = \{S_1, S_7, S_9, S_{11}, S_{15}, S_{17}, S_{23}, S_{25}, S_{27}, S_{31}\}$. 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 S_{11}, it contains p, but not $p \neq 2$. The time delay is set to [0..0], since $S_1 \cup \{p\} = S_3 \notin VES(\psi)$. 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 $PV_{out}(S_{15}, p) \subseteq PV_{out}(S_{11}, p)$. S_{17}, S_{23}, S_{25} and S_{27} causes t_3, t_4, t_5 and t_6 to be added respectively. No transition is added for S_{31} due to redundancy with S_{27}.

Let us continue with place p_9 containing a timer and is timer repeating. $TF(p_9) = \{S_9, S_{11}, S_{15}, S_{17}, S_{23}, S_{25}, S_{27}, S_{31}\}$ and $TTF(U(p_9)) = \{S_{17}, S_{23}, S_{25}, S_{27}, S_{31}\}$. 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_{23} 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_{31} is redundant.

Finally, the initial place is determined.

7.4 Verification Methodology Roadmap

This section continues the verification methodology roadmap from Section 6.6 and in particular Figure 6.19. Figure 7.22 shows the continuing 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

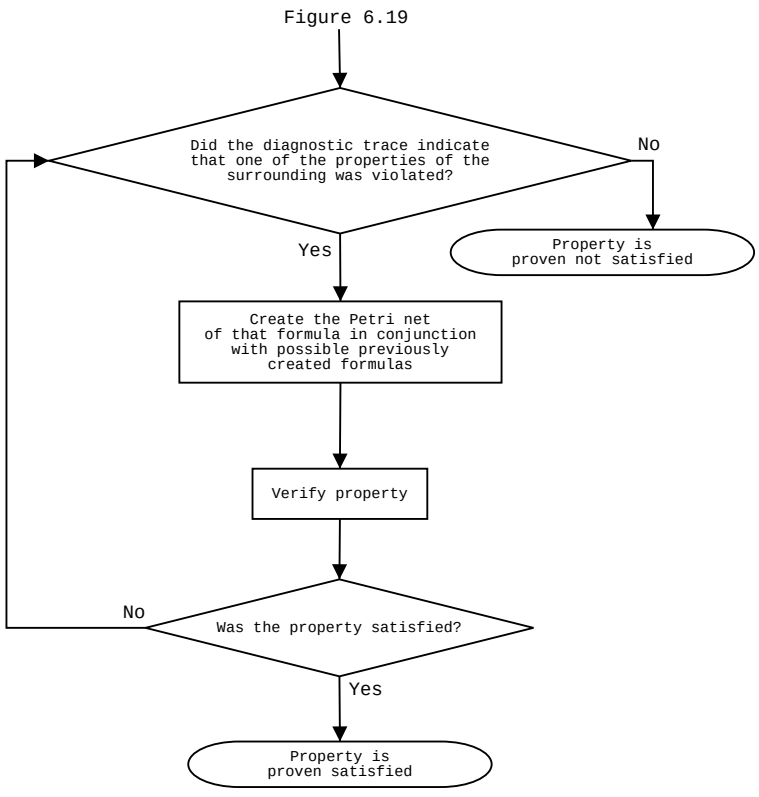


Figure 7.22: Continuation of the roadmap in Figure 6.19, useFormulas

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. If that is *not* the case, then the property is proven not satisfied. Otherwise, a PRES+ model is generated corresponding to that property.

The generated Petri-net 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 Petri-net is generated given both the previous formulas and the new one as a conjunction. The iteration continues until a final verification result is obtained.

Chapter 8
Example

THE PRESENTED VERIFICATION methodology gives a powerful means to verify large systems using a divide and conquer approach. This chapter provides an example in order to demonstrate how a system can be verified using the methodology.

8.1 The Mobile Telephone System

The model used as an example is a mobile telephone. Figure 8.1 shows an overview picture of the model and how the components forming the model are connected. It consists of seven components communicating via an AMBA bus.

1. Microphone. The microphone sends voice data to the transmitter.
2. Buttons. When dialing, 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 it to sound.

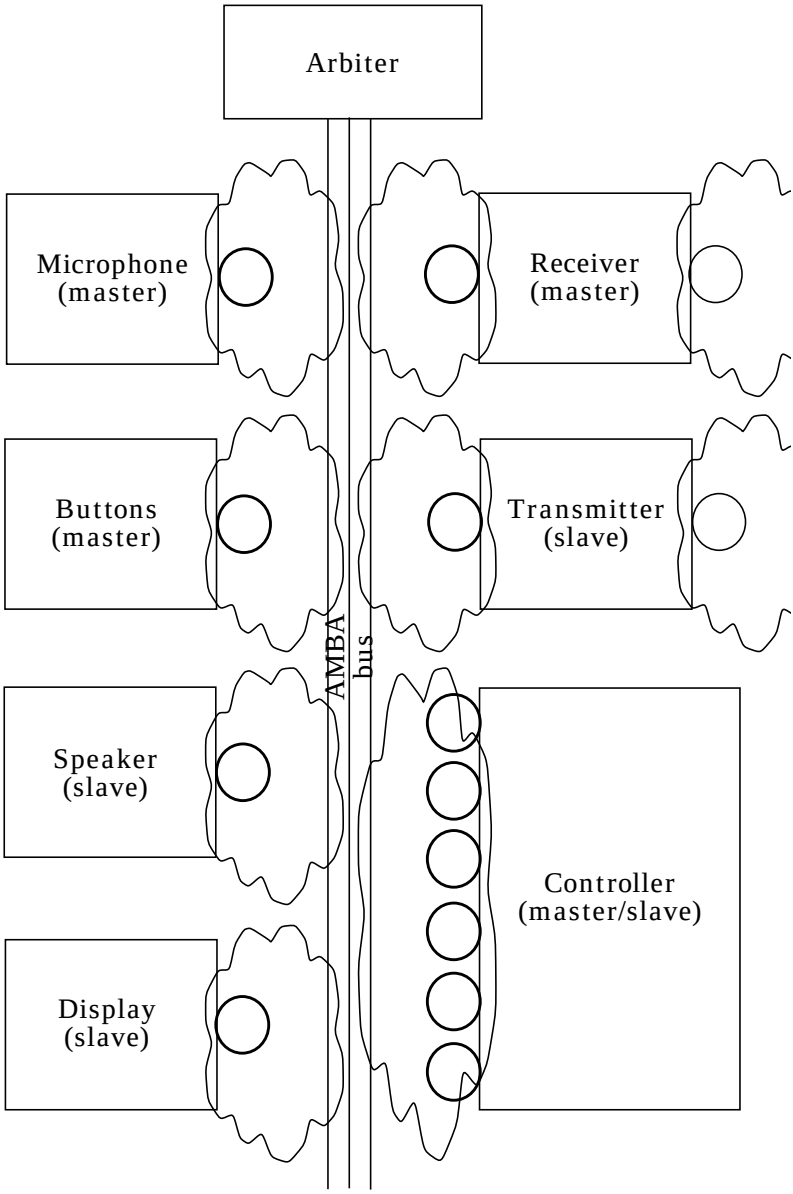


Figure 8.1: Overview model of the example system, a mobile telephone

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.

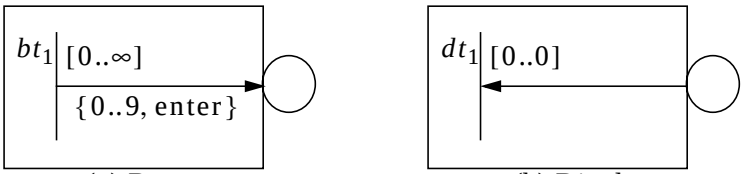
As mentioned previously, these components are supposed to communicate over an AMBA bus. However, since the AMBA bus requires a certain protocol and the components are not designed for that protocol, glue logics adapting the components to this protocol are inserted. These glue logics must be formally verified.

The components which are directly involved in the example are explained in more detail in the following sections.

8.1.1 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 8.2.

In this example, 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



(a) Buttons (b) Display

Figure 8.2: Models of the components Buttons and Display

request. From the point of view of Buttons, the 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 Petri-net 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 $[0..0]$.

8.1.2 CONTROLLER

The controller component keeps track of what is happening in the system and acts accordingly. Figure 8.3 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 dialing 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 going on 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.

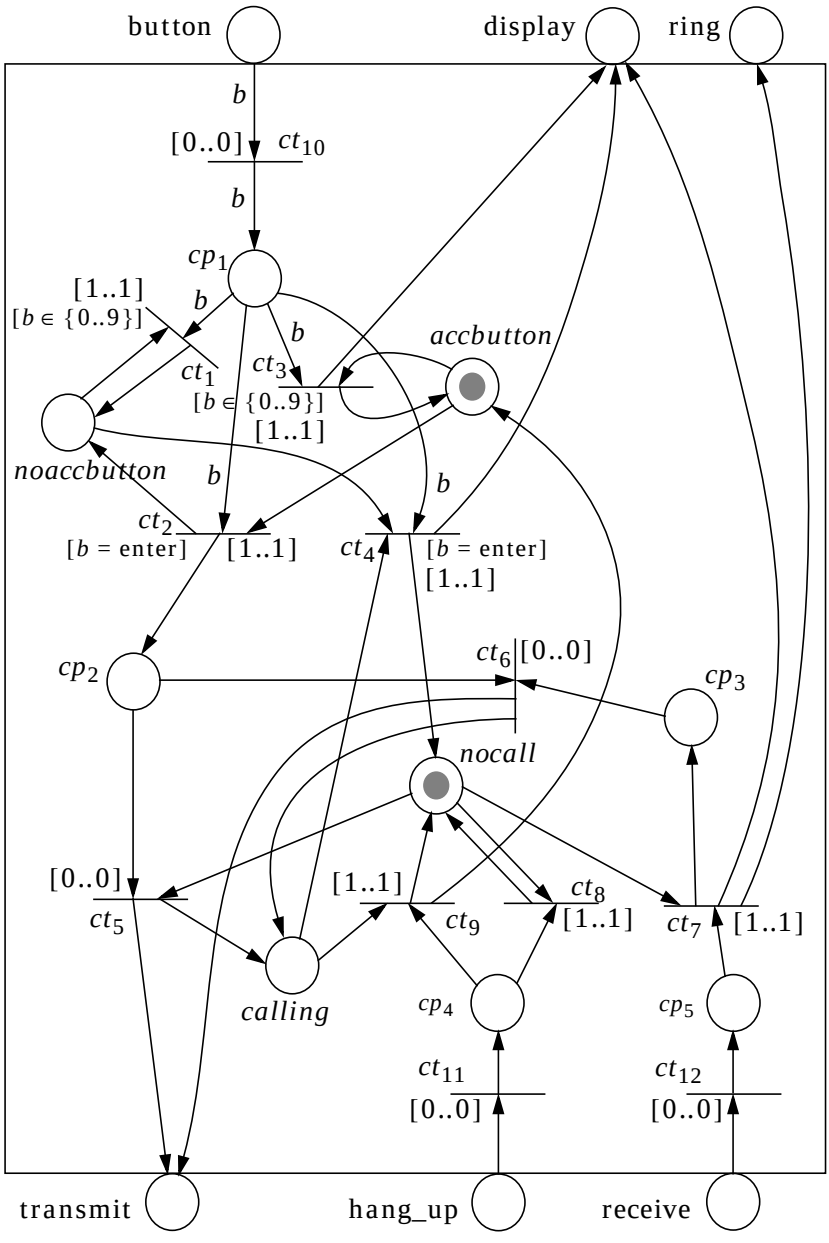


Figure 8.3: Model of the Controller component

8.1.3 AMBA BUS

All components communicate through an AMBA bus [Roy03]. The AMBA bus consists of two parts, Arbiter and Bus. The components communicating over the bus are furthermore divided into two categories, master and slave. Figure 8.1 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 Arbiter by emitting the signal (token) 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 do not necessarily have to address the same slave. When sending the last bunch, the master notifies this by emitting the signal (token) 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 of the bus to another requesting master. When the slave declares itself ready to receive again, 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 8.4 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 the round-robin schedule. The master holding the token has the opportunity to get access to the bus. If

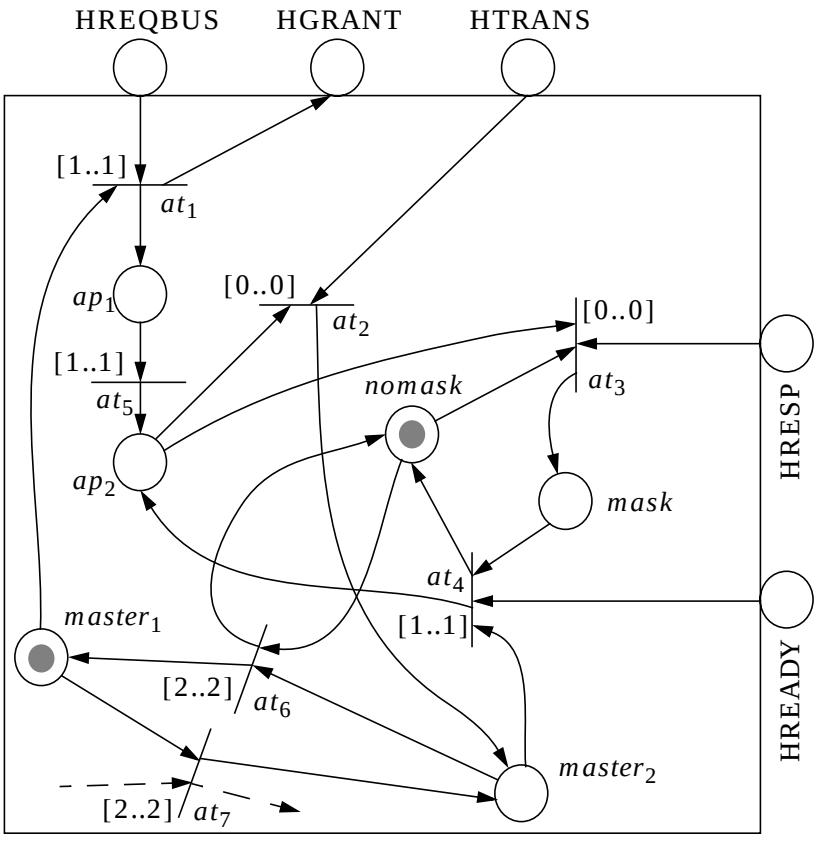


Figure 8.4: Model of the Arbiter component

a request has not arrived, the token moves to the next master, at_6. Place *mask* is marked when a slave has split the transaction of that master. *nomask* is marked otherwise.

The bus itself just distributes tokens sent to it to all components connected to it. Figure 8.5 shows a model of the Bus component. All transitions have time delay interval [0..0] and transition function identity. Consequently, it distributes exactly the same token to the rest of the components in zero time.

Port HRESP is directly connected to the arbiter through the port with the same name.

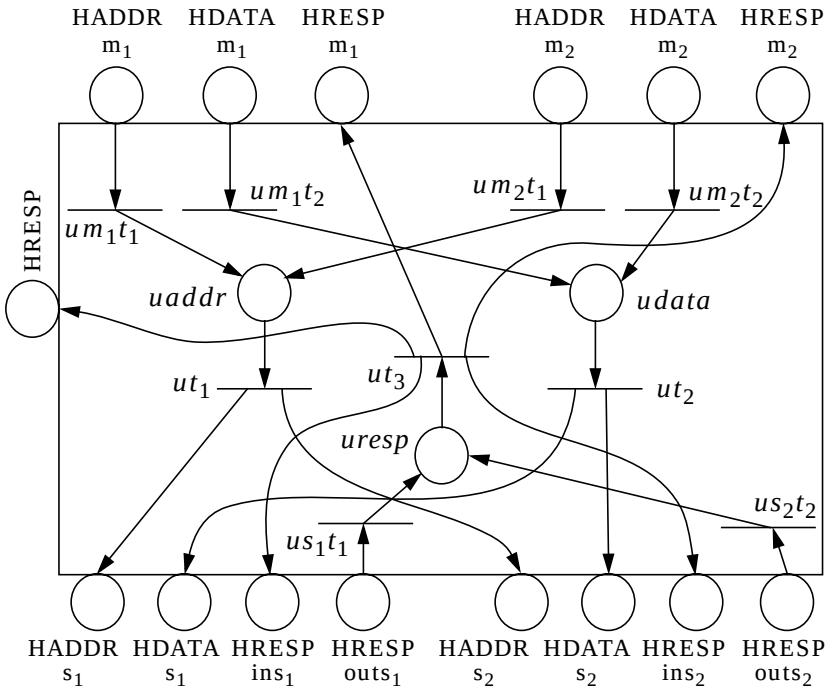


Figure 8.5: Model of the Bus component

8.1.4 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 (sometimes called wrapper) between the component and the bus.

Master functionality

A model of the glue logic which is active during the master functionality of the controller is shown in Figure 8.6. 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 reason, the glue logic must always remember the last transaction. The

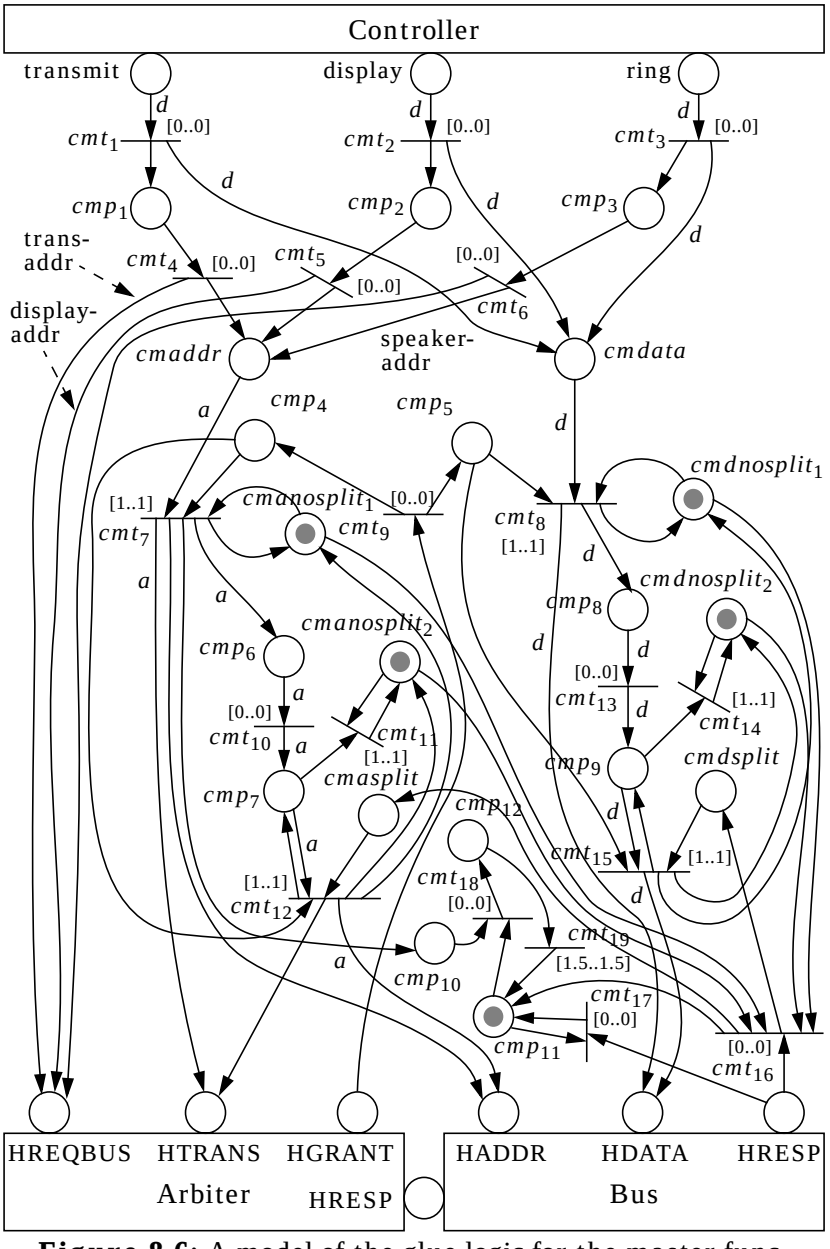


Figure 8.6: A model of the glue logic for the master functionality of the controller

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 resends 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 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 8.7.

When the slave is ready to receive data, a token is located in place $ready$. Otherwise, there are tokens in both places $notready_x$. Meanwhile, if the slave is not ready and data is sent to it, a token is placed in HRESPout to indicate that the trans-

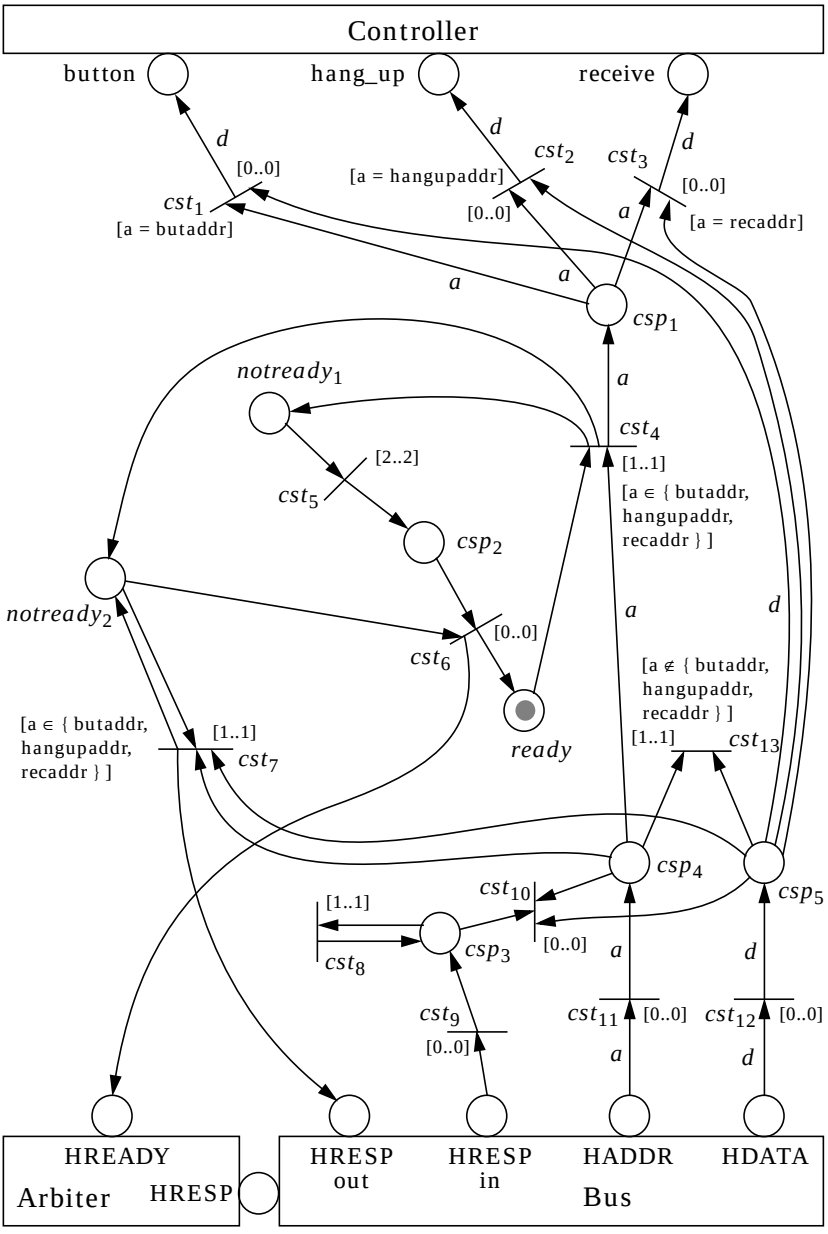


Figure 8.7: A model of the glue logic for the slave functionality of the controller

action 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 example, it is assumed that the controller is ready to receive data again 2 clock cycles after a previous reception (cst_5). After these two cycles the slave indicates to the arbiter that it is ready again (cst_6).

8.2 Verification of the Model

Three properties were verified in the system:

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 $\rightarrow$ **AF** HREADY)
3. When a master has been granted access to the bus, it must eventually close the transaction.
   AG (HGRANT $\rightarrow$ **AF** HTRANS)

8.2.1 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 and the slave functionality glue logic. Table 8.1 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.

Table 8.1: Verification results of property 1

Environment	Result	Time (s)
All empty stubs, bus generated	false	1.32
Add assumption on HDATA	true	125.33
Verify assumption, Buttons top-level stub, other stubs empty	true	7.58

Since the property was not satisfied, pessimism must be reduced in the stubs. According to the diagnostic trace, the bus produced a value on port HDATAs_x which is not allowed. However, the pessimism reduction algorithm failed to extinguish the value. It was thus necessary to make an assumption about the surrounding. In this case, it was assumed that only data in the set $\{0..9, \text{enter}\}$ can occur in port HDATAm_x. The property is formally given in Equation 8.1.

$$\mathbf{AG}\ (\text{HDATAm}_x \rightarrow \text{HDATAm}_x \in \{0..9, \text{enter}\}) \qquad (8.1)$$

A Petri-net for this formula was created together with a new version of the bus stub, now also including port HDATAm_x. 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 the positive result, the correctness of the assumption in Equation 8.1 must first be established.

The components involved in verifying the assumption were the buttons, arbiter, bus and master functionality glue logic. A top-level stub for buttons and empty stubs for the other components was enough for obtaining a result within 7.6 seconds.

8.2.2 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. Table 8.2 presents the result of the different stages in the verification process.

This verification has been started with a faulty glue logic. The fault consisted in that the slave functionality glue logic did not emit HRESP in time. This fault was finally fixed, after detection, by changing the time delay interval of transition cst_6.

At first, the property was verified using empty stubs on all components, except that the bus had one generated stub corresponding to interface $\{\text{HRESP}ins_x, \text{HRESP}outs_x, \text{HRESP}\}$. 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

Table 8.2: Verification results of property 2

Environment	Result	Time (s)
All empty stubs, except { HRESP*in*, HRESP*out* , HRESP }	false	2.47
Level 1 stubs for HADDR, HDATA	false	28.39
After correcting design error	false	87.57
Use top-level stub for Bus	true	246.14

the error, the property was reverified using the very same environment, but still with a negative verification result.

The problem this time was transition ut_3 in Figure 8.5 which had an infinite upper bound on the time delay interval. This caused the fact that no token would ever be placed in port HREADY. The situation is due to the pessimism of generated stubs. For this reason the generated stub was exchanged with a given one[1]. After additional 4 minutes, the property was finally satisfied.

8.2.3 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. Table 8.3 presents the result of the different stages in the verification process.

This verification was also stated 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 during verification, by adding places cmp_{10}, cmp_{11} and cmp_{12}, and the transitions cmt_{17}, cmt_{18} and cmt_{19} as indicated in Figure 8.6.

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 will not be sent and in particular the transaction will not be closed. Thus, the property is not satisfied.

---

1. Another way to continue the verification would have been to continue with less pessimistic stubs generated automatically and, if needed, with added PRES+ models corresponding to assumptions on the surrounding.

Table 8.3: Verification results of property 3

Environment	Result	Time (s)
All empty stubs	false	0.14
Arbiter stub	false	0.52
Add property 2 as assumption	false	2.58
After correcting design error	true	2467.42

To avoid this problem revealed by the diagnostic trace, the empty stubs of the arbiter were replaced with a given stub, such as the one shown in Figure 8.4. 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 Petri-net corresponding to the formula $\mathbf{AG}\,(\mathrm{HRESP} \rightarrow \mathbf{AF}_{\leq 5}\mathrm{HREADY})$ was generated and attached to Bus. 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 (places cmp_{10} to cmp_{12} together with neighbouring transitions) and the property was reverified with the same environment. After 41 minutes a positive result was obtained.

8.3 Discussion

This chapter has tried to give an example on how to use the verification methodology presented in this thesis, in practice. Moreover, it tries to convince the reader that the methodology is feasible to use.

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.

Chapter 9 Conclusions and Future Work

THIS THESIS HAS INTRODUCED a verification methodology which takes advantage of the fact that many designs are built using reusable components. The methodology assumes that these components are already verified, and concentrates on the parts of the design interconnecting the components.

This chapter summarises the thesis and points out interesting issues for future work.

9.1 Conclusions

Embedded systems are becoming increasingly common in our everyday lives. They are also becoming increasingly complex. In order to reduce the design complexity, predesigned and preverified components are used.

Due to the increasing complexity, the task of building such systems correctly becomes increasingly challenging. In order to meet this challenge, formal verification is introduced as part of the embedded systems design flow so that errors are found early in the design.

This thesis has presented a verification methodology which takes advantage of the fact that designs are built using reusable components. The methodology assumes that these components are already verified, and concentrates on the glue logics interconnecting the components. Every component has a number of properties associated to it which it requires the system to satisfy in order to work correctly.

The glue logics interconnecting the components are verified one at a time for the properties associated to the attached components. In order to be able to verify the glue logic, high-level models of the connected components must also be included in the verification, so that the glue logic may interact with an environment. 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 of a component. Since there are 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. The methodology iterates until a positive verification result is obtained or top-level stubs are used.

Until this point, it has been assumed that the stubs are given by the designer of the 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, which means that the 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 (T)ACTL formulas. An algorithm to generate a PRES+ model which produces all possible events still satisfying an ACTL formula has been presented. The generated Petri-net can then be attached to one of the components involved in the current verification.

An example has also been presented in order to demonstrate the feasibility of using the approach on realistic designs.

9.2 Future Work

Future work includes finding more efficient algorithms supporting the methodology. Efficient algorithms in this context means that they support the verification methodology in such a way that verification time is minimised. This can be obtained by, for instance, generating smaller stubs for the same degree of pessimism.

We have seen how a Petri-net can be generated from an ACTL formula. It is described in the thesis that this Petri-net is directly attached to the ports of a component or a stub of a com-

ponent. However, it might be possible, by analysing the structure of the component or stub, or by pattern matching, to modify the stub so that it complies with the ACTL formula. The implication of this would be that a smaller Petri-net is used in the verification with shorter verification times as a result.

It is also worth to examine the possibility of extending the algorithm in Chapter 7 for generating a Petri-net out of an arbitrary TACTL formula. In order to do this, it is needed to cope with deadlines on **G** and **R** operators. That would allow to express properties like "p must hold at least 5 time units", $\mathbf{AG}_{\leq 5} p$. Similarly, handling interval deadlines on properties could also be introduced, i.e. $\mathbf{AF}_{[3..5]} p$ (p must be true in between 3 and 5 time units).

References

[Ack00] B. Ackland, A. Anesko, D. Brinthaupt et al, ”A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP”, *Journal of Solid-State Circuits*, vol 35 no 3, 2000

[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

[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

[Cal99] A.E. Caldwell, H-J. Choi, A.B. Kahng, ”Effective Iterative Techniques for Fingerprinting Design IP”, in *Proc. DAC*, pp. 843-848, 1999

[Cam96] R. Camposano and 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

[Cou90] O. Coudert, J.C. Madre, ”A Unified Framework for the Formal Verification of Sequential Circuits”, in *Proc. ICCAD*, pp. 126-129, 1990

[Dal99] M. Dalpasso, A. Bogliolo, L. Benini, ”Virtual Simulation of distributed IP-based designs”, in *Proc. DAC*, pp. 50-55, 1999

[Edw97] S. Edwards, L. Lavagno, E.A. Lee et al, ”Design of Embedded Systems: Formal models, Validation, and Synthesis”, in *Proc. of the IEEE*, Vol.85 No 3, pp. 366-390, 1997

[Gaj00] D. Gajski, A C.-H. Wu, V. Chaiyakul et al, ”Essential Issues for IP Reuse”, in *Proc. ASP-DAC*, pp. 37-42, 2000

[Gar98] D. Garte, T. Kunjan, A. Reutter et al, ”Survey on a Practicable Design for Reuse Strategy Including Design Flow and Testbench Aspects”, in *Proc. Workshop on Reuse Techniques for VLSI Design*, 1998

[Gir93] E. Girczyc, S. Carlson, ”Increasing Design Quality and Engineering Productivity through Design Reuse”, in *Proc. DAC*, pp. 48-53, 1993

[Grä78] G. Grätzer, ”General Lattice Theory”, *Academic Press*, 1978

[Gra97] S. Graf, H. Saidi, ”Construction of abstract state graphs with PVS”, in *Lecture Notes in Computer Science*, Vol. 1254, pp. 72-83, 1997

[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

[Hon99] I. Hong, M. Potkonjak, ”Behavioral Synthesis Techniques for Intellectual Property Protection”, in *Proc. DAC*, pp. 849-854, 1999

[Jan03] A. Jantsch, ”Modeling Embedded Systems and SoC’s Concurrency and Time in Models of Computation”, *Morgan Kaufman*, 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 Reuse Methodology”, in *Proc. ISSS*, pp. 156-161, 2002

[Kea98] M. Keating, P. Bricaud, ”Reuse Methodology Manual for System-on-a-Chip Designs”, *Kluwer Academic Publishers*, 1998

[Koe98] M. Koegst, P. Conradi, D. Garte et al, ”A Systematic Analysis of Reuse Strategies for Design of Electronic Circuits”, in *Proc. DATE*, pp. 292-296, 1998

[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

[Pei99] H.P. Peixoto, M.F. Jacome, A. Royo et al. ”Design Space Layer: Supporting Early Design Space Exploration for Core-Based Designs”, in *Proc. DATE*, pp. 676-683, 1999

[Reu99] A. Reutter, R. Bosch, W. Rosenstiel, ”An Efficient Reuse System for Digital Circuit Design”, in *Proc DATE*, pp. 38-43, 1999

[Row97] J.A. Rowson, A. Sangiovanni-Vincentelli, ”Interface-Based Design”, in *Proc. DAC*, pp. 178-183, 1997

[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-??, 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

[Swa97] G. Swamy, ”Formal Verification of Digital Systems”, in *Proc. International Conference on VLSI Design*, pp. 213-217, 1997

[Tur99] J. Turley, ”Embedded Processors by the Numbers”, in *Embedded Systems Programming*, vol. 12, May 1999.

[UPP] Uppaal group, `http://www.uppaal.com/`

[Vah97] F. Vahid, L. Tauro, ”An Object-Oriented Communication Library for Hardware-Software CoDesign”, in *Proc. Workshop on HW/SW Codesign*, pp. 81-86, 1997.

[VSI] Virtual Socket Interface Alliance, `http://www.vsi.org/`

[Yal99] H. Yalcin, M. Mortazavi, R. Palermo et al, ”Functional Timing Analysis for IP Characterization”, in *Proc. DAC*, pp. 731-736, 1999