Des Autom Embed Syst (2007) 11:49–90
DOI 10.1007/s10617-006-9723-3

Formal verification of component-based designs

Daniel Karlsson · Petru Eles · Zebo Peng

Received: 7 June 2005 / Revised: 21 June 2006 / Accepted: 25 June 2006

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

This paper 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 them already formally verified by their designers. The components are considered correct given that the environment satisfies certain properties imposed by the component. The methodology verifies the correctness of the glue logic inserted between the components and the interaction of the components through the glue logic. Each such glue logic is verified one at a time using model checking techniques.

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

Keywords Formal verification · Petri-nets · Components · IP · Model checking · Embedded systems · Real-time systems

1 Introduction

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

D. Karlsson (✉) · P. Eles · Z. Peng
ESLAB, Department of Computer and Information Science, Linköpings Universitet, Linköping, Sweden
e-mail: danka@ida.liu.se

P. Eles
e-mail: petel@ida.liu.se

Z. Peng
e-mail: zebpe@ida.liu.se

music devices like Mp3-players. Moreover, other, traditionally mechanical, devices are becoming more and more computerised. Examples of such devices are cars and washing machines. Many of them are also highly safety critical such as aeroplanes or medical equipment.

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 [17, 19].

Not discovering a fault in the system in time can be very costly. Reusing predesigned IP blocks introduces the additional challenge that the exact behaviour of the block is unfamiliar to 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–$1 M per fault [31]. This suggests the importance of a structured design methodology based on a formal design representation, and, in particular, it suggests the need for formal verification.

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.

As the trend of building complex systems from reusable components is steadily growing, it is becoming 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 but still only covers a fraction of the statespace. 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 [33] and can be considered to be correct. What is still needed to be verified is the glue logic and the interaction between components over the glue logics. 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.

To formally verify the glue 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 glue logic to be verified is the components surrounding this logic. We assume that we have some high-level models of these components. However, due to several reasons, such as the fact that the components also need to interact with other glue logics, those high-level models cannot provide a complete specification of the environment. Therefore, it might be necessary to add assumptions about the unspecified part of the environment, namely the part which is connected to other interfaces of the components surrounding the logic under verification. The assumptions may be expressed in the same logic language as the verified properties. A model corresponding to the maximum set of behaviours still satisfying these assumptions can be built and included in the verification process.

In this paper 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. Once the model corresponding to the glue 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.

2 Related work

There exists a lot of work on issues related to reuse design methodologies [17], such as documentation, quality assurance [24], standardisation, integrity issues [9, 20], integration [28, 35], etc. The same is true for formal verification techniques such as equivalence checking [15], model checking [12] and theorem proving [30]. However, less effort has been put on combining the two, formal verification of systems built with reusable components. In particular, little work has been done on verification of the interconnect between two or more known components, which is the problem addressed in our work. However, there exists more work on the complementary problem, namely to verify that a component is correct given certain assumptions related to its environment.

The work by Cheung and Kramer [10] attacks a similar problem to ours in the sense that they use constraints on the environment in order to improve the verification of a component. The motivation for the work is that state explosion is still a problem even though modularity has been achieved. The modularisation may even make the problem worse, since the components are verified out of context. The context may restrict the possible behaviours of the component, therefore, without a context, many behaviours are unnecessarily examined. By adding constraints on the interfaces, this problem may be avoided. The constraints are automatically generated from the surrounding components and summarised in an interface process. Their work does, however, neither discuss the iterative refinement of interfaces (contexts), nor do they discuss timing aspects.

Xie and Browne [36] take a bottom-up approach to the verification of component-based systems. They divide components into two categories: composed and primitive. Composed components contain subcomponents as part of themselves, whereas primitive components generally are small and cannot be further divided into subcomponents. Due to the small size of primitive components, they can straight-forwardly be verified, e.g. by model checking, with conventional methods. The verification of composed components, on the other hand, relies on abstractions of the previously verified subcomponents. These abstractions might need to be iteratively refined in order to obtain a valid verification result. With this approach, the whole (possibly big) system must be simultaneously taken into consideration in the verification. Our approach focuses on each component interconnection at a time, considering at a moment only an as small part of the overall system as possible.

Interface automata have been proposed by de Alfaro and Henzinger [16] as a mechanism to check that the interfaces of two components are compatible. Each interface on each component has an associated automaton describing the legal behaviour on that interface. By composing the interface automata of two interconnected components, a new interface automaton describing the interface of the composed component is obtained. If this composition fails, the two interfaces are concluded incompatible. The approach concludes that two interfaces are compatible if there exists at least one environment that can make them work together. There is, however, no guarantee that the design at hand is such an environment. Properties other than interface compatibility cannot be addressed by this approach. The proposed interface automata can, moreover, not handle real-time aspects.

Barringer et al. [8] use learning to iteratively refine initially maximally abstract environment assumptions. During each iteration, model checking is performed to assert that certain properties are satisfied. If not, the environments are refined with the help of the diagnostic trace obtained from the verification, using a learning algorithm. If a contradiction is encountered, it is concluded that the property is not satisfied in the model. Our approach uses the same overall idea, but instead of learning, it exploits the structure of the connected components,

thus reducing the number of necessary iterations. In addition, their approach does not handle timing aspects.

Grumberg and Long [18] devise another approach based on abstraction and refinement of the component under verification, given a preorder (hierarchy) of such abstractions. They verify, with the help of model checking and an inference mechanism based on the preorder, that the component satisfies a certain property in all systems which can be built using that component. Our approach is also based on utilising a hierarchy of abstractions in the verification process. However, as opposed to Grumberg and Long's approach, the components involved are already verified and assumed to be correct. Our objective is to verify the interconnection of the components in a design, not the components themselves. Abstract models of the components can, therefore, be used to model the environment in the verification. The hierarchy of the abstract component models, in our case, is defined with respect to the interfaces of the components. This makes it relatively intuitive for designers to understand the behaviour of the particular abstraction in use. Grumberg and Long verify that components will work correctly in all environments, while we verify that the interconnection of several components will work correctly in the design at hand.

A common technique is to base abstraction and refinement, also in the context of model checking, on the simulation relation [6]. Clarke et al. [13] have proposed such an approach, where an abstraction of a design is refined by examining the diagnostic trace obtained from the model checker in case the verified property was unsatisfied. The diagnostic trace pinpoints which part of the abstracted model failed. If the failure is caused by a too abstracted model, the model is refined and the refined model is iteratively model checked. However, if the failure exists even in the unabstracted model, the result of the verification is returned to the designer. Our approach bears strong similarities with this work in the sense of the iterative diagnostic trace-guided refinement of abstract models of the system. However, our work takes the component-based nature of the designs into consideration and focuses on the abstraction and refinement of the interfaces between two or more components in such a framework. Section 6.4 describes this refinement approach in detail.

Schneider [32] proposes a formalisation of execution monitors. Such monitors watch that certain properties on the interfaces of a component are not violated. If violated, the monitor halts the execution. Ball and Rajamani have provided a realisation of this in SLIC, with which interface properties can be attached to C functions [7]. This approach, however, only investigates one particular execution trace, and hence cannot guarantee complete correctness. Our approach relies on formal techniques, which investigates the whole statespace and therefore can give such guarantee.

A slightly different, but related, issue is that of component matching described by Roop et al. [26]. The designer has a specification stating what functionality is needed at a certain place in the design. This specification has to be compared with the components in the repository. The problem amounts to determining if a certain component is able to implement the specification. The component may be able to do more than is required by the specification, but the specification must be a subset of what the component is able to perform. The proposed algorithm is based on the forced simulation relation [26], possibly augmented with a particular timing feature [27]. If the component is able to implement the desired behaviour, it is possible to automatically obtain a driver process which does the actual adaptation of the component. Consequently, we know by construction that this part of the system satisfied the specified set of properties. This is very different from our approach where component selection and interface generation are not performed automatically, and, consequently, the system is not assumed to be correct by construction. Our main concern is to check if the components and the designed glue logics are interconnected according to the requirements.

Assume-guarantee reasoning [12, 25] is not a methodology in the sense that properties are not explicitly verified. 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. However, this approach may easily lead to circular reasoning, in which case faulty conclusions might be drawn. Abadi et al. [1, 2] have furthermore developed a similar approach where the system is described as a set of temporal logic formulas, and composition is handled as conjunction. Our work can be used to perform the underlying verification task in these inference mechanisms. It does, however, not replace the inference mechanisms themselves.

In the component-based framework, which we assume, components may be designed by different providers. This situation might easily lead to a mismatch or incompatibility in their interfaces so that several such components cannot communicate with each other as intended. Therefore, glue logics have to be introduced between the components to adapt their interfaces in a way that they are able to properly communicate with each other. Such glue logic is too often designed and introduced ad hoc. Spitznagel and Garlan [34] present a systematic approach for designing glue logics which facilitates analysis of certain crucial issues, such as soundness, transparency and compositionality. Our work does not depend on whether the glue logics have been created using Spitznagel and Garlan's technique or with an ad hoc approach.

3 Methodology overview

In this paper, we consider systems which are built using predesigned components (IP blocks). Figure 1 illustrates such a system. In the figures throughout this paper, each component is depicted as a box with circles on its edge. The circles represent the ports of the component, which it uses for communication with other components. This notation conforms well with our design representation, introduced in Section 4.

The *glue logic* inserted between communicating components is depicted in Fig. 1 as clouds. For example, the interfaces of two or components connecting to each other may use different incompatible communication protocols. Thus, they cannot communicate directly with each other. For that reason, it is necessary to insert an adaptation mechanism between the components in order to bridge this gap. This adaptation mechanism would then, in this case, be the glue logic.

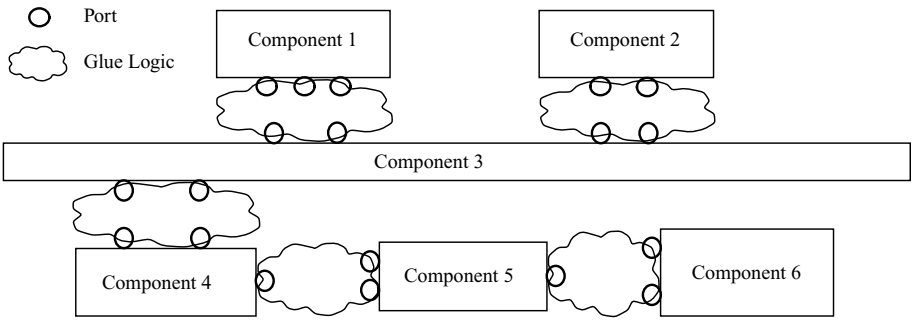


Fig. 1 Targeted system topology

3.1 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 two assumptions:

- The components themselves are already verified.
- The components have some requirements on their environment associated to them, expressed in a formal notation.

The first assumption states that the components themselves are already verified by their providers, so they are considered to be correct. What is still needed 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 by formulas in a formal temporal logic, 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 surrounding) to behave in order to work correctly. In this work, we use (timed) Computation Tree Logic, (T)CTL [4, 11] for expressing these requirements. However, other similar logics may be used as well.

CTL is a branching time temporal logic. This means that it can be used for reasoning about the several different futures (computation paths) that the system can enter during its operation, depending on which actions it takes. CTL formulas consist of path quantifiers (**A**, **E**), temporal quantifiers (**X**, **G**, **F**, **U**, **R**) and state expressions. Path quantifiers express whether the subsequent formula must hold in all possible futures (**A**), no matter which action is taken, or whether there must exist at least one possible future where the subformula holds (**E**). Temporal quantifiers express the future behaviour along a certain computation path (future), such as whether a property holds in the next computation step (**X**), in all subsequent steps (**G**), or in some step in the future (**F**). $p\,\mathbf{U}\,q$ expresses that p must hold in all computation steps until q holds. q must moreover hold some time in the future. $q\,\mathbf{R}\,p$ on the other hand expresses that p must hold in every computation step until q, or if q never holds, p holds indefinitely. State expressions may either be a boolean expression or recursively a CTL formula.

In TCTL, relations on time may be added as subscripts on the temporal operators (except the **X** operator). For instance, $\mathbf{AF}_{\leq 5}\,p$ means that p must hold before 5 time units and $\mathbf{AG}_{\leq 5}\,p$ means that p holds all the time during at least 5 time units.

$$\begin{aligned}&\mathbf{AG}\,((\mathit{status} = \text{disconnected} \vee \text{init}) \rightarrow \\ &\mathbf{A}\,[\mathit{status} = \text{connected}\,\mathbf{R}\,\neg\,\mathit{in} = \langle \text{send}, \_ \rangle])\end{aligned} \tag{1}$$

Equation (1) provides an example of a CTL formula capturing a constraint imposed by a component. This example, also shown in Fig. 2, is taken from a design using a connection-based protocol for communication. It consists of a component wanting to send a message to another component at regular time intervals. The sending component is not aware of the connection-based protocol and consequently all its messages must pass through a third component called Protocol Adapter. The protocol adapter implements the chosen protocol and was supplied and verified by a provider.

However, the protocol adapter needs an explicit command to connect to the receiving component, and messages to be sent must be preceded by a particular send command. Such

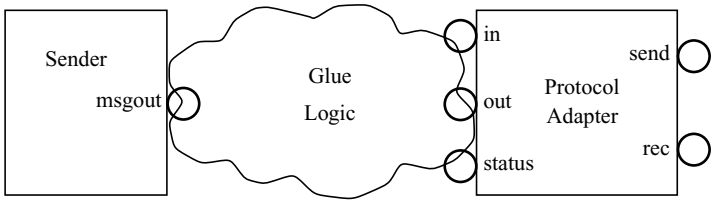


Fig. 2 A concrete example of a situation where the methodology can be applied

commands are received through port *in*. The adapter moreover provides the glue logic with information about the connection status through port *status*. Messages received by the protocol adapter are forwarded to the glue logic through port *out*. It is the task of the glue logic to supply the adapter with the appropriate commands and to take care of the messages forwarded by it.

Equation (1) is associated to the protocol adapter stating that it must always be true that if there either is a message in port *status* with the value "disconnected" or the system is in the initial state, then a message with value "send" may never occur in port *in* as long as there has not yet arrived a message in port *status* with value "connected". Intuitively, the formula states that *it is forbidden to send any message unless first connected*.

A set of such formulas is, as mentioned previously, associated to each interface of every component.

3.2 The verification process

The glue logic inserted between two components is to be verified so that it satisfies the requirements imposed by the connected components. Figure 3 illustrates the basic procedure. Model checking is used as the underlying verification technique. The model of the glue logic together with models corresponding to the interface behaviour of the interconnected components (called *stubs*) are given to the model checker together with the (T)CTL formulas describing the properties to be verified.

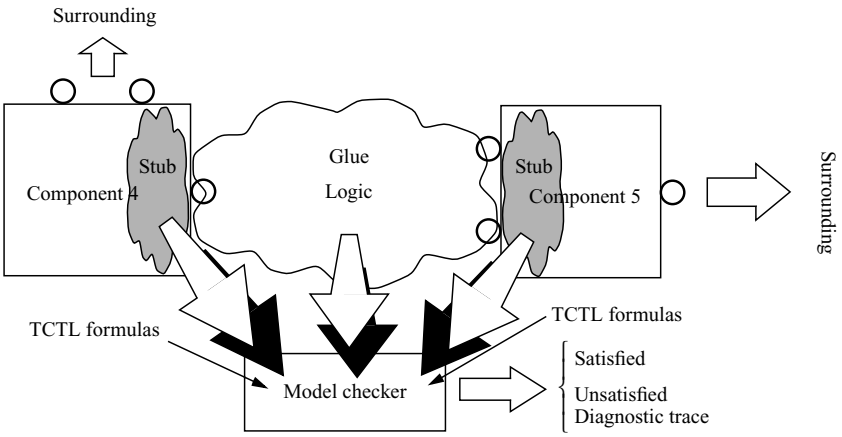


Fig. 3 Overview of the proposed methodology

A stub is a model which behaves in the same way as the component with respect to the interface consisting of ports connected to the glue logic under verification.

As a result of the verification, the model checker replies whether the properties are satisfied in the model or not. If they are not, the model checker provides a diagnostic trace in order to tell the designer what caused the properties to be unsatisfied.

As shown in Fig. 3, the part of the system not included in the verification of the particular glue logic is called the *surrounding* of the glue logic.

There are two main questions which must be answered:

- Where do stubs come from, and who models them?
- How can we incorporate the surrounding into the verification process?

These questions will be answered in detail in the subsequent sections. In general, it can however be said that stubs can originate from two different types of sources. Either they are given by the component provider, or they are generated by the designer, provided that a high-level model of the component is available. Section 6 assumes the situation where the stubs are given by the component provider. An important point to be made here is that, in this case, known properties of the surrounding are also included in this type of stubs. On the other hand, Section 7 presents an algorithm which automatically generates stubs from a model of a component. No aspects of the surrounding are, however, included in the stubs resulting from this procedure. If such aspects are needed to be considered for the verification of the glue logic, techniques outlined in Section 8 have to be applied.

4 The design representation: PRES+

In the discussion throughout this paper, as well as in the toolset we have implemented, the glue logic, the stubs and the components are assumed to be modelled in a design representation called *Petri-net based Representation for Embedded Systems* (PRES+) [14]. This representation is chosen over other representations, such as timed automata [5], due to its intuitivity in capturing important design features of embedded systems, such as concurrency and real-time aspects. Moreover, it clearly captures and exposes both the data and the control flow of the system, which is very important in the context of this work. The representation also induces graphically clear-cut interfaces between different components in the system. Together with the intuitively captured data and control flow, the dynamic behaviour on these interfaces can easily be followed and analysed. The representation is based on Petri-nets with some extensions as defined below. Figure 4 shows an example of a PRES+ model.

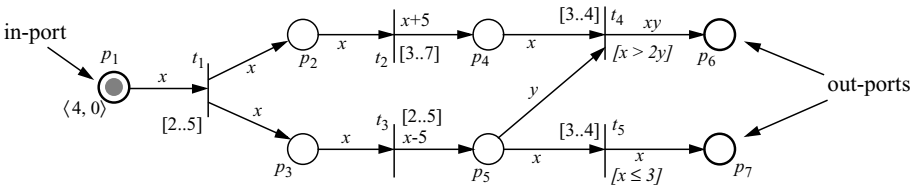


Fig. 4 A simple PRES+ net

Definition 1. PRES+ model. A PRES+ model is a 5-tuple $\Gamma = \langle P, T, I, O, M_0 \rangle$ where P is a finite non-empty set of *places*, T is a finite non-empty set of *transitions*, $I \subseteq P \times T$ is a finite non-empty set of *input arcs* which define the flow relation from places to transitions, $O \subseteq T \times P$ is a finite non-empty set of *output arcs* which define the flow relation from transitions to places, and M_0 is the initial *marking* of the net (see Item 2 in the list below).

We denote the set of places of a PRES+ model Γ as $P(\Gamma)$, and the set of transitions as $T(\Gamma)$. The following notions of classical Petri-nets and extensions typical to PRES+ are the most important in the context of this work.

1. A token k has values and timestamps, $k = \langle v, r \rangle$ where v is the value and r is the timestamp. In Fig. 4, the token in place p_1 has the value 4 and the timestamp 0. When the timestamp is of no significance in a certain context, it will often be omitted from the figures.
2. A marking M is an assignment of tokens to places of the net. The marking of a place P is denoted $M(p)$. A place p is said to be marked iff $M(p) \neq \emptyset$.
3. A transition t has a function (f_t) and a time delay interval ($[d_t^- \,..\, d_t^+]$) associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp of the new tokens is the maximum timestamp of the enabling tokens increased by an arbitrary value from the time delay interval. The transition must fire at a time before the one indicated by the upper bound of its time delay interval (d_t^+), but not earlier than what is indicated by the lower bound (d_t^-). The time is counted from the moment the transition became enabled. In Fig. 4, the functions are marked on the outgoing edges from the transitions and the time interval is indicated in connection with each transition.
4. The PRES+ net is forced to be safe, i.e. one place can at most accommodate one token. A token in an output place of a transition disables the transition.
5. The transitions may have guards (g_t). A transition can only be enabled if the value of its guard is true (see transitions t_4 and t_5).
6. 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 Fig. 4, ${}^\circ t_4 = \{p_4, p_5\}$ and $t_4^\circ = \{p_6\}$.
7. A transition t is enabled (may fire) iff there is one token in each input place of t, no token in any output place of t and the guard of t is satisfied.

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

Definition 2. Union. The union of two PRES+ models $\Gamma_1 = \langle P_1, T_1, I_1, O_1, M_{01} \rangle$ and $\Gamma_2 = \langle P_2, T_2, I_2, O_2, M_{02} \rangle$ is defined as $\Gamma_1 \cup \Gamma_2 = \langle P_1 \cup P_2, T_1 \cup T_2, \cup I_1 \cup I_2, O_1 \cup O_2, M_{01} \cup M_{02} \rangle$.

Other set theoretic operations, such as intersection and the subset relation, are defined in a similar manner.

Definition 3. Component. A component C is a subgraph of the graph of the whole system Γ such that:

1. Two components $C_1, C_2 \subseteq \Gamma$, $C_1 \neq C_2$, may only overlap with their ports (Definition 4), $P(C_1 \cap C_2) = P_{connect}$, where $P_{connect} = \{p \in P(\Gamma) | (p^\circ \subseteq T(C_2) \wedge^\circ p \subseteq T(C_1)) \vee (p^\circ \subseteq T(C_2))\}$.

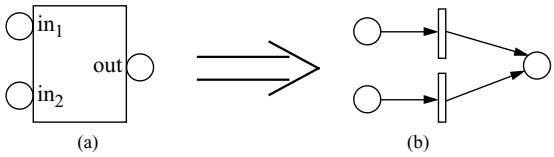


Fig. 5 Component substitution

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

Following Definition 3, a glue logic is formally also a "component". The reason that we call them differently is the way they are handled by the designer. A component acquired from a provider is, for instance, thoroughly verified and documented, whereas the glue logic is constructed by the designer himself and has to go through the verification process to see if it satisfies the imposed requirements.

Definition 4. Port. A place p is an out-port of component C if $(p^\circ \cap T(C) = \emptyset) \wedge (^\circ p \subseteq T(C))$ or an in-port of C if $(^\circ p \cap T(C) = \emptyset) \wedge (p^\circ \subseteq T(C))$. p is a port of C if it is either an in-port or an out-port of C.

Definition 5. Interface. An interface of component C is a set of ports $I = \{p_1, p_2, \ldots\}$ where $p_i \in P(C)$.

PRES+ can model the behaviour of a component at different levels of granularity. A component is generally drawn as a box surrounded by its ports, as illustrated in Fig. 5(a). Modelled in this way, it can be replaced with a PRES+ net as indicated by Fig. 5(b) without change in functionality.

The PRES+ representation is not required by the verification methodology itself, although the algorithms presented in this paper have been developed using PRES+. However, if another design representation is found more suitable, similar algorithms can be developed for that design representation.

In the context of PRES+, two forms of boolean state expressions are used within (T)CTL formulas: p and $p \Re v$. The expression p refers to a state where a token is present in place p. Expressions of the form $p \Re v$, on the other hand, describe states where also the value of a token is constrained. The value of the token in place p (there must exist such a token) must relate to the value v as indicated by relation $\Re$. For example, the expression $p = 10$ states that there is a token in place p, and that the value of that token is equal to 10.

In order to perform the model checking, the PRES+ model, as well as the (T)CTL properties, have to be translated into the input language of the particular model checker used. For the work in this paper, the models are translated into timed automata [5] for the UPPAAL model checking environment (http://www.uppaal.com/), using the algorithms described in [14]. The properties are also modified to refer to timed automata elements, rather than PRES+.

5 Glue logics

An example of a glue logic is provided in Fig. 6. The glue logic connects a component *Radar* to a component which implements a connection-based communication protocol. Component *Radar* emits a token containing radar information with an even time interval.

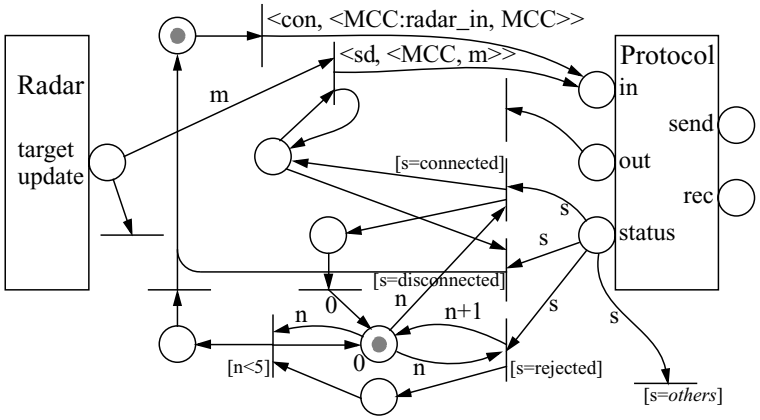


Fig. 6 A glue logic interconnecting two components

Since we use a connection-based protocol, a fact which component *Radar* is not aware of, the functionality related to establishing and maintaining a connection has to be implemented by the glue logic. In this and the following examples, the time delay intervals on the transitions are not shown for the sake of readability of figures. The transition connected to both the port *target_update* and the port *in* cannot be enabled until the protocol reported that it successfully has been connected. In this case, the token value ⟨sd, ⟨MCC, m⟩⟩ will be passed to the protocol component. 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 and the message itself. If, however, the connection failed, the glue logic will continue to attempt to connect, at most five times.

6 Verification of component-based designs

In this section, the theoretical framework underlying the verification methodology is presented. It gives formal definitions and presents important properties and relations, as well as some experiments. It is furthermore assumed in this section that the stubs are given by the component provider [21, 23].

6.1 Stubs

In Section 3.2, 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 6. 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.

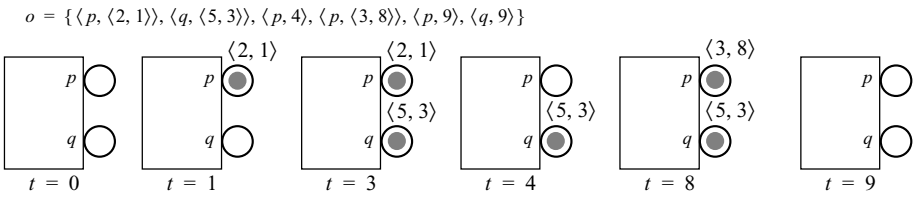


Fig. 7 Illustration of observations

Definition 7. Observation. An observation o is a set of events $o = \{e_1, e_2, \ldots\}$. Given observation o and an interface I, the *restricted* observation $o|_I = \{\langle p, k\rangle \in o | p \in I\} \cup \{\langle p, r\rangle \in o | p \in I\}$. An *input* observation *in* is an observation which only contains appearing events defined on in-ports and disappearing events defined on out-ports. An *output* observation *out* is an observation which only contains appearing events defined on out-ports and disappearing events defined on in-ports.

An observation is intuitively a sequence of events over time, where tokens appear and disappear in the places of the system. Exactly how and why the tokens appear and disappear is of no interest from the point of view of an observation. The events in an observation can be regarded as implicitly sorted with respect to their timestamps. Figure 7 illustrates the concept of observations according to Definition 7. 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 $t = 3$ 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 $o \mid_{\{p\}} = \{\langle p, \langle 2, 1\rangle\rangle, \langle p, 4\rangle, \langle p, \langle 3, 8\rangle\rangle, \langle p, 9\rangle\}$ and the one restricted with respect to $\{q\}$ is $o \mid_{\{q\}} = \{\langle q, \langle 5, 3\rangle\rangle, \langle q, 9\rangle\}$. Moreover, in this particular case, $o \mid_{\{p,q\}} = o$.

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 8. 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 nondeterminism. The set of all possible output observations *out* of C being the result of applying the input observation *in* to component C, is called the operation of component C from *in* and is labelled $Op_C(in)$. Given an operation $Op_C(in) = \{o_1, o_2, \ldots\}$ and an interface I of component C, the *restricted* operation $Op_C(in)|_I = \{o_1|_I, o_2|_I, \ldots\}$.

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

We are now ready to define what a stub is. In Section 3.2, stubs were described as a model of the behaviour of a component with respect to a specific interface. Ports belonging to other interfaces should be abstracted away.

Definition 9. 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 if and only if:

1. $I_S = I_C$.
2. For any valid input observation *in* of component C, $Op_C(in) \mid_{I_C} = Op_S(in \mid_{I_S})$.

The meaning of the expression on the left-hand side is the set of all possible output behaviours occurring in ports of I_C obtained by firing the PRES+ net of C given any possible valid input.

By applying the same input to S, restricted to the particular interface I_S, the set of behaviours denoted on the right-hand side of the equality sign is obtained from the output of S. 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 definition of stubs does not specify exactly how a stub with respect to a certain interface must be implemented. It only specifies its behaviour on that interface. Therefore, an interface might have several different stubs. Which of these stubs is chosen in a certain situation is of no theoretical importance as the interface behaviours are the same. From here on, for reasons of simplicity, an interface is considered to only have one stub.

The notion of *valid* input in Definition 9 is crucial. A valid input is an input satisfying all requirements which the component has on its environment. If these requirements are not satisfied, the component is not guaranteed to work correctly according to the discussion in Section 3.1. The definition only considers such input. Consequently, stubs obeying Definition 9 behave exactly like the corresponding component would have done when placed in a correctly designed system. In Section 7, this point will be relaxed.

Consider the Protocol Adapter component in Fig. 6. 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 8 shows how a stub for interface I of the Protocol Adapter might look like. Place *isconnected* contains a token with a boolean value indicating whether a connection is established or not. The value is initially *false*, which indicates that there is no connection. When the Protocol Adapter 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 a 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 Fig. 8 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 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

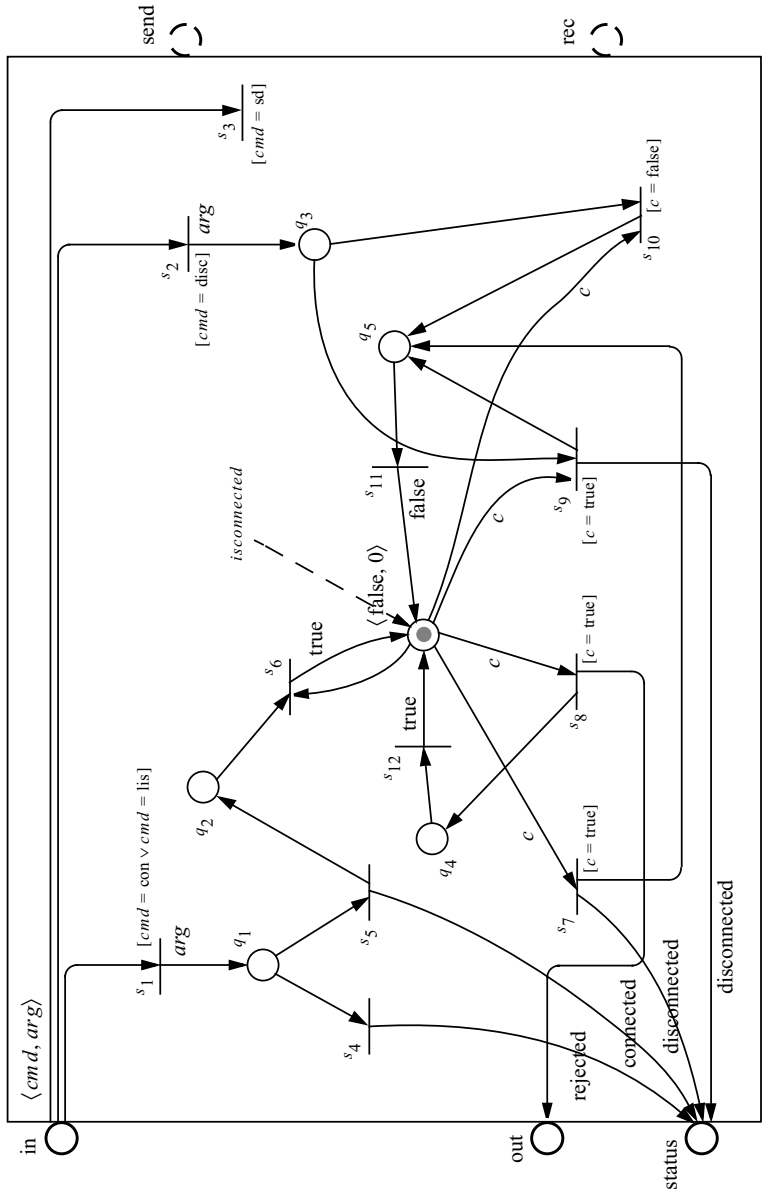


Fig. 8 A simple stub of the Protocol adapter

and s_{10}, depending on whether the Protocol was previously connected or not. Transition s_7 implements the case where the other party disconnects.

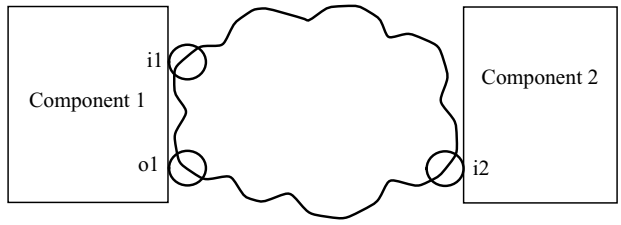
The introduced non-determinism naturally has some impact on the results of the verification process. A solution to this problem will be presented in Section 6.4 and Section 7. In addition, if the behaviour on the ignored ports needs to be constrained, techniques outlined in Section 8 have to be used.

6.2 Relationship between stubs

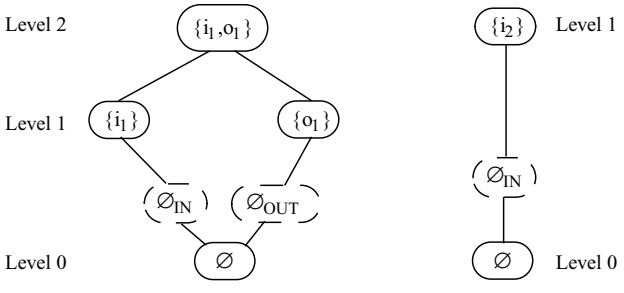
Components generally have several different interfaces. Since stubs are defined with respect to an interface, a component may have several different stubs as well. The stub in Fig. 8 is built with respect to the interface {*in*, *out*, *status*}, but the Protocol Adapter may also have, for instance, other stubs with respect to {*in*, *out*}, {*in*, *status*} or the single port interface {*status*}. Stubs can even be built for the interface {*in*, *status*, *send*}.

All potential stubs of a component may be hierarchically organised as a partial order (lattice) based on the subset relation of their interfaces. Such hierarchy has shown to be useful, as will be demonstrated in Section 6.4, in iteratively guiding the designer to choosing an appropriate stub. Figure 9(a) depicts two components with a glue logic in between. In Fig. 9(b), the relationships between the stubs of the two components are shown respectively.

Let us first look at the lattice of component 1. The lattice induces distinct levels of generality of the stubs. The top-level stub (the stub for the interface containing all ports of the component), with interface $I_{\max} = \{i_1, o_1\}$, exhibits exactly the same behaviour as its corresponding component. However, the implementation is not bound to be the same. In the bottom of the lattice, we have the empty interface, for which there does not exist any stub and



(a) Components and glue logic



(b) Interface lattices

Fig. 9 Components and corresponding interfaces

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

Fig. 10 The models of the empty stubs

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, $\emptyset_{IN}$, denotes the stub that consumes any token at any point in time. Similarly, the empty stub, $\emptyset_{OUT}$, denotes the stub that generates tokens with random values at any point in time. The models of these stubs are presented in Fig. 10. It is useful to introduce the notation $\emptyset_p$ to denote the empty stub at port p. Whether $\emptyset_p$ is equal to $\emptyset_{IN}$ or to $\emptyset_{OUT}$ depends on whether p is an in-port or an out-port.

Between $I_{\max}$ and $\emptyset$, 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 $\emptyset_{IN}$ and $\emptyset_{OUT}$, respectively. The empty stubs produce/consume tokens with random values and times with no regard to the possible behaviour of the actual 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 Fig. 8 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 *rejected* and *connected* as an answer to a *connect* request described previously in Fig. 8.

6.3 Verification environment

When a particular verification is going to take place, stubs must be selected from the lattices of the involved components. Each port connected to the glue logic must be covered by exactly one stub. Such a set of stubs is called *environment*. An environment, E, can be described by a set of interfaces, called *interface partition*, where each interface corresponds to a stub. If a port, p, does not exist in any interface of E, then the empty stub $\emptyset_p$ is added to p.

Definition 10. 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 5, all ports in the same interface must belong to the same component. For convenience, the set of all ports belonging to the interfaces in partition P is denoted $Ports(P) = \bigcup_{i \in P} i$.

$S = \{\{i_1\}, \{i_2\}\}$, $T = \{\{i_1, o_1\}\}$ and $U = \{\{i_1, o_1\}, \{i_2\}\}$ are all examples of interface partitions with respect to the components in Fig. 9. On the other hand, $V = \{\{o_1, i_2\}\}$ is not an interface partition, since the ports in the "interface" $\{o_1, i_2\}$ do not belong to the same component. Similarly, $W = \{\{i_1\}, \{i_1, o_1\}\}$ is not an interface partition either, since the included interfaces are not disjoint.

Definition 11. 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 $Env(P, J) = (\cup_{i \in P} S_i) \cup (\cup_{p \in J - Ports(P)} \emptyset_p)$ where each S_i is the stub for interface i, and $\emptyset_p$ is the empty stub attached to port p.

Let us consider the example in Fig. 9 and the partitions S, T and U again as defined above. For $J = \{i_1, o_1, i_2\}$, the environment corresponding to partition S, as defined above, ($Env(S, J)$) contains the stubs corresponding to the interfaces $\{i_1\}$, $\{i_2\}$ and the empty stub $\emptyset_{o_1}$. $Env(T, J)$ similarly contains the stub for $\{i_1, o_1\}$ and the empty stub $\emptyset_{i_2}$. $Env(U, J)$ does not contain any empty stub, since the interfaces in U cover all ports in J.

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*.

Environments can be organised hierarchically in a lattice, in a similar way as interfaces and stubs. This organisation is based on the precedence relation defined below. Figure 11 depicts the corresponding partition (environment) lattice.

Definition 12. 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. Intuitively, a partition P precedes partition Q if the interfaces in P are smaller, fewer and/or more fragmented than the corresponding interfaces in Q.

Considering the partitions S, T and U defined before, both $S \propto U$ and $T \propto U$, since every interface in S and T respectively is a subset of one interface in U. It is however not true that $S \propto T$, since $\{i_2\}$ is not a subset of any interface in T.

6.4 Formal verification with stubs

Having shown that there are many possibilities for choosing the stubs, i.e. the verification environment, for the verification problem at hand, a mechanism for helping the verifier making this choice is presented in this section, based on Theorem 1. A similar approach was also proposed by Clarke et al. [13]. The theorem is only valid for (T)ACTL formulas. Such formulas are (T)CTL formulas, which only have universal path quantifiers and negation may

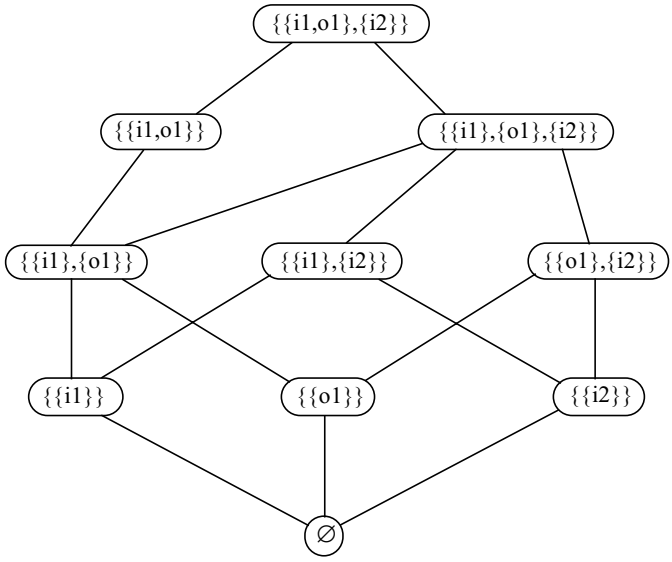
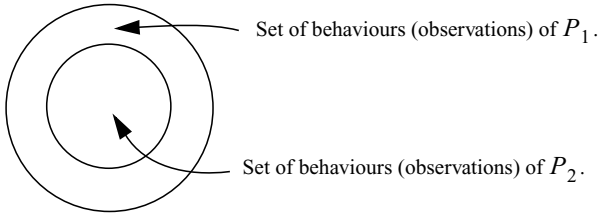


Fig. 11 Partition (environment) lattice of the situation in Fig. 9

Fig. 12 Illustration of Theorem 1



only occur in front of atomic propositions. (T)ACTL formulas express requirements in the sense that a certain condition must always hold or it must never hold.

Theorem 1. *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) Env(P_1, J), then it is also true that $M_0 \models \mathbf{AF}\,\phi$ for component Env(P_2, J).*

The intuition behind this theorem is that when verifying a property expressed as (T)ACTL, if it is satisfied when using a verification environment at a low level in the lattice, it is guaranteed to be satisfied also in higher levels, according to the precedence relation in Definition 12.

Figure 12 illustrates the intuition behind the proof of the theorem. The area inside the outer circle corresponds to the set of behaviours ($Op_{Env(P_1,J)}$) of $Env(P_1, J)$. According to Definition 11, this operation is produced by the union of all stubs corresponding to the interfaces in P_1. The area inside the inner circle similarly corresponds to the set of behaviours ($Op_{Env(P_2,1)}$) of $Env(P_2, J)$. This set is a subset of the first one 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 restriction to (T)ACTL formulas,

as opposed to an arbitrary (T)CTL formula, becomes obvious, since (T)ACTL formulas express properties which must always hold. This is due to the fact that they only contain universal path quantifiers. An arbitrary (T)CTL formula cannot guarantee that the property holds for all behaviours, only that there is at least one behaviour satisfying it. In contrast to (T)ACTL, arbitrary (T)CTL formulas are able to express possibility, e.g. it must be possible for p to occur in the future (**EF** p). A formal proof of the theorem is presented in Appendix A.

As a consequence of Theorem 1, the designer can start the verification process using stubs at low level. If the design satisfies the properties using this environment, the property is also satisfied for the complete design according to Theorem 1. However, if the property was unsatisfied, this does not mean that the property necessarily is unsatisfied in the complete design. The reason could be that the environment was able to produce to many excess behaviours. In this case, the verification has to be continued using higher level stubs. The diagnostic trace may help the designer in finding an appropriate stub. If the property is unsatisfied even with top-level stubs, it is not satisfied in the particular system.

As indicated by the experimental results presented in the next section, using low-level stubs leads to shorter verification times. For this reason, it is a good idea to start the verification with such stubs. The theorem also provides a means of verification in the case that top-level stubs, of which the behaviour is identical to the behaviour of the full component on the specified interface, are not given by the component provider and consequently are not available.

6.5 Experimental results

The following example demonstrates the use of alternative sets of stubs for verification of a split transaction bus (STB) in a multiprocessor DSP [3]. An overview of the system is shown in Fig. 13. Each processing element contains one 32-b V8 SPARC RISC Core with a co-processor and reconfigurable L-1 cache memory. The STB consists 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 acknowledgment 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 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

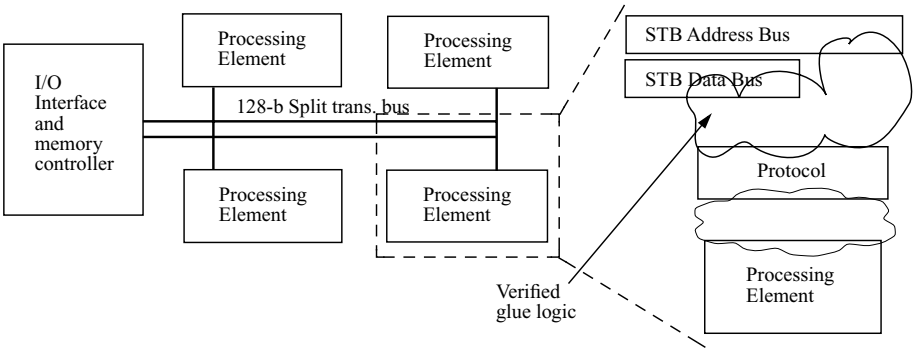


Fig. 13 Schematic view of the STB example

Table 1 Experimental results for the STB example

	Partition											
Property	1	2	3	4	5	6	7	8	9	10	11	12
A	F 0.41	F 3.28	F 0.34	F 162	T 156	F 345	F 330	F 68.2	T 17.7	F 636	T 30.4	T 12.6
B	T 0.14	T 0.41	T 0.16	T 17.6	T 24.8	T 16.9	T 23.6	T 1.69	T 1.38	T 26.9	T 1.54	T 1.29
C	F 0.23	F 0.74	F 0.23	F 19.7	F 29.7	F 18.6	F 28.8	F 3.25	F 3.27	F 32.7	F 4.09	F 4.01
D	F 0.38	F 0.89	F 0.37	F 129	F 45.9	F 97.7	F 313	F 20.1	T 3.32	F 292	T 10.2	T 7.04
E	T 0.20	T 0.58	T 0.21	T 28.1	T 54.2	T 29.2	T 48.9	T 2.80	T 1.20	T 53.3	T 4.48	T 4.39
F	F 0.34	F 0.68	F 0.31	T 18.7	T 26.2	T 16.5	T 25.2	F 6.51	F 2.85	T 28.8	T 1.76	T 1.36
G	F 0.41	T 0.43	F 0.44	T 18.5	T 26.3	T 17.0	T 26.7	T 2.47	T 0.94	T 30.0	T 2.36	T 1.94
H	T 0.21	T 1.30	T 0.22	F 167	F 438	F 344	F 325	F 66.4	F 11.9	F 689	F 87.2	F 38.0

F = property is unsatisfied, T = property is satisfied. Verification times are given in seconds.

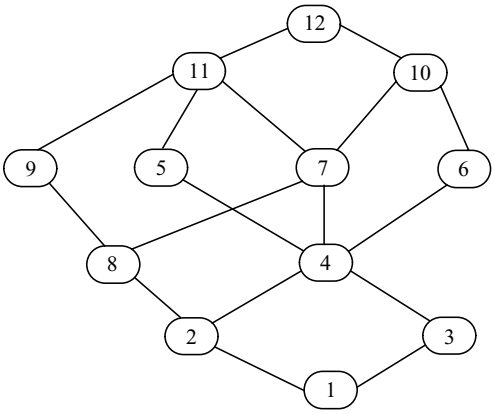


Fig. 14 Partition lattice in the STB example

and the same procedure continues until the whole block of data has been transmitted. One functionality of the glue logic being verified, see Fig. 13, is to deliver messages from the protocol to the correct bus. Another aspect is to process the results and acknowledgments so that they can correctly be treated by the protocol. For instance, in the case of an identifier broadcast, the protocol component expects two different commands from the address bus, depending on which of the following two situations occurred: (1) the protocol component currently in hold of the address bus is the component connected to this particular glue logic or (2) the broadcast is the result of another component proposing an identifier.

Table 1 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 Fig. 14 is not the full lattice but 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, i.e. Address bus, Data bus and Protocol. Environment 1 consists of only level 1 stubs on out-ports. Verification results marked with *T* mean that the verified property was satisfied, and conversely verification results marked with *F* mean that the property was unsatisfied in the particular environment. The numbers indicate the verification time in seconds.

In order to give a better understanding of the properties, we will have a closer look at two of them. Property B, for instance, is concerned 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 $\mathbf{AG}\,(rec \rightarrow rec \neq \langle \text{TRAN}, a\rangle \wedge a \neq \text{this\_component})$ 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, $\mathbf{AG}\,((addr.out = \text{ACK}) \rightarrow \mathbf{AF}\,addr.in = \text{drive\_addr})$, states that 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 satisfied at all in the system. That is why the verification result for that property is false, no matter which environment is used. On the other extreme we find properties B and E which are satisfied even with the lowest level environment. Hence, being expressed as an ACTL formula, the property is satisfied with any environment. Property H is not an ACTL formula and can hence not be expected to behave according to the same pattern. It is satisfied when verified with low-level stubs, but is not satisfied with high-level stubs. Property G, also expressed as an ACTL formula, is also satisfied. This can be verified by using the top-level environment, but also by verifying with environment 2. According to Theorem 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. The verification time with different environments is in the range 0.14–689 s. 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.

7 Automatic stub generation

Section 6 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, and the property to be verified is expressed as a (T)ACTL formula, other stubs at lower level can be used instead. An alternative situation is that a PRES+ model of the component is available, but no particular stubs. In this section, 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. According to the discussion in Section 3.2, here, we assume that we do not know anything about the surrounding environment, as opposed to Section 8 where we will add explicit knowledge about the surrounding, where needed.

The example component in Fig. 15 will be used to explain and analyse the stub generation algorithms in this section. In all cases, a stub for the marked interface $\{p_1, p_2\}$ will be generated.

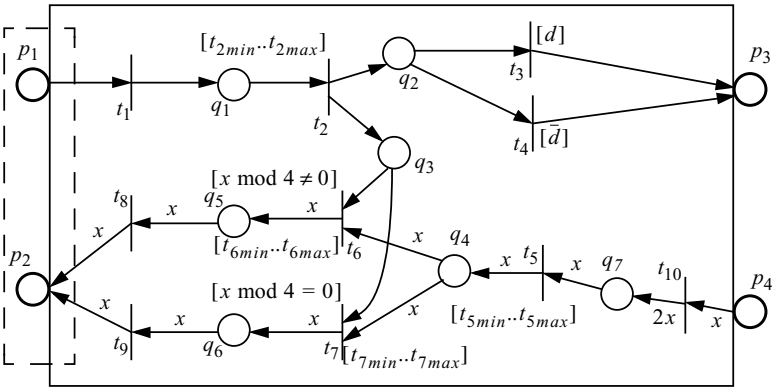


Fig. 15 Example of a component for stub generation

7.1 Pessimistic stubs

The stub definition presented in Section 6.1 (Definition 9) is quite strict, requiring equality between the operations of the component and stub. That strictness makes it very difficult to automatically create stubs. Definition 13, relaxes that definition:

Definition 13. 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. $I_C = I_S$
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 has a more "pessimistic" view of the component behaviour. 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 9 are in this section called *exact stubs* to differentiate between the two types.

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

Theorem 2. *Assume two environments E_1 and E_2 of the same set of components and $Op_{E_1} \subseteq Op_{E_2}$, an initial marking M_0 and a (T)ACTL formula, e.g.* $\mathbf{AF}\,\phi$ *expressed only on the ports of the stubs in E_1 and E_2. If $M_0 \models \mathbf{AF}\,\phi$ for component E_2, then it is also true that $M_0 \models \mathbf{AF}\,\phi$ for component E_1.*

The intuition behind this theorem is the same as in the case of Theorem 1. 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. A formal proof is presented in Appendix B.

Theorem 2 allows us to use pessimistic stubs when verifying (T)ACTL formulas. The behaviours of the exact stub (see Definition 9) are also produced by the pessimistic one

which, however, produces additional behaviours. This fulfils the assumptions of the theorem. So, if a property is satisfied using 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.

7.2 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 13, 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 discussion around Definition 9). The resulting stub is shown in Fig. 16.

To verify a design using naïve stubs could be quite time consuming, as shown by experiments. For this reason, an algorithm generating smaller stubs reducing verification time has been developed and is presented in the following sections.

7.3 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 and selected to be included in the stub, the parts of the model which were excluded must be compensated for. Apart from the pessimistic assumptions about the surrounding, this is an additional point where pessimism is introduced in the stub.

Hence, the stub generation algorithm consists of the following three parts:

1. Dataflow analysis
2. Identification of stub nodes

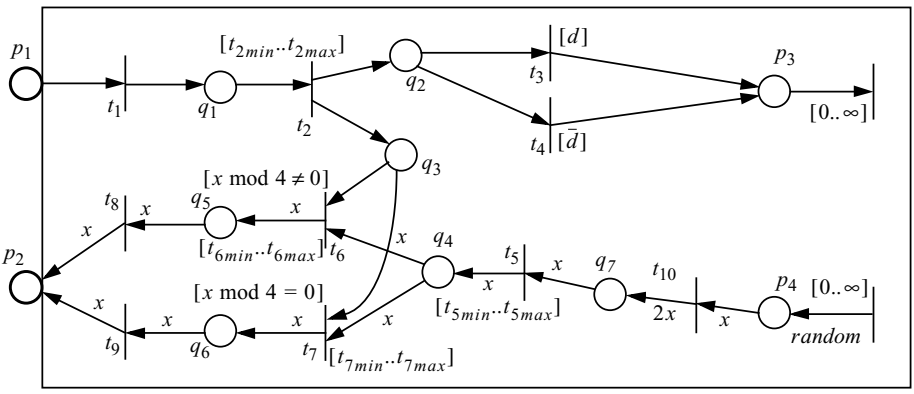


Fig. 16 A naïve stub of the component in Fig. 15

3. Compensation for the excluded parts of the component

Each of these three parts is discussed below. A more detailed description of these algorithms can be found in [22].

7.3.1 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 based on depth-first search on graphs. The procedure is called once for each port in the interface of the stub. For in-ports, the search is performed along the dataflow, whereas for out-ports it is performed against the dataflow. During the search through the graph, each node (place or transition) is marked with the node previously visited so that it is possible to obtain the path from an arbitrary node in the component to a port in the interface. The dataflow marking, obtained from the search, must consequently not only be able to distinguish the paths but also indicate the ports to which they lead. The dataflow marking is stored in a data structure for later use. The data structure associates a place or transition together with the original port from which the particular depth-first search started, to a set of neighbouring places or transitions which were visited immediately before the node just being visited.

Figure 17 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 and p_2. 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. A dataflow marking is, intuitively, the set of arrows (maintaining their types) associated to a node obtained from the search.

7.3.2 Identification of stub nodes

The parts of the component to be included in the stub are identified using the dataflow marking. The identification procedure is based on a depth-first search, starting from each port *not* belonging to the specified interface. The search continues until a so called *separation*

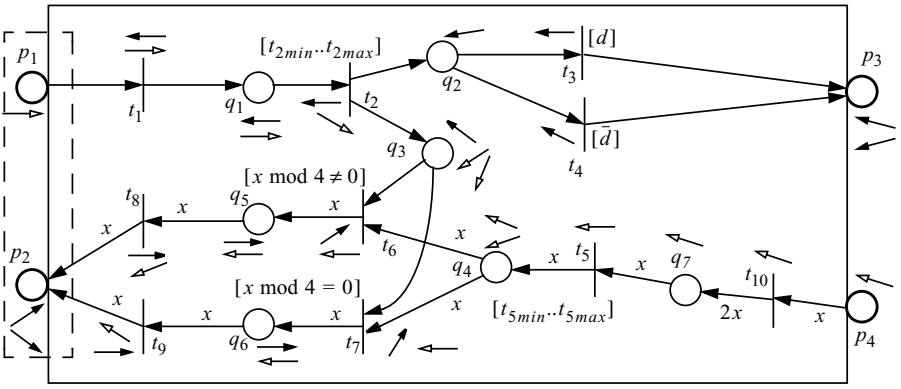


Fig. 17 The dataflow marking of the component in Fig. 15

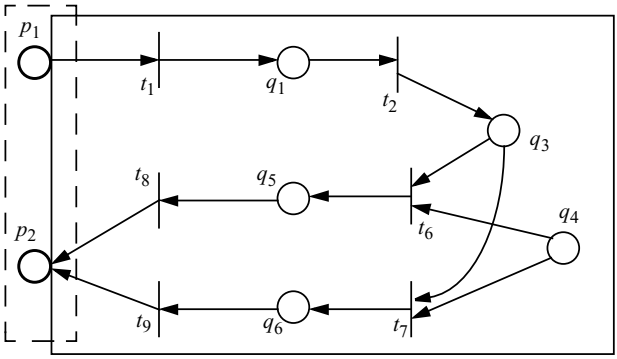


Fig. 18 The places and transitions in the automatically generated stub

point is found. A separation point 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. In principle, a separation node is the first node encountered which has dataflow arrows of several different types pointing in different directions. Such a node indicates influence from several different ports. For example in Fig. 17, transition t_2 is such a separation point. More details regarding the exact definitions and algorithms can be found in [22].

Once a separation point is found, a new depth-first search is started from that point, following the dataflow marking. Every node visited by the algorithm is added to the final stub. Figure 18 shows the stub resulting from this procedure.

7.3.3 Compensation

As can be seen in Fig. 18, some transitions in the stub do not have all their input or output places any more, as compared to the full component (Fig. 15). This means that they will not deliver (receive) all needed output (input). Similarly, some places do not have all input or output transitions. Consequently, some mechanism has to be developed in order to compensate for these missing places and transitions. This is done by introducing non-determinism. Four cases can be identified along two dimensions: place without input transition (join place), place without output transition (fork place), transition without input place (join transition) and transition without output place (fork transition). Their different compensation methods are described below:

1. Join places lack one or more input transitions in comparison with the full component. These transitions are compensated by adding them to the stub and further handle them as join transitions. For efficiency, all these transitions can be merged into one.
2. Fork places are processed analogously by adding the missing output transitions to the stub, and then handle them as fork transitions. For efficiency, these transitions can be merged into one.
3. Join transitions lack one or more input places. To compensate for this, the transition function is modified so that it may produce any random value still consistent with both its existing input values and its guard. Assume, for example, that the function of a certain join transition is $2x$ and that the place corresponding to x is missing. Assume further that the transition has a guard stating that x must be an even value. After compensation, the transition will have a random function only producing values divisible by 4. Besides

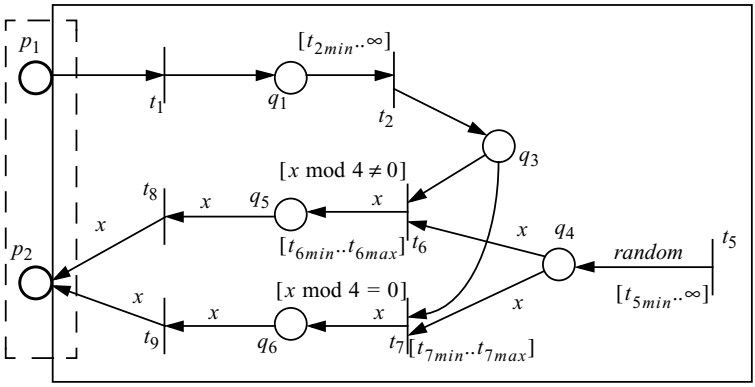


Fig. 19 An automatically generated stub

modifying the transition function, the upper bound of the time delay interval must be changed to infinity, to reflect the fact that the enabling time of the transition is unknown.

4. Fork transitions, on the other hand, lack one or more output places. Similar to join transitions, the enabling time of fork transitions is unknown, due to the forced safe property of PRES+. Tokens in an output place of a transition may consequently disable the transition. For this reason, the upper bound of the time delay interval is set to infinity. Both guard and function are kept intact.

In Fig. 18, transition t_2 has one output place not included in the stub. It is consequently a fork transition. According to the discussion above, there might be a token in the missing output place disabling t_2 forever. The upper bound of the time delay interval of that transition is therefore changed to infinity as indicated in Fig. 19. Similarly, place q_4 is a join place since it lacks its input transition, t_5. Hence, it is never able to receive any tokens. Thus, all transitions with q_4 as an output place are also added to the stub. The functions of these (join) transitions, t_5 in our example, are the same as in the full component, except that all missing arguments are considered random. The upper bound of the time delay interval of the transition is moreover set to infinity, since there is no guarantee that the transition will ever become enabled. Figure 19 shows the final result of the automatic stub generation algorithm.

7.4 Pessimism reduction

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 of the component which were excluded in the stub generation to the stub. However, in the general case, the designer does not have any detailed knowledge about the internals of the component and its stubs, so this procedure cannot be done 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 itself be a function, which is the most general approach, or defined by a stored table.

What the designer must know in order to use the component is, normally, not more than what is stated in the user documentation of the component, i.e. information regarding 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. The unwanted behaviour falls into one of the following three categories:

1. The unwanted behaviour is causal. The value itself is allowed at the particular port, but not in that particular ordering compared to other values. This case is not a matter of reducing pessimism, but it is a sign that the interface for which the stub was generated does not cover enough ports. The solution to this problem is, consequently, to generate another stub, using the algorithm described in Section 7.3, so that all relevant ports are included in the interface of the stub. If at least one such port connects to the surrounding, and not to the glue logic under verification, this solution may be combined with techniques described in Section 8 which allow the inclusion of certain assumptions on the surrounding into the verification process.
2. Late arrival or absence of tokens. Firing delays are overestimated with infinity in the stub generation algorithm. The reason for this was that there is no guarantee that those 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 given these assumptions. However, this problem can be solved by including certain assumptions on the surrounding into the verification process, as discussed in Section 8.
3. Incorrect values in a port. A value which appears in a port is not allowed to appear in that port in the complete component. This is a situation which is due to the pessimistic nature of the stub. The reason for the unwanted value is that there might be a transition in the part of the component not included in the stub, whose transition function cannot produce the particular value in question. But, since the excluded part was compensated with a random function, that value might still appear in the resulted stub.

Thus, pessimism reduction of stubs is applied in case 3, when there is a value in a port of the interface which cannot occur in that port in the full component. The pessimism is reduced by iteratively adding transitions and places, which were removed by the stub generation algorithm, until the unwanted value is eliminated. The value will be eliminated when the transition whose function cannot produce the particular value is added. It might happen that the unwanted value cannot be eliminated. The reason for this might either be a design error in the component, or that a too general surrounding was assumed, that feeds the component with values with which the component is not guaranteed to work correctly. In the latter case, it is necessary to make an assumption about the surrounding so that such unwanted values do not occur as an input. As a by-product of the pessimism reduction algorithm, the port on which the assumption has to be made is identified, and the specification of the component provides enough information as to which values to exclude from that particular port. The complete algorithm is presented in [22].

8 Inclusion of the surrounding into the verification process

Sometimes, it might be the case that the information captured by the models of the components or stubs is not sufficient to perform the verification. These are cases in which the correct

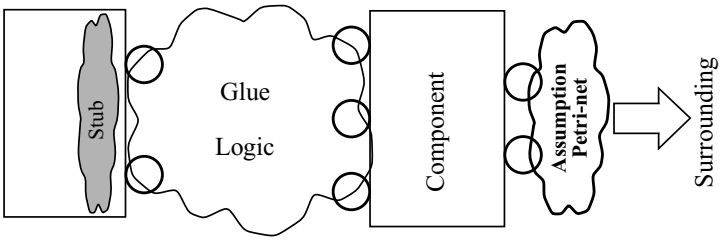


Fig. 20 Including the assumptions on the surrounding into the verification process

functionality of the involved components depends on certain assumptions relative to those inputs which are connected to the surrounding and, thus, have been ignored in the verification process so far. Hence, only considering models of the components may not be sufficient. We have to consider the constraints on the other interfaces. These constraints are also expressed as ACTL formulas. In order to solve this problem, we generate an "assumption Petri-net" which is connected to the component as demonstrated in Fig. 20. In this way, the additional constraints are incorporated into the verification process together with the component.

An algorithm which generates a model that is able to produce all possible events consistent with a particular ACTL formula, has been developed [22]. A model, consistent with the properties on the interfaces connecting to the surrounding, is thus created using this algorithm and attached to the model under verification, as indicated in Fig. 20.

The fundamental idea behind this algorithm is to find the maximum set of behaviours or states which do not violate the particular ACTL formula at hand [18]. A PRES+ model corresponding to this set of behaviours is then constructed. Let us outline the algorithm considering the example ACTL formula in Eq. (2). It states that if there is a token in port p, the value of that token must be less than 2.

$$\mathbf{AG}\,(p \rightarrow p < 2) \tag{2}$$

It is found that there are two states which satisfy the example formula. They are described by Eqs. (3) and (4) respectively. Either there is no token in p and the property continues to hold (Eq. (3)), or there is a token in p with a value less than 2 and the property continues to hold (Eq. (4)).

$$\neg p \wedge \mathbf{AXAG}\,(p \rightarrow p < 2) \tag{3}$$

$$p < 2 \wedge \mathbf{AXAG}\,(p \rightarrow p < 2) \tag{4}$$

In this simple example, there is only one way in which the property may continue to hold ($\mathbf{AXAG}\,(p \rightarrow p < 2)$). Except for the port p, the resulting model only has one place, corresponding to this future behaviour. In general, there is one place per such future behaviour. Cases with multiple possibilities of future behaviour arise, for instance, when several temporal operators occur in the formula.

Figure 21 shows the resulting model. It contains one place corresponding to the port p and another place (s_1) corresponding to $\mathbf{AXAG}\,(p \rightarrow p < 2)$.

For each place (possible future behaviour), transitions must be added for each possible state. There are two such states in the example, Eqs. (3) and (4). For Eq. (3), no transition needs to be added since it says that p should *not* have any token. The only action to take would

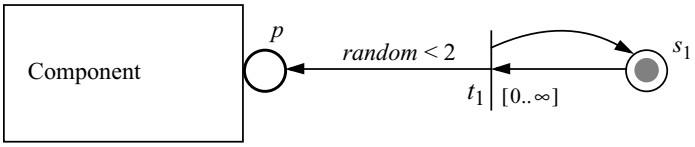


Fig. 21 The PRES+ model corresponding to the example formula in Eq. (2)

be to stay in the same place. For Eq. (4), a token must be put in place p with a random value less than 2, as performed by transition t_1. A token must also be put in the place corresponding to the continuation, in this case the same place. There is no time requirement on this action. Consequently, the transition may fire at any time, as reflected by time delay interval $[0\,..\,\infty]$. An initial token is finally added to the place.

9 Example and verification results

The presented verification methodology gives a powerful means to verify large systems using a divide and conquer approach. This section provides an example to demonstrate how a system can be verified using the methodology.

9.1 The mobile telephone system

The model used as an example is a mobile telephone. Figure 22 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 [23].

1. Microphone. The microphone sends digital voice data to the transmitter.
2. Buttons. When dialling, the buttons component sends to the controller information about which buttons were pressed.
3. Speaker. The speaker receives voice digital signals from the receiver and converts them to sound.
4. Display. The display shows on a small screen information sent to it by the controller.

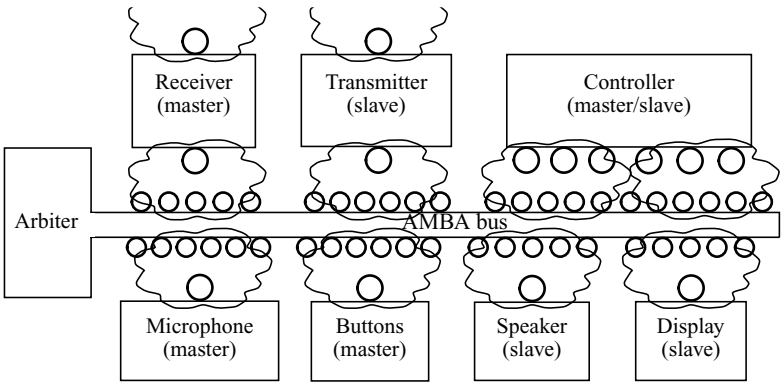


Fig. 22 Overview model of the example system, a mobile telephone

(a) Buttons (b) Display

Fig. 23 Models of the components Buttons and Display

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. Each component itself has previously been verified.

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

9.1.1 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 Fig. 23.

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 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 $\{0\ldots9, \text{enter}\}$". 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 $[0\,..\,0]$.

9.1.2 Controller

The controller component keeps track of what is happening in the system and acts accordingly. Figure 24 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 distinguishes between pressing a number, $b \in \{0\ldots9\}$, and pressing "enter", $b = \text{enter}$.

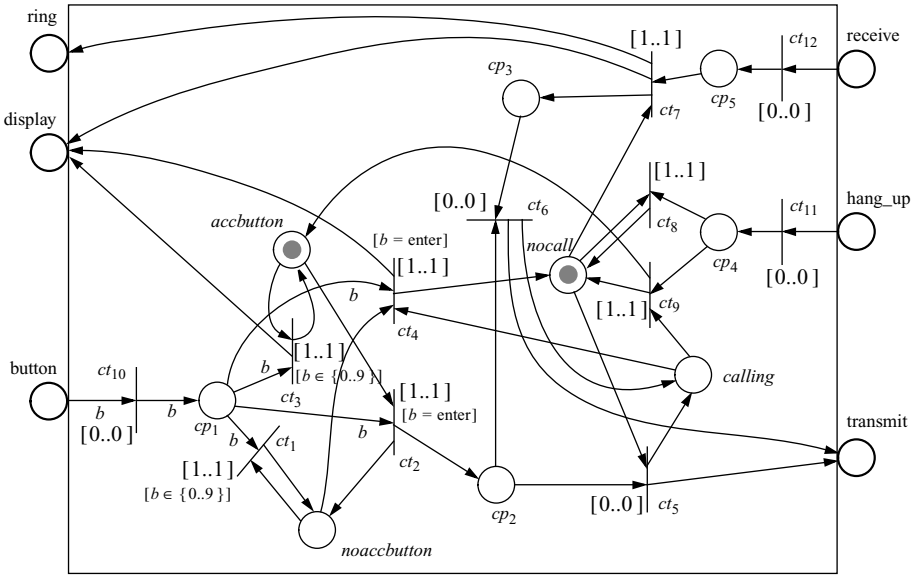


Fig. 24 Model of the Controller component

When dialling a number, signals (tokens) are also sent 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.

9.1.3 AMBA Bus

All components communicate through an AMBA bus [29]. 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 22 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 putting a token in the port HREQBUS. The arbiter will eventually grant access (token in port 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 putting a token in HTRANS.

If a slave is not ready to receive, it is able to put the transaction on hold, or in AMBA bus terms *split* (token in port HRESP), until it eventually becomes ready (token in port 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. 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.

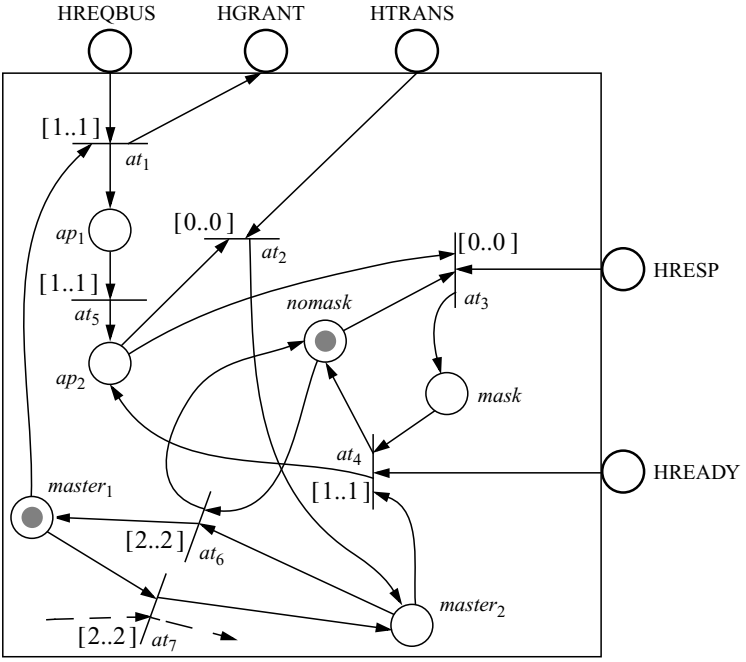


Fig. 25 Model of the Arbiter component

Figure 25 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 a request has not arrived, transition at_6 (or at_7, depending on which master currently holds the token) fires and the next master gets a chance to be granted access to the bus. 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 26 shows a model of the Bus component. All transitions have time delay interval [0..0], since we assume that all message passing on the bus is instantaneous. If another assumption is made, the transitions in the Bus component may be changed to another time interval. The transition function is identity, since the bus should only forward messages without altering them.

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

9.1.4 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 Fig. 27. 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 address is stored during one clock

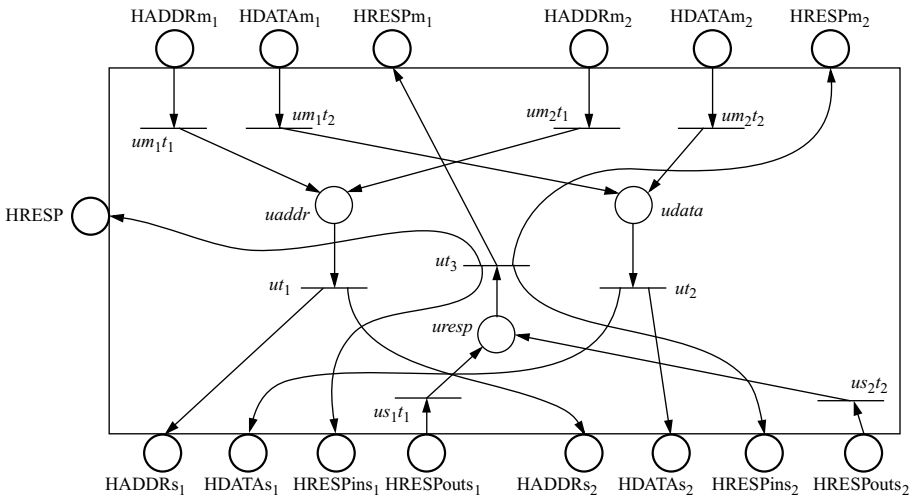


Fig. 26 Model of the Bus component

cycle in cmp_7 and the data in cmp_9. After one clock cycle the stored items are removed by cmt_{11} and cmt_{14} respectively. When the master is regranted access to the bus, transitions cmt_{12} and cmt_{15} become enabled and resend the data. The tokens, however, stay in their respective places in case the resent data is again split.

Meanwhile a transaction is split, no new data can be sent by the component. Presence or absence of tokens in $cmanosplit_1$ and $cmdnosplit_1$ regulate this behaviour.

The glue logic may receive tokens from HRESP even though its master did not send anything. This can be the result of a transaction of another master being split. Remember that the bus distributes split requests to all connected components. In 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 a message and if the receiving slave issues a split request, it is consequently intended for this particular master. 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 Fig. 28.

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 HRESP*out* to indicate that the transaction is split, (transition cst_7). Furthermore, the address and data received at the same 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).

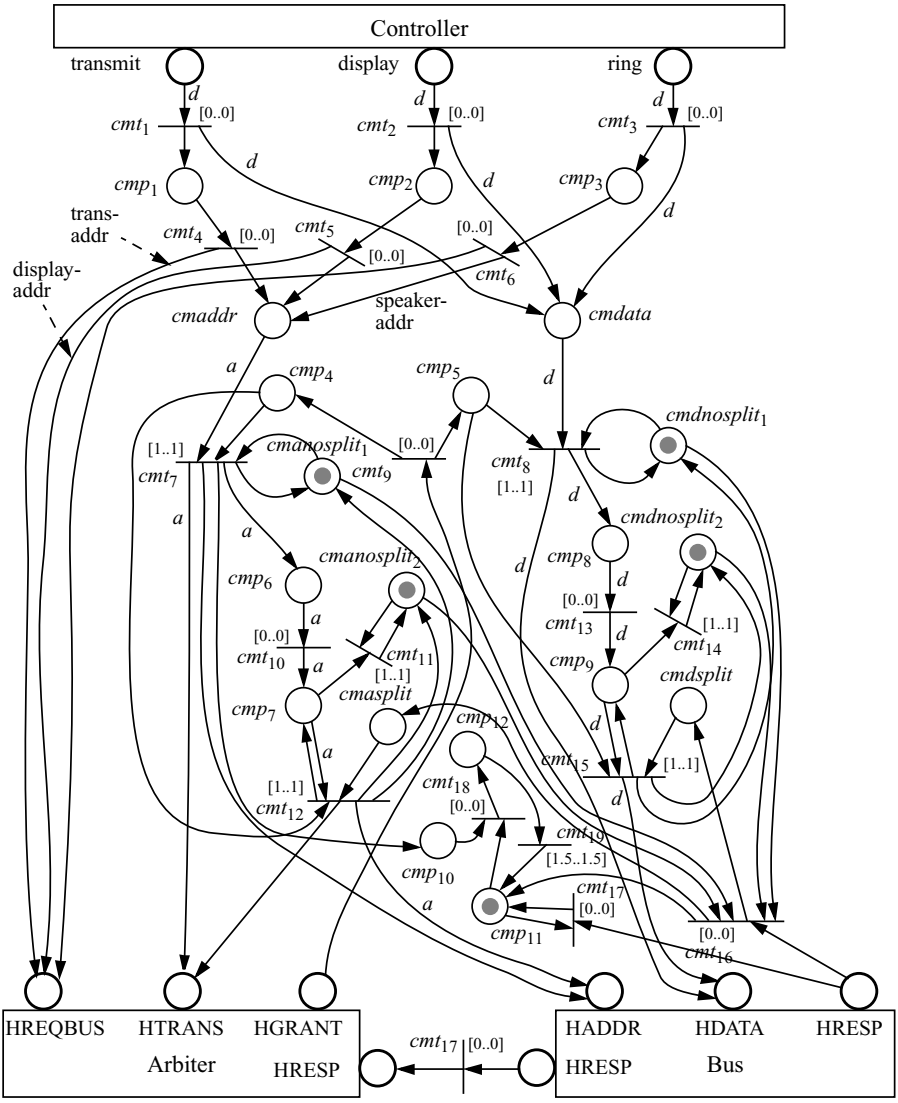


Fig. 27 A model of the glue logic for the master functionality of the controller

9.2 Verification of the model

Interesting properties to verify can be found in the specification of the system. In our example, a subset consisting of the following three properties, was verified:

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)

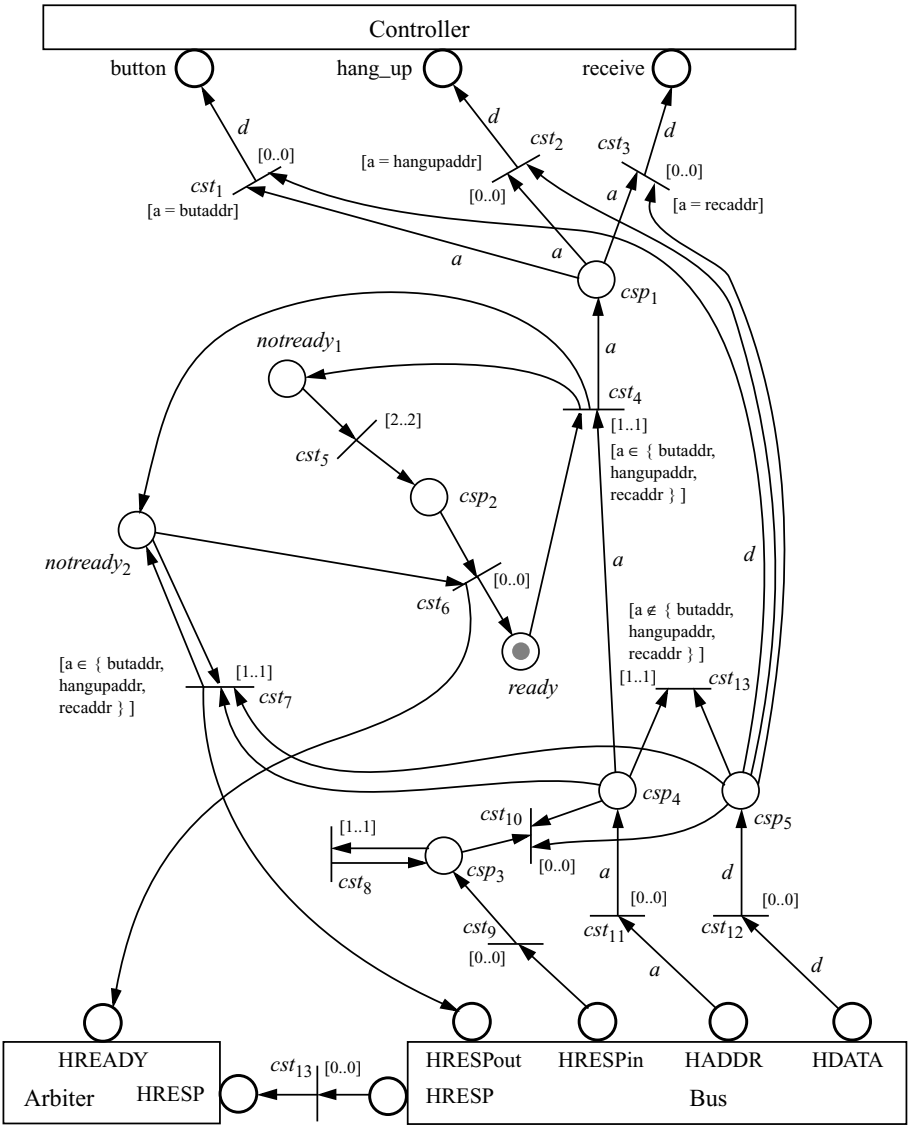


Fig. 28 A model of the glue logic for the slave functionality of the controller

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

9.2.1 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,

Table 2 Verification results of property 1

Step	Environment	Res	Time
Initial	All empty stubs, except bus generated	False	1.32 s
Add assumption on HDATA	All empty stubs, except bus, assumption	True	125.33 s
Verify assumption	Buttons top-level stub, other stubs empty	True	7.58 s

arbiter, bus and the slave functionality glue logic. Table 2 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 s to obtain this result.

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. Although, the pessimism reduction algorithm failed, it was able to trace the undesired value to port HDATAm_x. An assumption was made on that port stating that only data in the set $\{0 \ldots 9, \mathit{enter}\}$ can occur. The property is formally given in Eq. (5).

$$\mathbf{AG}\,(\mathrm{HDATA}\,m_x \rightarrow \mathrm{HDATA}\,m_x \in \{0 \ldots 9, \mathrm{enter}\}) \tag{5}$$

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 min 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 Eq. (5) 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 7.58 s.

9.2.2 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 3 presents the result of the different stages in the verification process.

Table 3 Verification results of property 2

Step	Environment	Res	Time
Initial	All empty stubs, except {HRESP*in*, HRESP*out*, HRESP}	False	2.47 s
Use higher level stubs	Initial except Level 1 stubs for HADDR, HDATA	False	28.39 s
Correct design error	Initial except Level 1 stubs for HADDR, HDATA	False	87.57 s
Use higher level stubs	Empty stubs for controller and top-level stub for the bus	True	246.14 s

Table 4 Verification results of property 3

Step	Environment	Res	Time
Initial	All empty stubs	False	0.14 s
Use higher level stubs	Initial except arbiter stub	False	0.52 s
Add property 2 as assumption	Arbiter stub given, button stub empty, bus with assumption	False	2.58 s
Correct design error	Arbiter stub given, button stub empty, bus with assumption	True	2467.42 s

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 a situation where there were tokens in all of $notready_2$, csp_4, csp_5 and csp_2. Both transitions cst_6 and cst_7 were consequently enabled. Following the trace, cst_7 was fired, thereby falsely splitting the transmission. The desired behaviour would have been to fire cst_6. Consequently, a design error in the glue logic was found. This fault was finally fixed, after detection, by changing the time delay interval of transition cst_6. After fixing the error, the property was reverified using the very same environment, but still with a negative verification result.

The problem this time was transition ut_3 in Fig. 26 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 of verification time, the property was finally satisfied.

9.2.3 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 4 presents the result of the different stages in the verification process.

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.

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 Fig. 25. After half a second's verification time, the property proved again unsatisfied. The diagnostic trace shows that

[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.

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 $\mathbf{AG}$ (HRESP $\rightarrow$ $\mathbf{AF}_{\leq 5}$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. Putting this fact aside, the property was already verified in the previous section anyway. Even with this extra assumption the property proved unsatisfied.

The diagnostic trace indicated an error in the 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 Fig. 27. The glue logic 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. A positive result was obtained after 41 min.

10 Conclusions

This paper 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 interconnecting the components. Each component has a number of associated properties 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. To verify the glue logic, high-level models of the connected components must also be included in the verification. 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 each component has several different interfaces, and stubs are defined with respect to an interface, there also exist several stubs to choose from for verification. 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.

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 pessimistic stubs which actually produce more events than the full component does. 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 outlined.

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. Such an assumption regarding the surrounding is usually 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 outlined. The generated Petri-net can then be attached to one of the components involved in the current verification.

Most of the activities in this verification methodology can be automatically performed and the corresponding tools have been implemented. The only activity which cannot be automated, and hence needs interaction with a human designer, is the examination of the diagnostic trace. The diagnostic trace guides the designer to either the design fault in the verified glue logic, or to the stub needed to be refined.

Appendix A

This appendix gives a formal proof of Theorem 1, on which the proposed methodology is based. Before presenting the proof, it is necessary to introduce a few auxiliary theorems and concepts.

Theorem 1A. *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 $Op_C(in) \mid_{I_C} = Op_S(in \mid_{I_S})$ (see Definition 9) filters out more elements from the unrestricted operation when $I_S = I_C = p_2$ than when $I_S = I_C = p_1$. Consequently o must also pass the filter of p_1 and can be an output of $Env(P_1, J)$, i.e. $o \in Op_{Env(P_1,J)}(in)$. □

Definition 14. Generalised operation. The generalised operation Op_C for component C is the union of all operations for every possible input observation, $Op_C = \cup_{in} Op_C(in)$.

According to Definition 8, 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 1 into the following corollary.

Corollary 1A. *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 1A and Definition 14. □

Definition 15. State sequence generator. A state, in this context, is a marking of ports. A state sequence generator is a function $\sigma(o, M_0)$, where o is an observation and M_0 is an initial state. The observation o may only contain appearing events and disappearing events on ports. The result of the function is a sequence of states obtained by iteratively applying the events in o to the previously obtained state (initially M_0) in the order indicated by their timestamps.

Let r_e denote the timestamp of an event $e \in o$. Assume $e = \langle p, \langle v, r_e\rangle\rangle$ or $e = \langle p, r_e\rangle$, depending on whether it is an appearing or disappearing event, and $E = \{e \mid \neg\, \exists\, e' \in o : (r_{e'} < r_e)\}$, i.e. the set of events with the lowest timestamp in o. Then Definition 15 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 7 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, Eq. (6) gives the solution.

$$\sigma(\mathrm{o}, \emptyset) = [\emptyset, \{\langle p, \langle 2, 1\rangle\rangle\}, \{\langle p, \langle 2, 1\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle\}, \{q, \langle 5, 3\rangle\rangle\}, \{p, \langle 3, 8\rangle\rangle, \langle q, \langle 5, 3\rangle\rangle\}, \emptyset] \tag{6}$$

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 [12] for the two example formulas $\mathbf{AF}\,\phi$ and $\mathbf{EG}\,\phi$ 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):

$$s \models \mathbf{AF}\,\phi \Leftrightarrow \forall \sigma \in P_M(s)\, \exists\, j \geq 0 : \sigma[j] \models \phi \tag{7}$$

$$s \models \mathbf{EG}\,\phi \Leftrightarrow \exists \sigma \in P_M(s)\, \forall\, j \geq 0 : \sigma[j] \models \phi \tag{8}$$

$P_M(s)$ 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 15, 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 15. Equations (9) and (10), where *IN* is the set of all possible input observations of component C, express the same semantics as Eqs. (7) and (8) in terms of observations and operations

$$M_0 \models \mathbf{AF}\,\phi \Leftrightarrow \forall o \in Op_C \forall i \in IN \exists j \geq 0 : \sigma(\mathrm{o} \cup i, M_0)[j] \models \phi \tag{9}$$

$$M_0 \models \mathbf{EG}\,\phi \Leftrightarrow \exists o \in Op_C \exists i \in IN \forall j \geq 0 : \sigma(\mathrm{o} \cup i, M_0)[j] \models \phi \tag{10}$$

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 Eqs. (7) and (8).

The work by Alur et al. [4] provides equivalent formulas to Eqs. (7) and (8) for TCTL. Based on the discussion above, they can be trivially extended to formulas similar to Eqs. (9) and (10).

Theorem 1. *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: $M_0 \models \mathbf{AF}\,\phi \Leftrightarrow \forall o \in Op_{Env(P_1,J)} \forall i \in IN \exists j \geq 0 : \sigma(\mathrm{o} \cup i, M_0)[j] \models \phi$, where *IN* is the set of all input observations on ports in the partitions, according to Eq. (9). As a consequence of Corollary 1A and the fact that o and i are universally quantified, it is possible to conclude $\forall o \in Op_{Env(P_2,J)} \forall i \in IN\ \exists j \geq 0 : \sigma(\mathrm{o} \cup i, M_0)[j] \models \phi$. □

Appendix B

This appendix gives a formal proof of Theorem 2, on which the stub generation and pessimism reduction algorithms are based.

Theorem 2. *Assume two environments E_1 and E_2 of the same set of components and $Op_{E_1} \subseteq Op_{E_2}$, an initial marking M_0 and a (T)ACTL formula, e.g.* $\mathbf{AF}\,\phi$ *expressed only on the ports of the stubs in E_1 and E_2. If $M_0 \models \mathbf{AF}\,\phi$ for component E_2, then it is also true that $M_0 \models \mathbf{AF}\,\phi$ for component E_1.*

Proof: $M_0 \models \mathbf{AF}\,\phi \Leftrightarrow \forall o \in Op_{E_2} \forall i \in IN \exists j \geq 0 : \sigma(\mathrm{o} \cup i, M_0)[j] \models \phi$, where *IN* is the set of all input observations on ports in the partitions, according to Eq. (9). As a consequence of the fact that o and i are universally quantified, it is straight-forward to conclude that $\forall o \in Op_{E_1} \forall i \in IN\ \exists j \geq 0 : \sigma(\mathrm{o} \cup i, M_0)[j] \models \phi$. □

References

1. Abadi, M. and L. Lamport. Composing Specifications. *Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
2. Abadi, M. and L. Lamport. Conjoining Specification. *Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
3. Ackland, B., A. Anesko, and D. Brinthaupt, et al. A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP. *Journal of Solid-State Circuits*, 35(3), 2000.
4. Alur, R., C. Courcoubetis, and D.L. Dill. Model Checking for Real-Time Systems. In *Proceedings of Symposium on Logic in Computer Science*, Philadelphia, USA, 1990, pp. 414–425.
5. Alur, R. and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
6. Asteroth, A. C. Baier, and U. Aßmann. Model Checking with Formula-Dependent Abstract Models. In *Lecture Notes in Computer Science*, 2102:155–165, 2001.
7. Ball, T., and S.K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical report, Microsoft Research, 2002.
8. Barringer, H., D. Giannakopoulou, and C.S. Pasareanu. Proof Rules for Automated Compositional Verification Through Learning. In *Proceedings of Specification and Verification of Component-Based Systems*, Helsinki, Finland, 2003, pp. 14–21.
9. Caldwell, A.E., H-J. Choi, and A.B. Kahng. Effective Iterative Techniques for Fingerprinting Design IP. In *Proceedings of Design Automation Conference*, New Orleans, USA, 1999, pp. 843–848.
10. Cheung, S.C. and J. Kramer. Context Constraints for Compositional Reachability Analysis. *Transactions on Software Engineering and Methodology*, 5(4):334–377, 1996.
11. Clarke, E.M., E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal logic specifications. *Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
12. Clarke, E.M., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, USA, 1999.
13. Clarke, E.M., O. Grumberg, S. Jha, et al. Counterexample-Guided Abstraction Refinement. In *Proceedings of International Conference on Computer Aided Verification*, Chicago, USA, 2000, pp. 154–169.
14. Cortés, L.A., P. Eles, and Z. Peng. Verification of Embedded Systems Using a Petri Net Based Representation. In *Proceedings of International Symposisum on System Synthesis*, Madrid, Spain, 2000, pp. 149–155.
15. Coudert, O. and J.C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of International Conference on Computer Aided Design*, Santa Clara, USA, 1990, pp. 126–129.

16. de Alfaro, L., and T.A. Henzinger. Interface Automata. In *Proceedings of the Annual ACM Symposium on Foundations of Software Engineering*, Vienna, Austria, 2001, pp. 109–120.
17. Gajski, D., A C.-H. Wu, V. Chaiyakul, et al. Essential Issues for IP Reuse. In *Proceedings of Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2000, pp. 37–42.
18. Grumberg, O. and D.E. Long. Model Checking and Modular Verification. *Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
19. Haase, J. Design Methodology for IP Providers. In *Proceedings of Design and Test in Europe*, Munich, Germany, 1999, pp. 728–732.
20. Hong, I. and M. Potkonjak. Behavioral Synthesis Techniques for Intellectual Property Protection. In *Proceedings of Design Automation Conference*, New Orleans, USA, 1999, pp. 849–854.
21. Karlsson, D., P. Eles, and Z. Peng. Formal Verification in a Component Reuse Methodology. In *Proceedings of International Symposium on System Synthesis*, Kyoto, Japan, 2002, pp. 156–161.
22. Karlsson, D. Towards Formal Verification in a Component-Based Reuse Methodology. Licentiate Thesis No 1058, Linköping Studies in Science and Technology, http://www.ep.liu.se/lic/science_technology/10/58/, Linköping, Sweden, 2003.
23. Karlsson, D., P. Eles, and Z. Peng. A Formal Verification Methodology for IP-based Designs. In *Proceedings of EUROMICRO Symposium on Digital System Design*, Rennes, France, 2004, pp. 372–379.
24. Keating, M. and P. Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs*. Kluwer Academic Publishers, Boston, USA, 1998.
25. Pnueli, A. In Transition for Global to Modular Temporal Reasoning About Programs. In *Logics and Models of Concurrent Systems*, vol. 13, Springer-Verlag, 1984.
26. Roop, P.S. and A. Sowmya. Forced Simulation: A Technique for Automating Component Reuse in Embedded Systems. *Transactions on Design Automation of Electronic Systems*, 6(4):602–628, 2001.
27. Roop, P.S., A. Sowmya, and S. Ramesh. k-Time Forced Simulation: A Formal Verification Technique for IP Reuse. In *Proceedings of International Conference on Computer Design*, San Jose, USA, 2002, pp. 50–55.
28. Rowson, J.A. and A. Sangiovanni-Vincentelli. Interface-Based Design. In *Proceedings of Design Automation Conference*, Anaheim, USA, 1997, pp. 178–183.
29. Roychoudhury, A., T. Mitra, and S.R. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Proceedings of Design and Test in Europe*, Munich, Germany, 2003, pp. 828–833.
30. Rushby, J. Theorem Proving for Verification. *Lecture Notes in Computer Science*, 2067:39–57, 2001.
31. Savage, W., J. Chilton, and R. Camposano. IP Reuse in the System on a Chip Era. In *Proceedings of International Symposium on System Synthesis*, Madrid, Spain, 2000, pp. 2–7.
32. Schneider, F.B. Enforceable Security Policies. *Transactions on Information and System Security*, 3(1):3–50, 2000.
33. Seepold, R., N.M. Madrid, A. Vörg, et al. A Qualification Platform for Design Reuse. In *Proceedings of International Symposium on Quality Electronic Design*, San Jose, USA, 2002, pp. 75–80.
34. Spitznagel, B., and D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proceedings of International Conference on Software Engineering*, Portland, USA, 2003, pp. 374–384.
35. Vahid, F. and L. Tauro. An Object-Oriented Communication Library for Hardware-Software CoDesign. In *Proceedings of Workshop on HW/SW Codesign*, Braunschweig, Germany, 1997, pp. 81–86.
36. Xie, F. and J.C. Browne. Verified Systems by Composition from Verified Components. In *Symposium on Foundations of Software Engineering*, Helsinki, Finland, 2003, pp. 277–286.