A Formal Verification Approach for IP-based Designs

Daniel Karlsson, Petru Eles, Zebo Peng
Linköpings universitet
{danka, petel, zebpe}@ida.liu.se

ABSTRACT

This paper proposes a formal verification methodology which is smoothly integrated 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 verified by their designers and which are considered correct under the assumption that the environment satisfies certain properties assumed by the component. What remains to be verified is the glue logic inserted between the components. Each such glue logic is verified one at a time using model checking techniques.

A big difficulty with such an approach is the question how to handle the connected components and the rest of the system in the verification of the glue logic, which only constitutes a small part of the design. In this paper, algorithms for generating a model corresponding to the rest of the system are discussed together with guidelines on how and when to use them. The methodology is illustrated by a small case study on a mobile telephone.

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 music devices such as Mp3-players. Moreover, other, traditionally mechanical, devices are becoming more and more computerised. Examples of such devices are cars or washing machines. Several such devices are, in addition, highly safety critical, such as aeroplanes or medical equipment. In fact, in 1999, 99% of all microprocessors were used in the type of systems mentioned above (embedded systems). Only the remaining 1% was used in general purpose computers [1]. This situation indicates the big importance of embedded systems.

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 [2, 3].

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.

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

There are several aspects which make this task difficult. One is the complexity of the systems, which makes simulation based techniques very time consuming. On the other hand, formal verification of such systems suffers from state explosion. However, it can often be assumed that the design of each individual component has been verified [4] and can be supposed to be correct. What remains to be verified is the interface logic and the interaction between components. Such an approach can handle both the complexity aspects (by a divide and conquer strategy) and the lack of information concerning the internals of predefined components.

Assume-guarantee reasoning [5] is a method of combining the results from the verification of individual components to draw a conclusion about the whole system following certain rules. This has the advantage of avoiding the state explosion problem by not having to actually compose the components, but each component is verified separately.

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

2. 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.

The *glue logic* inserted between communicating components is depicted in Figure 1 as clouds. For example, the interfaces of two or more 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 be the glue logic.

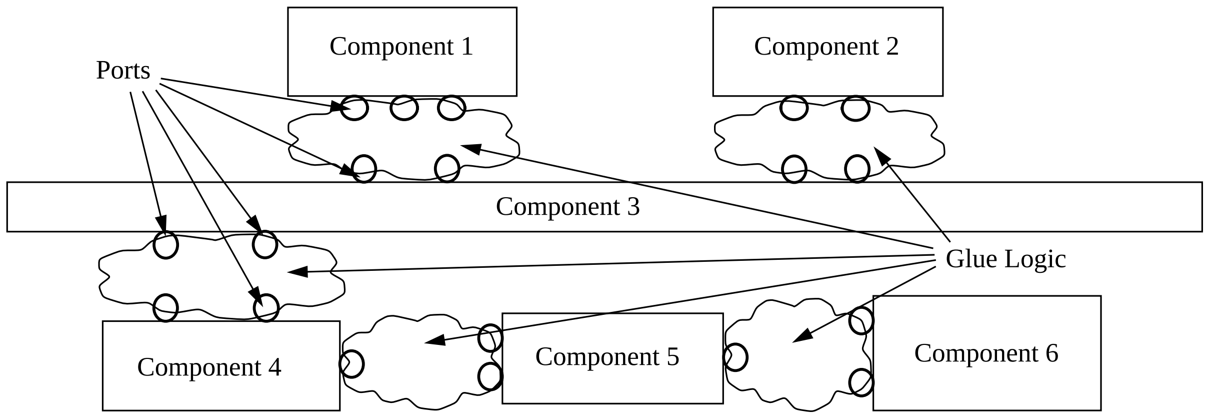

Figure 1. Targeted system topology

2.1 Objective and Assumptions

The objective of the proposed methodology is to 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 remains to be verified is the glue logic and the interaction between the components through the glue logic.

According to the second assumption, the components impose certain requirements on their environment. These requirements have to be satisfied in order for the component to function correctly. The requirements are expressed 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 [5] for expressing these requirements. However, other similar logics may be used as well.

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 always hold (**A**) or whether it must have the possibility to hold (**E**). Temporal quantifiers express the future behaviour along a certain computation path, 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 **U** q expresses that p must hold in all computation steps until q holds. q must moreover hold some time in the future. q **R** p on the other hand expresses that "q releases p", which means that p must hold in every computation step until q, or if q never appears, p holds indefinitely. State expressions may be either a boolean expression or recursively a CTL formula. In TCTL, relations on time may be added as subscripts on the temporal operators. 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.

$$\begin{aligned}&\mathbf{AG}\ ((status = \text{disconnected} \vee \text{init}) \rightarrow \\ &\mathbf{A}\ [status = \text{connected}\ \mathbf{R}\ \neg in = \text{send}])\end{aligned} \qquad \text{(eq. 1)}$$

Eq. 1 provides an example of a CTL formula capturing a constraint imposed by a component. This example, also shown in Figure 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.

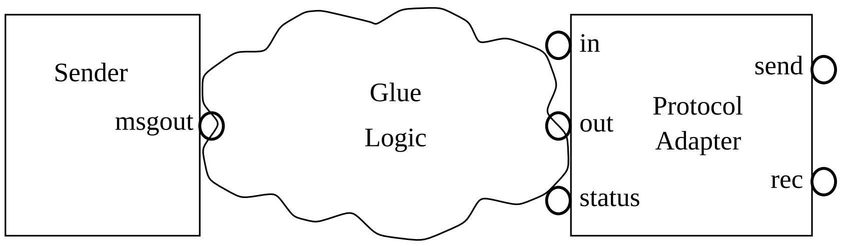

Figure 2. A concrete example of a situation where the methodology can be applied

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 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 produced by it.

Eq. 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 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.

2.2 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.

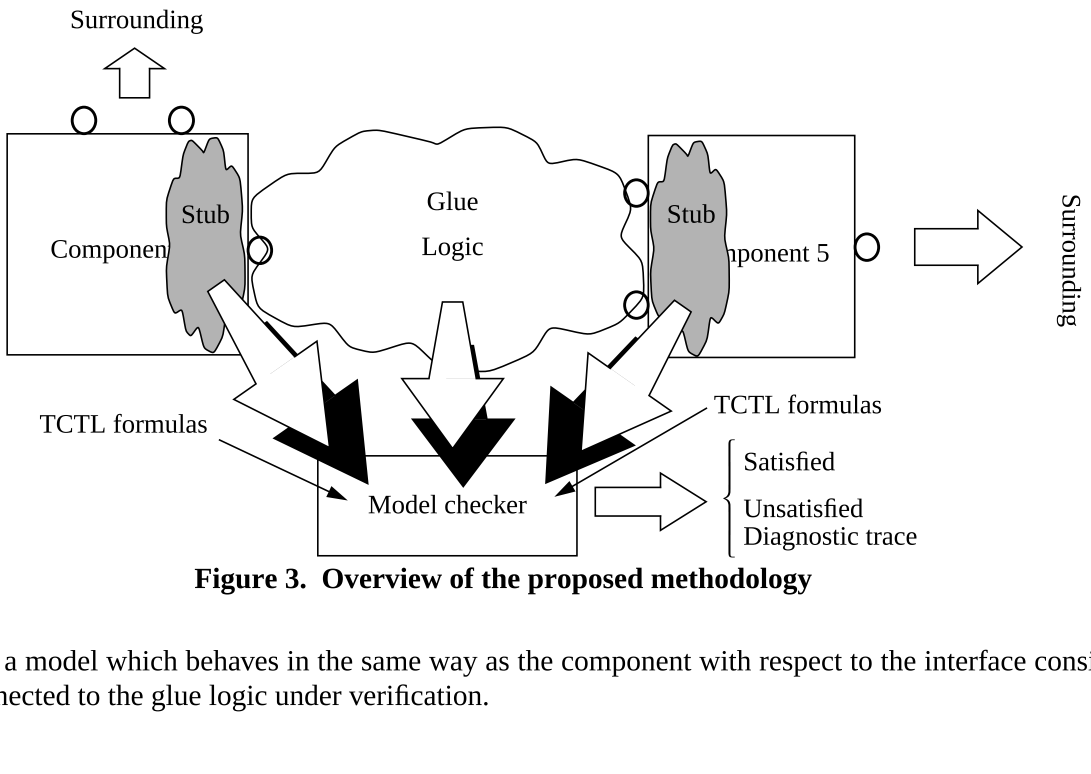

Figure 3. Overview of the proposed methodology

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

3. 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+) [6]. It is a Petri-net based representation with the extensions listed below. Figure 4 shows an example of a PRES+ model.

1. Each token has a value and a timestamp associated to it.
2. Each transition has a function and a time delay interval associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp is increased by an arbitrary value from the time delay interval. In Figure 4, the functions are marked on the outgoing edges from the transitions.
3. 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.
4. The transitions may have guards. A transition can only be enabled if the value of its guard is true (see transitions t_4 and t_5).

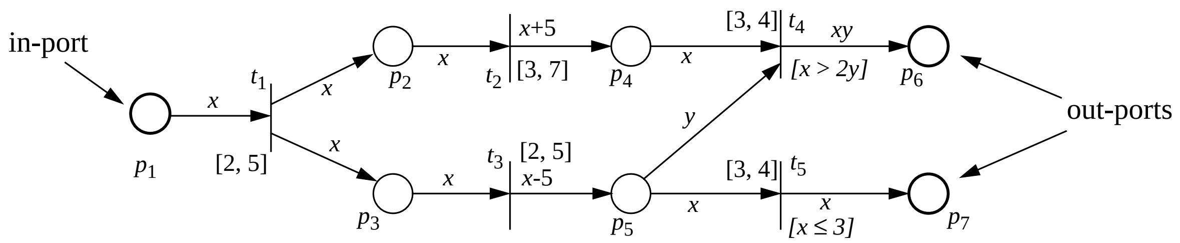

Figure 4. A simple PRES+ net

It should be pointed out that this design representation is not required by the methodology itself, but the algorithms mentioned in this paper have been developed considering PRES+ as the formal representation. However, if another design representation is found more suitable for a particular design, similar algorithms based on the same ideas can be developed for that design representation.

4. FORMAL VERIFICATION WITH STUBS

In this paper, we will concentrate on two possible scenarios and their slightly different approaches. Either the stubs are given by the component provider or they have to be generated by the designer. Emphasis will be put on the latter scenario in this paper (Section 5). This section concentrates on the scenario where the stubs are given by the component provider.

Remember that a stub is a model which behaves in the same way as a certain component with respect to a particular interface. In this section, this definition will be interpreted very strictly in the sense that the stub must behave *exactly* in the same way as the corresponding component. This also includes the behaviour of the surrounding to the extent expressed by the constraints imposed by the component on the surrounding. In Section 5, this definition will be somewhat relaxed.

Since a component generally has several interfaces, it also has several stubs, one for each interface. An interface is defined as a set of ports. Hence, interfaces can be partially ordered with respect to the subset relation. As a consequence, stubs can also be partially ordered by the same relation, due to the fact that they are defined with respect to a particular interface. In the bottom of the hierarchy, so called *empty stubs*[1] can be found. Such stubs either produce or consume (depending on whether the stub will be attached to an out-port or an in-port) events (tokens) containing any message (value) at any point in time, i.e. they behave completely randomly. On the other extreme, in the top of the hierarchy the top-level stub is found. It behaves exactly in the

[1] The name comes from the fact that they somehow correspond to the empty interface, which does not contain any ports at all.

same way as the full component, since it is defined with respect to all ports of the component. For more details, the reader is referred to [7, 8].

When verifying a particular glue logic using stubs given by the component provider, the designer has to select appropriate stubs from this hierarchy. In [7] we have shown that if a property, belonging to the sublogic ACTL[1], is satisfied using stubs near the bottom of the hierarchy (low level), it is guaranteed that the property is also satisfied using any higher level stub, including the full component.

Experiments have shown [8] that using low-level stubs generally leads to considerably shorter verification times. However, using low-level stubs makes it more likely that the property is unsatisfied than if high-level stubs were used.

Based on these facts, an iterative approach is suggested. Low-level stubs should initially be used in the verification. If the property is unsatisfied, a stub situated at a higher level in the hierarchy is used instead in the next verification round. The diagnostic trace obtained from the previous verification indicates which stubs violated the property and thus should be changed. This procedure iterates until either the property is verified to true, or the diagnostic trace indicated that the fault is situated in the glue logic (in which case a design error was found). It should be noted that the designer is always guided by the diagnostic trace.

Besides leading to shorter verification times, this methodology also provides a means to perform the verification even if not all stubs are available.

5. STUB GENERATION

In the second scenario when no stubs are given by the component providers, they must be generated by the designer. An algorithm which automatically generates stubs, given the model of a component and an interface, has been developed [8]. This section provides an outline of the basic ideas of the algorithms related to this issue. The example component in Figure 5 will be used to explain and analyse the stub generation algorithms in this chapter. In all cases, a stub for the marked interface $\{p_1, p_2\}$ will be generated.

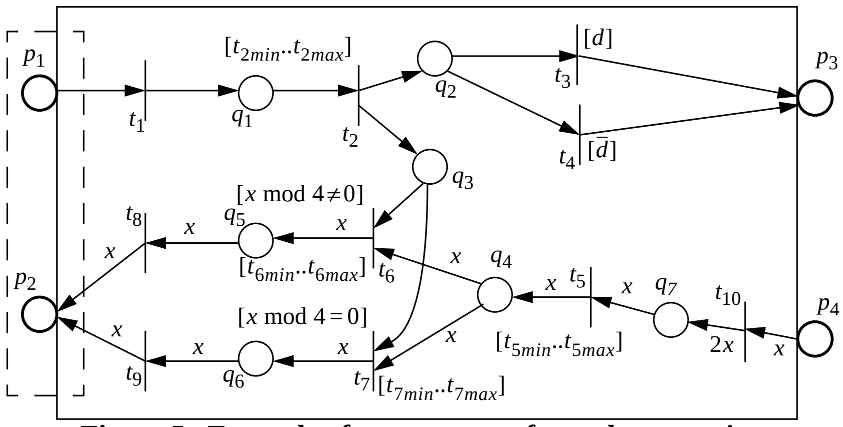

Figure 5. Example of a component for stub generation

In previous sections, a stub has been defined to be a model which has exactly the same behaviour as a certain component with respect to a particular interface (see [7] for a more formal definition). For future reference, such stubs will be called *exact*. In this section, that definition is relaxed in the following sense. Stubs are able to produce the same events as the corresponding components with respect to the particular interface, but they, in addition, produce more events not produced by the corresponding component. Such stubs are called *pessimistic*. As opposed to exact stubs, pessimistic stubs do not have to comply with the assumptions on the surrounding (see Section 4).

Due to similar theoretical results as discussed in Section 4, if a particular ACTL property is satisfied using pessimistic stubs, it is guaranteed that the property is also satisfied with the full component. Consequently, it is sufficient to verify the system using pessimistic stubs. However, if the property was not satisfied with the pessimistic stubs, it is necessary to reduce their pessimism, i.e. reduce the number of those events produced by the stub which cannot be produced by the full component [8].

[1] ACTL is a sublogic of CTL, which only allows universal path quantifiers and negation only in front of atomic propositions. All CTL formulas in this paper are ACTL.

During the verification process, the pessimism in the stubs is iteratively reduced until either the property is satisfied or the fault is found to be situated in the glue logic (design fault). It should be noted that this entire process, both pessimism reduction as well as identifying the stub to reduce pessimism on, is guided by the diagnostic trace from the previous verification round.

5.1 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 pessimistic stub, since it is able to produce the same events as the component is able to. It is also clear that it is able to produce additional events due to the extra random transitions added. The resulting stub is shown in Figure 6.

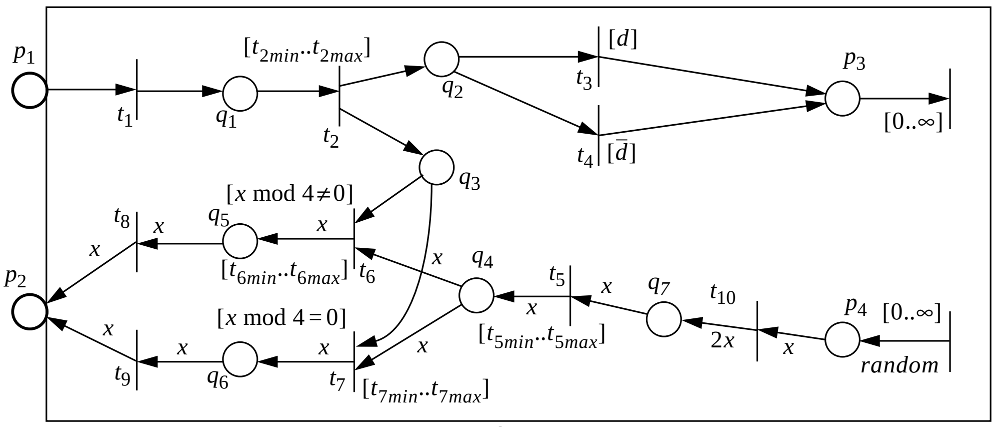

Figure 6. A naïve stub of the component in Figure 5

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

5.2 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. This is the 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
3. Compensation for the excluded parts of the component

Each of these parts is discussed below. A more detailed description of the algorithms can be found in [8].

5.2.1 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. These procedure is called once for each port in the interface of the stub. 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 to which ports 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 7 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.

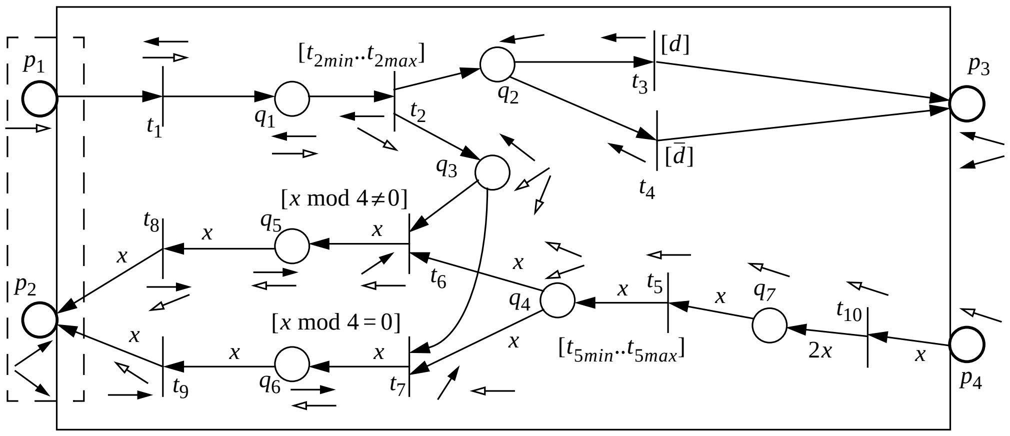

Figure 7. The dataflow marking of the component in Figure 5

5.2.2 *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 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, the separation point is the first node encountered which has several dataflow arrows pointing in different directions and which are of different type. Such nodes indicate influence from several different ports. In Figure 7, starting from the port p_3, which does not belong to the particular interface, the separation point found is t_2. Another separation point is q_4, which is obtained starting from p_4. More details regarding the exact definition and algorithms can be found in [8].

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 8 shows the stub resulting from this procedure.

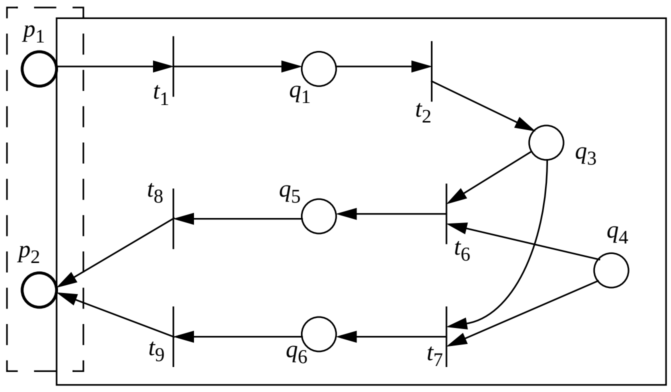

Figure 8. The places and transitions in the automatically generated stub

5.2.3 *Compensation*

As can be seen in Figure 8, some transitions in the stub do not have all input or output places anymore, compared to the full component (Figure 5). This means that they will not deliver (receive) all needed output (input). Consequently, some mechanism has to be developed in order to compensate for these places. This is done by introducing non-determinism.

Transition t_2 has one output place not included in the stub. In the worst case situation, there might be a token in that output place disabling t_2 forever. In order to reflect this, the upper bound of the time delay inter-

val of that transition is changed to infinity as indicated in Figure 9. Similarly, place q_4 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 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.
Apart from these two cases illustrated here, there are also other cases of compensating excluded parts of the components to take care of. However, the solution to these cases are quite similar [8]. Figure 9 shows the final result of the automatic stub generation algorithm.

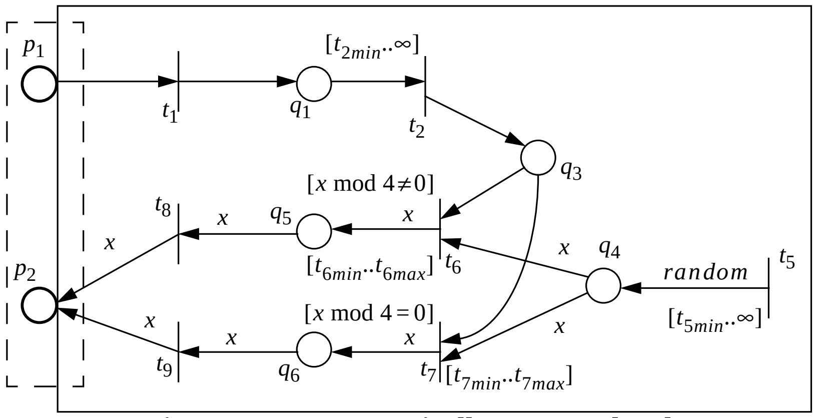

Figure 9. An automatically generated stub

5.3 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 in the glue logic to be verified. 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 to the stubs some parts of the component which were excluded during the stub generation. 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 a stored table.

What the designer must know in order to use the component is stated in the user documentation of the component, i.e. the events occurring on the ports. By following the diagnostic trace, obtained as a result from the verification, the designer can identify an unwanted behaviour on one of the ports of the component. 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 at 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 5.2, 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 6 which allow the inclusion of certain assumptions on the surrounding into the verification process.
2. Overestimation of firing delay. Firing delays are overestimated with infinity in the stub generation algorithm. The reason for this was that there is no guarantee that the transitions will ever become enabled. However, assuming the most hostile surrounding possible, this can never be guaranteed in the full component either. Consequently, no pessimism reduction algorithm may ever be able to reduce this type of pessimism given these assumptions. However, this problem can be solved by including certain assumptions on the surrounding into the verification process, as discussed in Section 6.

3. Unwanted values. A value which appears in a port, and which cannot appear in that port in the complete component, is in fact the only case where pessimism can be reduced. 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 stub.

Thus, pessimism reduction of stubs is only 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.

The new stub, with reduced pessimism, must compensate the parts of the component which are not included. This is done following the same procedure as in the stub generation algorithm, see Section 5.2.3.

6. 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 in order to perform the verification. These are cases in which the correct 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.

An algorithm which generates a model that is able to produce all possible events consistent with a particular ACTL formula, has been developed [8]. 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 Figure 10. Also in this situation, the diagnostic trace guides the designer to which properties need to be included in the model of the surrounding. The verification process iteratively adds properties to the surrounding until either the formula to be verified is satisfied or the diagnostic trace indicated a fault in the glue logic.

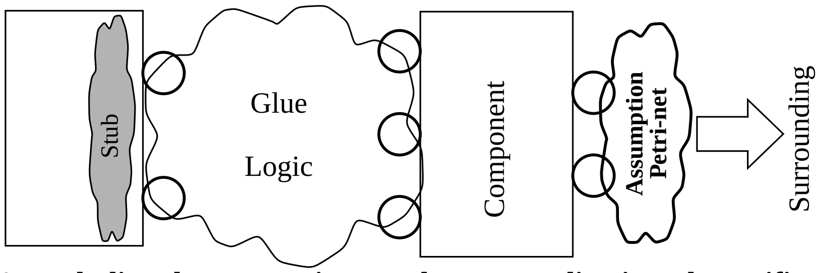

Figure 10. Including the assumptions on the surrounding into the verification process

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 [9]. 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.

$$\mathbf{AG}\ (p \rightarrow p < 2) \qquad \text{(eq. 2)}$$

It is found that there are two states which satisfy the example formula. They are described by Eq. 3 and Eq. 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.

$$\neg p \wedge \mathbf{AXAG}\ (p \rightarrow p < 2) \qquad \text{(eq. 3)}$$

$$p < 2 \wedge \mathbf{AXAG}\ (p \rightarrow p < 2) \qquad \text{(eq. 4)}$$

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

For each place (possible future behaviour), transitions must be added for each possible state. There are two such states in the example, Eq. 3 and Eq. 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 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.

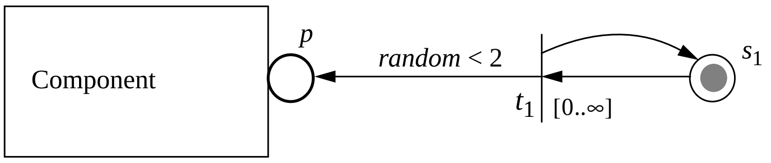

Figure 11. The PRES+ model corresponding to the example formula in Eq. 2

7. A CASE STUDY

The presented verification methodology gives a powerful means to verify large systems using a divide and conquer approach. We have implemented an environment which allows all the activities implied by the methodology to be performed automatically. The only action which needs interaction with the human designer is the examination of the diagnostic trace. The diagnostic trace guides the designer in deciding what action to be taken next.

This section provides an example in order to demonstrate how a system can be verified using the methodology.

7.1 The Mobile Telephone System

The application used as an example is a mobile telephone. Figure 12 shows an overview picture of the model and how the components forming the model are connected. It consists of seven components communicating via an AMBA bus.

1. Microphone. The microphone sends voice data to the transmitter.
2. Buttons. When dialling, the buttons component sends information about which buttons were pressed to the controller.
3. Speaker. The speaker receives voice signals from the receiver and converts it to sound.
4. Display. The display shows on a small screen information sent to it by the controller.
5. Receiver. The receiver receives data from the base-station of the mobile telephone network and passes it on to the designated component.
6. Transmitter. The transmitter receives data from other components in the telephone and passes it on to the base-station.
7. Controller. The controller coordinates the tasks of the other components.

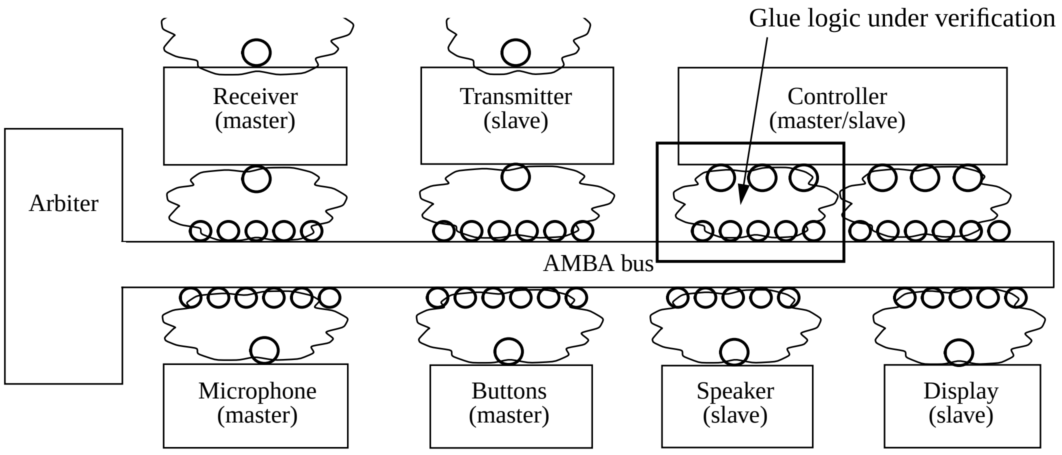

Figure 12. Overview model of the example system, a mobile telephone

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

The AMBA bus divides the components communicating over the bus into two categories, master and slave. Figure 12 indicates to which category each component in the example belongs. Components sending messages are masters and components receiving messages are slaves.

For this reason, this design principally contains two types of glue logic, one for handling the master functionality and one for handling the slave functionality for each type of component respectively. Consequently, the glue logics which are situated between the bus and a slave component is a slave functionality glue logic, whereas the glue logics situated between the bus and a master component is a master functionality glue logic.

7.2 Verification of the Model

In [8], a more exhaustive verification of this system is performed. Here, due to lack of space, the verification of only one property is presented. The property to be verified states that the controller must only receive legal values on the port $button$. The CTL formula expressing this property is stated in Eq. 5. It was verified using the UPPAAL model checking environment [10]. In order to be able to verify PRES+ models using this tool, the PRES+ models are automatically translated into the modelling language used by UPPAAL, namely Timed Automata [11]. Such a translation is discussed in [6].

$$\mathbf{AG}\ (button \rightarrow button \in \{0..9, \text{enter}\}) \qquad \text{(eq. 5)}$$

The components included in the verification of the property expressed by Eq. 5, were the controller, arbiter, bus and the slave functionality glue logic. Table 1 presents the result of the different stages in the verification process.

Table 1: Verification results of property 1

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

The 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 bus, as indicated by the diagnostic trace. It took about 1 second to obtain this result. The port of the bus to which data arrives is called HDATA. There are several such ports, one for each component connected to the bus. HDATA ports for master components are annotated with m_x and for slaves with s_x.

Since the property was not satisfied the diagnostic trace must be examined. According to the diagnostic trace, the bus produced a value on port $\text{HDATA}s_x$ which is not allowed. In order to do the verification, it was necessary to make an assumption about the surrounding. In this case, it has to be assumed that only data in the set $\{0..9, \text{enter}\}$ can occur in port $\text{HDATA}m_x$. The property is formally given in Eq. 6.

$$\mathbf{AG}\ (\text{HDATA}m_x \rightarrow \text{HDATA}m_x \in \{0..9, \text{enter}\}) \qquad \text{(eq. 6)}$$

A Petri-net model for this formula was automatically generated together with a new version of the bus stub, now also including port $\text{HDATA}m_x$, and attached to the interface connecting the buttons component. Using this new stub, the property was satisfied using approximately 2 minutes verification time.

The positive verification result was obtained by making an assumption about the surrounding. In order to finally conclude the positive result, the correctness of the assumption in Eq. 6 must first be established.

The components involved in verifying the assumption itself 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 seconds.

The successive steps through the methodology briefly outlined above, are guided by the diagnostic trace which all the time gives feedback to the user what to do next. It might indicate that too pessimistic stubs were used, that there is an error in the glue logic, or that assumptions regarding the surrounding have to be introduced.

8. 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 logics interconnecting the components. Every component has a number of properties associated to it which it requires the system to satisfy in order to work correctly.

An essential part of this methodology involves models of the component behaviour with respect to a certain interface of the component. These models are called *stubs*.

Two scenarios have been presented: either the stubs are given by the component provider, or they are generated by the designer given the model of the component. Both scenarios can be efficiently exploited in the verification process by adopting an iterative approach. Furthermore, properties regarding the surrounding of the glue logic under verification can also be incorporated into the process.

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

Most of the activities in this verification methodology can be automatically performed and have been implemented in a tool. The only activity which needs interaction with a human designer, is the examination of the diagnostic trace. The diagnostic trace constantly guides the designer in deciding what action to take next.

REFERENCES

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

[2] J. Haase, ”Design Methodology for IP Providers”, in *Proc. DATE*, pp. 728-732, 1999

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

[4] R. Seepold, N.M. Madrid, A. Vörg et al, ”A Qualification Platform for Design Reuse”, in *Proc. ISQED*, pp. 75-80, 2002

[5] E.M. Clarke, O. Grumberg, D.A. Peled, ”Model Checking”, *The MIT Press*, 1999

[6] L.A. Cortés, P. Eles, Z. Peng, ”Verification of Embedded Systems using a Petri Net based Representation”, in *Proc. ISSS*, pp. 149-155, 2000

[7] D. Karlsson, P. Eles, Z. Peng, ”Formal Verification in a Component Reuse Methodology”, in *Proc. ISSS*, pp. 156-161, 2002

[8] D. Karlsson, ”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/

[9] O. Grumberg, D.E. Long, ”Model Checking and Modular Verification”, in *ACM-TOPLAS*, Vol 16 No 3, pp. 843-871, 1994

[10] UPPAAL homepage: http://www.uppaal.com

[11] E.M. Clarke Jr, O. Grumberg, D.A. Peled, ”Model Checking”, MIT Press 1999.

[12] A. Roychoudhury, T. Mitra, S.R. Karri, ”Using formal techniques to Debug the AMBA System-on-Chip Bus Protocol”, in *Proc. DATE*, pp. 828-833, 2003