Formal Verification in a Component-based Reuse Methodology

Daniel Karlsson, Petru Eles, Zebo Peng
IDA, Linköpings universitet, 581 83 Linköping, Sweden
{danka, petel, zebpe}@ida.liu.se

ABSTRACT

There is an important trend towards design processes based on the reuse of predesigned components. We propose a formal verification approach which smoothly integrates with a component based system-level design methodology. Once a timed Petri Net 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 connected components and on abstract models of their functionality, without assuming any knowledge regarding their implementation. We have both developed the theoretical framework underlying the methodology and implemented an experimental environment using model checking techniques.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General - *Systems specification methodology*; B.7.2 [**Integrated Circuits**]: Design Aids - *Verification*; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD)

General Terms

Performance, Design, Verification

Keywords

Formal Verification, IP, Reuse, Model Checking, Timed Petri-Nets

1. INTRODUCTION

One of the important current trends is towards a design process based on the reuse of predesigned blocks [1]. Such blocks can be both hardware and software components. With such a design process, also called "interface based design" [2], the focus is on the interaction of components and, in particular, on interfaces, protocols and glue logics which interconnect independent blocks.

Once a design alternative has been produced, one crucial aspect is the verification of interfaces and of the global system functionality. There are several aspects which make this task very 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. Another problem is the lack of information about the internals of predesigned blocks. However, it can often be assumed that the design of each individual component has been verified and can be supposed to be correct [3]. What remains to be verified is the interface logic (hardware or software). 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.

Although several approaches have been proposed tackling aspects of component based design, there exists, to our knowledge, no work concerning the integration of such a design process with formal verification.

As a main contribution, 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 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, without assuming any knowledge regarding their implementation. We have developed both the theoretical framework underlying the methodology and implemented an experimental environment for its application, using model checking techniques. Our approach represents a contribution towards increasing both design and verification efficiency in the context of a methodology based on component reuse.

Section 2 of the paper introduces the design representation, followed by a preliminary discussion in Section 3. The verification technique is presented in Section 4 where we both develop the theoretical framework and discuss some examples. Experiments are presented in Section 5 and the conclusions are given in Section 6.

ISSS'02, October 2-4, 2002, Kyoto, Japan.

2. PRES+: THE DESIGN REPRESENTATION

In order to support our modeling approach, we have defined a Petri Net based representation called PRES+ [4]. The following extensions to classical Petri Nets are the most important in the context of this paper (see Figure 1):

1. A token k has values and timestamps, $k = \langle v, r \rangle$ where v is the value and r is the timestamp.
2. A transition has a function and a time delay interval associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp is increased by an arbitrary value from the time delay interval. In Figure 1, 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).
5. The preset ${}^\circ t$ (postset t°) of a transition t is the set of all places from which there are arcs to (from) transition t. Similar definitions can be formulated for the preset (postset) of places. In Figure 1, ${}^\circ t_4 = \{p_4, p_5\}$ and $t_4{}^\circ = \{p_6\}$.

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

Definition 1. Component. A component is a subgraph of the graph of the whole system $\Gamma = P \cup T$ (P is the set of places and T is the set of transitions) such that:

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

Definition 2. Port. A place p is an out-port of component C if $(p^\circ \cap C = \emptyset) \wedge ({}^\circ p \subseteq C)$ or an in-port of C if $({}^\circ p \cap C = \emptyset) \wedge (p^\circ \subseteq C)$. p is a port of C if it is either an in-port or an out-port of C.

Definition 3. Interface. An interface of component C is a set of ports $I = \{p_1, p_2, \ldots\}$ where $p_i \in C$.

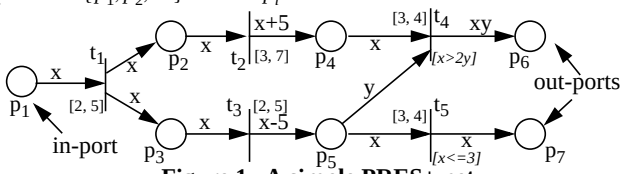


Figure 1. A simple PRES+ net

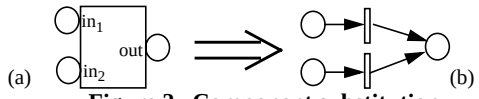


Figure 2. Component substitution

PRES+ can model the behaviour of a component at different levels of granularity. A component can be drawn as a box surrounded by its ports, as illustrated in Figure 2(a). Modelled in this way, it can be replaced with a PRES+ net as indicated by Figure 2(b).

3. PRELIMINARIES

Our verification methodology, presented in Section 4, can be used when the system is composed of preverified components connected by a so called glue logic, as indicated in Figure 3. Such a glue logic is sometimes called a wrapper. All boxes which represent components are abstractions of PRES+ nets in the way described above. The glue logics connecting the components are also modelled in PRES+.

An example of a glue logic is provided in Figure 4. 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 at regular rate.

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[1].

4. VERIFICATION

When all glue logics are constructed and all components are in place, it is time to verify that the components are assembled and interconnected correctly. Since the model is built with reusable components, we assume that they have been verified by the providers. What remains to be verified is the glue logic and the way it has adapted the various interfaces to work together. In our methodology, formal verification is performed using model checking [5].

4.1 Formulas and stubs

According to our methodology, the user acquires a verified component with a well-defined interface. For each such component, it is supposed that the following tasks have been previously performed:

1. Verify the component.
2. Provide (T)CTL [5] formulas as constraints on the inputs.
3. Provide stubs as description of the characteristics of outputs.

CTL (Computation Tree Logic) [5] is a logic language in which the possibility of events happening in different computation paths (futures) can be expressed. There are two computation path quantifiers, A (universal) and E (existential), and four operators on time, G (globally, in every computation step), F (some time in the future), U (until), R (releases). AG p has the meaning that p is always true in all possible computation paths. EF p means that in at least one computation path p holds at some point in the future. A [p U q] denotes that in all computation paths, p holds until q becomes true at some point in the future, while A [p R q] means

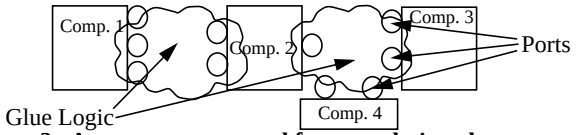


Figure 3. A system constructed from predesigned components

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

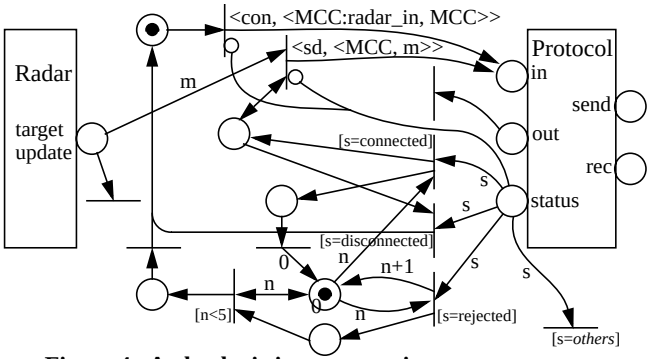


Figure 4. A glue logic interconnecting two components

that q holds until p becomes true or, if p never becomes true, q will hold globally. It is possible to explicitly include time in the formulas in which case the logic is called timed CTL or TCTL [7]. A TCTL formula could look like $\mathrm{AF}_{<5}p$, which means that p must always hold in the future within 5 time units. For the sake of simplicity we will use CTL formulas in our examples throughout the paper. However, the theoretical discussion as well as the verification methodology are valid for both CTL and TCTL.

The (T)CTL formulas associated to components impose constraints on the environment in which the component is placed. These constraints are expressed only in terms of the in- and out-ports of the component.

Review the example introduced in Section 3. The connection-based protocol component (Figure 4) comes together with (T)CTL formulas which describe the expected input on each interface of the component. Two of these formulas are:

$$\mathrm{AG}\ ((status = \langle \mathrm{disconnected}, p\rangle \vee \mathrm{init}) \rightarrow \mathrm{A}\ [status = \langle \mathrm{connected}, p\rangle\ \mathrm{R}\ \neg in = \langle \mathrm{send}, \langle p, \_\ \rangle\rangle]) \quad \text{(eq. 1)}$$

$$\mathrm{AG}\ (status = \langle \mathrm{connected}, p\rangle \rightarrow \mathrm{A}\ [status = \langle \mathrm{disconnected}, p\rangle\ \mathrm{R}\ (\neg in = \langle \mathrm{connect}, \langle p, \_\ \rangle\rangle \wedge \neg in = \langle \mathrm{listen}, \langle p, \_\ \rangle\rangle)]) \quad \text{(eq. 2)}$$

(eq. 1) states that the protocol can never receive a send command when it is disconnected at a certain protocol-port p. (eq. 2) requires that, as long as the protocol is already connected at protocol-port p, it is prohibited to connect again.

For the verification process of the glue logic, the (T)CTL formulas define the requirements on the input ports of the connected components. However, information regarding the output produced by these components is also needed for the verification process. For this purpose, the concept of a stub is introduced. Before defining a stub, some auxiliary concepts have to be defined.

Definition 4. Interface compatibility. Interfaces I_1 and I_2 are compatible iff there exists a bijection $f: I_1 \rightarrow I_2$ such that if $f(p) = q$, then p and q are both either in-ports or out-ports.

Definition 5. Event. An *appearing event* is a tuple $e^+ = \langle p, k\rangle$, where p is a place and $k = \langle v_k, r_k\rangle$ is a token. Appearing events represent the fact that a token k with value v_k is put in place p at time moment r_k. A *disappearing event* is a tuple $e^- = \langle p, r\rangle$ where p is a place and r is a timestamp. Disappearing events represent the fact that a token in place p is removed at time r. Observe that for disappearing events we are not interested in the token value. An *event* e is either an appearing event or a disappearing event.

Definition 6. 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.

Definition 7. 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. For each in, several different observations out are possible. The set of all possible output observations out of C being

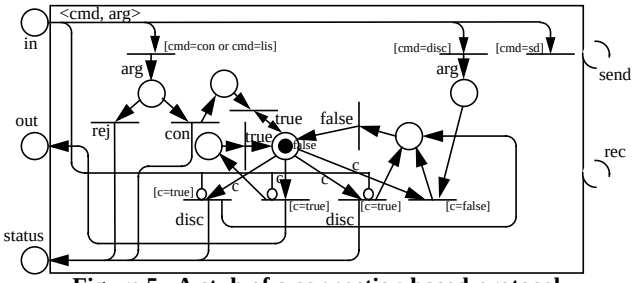


Figure 5. A stub of a connection based protocol

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 $Op_C(in)|_I = \{o_1|_I, o_2|_I, \ldots\}$.

Definition 8. Stub. Let us consider two components, S and C. I_S is the interface of S containing all ports of S. I_C is any interface of C. S is a stub of C with respect to interface I_C iff:

1. Interface I_S is compatible with interface I_C.
2. For any possible input observation in of component C, $Op_C(in)|_{I_C} = Op_S(in|_{I_S})$.

Note that it is not important what happens on other interfaces than I_C. Figure 5 gives an example of a stub of a connection-based protocol component with respect to the interface $I_C = \{\text{in}, \text{out}, \text{status}\}$. Since not all ports are part of the interface, the stub cannot avoid to express a non-deterministic behaviour. For instance, if the stub receives a *connect* request, it can issue non-deterministically either a *rej* (rejected) or *con* (connected) message as an answer to this request. However, in the full component this choice is deterministic, depending on the data exchange on the ports *send* and *rec* with the rest of the system. The degree of non-determinism of a stub depends on the number of component ports which are not included in the interface of the stub.

Definition 9. Top-level interface. The top-level interface of a component C, with respect to a glue logic G, is the set of ports of the component to which the glue logic is connected, $I_{max}^{C,G} = C \cap G$. We will use the simple notation I_{max}, if it is either not important or it is clear from the context, to which component and glue logic we refer.

The ports of a component C, connected to the glue logic G, can be divided into interfaces in many different ways. More precisely, every subset of I_{max} can be considered an interface for which a stub can be constructed. Figure 6 presents a partial order (lattice) of interfaces and hence also of stubs of a component connected to a glue logic through two in-ports (I_1 and I_2) and two out-ports (O_1 and O_2). The lattice induces distinct levels of generality of the stubs. The top-level stub (the stub for the top-level interface), with interface $I_{max} = \{I_1, I_2, O_1, O_2\}$, exhibits exactly the same behaviour as its corresponding component. However, the implementation is not bound to be the same. In the bottom of the lattice, we have the empty interface, for which there does not exist any stub and which is only of theoretical interest. If, for a certain verification, no stubs situated at level 1 or higher are applied at a certain port, then a so called empty stub is connected to that port. In the case of in-ports, the empty stub, $\varnothing_{IN}$, denotes the stub that consumes any token at any point in time. Similarly, the empty stub, $\varnothing_{OUT}$, denotes the stub that generates tokens with random values at any point in time. The models of these stubs are presented in Figure 7. It is useful to introduce the notation $\varnothing_p$ to denote the empty stub at port p. Whether $\varnothing_p$ is equal to $\varnothing_{IN}$ or to $\varnothing_{OUT}$ depends on whether p is an in-port or an out-port. We further elaborate on the use of empty stubs in Section 4.3. Between I_{max} and $\varnothing$, stubs of different levels of generality can be found.

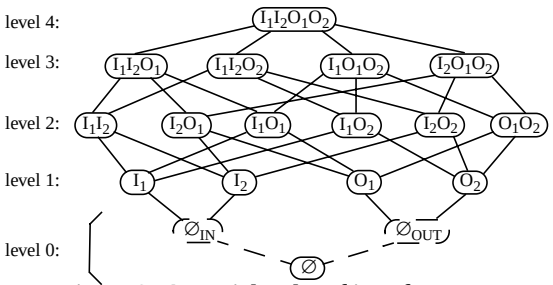


Figure 6. A partial order of interfaces

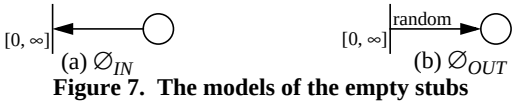


Figure 7. The models of the empty stubs

On level 1, stubs for one-port interfaces can be found. If the interface only contains an in-port, the functionality of the stub is to consume the token at random times which, however, correspond to times when the full component could be able to consume the token, if it would be consumed at all. If it only contains an out-port, the functionality is to issue a new token with random value at random occasions. The value and time are random to the extent that the issued values could, in some circumstance, be issued by the full component at the time in question (note the difference between these stubs and $\varnothing_{IN}$ and $\varnothing_{OUT}$, respectively).

If higher level (level > 1) stubs contain both in-ports and out-ports, a certain degree of causality is introduced. The out-ports can no longer produce any arbitrary value on the tokens, but rather any value still consistent with the token values arriving at the in-ports given the behaviour of the full component. Hence, for instance, in Figure 5 no token on port *out* can be issued unless the stub has received a connection or listen request at port *in* and accepted it. If there are other in-ports of the component, not represented in the interface of the stub, the output is considered non-deterministic from the point of view of the absent in-port, as in the case with the non-deterministic issuing of *rej* and *con* as an answer to a *connect* request described previously in Figure 5.

4.2 The verification process

Our verification approach is illustrated in Figure 8. To verify the glue logic we need to integrate its model with stubs of the components it is connected to. These stubs capture the characteristics of the outputs of the components and, by this, they provide the environment for the glue logic to be verified. The net composed of one or more stubs and the glue logic itself is then passed to the model checker together with (T)CTL formulas corresponding to the involved interfaces of the components. It should be mentioned that, in order to perform the model checking, PRES+ has to be translated into the input language of the particular model checker used. We have discussed the problems related to such a translation into timed automata [5] for the UPPAAL model checking environment [6] in [4]. In addition to the formulas provided together with the components, the designer can add formulas invented by himself to the verification process.

In order to illustrate the verification process, a small example is provided in Figure 9. *Doubler* accepts a token with an integer value at in-port *arg*. In response, it will issue a token at out-port *output* with the value two times the value it received. Component *Strange* will issue one token on out-port *action* as an answer to each token it receives on in-port *input*. These two components are connected through the glue logic in the figure. The glue logic will provide the *Doubler* with an argument, starting with value 0 and increasing each time by one. The reply of the *Doubler* is given to *Strange* which will acknowledge by issuing a token on out-port

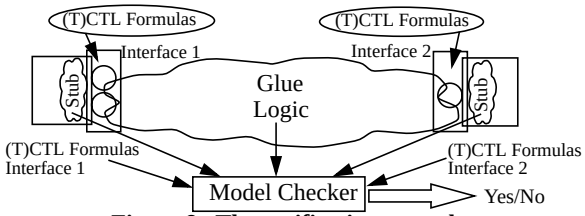


Figure 8. The verification procedure

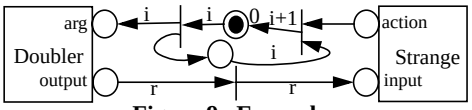


Figure 9. Example

action, which in turn will cause a new integer to eventually be provided to the *Doubler*.

Figure 10 shows the stubs corresponding to the example in Figure 9. Let us elaborate on how different levels of stubs can be used for verification considering the following formulas:

AG ($input \rightarrow input \equiv 0 \pmod 2$) (eq. 3) AG ($arg \rightarrow arg \geq 0$) (eq. 5)

AG ($arg \rightarrow$ A [$output$ R $\neg arg$]) (eq. 4) AGEF $input < 0$ (eq. 6)

To check (eq. 3) (if there is a token in place *input*, then the value of that token must be an even number), only the stubs for the interfaces {*output*} and {*input*} are needed. (eq. 4) (if one argument is received by *Doubler*, another argument may not arrive until the result of the first one is produced), requires top-level stubs at both sides. (eq. 5) (if there is a token in place *arg*, then the value of that token is positive) can be checked with only empty stubs. Let us look at (eq. 6) (there is always a possibility that a negative value may arrive at port *input*) which obviously is not satisfied. However, if empty stubs are used, the verification will indicate that the formula is satisfied. But if the stub {*arg*, *output*} is used, the verification will point out that the property is not true, which is the correct conclusion.

It is obvious that using top-level stubs for all components we will get a correct verification for properties specified by any formula. However, we have a whole lattice of stubs for each component. Thus, the following question has to be answered: Do we always have to use the top-level stubs in order to verify a certain formula? If the answer is "no", then which stub or combination of stubs to use for verification? These questions are of both theoretical and practical importance. From the practical point of view, selecting a certain combination of stubs can reduce the complexity of the verification process and, by this, the verification time. On the other hand, it can happen that certain stubs, possibly the top-level ones, are not available. Thus, it is important to provide a theoretical platform which allows to decide if it is possible to perform a correct verification with a certain combination of available stubs. This theoretical framework will be developed in Section 4.3.

4.3 Properties and relationships of stubs

Definition 10. Interface partition. An interface partition P is a set of non-empty interfaces $P = \{P_1, P_2, \ldots\}$ such that $P_i \cap P_j = \varnothing$ for any i and j, $i \neq j$.

It should be pointed out that each port can at most belong to one interface in every partition. As a consequence of Definition 3, all ports in the same interface must belong to the same component. By convenience, the set of all ports belonging to the interfaces in partition P is denoted $Ports(P) = \bigcup_{i \in P} i$.

Definition 11. 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.

Returning to the example in Figure 9, some possible partitions are: $P = \{\{\text{arg}\}, \{output\}\}$, $Q = \{\{\text{arg}\}, \{output\}, \{action, input\}\}$ and $R = \{\{\text{arg}, output\}, \{action, input\}\}$. $P \propto Q$, since all interfaces in P are subsets of an interface in Q. It is also true that $P \propto R$ and $Q \propto R$. However, it is *not* the case that $R \propto Q$. Intuitively, the interfaces in R are more accurate than those in P or Q, since they capture more of the causalities and dependencies between their ports.

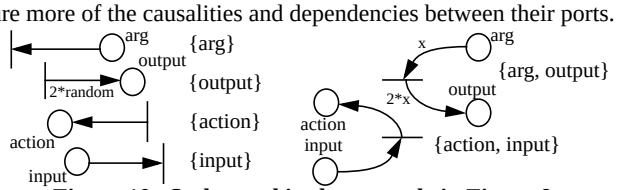


Figure 10. Stubs used in the example in Figure 9

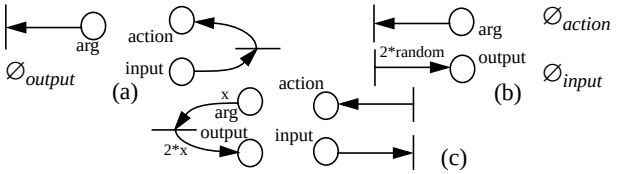


Figure 11. A few environments for the example in Figure 9

Because of lack of space, we have omitted the relatively straight-forward proofs of the following two theorems:

Theorem 1. The partition precedence relation is a partial order.

Theorem 2. The partition precedence relation is a lattice with top element P_{max}, including the top-level interfaces of all connected components, and bottom element $P_{min} = \varnothing$.

Definition 12. Environment. The environment corresponding to a partition $P = \{I_1, I_2, \ldots\}$ with respect to a set of ports J where $Ports(P) \subseteq J$, is defined as $Env(P, J) = (\bigcup_{i \in P} S_i) \cup (\bigcup_{p \in J - Ports(P)} \varnothing_p)$ where each S_i is the stub for interface i, and $\varnothing_p$ is the empty stub attached to port p [1].

Let us consider the example in Figure 9 with the stubs of the components in Figure 10. With $J = \{arg, output, action, input\}$, Figure 11(a) shows the environment $Env(\{\{arg\}, \{action, input\}\}, J)$. Since port *output* is not included in the partition, the empty stub $\varnothing_{output}$ (see Figure 7) has been added. Figure 11(b) shows a similar example for $Env(\{\{arg\}, \{output\}\}, J)$. In Figure 11(c), no empty stub needs to be added for $Env(\{\{arg, output\}, \{action\}, \{input\}\}, J)$, since all ports in J are included in the partition.

If all the individual stubs in $Env(P, J)$ together are viewed as one single component, we obtain the environment corresponding to partition P with respect to the set of ports J. The name stems from the fact that such a component acts as the environment of the glue logic, connected to the ports in J, in the verification process. Based on Theorem 2, it is possible to construct a lattice of partitions, i.e. environments, similar to that done with individual stubs and their interfaces (Figure 6). Figure 12 introduces a very simple example consisting of two interconnected components. In Figure 12(b), we show the interface (stub) lattice corresponding to each of the components. Figure 12(c) depicts the partition (environment) lattice.

Theorem 3. Given an input observation in, two partitions P_1 and P_2, $P_1 \propto P_2$, and a set of ports J where $Ports(P_1), Ports(P_2) \subseteq J$, then $Op_{Env(P_1, J)}(in) \supseteq Op_{Env(P_2, J)}(in)$.

Proof. Assume an observation $o \in Op_{Env(P_2, J)}(in)$. This means that o is a possible output observation given the input observation in. By definition of partition precedence, $\forall p_1 \in P_1 \exists p_2 \in P_2 : p_1 \subseteq p_2$. Hence the restriction operator in $Op_C(in)|_{I_C} = Op_S(in|_{I_S})$ (see Definition 8) 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 13. Generalized operation. The generalized operation Op_C for component C is the union of all operations for every possible input observation, $Op_C = \bigcup_{in} Op_C(in)$.

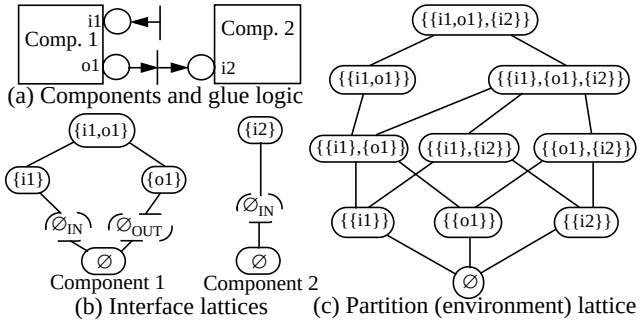


Figure 12. Component, interface and partition relationships

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

According to Definition 7, an operation is the set of all possible outputs given a certain input. The generalized operation is the set of all possible outputs no matter what the input is. The generalized operation allows us to generalize Theorem 3 into the following corollary.

Corollary 1. Given partitions P_1 and P_2, $P_1 \propto P_2$, and a set of ports J where $Ports(P_1), Ports(P_2) \subseteq J$, then $Op_{Env(P_1,J)} \supseteq Op_{Env(P_2,J)}$.

Proof. Follows directly from Theorem 3 and Definition 13. ❑

Definition 14. 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 14 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 $\sigma(\varnothing, M_0) = [\,]$.

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

$$s \models \mathrm{AF}\,\phi \Leftrightarrow \forall \sigma \in P_M(s) \exists j \geq 0 : \sigma[j] \models \phi \qquad \text{(eq. 7)}$$

$$s \models \mathrm{EG}\,\phi \Leftrightarrow \exists \sigma \in P_M(s) \forall j \geq 0 : \sigma[j] \models \phi \qquad \text{(eq. 8)}$$

$P_M(s)$ 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 14, but is a quantified variable. 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 14. (eq. 9) and (eq. 10), where IN is the set of all possible input observations of component C, express the same semantics as (eq. 7) and (eq. 8) in terms of observations and operations.

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

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

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 (eq. 7) and (eq. 8).

In [7] equivalent formulas to (eq. 7) and (eq. 8) are given for TCTL. Based on the discussion above, they can be trivially extended to formulas similar to (eq. 9) and (eq. 10).

Before presenting Theorem 4, it is necessary to introduce a subcategory of CTL formulas, namely ACTL. ACTL formulas are formulas which only contain universal path quantifiers. Moreover, the use of negations is restricted to only be allowed in front of atomic propositions[1]. For example, $\mathrm{AG}\,(p \rightarrow \mathrm{AF}\,q)$ and $\mathrm{AG}\,(p \rightarrow \mathrm{AF}\,\neg q)$ are ACTL formulas while $\mathrm{AG}\,(p \rightarrow \neg\mathrm{AF}\,q)$ and $\mathrm{AG}\,(p \rightarrow \mathrm{EF}\,q)$ are not. ACTL formulas can be extended with time in a similar way as for CTL formulas, whereby TACTL formulas are obtained.

Theorem 4. Assume the partitions P_1 and P_2, $P_1 \propto P_2$, a set of ports J where $Ports(P_1), Ports(P_2) \subseteq J$, an initial marking M_0 on the ports in J and a (T)ACTL formula, e.g. $\mathrm{AF}\,\phi$, also expressed only on the ports in J. If $M_0 \models \mathrm{AF}\,\phi$ for component $Env(P_1, J)$, then it is also true that $M_0 \models \mathrm{AF}\,\phi$ for component $Env(P_2, J)$.

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

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

4.4 Discussion

Section 4.3, in particular Theorem 4, provides the answer to the questions identified at the end of Section 4.2. Let us assume that we have a set C of two or more components which have been interconnected by a glue logic. It has to be verified that a certain property, expressed as a (T)CTL formula ϕ, holds. The following situations can occur:

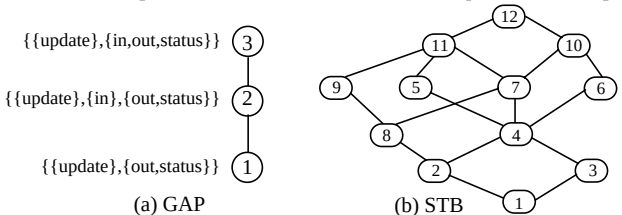
1. Formula ϕ is not a (T)ACTL formula. In this case the verification has to be performed with top-level stubs for all connected components.
2. Formula ϕ is a (T)ACTL formula. In this case, if the formula is satisfied using stubs at any level, the property can be considered as satisfied (this is a direct consequence of Theorem 4 and of the fact that, according to Theorem 2, for any partition P and top-level partition P_{max}, $P \propto P_{max}$).

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

5. EXPERIMENTAL RESULTS

The following experiments concern the verification of systems resulted after the interconnection of components. In the first set of experiments we have verified the glue logic in Figure 4, which interconnects the *Radar* and *Protocol* components of a General avionics platform (GAP) [8]. We illustrate the verification of four properties. Property A is $\mathrm{AGAF}\,\neg update$ (the tokens in port *update* will always be consumed). Property D is $\mathrm{AGAF}\,\neg out$ (the tokens in port *out* will always be consumed). Properties B and C are identical to (eq. 1) and (eq. 2). Consequently, all these formulas are ACTL. Three possible partitions were used whose relationships are shown in the lattice in Figure 13(a). The results of the verification are shown in Table 1. It can be observed that all four properties imposed by the interconnected components are satisfied with the actual glue logic. For property D, the verification can be done using the lowest level of the three partitions (as the property is expressed by an ACTL formula, point 2 in Section 4.4 applies).

The second example refers to a split transaction bus (STB) in a multiprocessor DSP [9]. An overview of the system is shown in Figure 14. 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 pro-



(a) GAP (b) STB

Figure 13. Partition lattices for the GAP and STB examples

[1] An atomic proposition is the lowest level of a formula which does not contain any subformula.

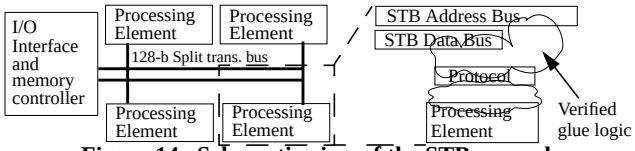


Figure 14. Schematic view of the STB example

cessing 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 and the same procedure continues until the whole block of data has been transmitted. One functionality of the glue logic being verified 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 2 shows the verification results with the STB example. The high number of ports yields a large lattice of partitions. In Figure 13(b) only those which are involved in this particular experiment are included. Partition 12 consists of the top-level stubs for all three connected components. Partition 1 consists of interfaces containing only out-ports.

In order to give a better understanding of the properties, we will have a closer look at two of them. Property B concerns with the fact that the glue logic must issue different commands to the protocol component when the address bus broadcasts the identifiers, depending on the source causing this event to happen. It is formulated as $\mathrm{AG}\,(rec \rightarrow rec \neq \langle \mathrm{TRAN}, a \rangle \wedge a \neq \mathrm{this\_component})$ where TRAN (transaction) is the command to be received by the protocol component when the source causing the event is the one connected to the glue logic under verification. It should not be possible to receive such an event when the address is different from the one of the current component. Property I includes also timing aspects (it is a TCTL formula): $\mathrm{AG}\,((addr.out = \langle \mathrm{ACK}, a \rangle \wedge a = \mathrm{thisComp}) \rightarrow \mathrm{AF}_{<10} addr.in = \mathrm{DRVADD})$. When the address bus has sent an acknowledgment, it expects the command DRVADD to arrive within 10 time units.

Properties A to G are expressed as ACTL formulas, while property H is not (it is a general CTL formula). The last three properties include timing aspects. I and J are TACTL formulas while K also includes existential quantifiers. It can be noticed that property C is not satisfied in the system. The verification result for that property is false, no matter which environment is used. On the other extreme we find properties B, E and J which are satisfied even with the lowest level environment. Hence, being expressed as (T)ACTL formulas, the properties are satisfied with any environment. Property G, also expressed as an ACTL formula, is also satisfied. This can be verified by using the top-level environment, but also by verifying with environment 2. According to point 2 in Section 4.4, 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 properties H and K which are expressed by non-(T)ACTL formulas. Verification with low-level environments is not relevant. The only valid verification is using the top-level environment.

Table 1: Experimental results for GAP example

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

Let us have a look at verification times. For the different properties and environments they are in the range 0.12-689 seconds. For a given property the verification times are small for the very low-level stubs and for the top-level stubs. This is due to the simplicity of the low-level stubs, on the one side, and the high degree of determinism of the top-level stubs (which reduces the state space) on the other side. Between these two limits we can observe a, sometimes very sharp, increase of verification times for the stubs which are at a level close to the top. If they are 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.

6. CONCLUSIONS

We have introduced a methodology to perform formal verification of embedded systems in the context of component reuse. A timed Petri Net based design representation is used.

The verification technique smoothly integrates with communication based design and component reuse. The verification is performed against a set of (T)CTL formulas associated to the components and is based on abstract models of the interconnected components, without assuming any knowledge regarding their implementation. The theoretical framework underlying the methodology has been presented together with examples and experimental results.

7. REFERENCES

[1] J. Haase, "Design Methodology for IP Providers", *Proc. DATE 1999*, 728-732.
[2] J.A. Rowson, A. Sangiovanni-Vincentelli, "Interface-Based Design", *Proc. DAC 1997*, 178-183.
[3] R. Seepold, N. Martínez Madrid, et al., "A Qualification Platform for Design Reuse", *Proc. ISQED 2002*, 75-80.
[4] L.A. Cortés, P. Eles, Z. Peng, "Verification of Embedded Systems using a Petri Net based Representation", *Proc. ISSS 2000*, 149-155.
[5] E.M. Clarke Jr, O. Grumberg, D.A. Peled, "Model Checking", MIT Press 1999.
[6] UPPAAL homepage: http://www.uppaal.com.
[7] R. Alur, C. Courcoubetis, et al., "Model-checking for Real-time Systems", *Proc. Logic in Computer Science 1990*, 414-425.
[8] C.D. Locke, D.R. Vogel, et al., "Building a Predictable Avionics Platform in Ada: A Case Study", *Proc. RTSS 1991*, 181-189.
[9] B. Ackland, A. Anesko, et al., "A Single-Chip, 1.6-Bill., 16-b MAC/s Multiprocessor DSP", *Journ. of Solid-State Circ.*, 35, 3, 2000.

Table 2: Experimental results for 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
I	F 1.4	F 1.7	F 1.5	F 2.5	F 1.2	F 3.2	F 2.6	F 2.3	T 148	F 1.8	T 93.6	T 7.5
J	T 9.4	T 12.4	T 10.0	T 12.5	T 7.0	T 8.0	T 19.0	T 17.7	T 10.7	T 11.4	T 9.7	T 4.0
K	T 0.5	T 0.7	T 0.4	T 0.4	F 54.8	T 0.4	T 0.3	T 0.4	F 24.2	T 0.3	F 26.1	F 5.1

F - property is unsatisfied in the corresponding environment, T - property is satisfied in the corresponding environment. Verification times are given in seconds.