A Theory of Consistency for Modular Synchronous Systems*

Randal E. Bryant[1], Pankaj Chauhan[1], Edmund M. Clarke[1] and Amit Goel[2]

[1] Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA 15213 USA
{bryant, pchauhan, emc}@cs.cmu.edu
[2] Electrical and Computer Engineering Department,
Carnegie Mellon University,
Pittsburgh, PA 15213 USA
{agoel}@ece.cmu.edu

Abstract. We propose a model for modular synchronous systems with combinational dependencies and define consistency using this model. We then show how to derive this model from a modular specification where individual modules are specified as Kripke Structures and give an algorithm to check the system for consistency. We have implemented this algorithm symbolically using BDDs in a tool, SpecCheck. We have used this tool to check an example bus protocol derived from an industrial specification. The counterexamples obtained for this protocol highlight the need for consistency checking.

1 Introduction

The correctness of a system is defined in terms of its specification. In model checking [6], for example, a model of the design is verified against a set of temporal logic properties. However, specifications might have errors themselves. The authors have discovered errors in protocol specifications for the PCI bus and the CoreConnect bus [4, 9]. This highlights the need to examine specifications more carefully.

These problems often occur due to the limitations of natural languages used to describe specifications. However, even formal specifications can have problems. Sometimes, it is not possible to realize the specification in any implementation, while at other times, the system could deadlock. Bus specifications often allow *combinational dependencies* which are sometimes desirable for efficiency reasons. These dependencies might cause a module to exhibit deadlock only on

* This work was supported in part by the National Science Foundation under Grant no. CCR-9803774 and in part by the MARCO/DARPA Gigascale Silicon Research Center (http://www.gigascale.org). Their support is gratefully acknowledged. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

certain inputs. Problems like these are often attributed to inconsistencies in the specification. We propose a theory that formalizes the notion of *consistency* for modular synchronous systems.

The main contributions of this paper are a model for representing modular synchronous systems that incorporates combinational dependencies, a definition of consistency for this model, a method to obtain this model from any given specification in which each module can be described by a Kripke structure [6], and an algorithm to check for consistency for this model.

We motivate the paper with the following small examples in which the specifications are expressed in Linear Temporal Logic (LTL) [6]. They demonstrate inconsistent behavior as explained.

Example 1. Consider the following trivial example. The specification for a module consists of two properties:

a. $\mathbf{G}\,\neg reset$: *reset* should never be asserted.
b. $reset$: *reset* should be asserted initially.

We can see that it is impossible to satisfy this specification.

Example 2. As another example, consider an arbiter that receives requests from two masters with the following specification:

a. $\mathbf{G}\,(req_0 \rightarrow \mathbf{X}\,ack_0)$: If master 0 makes a request, it should be acknowledged in the next cycle.
b. $\mathbf{G}\,(req_1 \rightarrow \mathbf{X}\,ack_1)$: If master 1 makes a request, it should be acknowledged in the next cycle.
c. $\mathbf{G}\,\neg\,(ack_0 \wedge ack_1)$: Both masters should not be acknowledged at the same time.

If both req_0 and req_1 are asserted, there is no valid next state assignment for ack_0 and ack_1. Whatever the arbiter does, at least one property will **not** be satisfied.

Before the next example, let us consider a *Master* and *Slave* that communicate using the signals *req* and *ack*. Suppose that one of the requirements of the Master is expressed by the LTL formula $\mathbf{G}(req \rightarrow req\ \mathbf{U}\ ack)$. Once asserted, *req* should stay asserted until it is acknowledged. This property allows for *req* to be deasserted in the same cycle in which *ack* is asserted. Therefore, the next state value of *req*, i.e. req', depends not only on the current state value of *req* and *ack*, but also on ack', the *next state value* of *ack*. The timing diagram in figure 1 illustrates this. A combinational dependency such as this can sometimes lead to inconsistent behavior as seen in the next example.

Example 3. The following is a specification for a device with output *req* and inputs *busy* and *ack*:

a. $\mathbf{G}\,(req \rightarrow req\,\mathbf{U}\,ack)$: Once request is asserted it remains asserted till the request is acknowledged.
b. $\mathbf{G}\,(busy \rightarrow \mathbf{X}\,(\neg req))$: A request should not be made if the bus is busy.

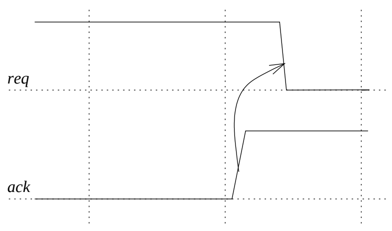

Fig. 1. Timing diagram showing combinational dependency between *req* and *ack*.

In this case, the problem arises when a request has already been made and the bus becomes busy (possibly because of some other device). The second property requires *req* to be deasserted but the first property then requires that *ack* be asserted. However, *ack* is an input and cannot be controlled by the device under consideration. This means that the specification breaks down on certain inputs. This problem is similar to that of receptiveness defined by Dill in [8] for *asynchronous systems*. However, if the specification of the module controlling *ack* is constrained ($\mathbf{G}(busy \to \mathbf{X}\, ack)$), then the problem disappears.

In our model, which we call the Synchronous Consistency Model (SCM), we divide a synchronous step into several phases which we call *micro-transitions*. After each micro-transition, the state of the system is partially updated. A transition to the next state is complete after the last micro-transition. The partial updates to the states due to micro-transitions are represented by *micro-states*. Micro-transitions allow us to capture combinational dependencies between signals.

The concept of micro-transitions is not entirely new. Most hardware description languages such as VHDL and VERILOG have some notion of combinational dependency. ESTEREL [2] allows dynamically scheduled sub-rounds. Alur and Henzinger [1] also break a round (synchronous step) into sub-rounds (micro-transitions) in the Reactive Module Language(RML). While Alur and Henzinger describe an operational modeling language, we do not present any language. Instead, we decribe how to derive the SCM from modular specifications and check it for consistency. In RML, the model explicitly specifies the behavior of sub-rounds. We want the micro-transitions to be synthesized automatically from high level, declarative specifications.

For our experiments, we have used LTL for specification and tableau construction to derive Kripke structures. Other approaches might be more suitable. Shimizu et. al. [12] describe a monitor based formal specification methodology for modular synchronous systems. Clarke et. al. [7] describe a way to obtain executable protocol specification and describe algorithms that enable them to debug specifications. However, neither of them incorporate combinational dependencies. We believe that these two approaches are complementary to our work. Given monitors or executable models, our approach can be used to check for consistency.

The organization of the rest of the paper is as follows. In Section 2, we discuss preliminaries needed for the rest of the paper. We define the Synchronous Consistency Model and **consistency** for systems using this model in Section 3. Section 4 describes how to derive a Synchronous Consistency Model from a modular specification. We present our algorithm to check the consistency of an SCM in Section 5. In section 6 we introduce SpecCheck, a prototype tool we have built for consistency checking. We also describe a system and its specification, based on an industrial bus protocol, that we checked using our tool. Finally, we conclude our paper with directions for future research in section 7. In Appendix A we present the complete specification for the example system described in Section 6.

2 Preliminaries

As mentioned in the introduction, we derive a Synchronous Consistency Model from modular specifications where each module is specified as a Kripke structure. Section 2.1 defines a Kripke structure. Section 2.2 defines parallel composition for Kripke structures. We will use parallel composition to derive a global state transition graph for the specification. The example specifications we use in this paper are expressed in LTL, which is defined in Section 2.3.

2.1 Kripke Structures

A Kripke structure [6] T is a tuple (S, S_0, R, AP, L) where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation, AP is the set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

If a specification is expressed in temporal logic, then tableau construction methods [5, 10] produce a Kripke structure for that specification. For example, the method described for LTL properties in [5] produces a Kripke structure with every path that satisfies the LTL property. Figure 2(a) shows a tableau for a module with the only requirement being $\mathbf{G}(a \rightarrow \mathbf{X}\,\neg b)$. Figure 2(b) shows a tableau for $\mathbf{G}(b \rightarrow b\,\mathbf{U}\,c)$.

2.2 Synchronous Parallel Composition

Let $T' = (S', S'_0, AP', L', R')$ and $T'' = (S'', S''_0, AP'', L'', R'')$ be two tableaux. The synchronous parallel composition [6] of T' and T'' denoted by $T' \parallel T''$ is the structure $T = (S, S_0, AP, L, R)$ defined as follows.

1. $S = \{(s', s'') \mid L'(s') \cap AP'' = L''(s'') \cap AP'\}$.
2. $S_0 = (S'_0 \times S''_0) \cap S$.
3. $AP = AP' \cup AP''$.
4. $L\left((s', s'')\right) = L'(s') \cup L''(s'')$.
5. $R\left((s', s''), (t', t'')\right)$ if and only if $R'(s', t')$ and $R''(s'', t'')$.

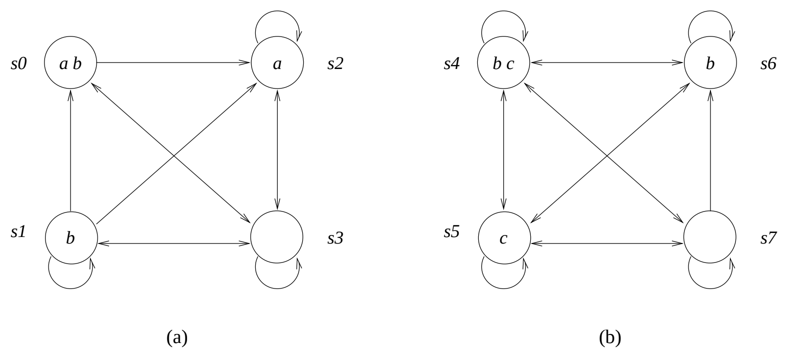

Fig. 2. (a) A tableau for $\mathbf{G}(a \rightarrow \mathbf{X}\,\neg b)$. (b) A tableau for $\mathbf{G}(b \rightarrow b\,\mathbf{U}\;c)$.

If $s = (s', s'') \in S$ then we say that s' and s'' are components of s. Figure 3 shows the structure obtained from the synchronous parallel composition of the two structures in Figure 2.

This definition of composition models *synchronous* behavior. States of the composition are pairs of component states that agree on the common atomic propositions. Each transition of the composition involves a joint transition of the two components. The parallel composition of more than two tableaux is defined similarly.

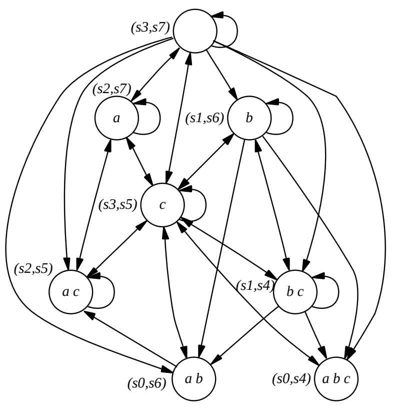

Fig. 3. Synchronous Parallel Composition of the Tableaux in Figure 2

2.3 LTL

For a set of atomic proposition AP, the set of LTL formulas is defined as follows:

- an atomic proposition $p \in AP$ is an LTL formula.
- If f and g are LTL formulas, then $\neg f$, $f \vee g$, $\mathbf{X}\, f$ and $f\,\mathbf{U}\, g$ are LTL formulas.

The following abbreviations are also used:

- $f \wedge g = \neg(\neg f \vee \neg g)$
- $\mathbf{F}\, f = \mathit{true}\, \mathbf{U}\, f$
- $\mathbf{G}\, f = \neg\, \mathbf{F} \neg f$

For a complete discussion of LTL and its semantics refer to [6].

3 Synchronous Consistency Model

The model we are proposing augments the state transition graph of the system with extra information to capture combinational dependencies. As discussed in the introduction, these dependencies are an important aspect of many bus protocol specifications [9].

Definition 1 (Synchronous Consistency Model (SCM)). *A* synchronous consistency model *M is a six-tuple (S, R, P, C, n, Φ) where*

1. S is the set of observable states.
2. $R \subseteq S \times S$ is a transition relation.
3. P is the set of micro-states.
4. $C : S \to P$ maps observable states to micro-states.
5. $n \in \mathbf{N}$ is the number of micro-transitions in a sequential step.
6. $\Phi = \{\phi_{s,i} \subseteq P \times P | s \in S, 1 \leq i \leq n\}$ is the set of micro-transition relations.

(S, R) defines a synchronous state transition graph. We say that the states in S are *observable*. The micro-states in P capture the intermediate stages of computation within a synchronous transition and are *unobservable*. For each observable state $s \in S$, there is a corresponding micro-state $C(s) \in P$. We represent a synchronous transition from $s \in S$ as a sequence of micro-transitions beginning at $C(s)$. Without loss of generality, we assume that all transitions between observable states have the same number n of micro-transitions. The relation $\phi_{s,i} \subseteq P \times P$ describes the allowed partial updates in the i^{th} micro-step, starting from the observable state s. We require that $\phi_{s,i}$ be defined for all observable states s and for each micro-step up to n. In order to define what *consistency* means, we need the following definitions.

A *valid sequence* of micro-states is required to begin in a micro-state corresponding to an observable state and obey the micro-transition relations beginning in that state.

Definition 2 (Valid Sequence). *A sequence of micro-states $\langle p_0, p_1, \ldots, p_l \rangle$ is a* valid sequence *with respect to a model $M = \langle S, R, P, C, n, \Phi \rangle$ iff there exists an observable state $s \in S$ such that $C(s) = p_0$, and for all i such that $1 \leq i \leq l$, $(p_{i-1}, p_i) \in \Phi_{s,i}$.*

A *valid transition sequence* is a valid sequence that ends in a micro-state corresponding to an observable state. A valid transition sequence corresponds to a synchronous transition in the state transition graph (S, R).

Definition 3 (Valid Transition Sequence). *A sequence* $\langle p_0, p_1, \ldots, p_n\rangle$ *is a* valid transition sequence *with respect to a model* $M = \langle S, R, P, C, n, \Phi\rangle$ *iff there exist observable states* s *and* s' *such that:*

1 $C(s) = p_0$

2 for all i *such that* $1 \leq i \leq n$, $(p_{i-1}, p_i) \in \Phi_{s,i}$

3 $C(s') = p_n$

4 $(s, s') \in R$

A valid sequence which can not be extended to form a valid transition sequence represents inconsistent behavior. We call these sequences *divergent sequences.*

Definition 4 (Divergent Sequence). *A sequence is a* divergent sequence *with respect to a model* $M = \langle S, R, P, C, n, \Phi\rangle$ *iff it is a valid sequence with respect to* M*, and it is not a prefix of any valid transition sequence in* M*.*

Definition 5 (Micro-transition Conformance). *The set of micro-transition relations* Φ *for a model* $M = \langle S, R, P, C, n, \Phi\rangle$ conforms *to the transition relation* R *iff for every* $(s, s') \in R$ *there exists a valid transition sequence beginning and ending in micro-states* $C(s)$ *and* $C(s')$*, respectively.*

We are now ready to define what it means for a model to be *consistent.* If a model satisfies this definition, then the consistency problems described in the introduction can be avoided.

Definition 6 (Consistent Model). *A model* $M = \langle S, R, P, C, n, \Phi\rangle$ *is said to be* consistent *iff it satisfies the following conditions:*

1 $S \neq \emptyset$.

2 R *is total, i.e., for every state* $s \in S$ *there is a state* $s' \in S$ *such that* $R(s, s')$.

3 The set of micro-transition relations Φ *conforms to the transition relation* R.

4 There are no divergent sequences in M.

If the model is derived from a specification, then the first condition ensures that the specification is satisfiable, while the second checks for the absence of deadlocks. Conformance implies that the micro-transition relation implements the global transition relation. The absence of divergent sequences guarantees that the system does not get stuck after a valid sequence of micro-transitions.

4 Deriving an SCM from Modular Specifications

We have defined *consistency* for an SCM. We now describe a way to derive an SCM $M = (S, R, P, C, n, \Phi)$ from modular specifications. We are given a set of modules M_i and a partial order $\prec$ on AP. The relation $\prec$ is used to capture combinational dependencies. If b depends on a then $a \prec b$. Each module is described by a Kripke structure T_i.

The variables occurring in the structure for each module are classified either as input or output variables for that module. We say that a module M_i controls its output variables and require that a variable be controlled by exactly one[1] module. This allows us to define a function γ from the set of variables to the set of modules. $\gamma(v_a) = M_a$ iff the variable v_a is controlled by the module M_a.

Let $T = \big(S, S_0, AP, L, R\big)$ be the global Kripke structure obtained by the parallel composition $T_1 \parallel T_2 \parallel \ldots \parallel T_m$. We restrict the states of T to only those reachable from S_0 to obtain (S, R).

The set of micro-states P is defined as the power set of the atomic propositions in T, i.e., $P = 2^{AP}$. The mapping C from observable states to micro-states is then defined by L.

We use $\prec$ to derive the number of micro-transitions in a synchronous transition, n, and the set of micro-transition relations, Φ. We can partition the atomic propositions AP into disjoint sets by levelizing $\prec$. A set in the levelized partition may contain variables controlled by different modules. We further split each set to obtain the partition Y so that the resulting sets Y_i in $Y \doteq \big\langle Y_1, \ldots, Y_n \big\rangle$ have variables controlled by only one module. The number of sets n in Y determines the number of micro-transitions in M. Y satisfies the following properties:

1. $\cup_{i=1}^{n} Y_i = AP$, i.e. the partitioning is exhaustive,
2. $Y_i \cap Y_j = \emptyset, i \neq j$, i.e. the sets in the partition are disjoint,
3. $[x_i \prec x_j \text{ and } x_i \in Y_k,\ x_j \in Y_l] \Rightarrow k < l$, i.e. the set of atomic propositions is *levelized* by $\prec$,
4. $\forall v_a, v_b \in Y_i \cdot \gamma(v_a) = \gamma(v_b)$, i.e. all variables in the same partition are controlled by the same module.

Define the function $\pi : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, m\}$ such that $M_{\pi(k)}$ is the controlling module for variables in Y_k, i.e., $\gamma(v) = M_{\pi(k)}$ for all $v \in Y_k$. This function is well defined since the controlling module for all variables in Y_k is the same.

The partitioning Y tells us the order in which next state variables are updated. We begin with an observable micro-state p_0 in which no variable has been updated yet. Then the next state values of variables in $Y_1, Y_2, \ldots, Y_n$ are computed in that order. The i^{th} micro-transition depends on transitions in the tableau for the controlling module for that micro-step.

Consider the example in Figure 4, which shows an observable state s such that $L(s) = \{a, b\}$ in the global Kripke structure of specifications for two modules M_1 and M_2. M_1 controls a and b and its properties are $\mathbf{G}(a \rightarrow \mathbf{X}\,\neg b)$ and $\mathbf{G}(b \rightarrow b\,\mathbf{U}\,c)$ (Figure 3). M_2 controls c and is unconstrained. The partitioning $Y = \langle\{a\}, \{c\}, \{b\}\rangle$ of variables is a legal partitioning for the given partial order $c \prec b$. The micro-state corresponding to s is p_0. Since there are three sets in Y, there are three micro-transitions. Figure 4, shows all the legal micro-transitions for the system starting from observable state s.

[1] This allows only closed systems. However, that is not a problem since we can always introduce an extra module to control the primary inputs to the system, without restricting their behavior.

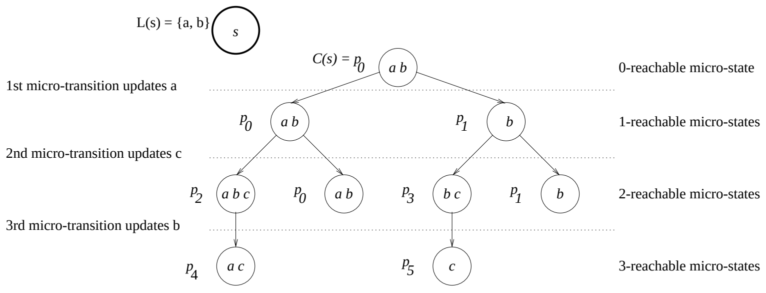

Fig. 4. Legal micro-transitions possible from $C(s_0) = p_0 = \{a, b\}$ for specifications $b \rightarrow b \, \mathbf{U} \, c$ and $a \rightarrow \mathbf{X} \neg b$

In the second micro-transition, M_2 updates the next state value of c, while the next state value of a has already been updated, and the next state value of b will be updated in the next micro-transition. Let s_j be the component of s in M_2. Suppose that after the first micro-transition we get to p_1, then the partially updated value of a is false. This restricts transitions in M_2 to those next states where a is false. M_2 can update c to either true or false. If c is updated to false, then M_1 is restricted to those transitions in which the next state values of a and c are both false. But there is no such transition in M_1 from s_j. Hence there is no third micro-transition from p_1 starting from s. The following recursive definitions formally capture this idea.

Definition 7 (i-reachability). *A micro-state $p_k \in P$ is* i-reachable *from $s \in S$ iff there exists a valid sequence of length $i+1$ beginning with $C(s)$ and ending with p_k.*

Definition 8 ($\mathbf{\Phi_{s,i}}$). *For $p_a, p_b \in P$, $(p_a, p_b) \in \Phi_{s,i}$ iff p_a is $(i-1)$-reachable from s and there exist $s_j, s_k \in S_{\pi(i)}$ such that:*

1. *s_j is a component of s.*
2. *$R_{\pi(i)}(s_j, s_k)$.*
3. *$p_b = \big(p_a - Y_i\big) \cup \big(L_{\pi(i)}(s_k) \cap Y_i\big)$. The micro-state p_b is obtained from p_a by retaining the values of all variables except those in Y_i. The updated value of Y_i is extracted from s_k.*
4. *$p_a \cap \big(Y_1 \cup \ldots \cup Y_{i-1}\big) \cap AP_{\pi(i)} = L_{\pi(i)}(s_k) \cap \big(Y_1 \cup \ldots \cup Y_{i-1}\big)$. The values of variables updated before the i^{th} micro-transition agree with their values in s_k.*

For every observable state s there is only one 0-reachable micro-state, $C(s)$. This allows us to compute $\Phi_{s,1}$ which in turn gives us the set of 1-reachable micro-states from s. Using these definitions recursively, we can compute Φ. The following theorem guarantees that the Φ obtained in this manner conforms to R.

Theorem 1. *The set of micro-transition relations Φ obtained from definitions 7,8 for the SCM $M = (S, R, P, C, n, \Phi)$ conforms to the transition relation R.*

Proof: Given $s = (s_1, s_2, \ldots, s_m)$ and $s' = (s'_1, s'_2, \ldots, s'_m)$ such that $R(s, s')$, we need to prove that there exists a valid transition sequence beginning and ending in $C(s)$ and $C(s')$ respectively. We construct a sequence $\langle p_0, p_1, \ldots, p_n \rangle$ as follows:

- $p_0 = C(s)$
- $p_i = (p_{i-1} - Y_i) \cup (L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i)$ for all $1 \leq i \leq n$

If we can prove that $(p_{i-1}, p_i) \in \Phi_{s,i}$ for all $i \in \{1, \ldots, n\}$ and that $C(s') = p_n$ then $\langle p_0, p_1, \ldots, p_n \rangle$ is one such valid transition sequence.To see that $(p_{i-1}, p_i) \in \Phi_{(s,i)}$, we consider $s_{\pi(i)}$ and $s'_{\pi(i)}$. State $s_{\pi(i)}$ is a component of s. $R(s, s') \Rightarrow R_{\pi(i)}(s_{\pi(i)}, s'_{\pi(i)})$, and $p_i = (p_{i-1} - Y_i) \cup (L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i)$ by construction. It then remains to be shown that $p_{i-1} \cap (Y_1 \cup \ldots \cup Y_{i-1}) \cap AP_{\pi(i)} = L_{\pi(i)}(s'_{\pi(i)}) \cap (Y_1 \cup \ldots \cup Y_{i-1})$.
We now prove by induction that $p_i \cap (Y_1 \cup \ldots \cup Y_i) = L(s') \cap (Y_1 \cup \ldots \cup Y_i)$. It then follows that (a) $p_{i-1} \cap (Y_1 \cup \ldots \cup Y_{i-1}) \cap AP_{\pi(i)} = L_{\pi(i)}(s'_{\pi(i)}) \cap (Y_1 \cup \ldots \cup Y_i)$, and (b) $p_n \cap (Y_1 \cup \ldots \cup Y_n) = L(s') \cap (Y_1 \cup \ldots \cup Y_n)$. Since the partitioning Y is exhaustive, $(Y_1 \cup \ldots \cup Y_n) = AP$, $p_n = L(s') = C(s')$.

For the base case, we consider p_1. By our construction $p_1 = (p_0 - Y_1) \cup (L_{\pi(1)}(s'_{\pi(1)}) \cap Y_1)$. This implies $p_1 \cap Y_1 = (L_{\pi(1)}(s'_{\pi(1)}) \cap Y_1)$. Since $s'_{\pi(1)}$ is a component of s', $p_1 \cap Y_1 = L(s') \cap Y_1$.

For our inductive hypothesis, assume that $p_{i-1} \cap (Y_1 \cup \ldots \cup Y_{i-1}) = L(s') \cap (Y_1 \cup \ldots \cup Y_{i-1})$.

$$\begin{aligned}
p_i &= [p_{i-1} - Y_i] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i] \\
\Rightarrow p_i \cap (Y_1 \cup \ldots \cup Y_i) &= [(p_{i-1} - Y_i) \cap (Y_1 \cup \ldots \cup Y_i)] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap (Y_1 \cup \ldots \cup Y_i)] \\
&= [p_{i-1} \cap (Y_1 \cup \ldots \cup Y_{i-1})] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i] \\
&= [L(s') \cap (Y_1 \cup \ldots \cup Y_{i-1})] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i] \\
&\quad \text{by induction hypothesis} \\
&= [L(s') \cap (Y_1 \cup \ldots \cup Y_{i-1})] \cup [L(s') \cap Y_i] \\
&\quad \text{because } s'_{\pi(i)} \text{ is a component of } s' \\
\Rightarrow p_i \cap (Y_1 \cup \ldots \cup Y_i) &= L(s') \cap (Y_1 \cup \ldots \cup Y_i) \,\Box
\end{aligned}$$

5 Algorithm to Check Consistency

The heart of our algorithm for checking consistency is a procedure for computing the set of i-reachable micro-states from an observable state s. We compute the i-reachable micro-states from s by updating the next state values of Y_i variables in the $(i-1)$-reachable micro-states from s. The valuation of $\tilde{Y} = Y_{i+1} \cup \ldots \cup Y_n$ remains unchanged. As in the previous section, $M_{\pi(i)}$ is the controlling module for the variables in the partition Y_i. Given γ, it is easy to compute $\pi(i)$. In this computation, if we find that there exists an $(i-1)$-reachable micro-state for which there is no successor micro-state, then we have a divergent sequence of length $i - 1$. This implies that the model is inconsistent by definition 6.

The explicit-state algorithm RECCHECK shown in Figure 5 returns false if there is a divergent sequence beginning with $C(s)$. The loop in line 4 iteratively computes P_i, the set of i-reachable micro-states from s. Line 6 assigns $Y_{i+1} \cup \ldots \cup Y_n$ to Y_{after}. In line 7, we initialize P_i to the empty set. The loop beginning in line 8 checks for every $(i-1)$-reachable micro-state p_{i-1}, if we can extend any valid sequence of length i ending in p_{i-1}. If it can be extended, then all i-reachable micro-states p_i which extend these sequences are added to P_i (line 13) and the flag *extended* is set to true. If there is a divergent sequence, *extended* remains false in line 15 and the procedure terminates by returning false. In line 9, l denotes the valuation of the variables that have been updated in the previous steps ($Y_{before} = Y_1 \cup \ldots \cup Y_{i-1}$). Line 10 initializes the extended flag to false. The loop beginning in line 11 iterates over all states s_k in the image of s_j in the controlling module for the i^{th} micro-transition. Line 12 checks if s_k is compatible with the updates made in the previous micro-transitions (l). Line 17 adds Y_i to Y_{before}.

The explicit-state procedure SPECCHECK for checking consistency is also described in Figure 5. Note that we do not check the micro-transition relation for conformance since that is guaranteed by Theorem 1. In fact, we do not explicitly construct Φ. We check for divergent sequences by the i-reachability procedure in Figure 5.

As mentioned, the algorithms presented in this section are explicit-state. We have implemented symbolic state versions of these algorithms in our tool, SpecCheck, described in the next section.

6 Implementation and Experimental Results

We implemented a prototype tool, SpecCheck, that performs consistency checking on a given specification for a synchronous modular system. The input to the tool is a list of modules along with a set of properties expressed in LTL for each module, and a partial order, $\prec$, on the atomic propositions used. To derive tableaux for LTL properties, we used a slight modification of the construction described by Clarke et. al. in [5] symbolically using BDDs [3]. After deriving a tableau for each LTL property, we compose tableaux of all LTL properties to derive a Kripke structure for a module and in turn compose these modules to form a global Kripke structure. There is no unique tableau for an LTL property. We have observed spurious deadlocks in some cases, depending on the tableau construction used. Our modification to the method of [5] gets rid of the spurious deadlocks in our examples. We compute the set of i-reachable micro-states, P_i, symbolically using BDDs. P_i is a set of tuples (s, p), where $(s, p) \in P_i$ if and only if p is i-reachable from s. The set of states is encoded using $AP \cup A$, where A is a set of auxiliary state variables which are used to differentiate between states with the same labelling. The label of a state can be obtained by existentially quantifying the auxiliary variables. The set of micro-states is encoded using AP. SpecCheck is implemented in Moscow ML [15] which is an implementation of

```
RecCheck($s, n, Y, \pi, \langle T_1, T_2, \ldots, T_m \rangle$)
$Y_{\text{before}} \leftarrow \emptyset$
$Y_{\text{after}} \leftarrow Y$
$P_0 \leftarrow L(s)$
for $i = 1$ to $n$
{
        $s_j \leftarrow$ Component($s, \pi(i)$)
        $Y_{\text{after}} \leftarrow Y_{\text{after}} - Y_i$
        $P_i = \emptyset$
        for every $p_{i-1} \in P_{i-1}$
        {
                $l \leftarrow p_{i-1} \cap Y_{\text{before}}$
                $extended \leftarrow$ false
                for every $s_k \in S_{\pi(i)}$
                {
                        if ($R_{\pi(i)}(s_j, s_k)$ and $(L_{\pi(i)}(s_k) \cap Y_{\text{before}}) = l$)
                                $P_i = P_i \cup \{(p_{i-1} - Y_i) \cup (L_{\pi(i)}(s_k) \cap Y_i)\}$
                                $extended \leftarrow$ true
                }
                if ($extended =$ false)
                        return false
        }
        $Y_{\text{before}} \leftarrow Y_{\text{before}} \cup Y_i$
}
return true
```

```
SpecCheck($\langle T_1, T_2, \ldots, T_m \rangle, \gamma, \prec$)
$(S, R) \leftarrow$ Compose($\langle T_1, T_2, \ldots, T_m \rangle$)
if ($S = \emptyset$) return false
if $R$ is not total return false
$\hat{Y}$ = Levelize($AP, \prec$) /* levelize $\prec$ */
$(Y, \pi)$ = ModPartition($\hat{Y}, \gamma$)
/* refine $\hat{Y}$ so that each partition has variables controlled by only one module */
for every $s \in S$
{
        if (RecCheck($s, n, Y, \pi, \langle T_1, T_2, \ldots, T_m \rangle$) $= false$) return false
}
return true
```

Fig. 5. RecCheck, algorithm to check for the absence of divergent sequences and SpecCheck, algorithm to check consistency

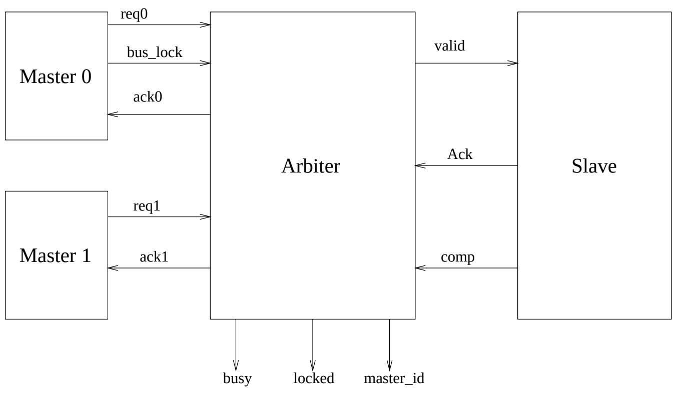

Fig. 6. Example System

standard ML [11]. The symbolic computations are performed using MuDDy [13] which is an ML interface to the BuDDy [14] BDD package.

We used our tool to check specifications for the system depicted in figure 6. Two version of specifications for this system are described in more detail in the appendix A. The system consists of two master devices and one slave connected to an arbiter.

The general flow of control is as follows. The masters request control of the bus using their respective *req* signals. The arbiter passes the request to the slave (*valid*) and when the slave acknowledges the request (*Ack*), the arbiter passes the acknowledgement to the requesting master (*ack0* or *ack1*). After acknowledging a request, the slave indicates completion of the request by asserting *comp*.

The *busy* signal is used to indicate when the bus is busy. Master 0 can also try to lock the bus by asserting *bus_lock* along with its request. If the bus is locked, this is indicated by *locked*. In the locked state, the arbiter ignores requests made by master 1.

The *master_id* signal indicates which master currently controls the bus. If the bus is idle, the value is undefined. Arbitration occurs when the bus is free. The first master to make a request when the bus is free is granted the bus (by setting *master_id* appropriately). If both masters make a request, then the request from master 0 gets priority. In version A, the arbiter asserts *valid* one cycle after arbitration, whereas in version B, *valid* is asserted in the same cycle. In the appendix A, we have precisely described the properties of these modules in LTL.

Figure 7(a) shows a counterexample for version A, demonstrating a deadlock, while figure 7(b) demonstrates a receptiveness problem with version B.

In figure 7(a), there is no valid next state because of the following properties.

1. $\mathbf{G}(((\mathit{req0} \vee \mathit{req1}) \wedge \neg \mathit{busy}) \rightarrow \mathbf{X}\, \mathit{valid})$: If there is a request and the bus not busy, then in the next cycle *valid* must be asserted.
2. $\mathbf{G}(\mathit{locked} \wedge \neg \mathit{req0} \rightarrow \mathbf{X}\, \neg \mathit{valid})$: If the bus is locked and master 0 is not making a request, then *valid* cannot be asserted in the next cycle.

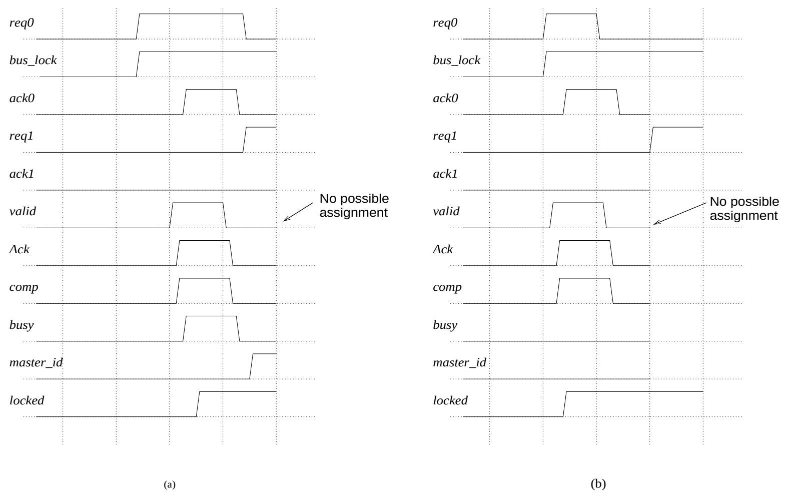

Fig. 7. (a) Deadlock in the specifications. (b) Receptiveness problem in specifications

Property 1 requires *valid* to be asserted in the next state, while property 2 requires it to be deasserted. Hence there is no possible next state in the trace shown. This inconsistency can be removed by replacing the first property with the following two properties:

1(a). $\mathbf{G}((\mathit{req0} \wedge \neg \mathit{busy}) \rightarrow \mathbf{X}\, \mathit{valid})$: If master 0 makes a request and the bus is not busy, then in the next cycle *valid* must be asserted.

1(b). $\mathbf{G}((\mathit{req1} \wedge \neg(\mathit{busy} \vee \mathit{locked})) \rightarrow \mathbf{X}\, \mathit{valid})$: If master 1 makes a request and the bus is neither busy nor locked, then in the next cycle *valid* must be asserted.

In figure 7(b), an incomplete trace for version B is shown. In the last cycle, only *req0*, *bus_lock*, *req1* and *locked* are assigned. If *req1* had remained deasserted in the last cycle, there would have been no problem. But because it is asserted, there is no possible assignment for *valid*. The following properties lead to this receptiveness problem.

3. $\mathbf{G}((\neg \mathit{valid} \wedge \neg \mathit{busy}) \rightarrow \mathbf{X}((\mathit{req0} \vee \mathit{req1}) \leftrightarrow \mathit{valid}))$: If *valid* is deasserted and the bus is not busy, *valid* is asserted in the next cycle if and only if there is a request in that cycle.
4. $\mathbf{G}(\mathit{locked} \wedge \mathit{valid} \rightarrow \mathit{req0})$: When the bus has been locked by master 0, ignore all requests by master 1, so *valid* just reflects master 0's requests.
5. $\mathbf{G}(\mathit{locked} \rightarrow \mathbf{X}(\mathit{locked} \leftrightarrow \mathit{bus\_lock}))$: If the bus is locked, it remains locked while master 0 keeps *bus_lock* asserted.

Just like for version A, this inconsistency can be resolved by replacing property 3 with the following two properties:

3(a). $\mathbf{G}((\neg \mathit{valid} \wedge \neg \mathit{busy} \wedge \mathit{locked}) \rightarrow \mathbf{X}(\mathit{req0} \leftrightarrow \mathit{valid}))$:If *valid* is deasserted and the bus is not busy but locked, *valid* is asserted in the next cycle if and only if master 0 makes a request in that cycle.

3(b). $\mathbf{G}((\neg valid \wedge \neg busy \wedge \neg locked) \rightarrow \mathbf{X}((req0 \vee req1) \leftrightarrow valid))$:If *valid* is deasserted and the bus is neither busy nor locked, *valid* is asserted in the next cycle if and only if there is a request in that cycle.

7 Conclusion and Future Research

We have proposed a theory for consistency of modular synchronous systems with combinational dependencies, and developed an algorithm that allows us to check for inconsistencies. The algorithm has been implemented in a prototype tool, SpecCheck, which has been used to find bugs in the examples described in the paper.

Nevertheless, there are a number of directions for future research. We have assumed that the partial order $\prec$ on AP is specified by the user. We believe that it may be possible to derive this order automatically from a specification in temporal logic. The starting point in our check for consistency is a modular specification given as a collection of Kripke structures. We have observed that it is possible to obtain spurious deadlocks with SpecCheck, depending on the tableau construction method used. Similar issues are discussed in [6]. A better understanding of tableau construction methods and how they are used in SpecCheck should enable us to eliminate the spurious deadlocks that we have observed. We need to understand better what consistency means for actual implementations. The standard notion of simulation between an implementation and a more abstract model is not sufficient to guarantee consistency for the implementation. Finally, we need to extend SpecCheck to handle hierarchical systems. We believe that this can be done within our current framework.

References

1. R. Alur and T.A. Henzinger. "Reactive modules." In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pp. 207-218. 1996.
2. G. Berry, G. Gonthier. "The synchronous programming language ESTEREL: Design, semantics, implementation." Technical Report 842, INRIA. 1988.
3. R. E. Bryant. "Graph-based algorithms for boolean function manipulation." *IEEE Transactions on Computers, C-35(8)*, pp. 677-691. 1986.
4. P. Chauhan, E. Clarke, Y. Lu, D. Wang. "Verifying IP-Core based System-On-Chip designs." In *Proceedings of the IEEE ASIC/SOC Conference*, pp. 27-31. 1999.
5. E.Clarke, O. Grumberg. H. Hamaguchi. "Another look at LTL model checking." *Formal Methods in System Design, 10*, pp. 47–71. 1997.
6. E. Clarke, O. Grumberg, D. Peled. *Model Checking.* MIT Press. 1999.
7. E. Clarke, Y. Lu, H. Veith, D. Wang, S. German. "Executable Protocol Specification in ESL." *Formal Methods in Computer-Aided Design (FMCAD'00).* 2000.
8. D.L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits.* ACM Distinguished Dissertations. MIT Press, 1989.
9. A. Goel, W. R. Lee. "Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core." *37th ACM/IEEE Design Automation Conference.* 2000.

10. D. E. Long. "Model Checking, Abstraction and Compositional Reasoning." PhD Thesis, Carnegie Mellon University, 1993.
11. Milner, Tofte, Harper, MacQueen. *The Definition of Standard ML.* MIT Press, 1997.
12. K. Shimizu, D. L. Dill, A. J. Hu. "Monitor-Based Formal Specification." *Formal Methods in Computer Aided Design (FMCAD'00).* 2000.
13. K. F. Larsen, J. Lichtenberg. MuDDy. Version 1.7. `http://www.itu.dk/research/muddy`.
14. J. L. Nielsen. BuDDy–A Binary Decision Diagram Package. Version 1.7. `http://www.itu.dk/research/buddy`.
15. S. Romanenko, P Sestoft. Moscow ML. Version 1.44. `http://www.dina.dk/~sestoft/mosml.html`.

A Example Specifications

Slave Specification

Slave specifications are identical for both versions. We require that:

1. Slave should acknowledge only valid requests.
2. Slave should not assert *comp* before the first request is acknowledged.
3. *comp* should be asserted only once per request. If *comp* has been asserted once, then it will be deasserted until another request has been acknowledged.

Master Specification

Version A

1. Once asserted *req0* should remain asserted until it is acknowledged.
2. Once asserted *req1* should remain asserted until it is acknowledged.
3. Master 0 can request to lock the bus by asserting *bus_lock* only when *req0* is asserted.

Version B

1. Once asserted *req0* can only be deasserted one cycle after the cycle in which *ack0* is asserted.
2. Once asserted *req1* can only be deasserted one cycle after the cycle in which *ack1* is asserted.
3. Master 0 can request to lock the bus by asserting *bus_lock* only when *req0* is asserted.

Valid Signal

Version A

1. If there is a request and the bus is not busy, then in the next cycle *valid* must be asserted.

2. Once asserted *valid* can only be deasserted one cycle after the cycle in which *Ack* is asserted.
3. If *valid* is deasserted, and either the bus is busy or there is no request then *valid* should remain deasserted in the next cycle.
4. If *Ack* is asserted, *valid* should be deasserted in the next cycle.
5. If the bus is locked and master 0 is not making a request, then *valid* cannot be asserted in the next cycle.

Version A	**Version B**
Slave Specification	
$\mathbf{G}(\mathit{Ack} \rightarrow \mathit{valid})$	
$\neg \mathit{comp}\ \mathbf{U}\ \mathit{Ack}$	
$\mathbf{G}(\mathit{comp} \rightarrow \mathbf{X}\,(\neg \mathit{comp}\ \mathbf{U}\ \mathit{Ack}))$	
Master Specification	
$\mathbf{G}\,(\mathit{req0} \rightarrow \mathit{req0}\ \mathbf{U}\ \mathit{ack0})$	$\mathbf{G}(\mathit{req0} \wedge \neg \mathit{ack0}) \rightarrow \mathbf{X}\ \mathit{req0}$
$\mathbf{G}\,(\mathit{req1} \rightarrow \mathit{req1}\ \mathbf{U}\ \mathit{ack1})$	$\mathbf{G}(\mathit{req1} \wedge \neg \mathit{ack1}) \rightarrow \mathbf{X}\ \mathit{req1}$
$\mathbf{G}(\neg \mathit{bus\_lock} \rightarrow (\neg \mathit{bus\_lock}\ \mathbf{U}\ \mathit{req0}))$	$\mathbf{G}(\neg \mathit{bus\_lock} \rightarrow (\neg \mathit{bus\_lock}\ \mathbf{U}\ \mathit{req0}))$
Valid Signal	
$\mathbf{G}(((\mathit{req0} \vee \mathit{req1}) \wedge \neg \mathit{busy}) \rightarrow \mathbf{X}\ \mathit{valid})$	$\mathbf{G}\,((\neg \mathit{valid} \wedge \neg \mathit{busy})$ $\rightarrow \mathbf{X}((\mathit{req0} \vee \mathit{req1}) \leftrightarrow \mathit{valid}))$
$\mathbf{G}((\mathit{valid} \wedge \neg \mathit{Ack}) \rightarrow \mathbf{X}\ \mathit{valid})$	$\mathbf{G}((\mathit{valid} \wedge \neg \mathit{Ack}) \rightarrow \mathbf{X}\ \mathit{valid})$
$\mathbf{G}\,(\neg \mathit{valid} \wedge \neg((\mathit{req0} \vee \mathit{req1}) \wedge \neg \mathit{busy})$ $\rightarrow \mathbf{X}\,\neg \mathit{valid})$	$\mathbf{G}(\neg \mathit{valid} \wedge \mathit{busy} \rightarrow \mathbf{X}\,\neg \mathit{valid})$
$\mathbf{G}(\mathit{Ack} \rightarrow \mathbf{X}\,\neg \mathit{valid})$	$\mathbf{G}(\mathit{Ack} \wedge \mathit{valid} \rightarrow \mathbf{X}\,\neg \mathit{valid})$
$\mathbf{G}(\mathit{locked} \wedge \neg \mathit{req0} \rightarrow \mathbf{X}\,\neg \mathit{valid})$	$\mathbf{G}(\mathit{locked} \wedge \neg \mathit{req0} \rightarrow \neg \mathit{valid})$
Arbiter Acknowledgement Signals	
$\mathbf{G}(\neg \mathit{master\_id} \wedge \mathbf{X}\ \mathit{Ack} \leftrightarrow \mathbf{X}\ \mathit{ack0})$	$\mathbf{G}(\mathit{ack0} \leftrightarrow \mathit{Ack} \wedge \neg \mathit{master\_id})$
$\mathbf{G}(\mathit{master\_id} \wedge \mathbf{X}\ \mathit{Ack} \leftrightarrow \mathbf{X}\ \mathit{ack1})$	$\mathbf{G}(\mathit{ack1} \leftrightarrow \mathit{Ack} \wedge \mathit{master\_id})$
Busy Signal	
$\mathbf{G}(\neg \mathit{busy} \rightarrow \neg \mathit{busy}\ \mathbf{U}\ \mathit{Ack})$	
$\mathbf{G}(\mathit{Ack} \rightarrow \mathit{busy})$	$\mathbf{G}(\neg \mathit{busy} \rightarrow \mathbf{X}(\mathit{busy} \leftrightarrow (\mathit{Ack} \wedge \neg \mathit{comp})))$
$\mathbf{G}(\mathit{busy} \wedge \neg \mathit{comp} \rightarrow \mathbf{X}\ \mathit{busy}$	
$\mathbf{G}(\mathit{comp} \rightarrow \mathbf{X}\,\neg \mathit{busy})$	$\mathbf{G}(\mathit{busy} \rightarrow \mathbf{X}(\neg \mathit{busy} \leftrightarrow \mathit{comp}))$
Master_id Signal	
$\mathbf{G}((\mathit{locked} \vee \mathit{busy}) \rightarrow (\mathit{master\_id} \leftrightarrow \mathbf{X}\ \mathit{master\_id}))$	
$\mathbf{G}(\neg(\mathit{locked} \vee \mathit{busy}) \rightarrow \mathbf{X}(\mathit{req0} \rightarrow \neg \mathit{master\_id}))$	
$\mathbf{G}(\neg(\mathit{locked} \vee \mathit{busy}) \rightarrow \mathbf{X}(\mathit{req1} \wedge \neg \mathit{req0} \rightarrow \mathit{master\_id}))$	
Locked Signal	
$\mathbf{G}\,(\neg \mathit{locked} \rightarrow \mathbf{X}(\mathit{locked} \leftrightarrow \mathit{bus\_lock} \wedge (\neg \mathit{master\_id} \wedge \mathit{Ack})))$	
$\mathbf{G}(\mathit{locked} \rightarrow \mathbf{X}(\mathit{locked} \leftrightarrow \mathit{bus\_lock}))$	

Table 1. LTL specifications for the example system depicted in Figure 6

Version B

1. If *valid* is deasserted and the bus is not busy, *valid* is asserted in the next cycle if and only if there is a request in that cycle.
2. Once asserted *valid* can only be deasserted one cycle after the cycle in which *Ack* is asserted.

3. If *valid* is deasserted and the bus is busy then *valid* should remain deasserted in the next cycle.
4. If *Ack* is asserted in response to a *valid* then *valid* should be deasserted in the next cycle.
5. When the bus has been locked by master 0, ignore all requests by master 1.

Arbiter Acknowledgement Signals
Version A

1. If current master is 0 and an acknowledge is received in the next cycle then *ack0* is asserted in the next cycle, otherwise *ack0* remains deasserted.
2. If current master is 1 and an acknowledge is received in the next cycle then *ack1* is asserted in the next cycle, otherwise *ack1* remains deasserted.

Version B

1. If current master is 0 and an acknowledge is received then *ack0* is asserted, otherwise *ack0* remains deasserted.
2. If current master is 1 and an acknowledge is received then *ack1* is asserted, otherwise *ack1* remains deasserted.

Busy Signal
Version A

1. If *busy* is deasserted, it remains deasserted until an *Ack* is received.
2. If an *Ack* is received, *busy* is asserted.
3. Once asserted *busy* can only be deasserted one cycle after the cycle in which *comp* is asserted.
4. If *comp* is asserted, *busy* should be deasserted in the next cycle.

Version B

1. If the bus is not busy then in the next cycle the bus is busy if and only if the slave acknowledge a request and does not complete it in that cycle.
2. If the bus is busy then the bus remains busy till the slave asserts *comp*. Once *comp* is asserted, the bus is no longer busy.

Master_id Signal

The arbiter arbitrates by setting *master_id* to denote which master is controlling the bus. Both versions behave identically when setting *master_id*.

1. *master_id* remains unchanged if the busy is either busy or locked.
2. If the bus is neither busy nor locked then master 0 is given control of the bus if it makes a request.
3. If the bus is neither busy nor locked then master 1 is given control of the bus if it makes a request and master 0 is not making a request.

Locked Signal

1. If the bus is not locked then in the next cycle the bus is locked if and only if a bus locking request from master 0 is acknowledged by the slave.
2. If the bus is locked, it remains locked while master 0 keeps *bus_lock* asserted.

The initial state of all the signals is deasserted.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.