Monitor-Based Formal Specification of PCI

Kanna Shimizu[1], David L. Dill[1], and Alan J. Hu[2]

[1] Computer Systems Laboratory, Stanford University, Stanford, CA 94305
{kanna.shimizu, dill}@cs.stanford.edu
http://radish.stanford.edu/pci
[2] Dept. of Computer Science, Univ. of British Columbia, Canada ajh@cs.ubc.ca

Abstract. Bus protocols are hard to specify correctly, and yet it is often critical and highly beneficial that their specifications are correct, complete, and unambiguous. The informal specifications currently in use are not adequate because they are difficult to read and write, and cannot be functionally verfied by automated tools. Formal specifications, promise to eliminate these problems, but in practice, the difficulty of writing them limits their widespread acceptance. This paper presents a new *style* of specification based on writing the interface specification as a formal monitor, which enables the formal specification to be simple to write, and even allows the description to be written in existing HDLs. Despite the simplicity, monitor specifications can be used to specify industry-grade protocols. Furthermore, they can be checked automatically for internal consistency using standard model checker tools, without any protocol implementations. They can be used without modification for several other purposes, such as formal verification and system simulation of implementations. Additionally, it is proved that specifications written in this style are *receptive*, guaranteeing that implementations are possible. The effectiveness of the monitor specification is demonstrated by formally specifying a large subset of the PCI 2.2 standard and finding several bugs in the standard.

1 Introduction

Given the importance of conforming to bus protocols, it is surprising that the current state of specification, even for widely used standards, is a long document written in English. Natural languages are ill-suited for precise specification because they tolerate vagueness, and are difficult for computers to analyze. The situation would be better if official standards, or even module interface specifications internal to companies, were written in a precisely-defined notation. However, until now, due perhaps to language and tool barriers, industry has largely ignored formally specifying interfaces. And so we present a specification *style* where the user need not learn any language beyond Verilog or VHDL and still be able to write formal specifications.

The style is based on writing the specification as a formal *monitor* (Figure 2). A monitor is an observer in a group of interacting modules, or *agents* which communicate via a set of protocol rules. It's main task is to flag agents when they fail to uphold the protocol. Writing the specification as a monitor enables the specification to be written as a list of simple rules, thus, allowing formal specification to be a relatively easy process. It also allows the specification to be checked "stand-alone" where no implementation needs to be written to verify the protocol. Furthermore, it results in a synthesizable specification which can be directly used in testing environments where cycle-based models are needed. Another direct use is for modeling environments when model checking an implementation. And despite its simplicity, a monitor specification can be written for "real" protocols such as the widely-used PCI local bus protocol.

We also describe several highly effective debugging methods for monitor-style specifications. It is explained how certain requirements on the specification style discourages errors and how the debugging methods further ensure correctness and absence of contradictions. One highlight with a monitor specification is that debugging can be done on the protocol based on its internal consistency, before any implementations are designed. Furthermore, if two easy-to-check properties holds for the specification, it is guaranteed that the specification is receptive. Receptiveness guarantees that an implementation exists for the specification, and that there is no illusory freedom in the specification. On a practical level, these debugging methods found several problems in the official PCI protocol when they were applied to a specification of PCI.

The primary contributions of this paper are:

- *the definition of a simple yet powerful specification style that is resistant to specification errors;*
- *presentation of general specification debugging methodology, which does not require any implementations;*
- *a report on the successful application of the specification style and debugging methodology to PCI, and the resulting discovery of bugs in the protocol*
- *a theorem stating that the specification style together with a simple-to-check property, guarantees the receptiveness of a specification.*

Previous Work Some of the same ideas were explored in a 1998 paper by Kaufmann, Martin, and Pixley [1], which proposed using logical constraints for environment modeling. All of the contributions listed above go beyond the Kaufman *et al.* paper. In 1999, Chauhan, Clarke, Lu and Wang [5] specified PCI using CTL and then model-checked the state machines that appeared in the appendix of the official PCI specification document [2] against their CTL specification. With our specification style, CTL is not used to specify the protocol; consequently, along with simplicity, our specification is executable and it can be used directly for a variety of applications, such as simulation, that a CTL specification cannot. In 1998, Mokkedem, Hosabettu, and Gopalakrishnan formalized and proved some higher-level properties of PCI involving communication over

bus bridges [8]. Their specification is almost unrelated to the one here, which focuses on the low-level behavior of individual signals and ignores the higher-level transaction ordering properties which Mokkedem *et al.* concentrate on; the difference is in the levels of abstraction of the specifications. Many specification languages for reactive systems and protocols have been proposed (too many to cite). However, it is important to note that we are specifically *not* proposing a new specification language, but a specification style that is simple enough to be implemented in a number of existing hardware description languages.

Notation In the examples of the specification appearing in the paper, a logical notation is used. Individual variables (e.g. *frame*) are true when asserted and false when deasserted. This is a warning to readers who are accustomed to control signals being low when asserted. Logical connectives "$\rightarrow$", "$\leftrightarrow$", and "$\neg$" represent "IMPLIES", "IFF", and "NOT," respectively. $prev(x)$ means the value of x in the previous cycle, $prev(prev(x))$ is the value two cycles ago. In an HDL, $prev(x)$ would be a state variable which is assigned the current value of x on each clock.

Structure of the Paper Section 2 describes the specification style, what such a specification can be used for, and the rules for the style that ensures some of the correctness. Section 3 outlines the debugging methods and the bugs founds when applied to the PCI protocol. In section 4, the proof of receptiveness for a monitor specification is given. Section 5 is the conclusion.

2 The Monitor Specification

2.1 Description - the Specification Written as a Monitor

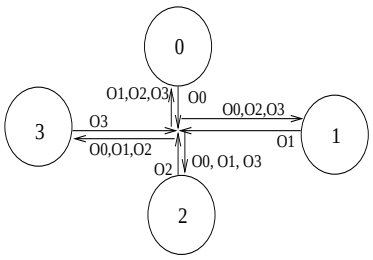

Fig. 1. The System View

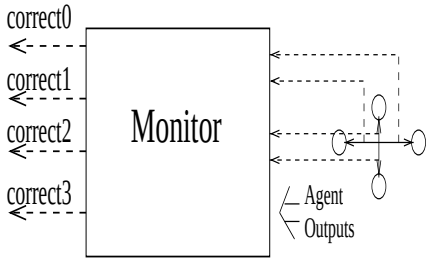

Fig. 2. The Monitor

A bus protocol specification can be viewed as a specification for a complete, closed system of agents using the bus. In Figure 1, agents 0,1,2,3 are using the bus and O0, O1, O2, O3 are the corresponding output sets. Because of the bus, the inputs for each agent are the outputs of other agents. (For agent

1, its inputs are O0, O2, and O3). A bus protocol specification dictates the behavior of all the outputs relative to each other. A *monitor* that checks the agents' compliance to the protocol at each execution step, can be written. It is a machine with the agent output signals as its inputs, and boolean $correct_i$ signals as its outputs (figure 2). The monitor is such that as soon as an agent (or several agents) breaks the bus protocol, it singles out the erring agent(s) by making the corresponding $correct_i$ signal false. If $correct_i$ is true, agent i has upheld the specification so far and its current outputs also conform to the specification. If $correct_i$ is false, agent i has broken a specification requirement currently or sometime in the past. Thus, $correct_i$ is "sticky"; once a rule has been broken, the corresponding $correct_i$ stayes false forever. *The specification style is based on writing the specification as such a monitor.*[1] After all, the monitor must have exactly the protocol information to decide on agent compliance so it is natural for the protocol specification to be in the form of a monitor; it differs from the conventional view of a specification only because it is an active machine as opposed to a passive documentation. The immediate benefits of this are the direct applications of such a specification.

For Model Checking a Single Implementation To verify a single agent implementation, one needs to create an environment for it, namely other agents on the bus. This is a non-trivial, tedious task. However with a monitor, an environment can be created without writing any implementation code. It does this by specifying which input sequences to the agent are correct according to the interface specification. Namely, one would model check the single agent by conditioning all the properties to be verified, with "if the interfacing agents have been correct so far according to the monitor". For example, if p is the property to be model checked, and agents i and j form the environment, the property to be model checked is "$correct_i \wedge correct_j \Rightarrow p$" where $correct_i$ and $correct_j$ correspond to the output signals of the monitor for i and j. The monitor and the condition in the model checking properties correctly constrain the inputs to the agent. This is an example of assume guarantee reasoning where the specification for one (or more) agent(s) is used to verify the implementation of another agent. This use of the monitor is very similar to what is described in [1]. As an execution example of this technique, Govindaraju used our PCI monitor to successfully verify a PCI controller implementation [10].

For Simulating Complete System Implementations In a testing environment, an interface monitor, if written in the language of the implementation, can be directly connected to a design and flag errors and correctly assign blame to the erring module in a system-level simulation. Since monitors can be written in synthesizable RTL, they can be used for tools that need cycle-level models instead of event-based simulation models, such as formal verifiers or emulators.

[1] Only the monitor is written by the specification writer. The agents in the figure are to be later implemented by someone else.

2.2 Construction of a Monitor Specification

A further advantage of the monitor-style specification is that they are very easy to construct. First, it is noted that a specification is a list of rules. In particular, the official PCI 2.2 specification is written that way. Thus, it is natural for the monitor to check for each of these rules independently. For clarity, these rules will be called *constraints* here. Here are some examples of PCI constraints,

``TRDY# cannot be driven until DEVSEL# is asserted.'' (section 3.3.1)
``Only when IRDY# is asserted can FRAME# be deasserted'' (section 3.3.1)

As logic formulas, these can be written as follows; $trdy \rightarrow devsel$ (if *trdy* is true, then *devsel* must be true) and $prev(frame) \rightarrow frame \vee irdy$ (if *frame* is true in the last cycle, then it must either be true in this cycle or if it's not, irdy must be true). The goal was to keep the constraints as simple as possible to prevent the overall specification from getting complicated. When specifying PCI, it was found that the following constraint characteristics can be kept true, and the specification can still fully describe the protocol.

1. **No CTL or LTL** For the monitor specification, all of the PCI constraints can be written without using any CTL [6] constructs nor is knowledge of any linear time temporal logic (LTL) specifically needed. This is the basis for the claim that the specification style can be used with HDLs such as Verilog. In Verilog, the above example becomes, (where $correct_i$ is initialized to 1.)

```
if(trdy && !devsel) {correct = 0;}
```

2. **No complex state machines** Only two types of very simple state machines were needed as auxiliary variables for the PCI constraints. One type is a event-recoding state machine which becomes true when a *set* event happens and remains true until a *reset* event occurs and is false otherwise. This is needed, for example, to "remember" whether the transaction is a read or a write. The second type is a counting state machine which starts to count after a *set* event, and stops counting either when a *reset* event happens or a *limit* is reached, whichever comes first.

3. **Small time frames** With the help of the state machines described above, all of the constraints can be written with less than three time frames. Thus, the most complicated PCI constraint looks like this: $prev(devsel) \wedge prev(prev(stop)) \wedge prev(stop) \wedge prev(final\_dphase\_done) \rightarrow \neg req$. For most constraints, only two time frames are needed, and thus, most are as follows: $prev(stop) \wedge \neg prev(devsel) \wedge \neg prev(dphase\_done) \rightarrow \neg devsel$. This property keeps the constraints compact.

From a preliminary inspection of a more complex protocol than PCI, such as Intel's Merced bus, properties 1 and 3 seem to hold for other protocols. Thus, a specification can be a list of compact constraints which are easy to maintain. And to construct the desired monitor, the constraints are directly used to determine the $correct_i$'s. Assuming that each constraint constrains the behavior of only one agent, the constraints are grouped by the agent which they constrain. When the

agent output signals make all the constraints of one agent true, the corresponding $correct_i$ is true; otherwise, the $correct_i$ is false. Thus, $correct_i$ is a conjunction of all the constraints specifying the behavior of agent i. The following is the assignment statement for $correct_i$, where $constraint_i^j$ pertains to agent i.

if $(constraint_i^0 \wedge constraint_i^1 \wedge ... \wedge constraint_i^n)$
then $correct_i$ = true, else $correct_i$ = false

Therefore, the monitor is a list of propositional formulas, auxiliary state variable assignments, and $correct_i$ assignments. There is no conversion of this to a state machine; this is precisely the code for the monitor.

(propositional formulas)
$constraint_i^0 = trdy \rightarrow devsel$
...
(state variable assignments and the two types of small state machines)
$prev(trdy) = trdy$
...
($correct_i$ assignments)

2.3 Detailed Style Requirements for a Monitor-Style Specification

Some requirements on the constraints were discovered and developed. In this section, the motivation behind them will be discussed.

Separability of the Constraints Rule

Each constraint can only constrain outputs of one agent.

For each constraint, there should only be one agent to blame when the constraint is broken. Consequently, a constraint can only restrict the outputs of one agent; if it dictates the output behavior of two or more agents, multiple agents can be held responsible for a single broken constraint. Since it is exactly the current state variables that are constrained by the constraints, all current state variables of a constraint, must be outputs of the same agent. Equivalently, since for a particular agent, outputs of other agents are its inputs, the constrained variables must all be the agent's outputs and not its current inputs.

This rule is called the *separability* rule because it allows constraints for different agents to be separated and evaluated independently. This separability factor is important for a specification because with it, a specification can be guaranteed to have an implementation as proved in section 4. The main need for the separability of a constraint is to uphold an important principle of specifications: *it should be possible to implement an agent based on information solely from the specification, and know that the implementation can interact correctly with any other agent upholding the specification.*

Independent Implementability If multiple agents are responsible for upholding an inseparable constraint, the implementation of a single agent must be able

to do the impossible act of predicting the behavior of the other agents. Thus, the only way such a system can be designed is for the different agents to be designed *together* which runs counter to the above principle. For example, a bad, inseparable specification would be $a = b$, where a and b are outputs of two different agents. An implementer of one of the agents cannot safely set the value of a without knowing what the implementer of the other agent will set the value of b to. Such a functionality is not independently implementable. Equivalently, a monitor for such a specification cannot blame a single agent when the constraint is broken. If $a = b$ does not hold, it is impossible to decide whether a is wrong or b is wrong. Thus, it is apparent why "a broken constraint can only blame one agent" is a sufficient condition for the specification to uphold the above principle.

Removing Illusory Freedom Consider the specification "$\neg(a \wedge b)$," where a and b are outputs of different agents. Such a formula might result from an attempt to specify mutually exclusion. In this case, an implementer can only set $a = 0$ safely, in case the implementer of the other agent may set $b = 1$. But the implementer of b can only safely set $b = 0$, too, so the specification could just as well be $a = b = 0$. In other words, this specification allows *illusory freedom*, which is also undesirable. The separability style rule disallows such situations.

Specifying Moore Machines As the clock speeds of busses gets faster and faster, almost all bus interface protocols assume a Moore machine timing; namely, it is expected that the interfacing agent needs at least one cycle to respond to its inputs. The output separability rule can be thought of as a result of modeling the agents as Moore machines. For example, machine A has an input in_A and outputs o_A and r_A and is specified a constraint that breaks the separability style rule: $\neg in_A \vee o_A \wedge r_A$. This can be interpreted as whenever in_A is true, machine A must react immediately and assert its outputs o_A and r_A true. But since machine A is a Moore machine, this is an unreasonable specification.

The example of mutual exclusion is interesting because although it is frequently a desired property of a bus, "only one agent can be driving the bus at a time", the style rule disallows it as a constraint as outlined in the "Removing Illusory Freedom" paragraph. Mutual exclusion is not a specification that can be implemented independently by several agents. Instead, it is an emergent property that is implied by other constraints that can be specified independently for the agents. For example, in PCI only the agent that is the current *master* can drive certain bus signals, which is a property that can be specified as a separable constraint. The arbiter in the system ensures that only one agent is the master at any time and this is also a separable constraint. Together, these separable constraints ensure the non-separable mutual exclusion property of the bus.

An Un-Implementable PCI 2.2 Requirement

Interestingly, there is an official PCI 2.2 specification [2] requirement which does not satisfy this separability rule, and is consequently un-implementable as stated. However, there is an equivalent, implementable requirement to replace it.

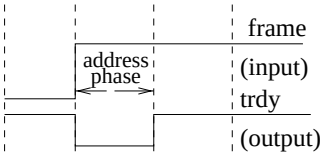

Fig. 3. trdy behavior during the address phase

The requirement, which is in section 3.2.4 of the official PCI documentation, states that the signal *trdy* must use the address phase as a turnaround cycle. The "address phase" is when the signal *frame* is asserted when it was de-asserted in the previous cycle (figure 3), and a "turnaround cycle" for a bus signal is a cycle where no agent is allowed to drive that signal. Thus, the requirement translates to "if *frame* just became asserted in this state, do not drive *trdy*." The problem is *frame* and *trdy* can be driven by different agents, and so both must decide simultaneously how to meet the requirement *together*. And a Moore machine agent cannot react (via the value of *trdy*) to its input (in this case, *frame*) in the same cycle. Thus, this requirement cannot be implemented in an agent as stated in the standard. However, this requirement can be stated in a different way with the same intended effect. The requirement should be "no agent may drive *trdy* if *frame* and *irdy* were both deasserted in the previous state.[2]" This obeys the output separability rule and it will enforce the desired property that all agents not drive *trdy* in the address phase.

The Importance of Isolating Current Variables in a Constraint

If the constraint spans more than one time frame (i.e. involves at least the previous state) the constraint must be written in the form "past_conditions → current_state". Thus, all multi-state constraints must be written as implications, and all prior states variables must be in the antecedent and current state variables must be in the consequent.

Example : ``Only when IRDY# is asserted can FRAME# be deasserted.''
Correct : $prev(frame) \rightarrow frame \vee irdy$
Incorrect : $prev(frame) \wedge \neg frame \rightarrow irdy$

Unlike the previous rule, this rule is not required for the implementability of the specification, but writing the constraints in this form makes the specification easier to understand and debug. This style rule separates the conditioning element, the *past* history, from the constraining element, the *current* constraint. Also, this form makes contradictions in the constraints easier to spot, as demonstrated in section 3.1.

[2] A note to PCI experts: for an address phase following a back-to-back transaction, which won't have *frame* and *irdy* deasserted in the previous state, other constraints ensure that there is a turnaround cycle for *trdy*.

3 Debugging the Monitor Specification

A very important practical question is how to debug a specification. The following debugging methods work with the monitor-style specification *without requiring any implementations to be written.* Once the specification is written, these methods can be immediately and directly applied to it. They check whether the monitor is overly restrictive where it flags correct actions as errors, or underrestrictive where it does not catch incorrect actions. Therefore, they check for contradiction and completeness, respectively.

This section also includes the bugs found by these methods in the monitor-style formal PCI specification. Some were translation errors which are significant because they support the claim that an informal specification is prone to misinterpretations. However more importantly, some inherent problems in the official PCI protocol were discovered. These discovered bugs further stress the importance of using a formal specification style to develop and review a protocol. For these debugging methods, a CTL model checker is needed. Good model checking tools exist (such as CMU's SMV [3] and Cadence's SMV [4] which were both used successfully with the PCI specification) that will take a monitor-style description and answer queries about the defined state graph. The queries are written in the branching-time temporal logic CTL [6].

3.1 Dead State Check

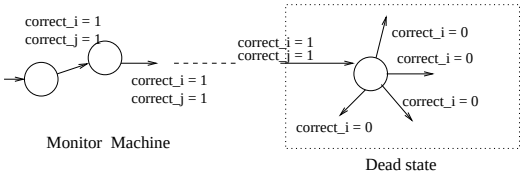

Fig. 4. A dead state for agent i

Dead states arise due to contradictions in the specification. For example, if one constraint for agent i requires p to be true in the current state and another requires $\neg p$ to be true in the same state, there is a dead state. Defining dead states precisely requires defining a few other concepts first. A transition in the monitor machine from a state is said to be *correct for agent i* if the monitor asserts $correct_i$ during the transition. A monitor state s is *correctly reachable*, if there exists a sequence of agent outputs that causes the monitor state to enter s from the initial state, while **all** $correct_i$ signals are continuously asserted. A dead state of the monitor for agent i is a correctly reachable state that has no correct exiting transitions for agent i; for all outgoing transitions of the dead state, $correct_i$ is

false. Intuitively, a good specification should have no correctly reachable dead states because then, all possible agent outputs are incorrect according to the monitor. (Others have observed the importance of dead states [1].)

To ensure the absence of a dead state in a monitor specification, a certain characteristic needs to hold for all the agents in the specification: "for every state in the monitor where no constraints for any agent have been broken so far, there must exist at least one next state where all of the constraints for the particular agent hold". This characteristic can be checked easily using a CTL model checker with the formula, for a particular agent i, $AG(\bigwedge_{j \in Agents} correct_j \rightarrow EX\ correct_i)$. If there are any contradictions in the specification, the model checker for this property returns a counterexample indicating a dead state.

The following is an incorrect logical translation of some PCI requirements, a mistake actually made by the first author. The dead state check found a dead state which resulted from the conflicting constraints,

$prev(address\_phase) \rightarrow \neg trdy$
$prev(trdy) \rightarrow trdy \vee (irdy \wedge (stop \vee trdy)) \vee prev(irdy \wedge (stop \vee trdy))$
$prev(trdy) \rightarrow (prev(stop) \leftrightarrow stop)$

The contradiction is not obvious from the expression above but if it is re-written to obey the *Isolate Current Variables* rule, and the values of the state variables in the dead state are known from the dead state check, it is possible to see that there is no legal next state when $prev(address\_phase \wedge \neg irdy \wedge trdy \wedge \neg stop)$ holds because $\neg trdy \wedge (trdy \vee (stop \wedge irdy)) \wedge \neg stop$ is unsatisfiable.

$prev(address\_phase) \rightarrow \neg trdy$
$prev(\neg irdy \wedge trdy) \rightarrow trdy \vee (stop \wedge irdy)$
$prev(trdy \wedge \neg stop) \rightarrow \neg stop$
$prev(trdy \wedge stop) \rightarrow stop$

Since the dead state check returns the dead state as "..., $address\_phase = true, irdy = false, trdy = true, stop = false$, ...", these variable assignments can be used to see which constraints are in effect, by plugging these values into the left-hand-side of the implications (the antecedents). In this case, the first three constraints are in effect and their consequents form the contradiction. This process is effective if the constraints follow the *Isolate...* rule and the dead state check can return the dead state's state variable values. One advantage this check has over the other checks is its simplicity. No creativity or expertise is required; only the CTL formula given above and a model checker are needed.

Another similar check tests for under-restriction in the specification. It is reasonable to assume that in all states, at least one constraint is in effect. The check searches for correctly reachable states where *all* possible outputs for agent i are correct according to the monitor. The monitor can be checked for this property with the CTL formula, $EF((\bigwedge_{i \in Agents} correct_i) \wedge AX correct_i)$ "There is a correctly reachable state where *all* the possible next states for an agent are correct." Although this check turned up no bugs in the PCI monitor specification, it is still

a worthwhile check. Like the dead state check, this check requires no creativity or extra work.

The bus agents are assumed to be Moore machines in this paper but Bryant, Chauhan, Clarke, and Goel define and describe inconsistencies for combinational Mealy machine circuits in [11].

Results From Applying the Dead State Check to PCI This check proved to be more effective in catching errors introduced in the monitor writing stage rather than serious problems in the protocol itself. Specifically, the dead state check pinpointed typos by the monitor specification writer, misinterpretations by the monitor writer due to the ambiguous wording in the protocol text, and exceptions to general rules not mentioned by the official specification. Because this check proved to be effective in finding the exact intent of the requirement, it proved indispensable for making the constraints precise. (See example below.) Four bugs in the formal specification which were due to misinterpretation of (arguably) ambiguous wording in the official documentation were found by dead state checking. Furthermore, the test aids the specification writer in realizing the boundary cases for general rules by demonstrating how a contradiction can occur in special cases. Thus, the dead state check helps the specification writer identify exceptions to the too generally-stated rules, *which are not mentioned by the official document.* The dead state check discovered six such cases where the monitor writer needed to refine the constraints to account for the special cases.

As an example of an ambiguously worded requirement which was misinterpreted and thus caused a contradiction in the monitor, consider the following from section 3.3.3.1: "IRDY# must remain asserted for at least one clock after FRAME# is de-asserted" which seemingly translates to the constraint

1. $prev(prev(frame)) \wedge prev(\neg frame) \rightarrow irdy$

However, in section 3.3.3.2.1, it is stated that "the master must de-assert IRDY# the clock after the completion of the last data phase."

$last\_data\_phase = \neg frame \wedge irdy \wedge (trdy \vee stop)$
$prev(last\_data\_phase) \rightarrow \neg irdy$
$\Rightarrow$ 2. $prev(\neg frame \wedge irdy \wedge (trdy \vee stop)) \rightarrow \neg irdy$

The conjunction of these two constraints causes a conflicting requirement on $irdy$ in the correctly reachable state where both antecedents are true: $prev(\neg frame \wedge irdy \wedge (trdy \vee stop) \wedge prev(frame))$. In the next state, the first constraint states that $irdy$ must be true and the second, $irdy$ must be false. It was concluded from guessing at the intention of the requirement that the first rule was misinterpreted and the correct interpretation of it is $prev(frame) \rightarrow irdy \vee frame$ which admittedly is puzzling because this constraint doesn't require the "one clock after" part of the requirement.

3.2 Characteristic Check

Another more powerful debugging method is checking for specific properties, or *characteristics* in the specification. These characteristics are mainly logical statements about agent events. If the monitor is too constricting, certain agent actions which should be possible will not be allowed by the monitor. This method also catches loopholes in the specification which allow behavior that should be illegal, and so completeness of the specification can be gauged. This debugging method is not new but it furthers the case for formally specifying protocols and more importantly, it found several bugs in the official PCI protocol standard. These characteristics are expressed as CTL formulas and are checked against the monitor using CTL model checking. It must be emphasized that the *specification* constraints are simple, bounded, linear time properties and the *checking* characteristics are more complex, unbounded, CTL formulas. *This allows the specification to be synthesizable and yet guarantee rich properties.* One drawback of this characteristic checking is that a user must come up with the characteristic statements. They cannot be automatically deduced from the specification. It is also subject to false error reports when the characteristics themselves are incorrect.

Results From Applying the Characteristic Check to PCI 114 characteristics were written in CTL and checked against the monitor-style PCI specification. This checking method found sixteen bugs in the monitor which resulted from errors in the monitor writing process, but more importantly, seven bugs in the official standard were found by this method. For finding actual problems in the official specification, this test proved to be more effective than the simple dead state check. Here are some characteristics that exposed problems in the official specification. These or similar characteristics are probably applicable for protocols other than PCI and are considered general system properties that should be checked for.

1. System Must Always, Eventually Return to the Idle State It is reasonable to assume that the system should always be *able* to reset into the *idle* state; if there are any deadlocks states which forbid this from happening, checking for this characteristic should find such a problem. However, it is the stronger property, "the system must always *inevitably* reset and go back to the idle state" which found problems in the PCI protocol. This can be expressed in CTL as $AG\,(\bigwedge_{i \in Agents} correct_i \rightarrow \neg EG((\bigwedge_{i \in Agents} correct_i) \wedge \neg idle))$ where $idle$ is defined as $(\neg irdy \wedge \neg frame)$. "If in a state where all agents have been correct so far, then the system must eventually reach a state where all agents are still correct and the bus is in the idle state." This characteristic must be true for the PCI protocol because only when the bus is idle, can a new agent start a transaction. If the bus is never idle because one agent is constantly driving it, this agent has effectively taken over the bus never allowing other agents to use it. To avoid such a situation, the specification must force an agent to always, eventually relinquish the use of the bus as a master and let the bus state be idle.

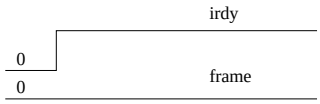

Fig. 5. A non-Idle State

There are three legal ways the PCI protocol allows an agent to not relinquish the bus. Essentially in all three cases, *frame* is deasserted while *irdy* is asserted by the agent (Figure 5). An agent can keep the bus in this state because of the following. There is a timer counter which counts the number of cycles *frame* has been asserted and when it exceeds a preset value, the specification requires the agent to deassert *frame*. Thus, the protocol intends to limit one agent's use of the bus by observing the assertion of *frame*. The protocol's shortcoming is in not recognizing that in an $irdy \wedge \neg frame$ state, $\neg frame$ keeps the timer counter deactivated but *irdy* keeps the bus non-idle.

1. From an idle state, the protocol allows an agent to assert *irdy* and remain in this non-idle-bus state ($irdy \wedge \neg frame$) forever. (Figure 5)
2. During the data phase of a single data phase transaction, *frame* is deasserted and *irdy* is asserted. If a target doesn't respond with a *trdy* or a *stop*, the agent can remain in this non-idle-bus state forever.
3. During the last data phase of a transaction, *frame* is deasserted and *irdy* is asserted. If a target doesn't respond with a final *trdy* or a *stop*, the agent can remain in this non-idle-bus state forever.

2. Definitions are Disjoint Protocols allow an agent to communicate to other agents abnormal terminations when a transaction cannot be carried out. Usually, there are several different types of terminations and an agent asserts and de-asserts different bus signals to indicate which termination type it is executing. For example, in PCI, one termination type, *target_abort* is defined as $target\_abort = \neg devsel \wedge stop$ and another type, a *retry* as $retry = stop \wedge \neg trdy \wedge initial\_data\_phase$ in section 3.3.3.2 of the documentation. The other agent involved in the transaction, namely the master agent, must react differently to each target termination type, so the ability to identify a termination type uniquely from the signals of the terminating agent is important. We can check whether these terminations are distinct by checking the CTL formula $AG\neg((\bigwedge_{i \in Agents} correct_i) \wedge target\_abort \wedge retry)$, ("There is never a state where the specification holds and *target_abort* and *retry* are signaled simultaneously") which reveals that there is a correctly reachable state where $\neg trdy \wedge stop \wedge \neg devsel \wedge initial\_data\_phase$ holds, which is consistent with either *retry* or *target_abort*. Therefore, the protocol allows an agent to signal both termination types simultaneously. If the PCI protocol had been originally written in a formal form and tested for this characteristic, this ambiguity in the protocol could easily have been found and resolved before the protocol became a public standard.

3. Termination Types Should Not Change During a Single Transaction The third characteristic checks whether termination types can change during one transaction. For example, it checks whether an agent can signal a *target abort* in one clock cycle and then a *retry* in the next clock cycle before the transaction ends. Amazingly, checking the property $AG\neg((\bigwedge_{i\in Agents} correct_i) \wedge target\_abort \wedge EX((\bigwedge_{i\in Agents} correct_i) \wedge retry)$ ("This never happens: the specification holds so far when a target abort is signaled, and there is a possible next state where the specification still holds and retry is signaled") reveals that PCI allows this. It is ambiguous, for example, how the master agent should be implemented to react to a target which signals a *target abort* which indicates that the target is not capable of handling the requested data but then, signals a *disconnect with data* which allows the target to transfer data.

4 The Receptiveness Proof

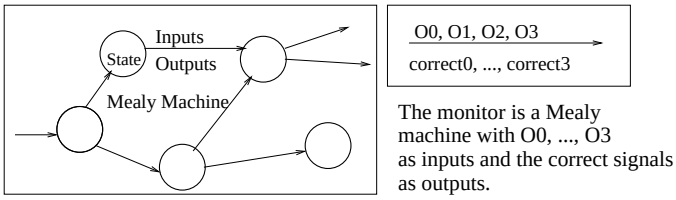

Fig. 6. A Mealy Machine

A Mealy machine has its inputs *and* its outputs on its edges; the output is not associated with a state as with Moore machines. A monitor is a Mealy machine because in order to determine the output $correct_i$ values, a combinational function on the input observed signals and internal state variables is sufficient (Figure 2, 6).

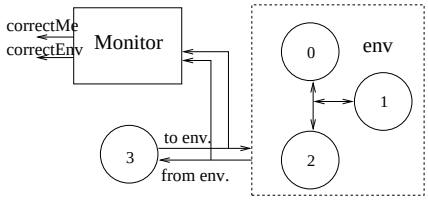

Fig. 7. One Agent and the Environment

One can view the system of agents such that one agent is an object of interest and the other agents form its environment. Using the same monitor, we now have outputs $correct_{me}$ for the former and a $correct_{env}$ for the latter. ($correct_{me}$ can be a particular $correct_j$ and $correct_{env}$ would be a logical conjunction of $correct_i$ for $i \neq j$.) (Figure 7) This setup can be viewed as a game between the agent of interest, *me*, and the environment, *env*. Agent *me* and *env* output signals in a locked synchrony, and do not alternate driving an output. It is deemed that as soon as *env* breaks a specification rule ($correct_{env}$ becomes false), *me* has "won" and the monitor's $correct_{me}$ value will remain true regardless of *me*'s outputs. This is a reasonable restriction because an agent should not be required to uphold the specification if the environment has fed it illegal inputs. It will be proven that if two requirements hold for the specification of the system, it is guaranteed that a Moore machine K exists where no matter what the environment outputs to it, K will always output signals that will keep $correct_{me}$ true; K *implements* the specification. With such a K, the environment will never be able to force K to output an illegal sequence.

The first requirement on the monitor is the output separability rule (section 2.3) which is restated here as "the function which determines $correct_i$ must only be a function of the current state of the monitor machine and the current output of agent i, and *not the current outputs of any other agent.*" The information of the output values of the other agents is incorporated into the next state of the monitor machine. The second requirement is that there are no dead states (section 3.1) for agent *me* in the monitor. For every correctly reachable state of the monitor, there is at least one transition out of that state with the $correct_{me}$ on the edge as true.

Theorem 1. *If a Mealy machine monitor, M, which obeys the following requirements exists for some specification, then a Moore machine implementation for the single agent* me *is guaranteed to exist. The restrictions are,*

1. The monitor must not have any dead states for agent me.
2. The monitor must observe the output separability rule.
And it is assumed that once the environment breaches the specification, $correct_{me}$ *is infinitely true.*

Proof Sketch: Because of assumption 2, a correct output for the agent can be determined at every state independent of the current input. Assumption 1 guarantees the existence of a correct output for *me* for every correctly reachable state.

Proof: Please see Appendix A for the full proof.

If one views this system once again as a multiple agent model (Figure 1), an interesting corollary can be deduced from Theorem 1.

Corollary 1 : A Set of Implementations Exist for a Specification *If a Mealy machine monitor, M, which obeys the following restriction exists for some*

specification, then a set of Moore machines which implement the specification is guaranteed to exist. The restrictions are,

1. *The monitor does not have any dead states for all agents in the specification.*
2. *The monitor observes the output separability rule.*

Proof Sketch Apply theorem 1 to each agent in the specification and the theorem guarantees a correct implementation of all agents. Since the theorem guarantees specification compliance, independent of the inputs to the agent, the agents can be implemented independently and be guaranteed correctness when composed together.

An implementation can always choose the left branch in the specification to avoid the dead state. Receptiveness disallows such illusory freedom in the specification.

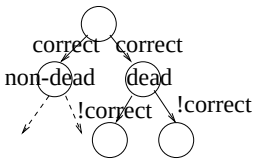

Fig. 8. How even with Dead States in the Specification, an implementation can exist

Corollary 2 : Receptiveness A monitor specification is defined to be *receptive* [9] if for every correctly reachable state in the monitor, there exist agent implementations, when connected to each other and to the monitor, can cause the monitor to reach that state.[3] Receptiveness ensures that there is no illusory freedom in the specification. For example,

$prev(a) \rightarrow out_0 \vee out_1$
$prev(out_1) \rightarrow \neg c$
$prev(out_1) \rightarrow c$

out_0 and out_1 are outputs of one agent and so the agent can always choose to assert out_0 instead of out_1 to avoid the inevitable error state caused by asserting out_1 (because of the last two constraints). Thus, even with a dead state in the specification, there exists an implementation; this example illustrates how the absence of dead states is not a necessary condition for Theorem 1. Receptiveness is a tougher condition to satisfy than implementability, and ensures that there is no illusory freedom in the specification such as "$\vee\ out_1$" in the first constraint. Every correctly reachable state must have a correct next state in order for receptiveness to hold for a specification.

A specification is receptive if

1. *The monitor does not have any dead states for all agents in the specification.*
2. *The monitor observes the output separability rule.*

[3] Ed Clarke has also recognized the relevance of receptiveness to bus specifications, but proposes using model checking algorithms that can check the property directly.

Proof Sketch : State s is a correctly reachable monitor state and the sequence of correct agent i outputs $\{O_0^i, O_1^i, ..., O_n^i\}$ which lead to state s is known for $\forall$i *in Agents*. These agents are individually constructed such that they output this sequence. Thus, the set of agents can take the monitor to state s; it remains to be shown that the agents implement the specification, namely, that their outputs keep the $correct_i$'s true. Up till state s, it is obvious that the agents are correct because the output sequences were chosen along the edges where all $correct_i$'s are true. Such a sequence exists because s is correctly reachable. As for after state s, there exists a next state s' such that the transition from s to s' is *correct*, because of assumption 1. The outputs along this transition can be used for the agent implementations. Inductively, it can be shown that at each step, a correct output can be independently chosen for all agents because of the assumptions. Thus, the agents implement the specification.

5 Conclusion

The monitor-style specification of PCI consists of 83 rules. The entire description file has 280 lines excluding comments. The specification covers almost all signal-level requirements from section 3.1 to section 3.6 of the official 2.2 PCI specification documentation [2]. Bus bridges and 64 bit extensions are not included. The PCI formal specification was fully debugged using the different checks introduced in section 3. There are no dead states in the current specification and all characteristics written, hold. Most of the model checking was done using Cadence SMV [4] on a Pentium Pro system with 128M of memory where the model checking runs took under 5 minutes and model checking the specification was not a problem. The monitor-style PCI specification, written in Verilog, is available at http://radish.stanford.edu/pci.

An obvious next step for this line of work would be to attempt specifications of other interfaces, especially those with characteristics different from PCI, such as pipelined busses. It is likely that different stylistic issues will arise, as well as complexity issues. A second direction is to find additional uses for monitor specifications, to maximize their value. One possibility would be interface synthesis for which work has been started by Clarke, Lu, Veith, Wang, and German [12]. A more modest goal would be to extract *don't cares* from the specification to aid in optimizing a synthesizable description of an interface implementation. Another possibility is to use the monitor as an activator which generates test vectors for an implementation automatically from the monitor description, or as a checker to measure testing coverage. This is to quantify the amount of testing that needs to be done in order for the implementation to be considered thoroughly tested. A third direction of interest is to develop better tools for debugging the specification. For example, the dead state check is limited in that although it can return a state description of the dead state, it cannot pinpoint the conflicting constraints. The problem is exacerbated by the fact that any number of constraints can contribute to an unsatisfiable specification. A greedy algorithm using satisfiability

on the constraints such that the conflicting requirements can be found, should be developed for a more complete specification debugging tool environment.

Acknowledgements The authors will like to thank K. McMillan for help with Cadence SMV, S. Berezin for help with CMU SMV, H. Kapadia for answering PCI questions, and our colleagues in the Gigascale Silicon Research Center (GSRC) for useful feedback. This research was supported by GSRC contract SA2206-23106PG-2.

References

1. M. Kaufmann, A. Martin, and C. Pixley. "Design Constraints in Symbolic Model Checking" in *International Conference on Computer-Aided Verification*, 1998.
2. PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.2*, December 18 1995.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. "Sequential circuit verification using symbolic model checking" in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
4. K. McMillan. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.
5. P. Chauhan, E. M. Clarke, Y. Lu and D. Wang. "Verifying IP-Core based System-On-Chip Designs" in *Proceedings of the IEEE ASIC conference*, September 1999.
6. E.M. Clarke and E.A. Emerson. "Synthesis of synchronization skeletons for branching time temporal logic" in *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981* Lecture Notes in Computer Science, vol. 131, Springer-Verlag. 1981.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic" in *ACM Transactions on Programming Languages and Systems* 8(2):244-263, April, 1986.
8. A. Mokkedem, R. Hosabettu, and G. Gopalakrishnan. "Formalization and Proof of a Solution to the PCI 2.1 Bus Transaction Ordering Problem" in *Proceedings of the Second International Conference, Formal Methods in Computer-Aided Design,* 1998. Lecture Notes in Computer Science, vol. 1522, Springer-Verlag.
9. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*, MIT Press, 1989.
10. G.S. Govindaraju and D.L. Dill. "Counterexample-guided Choice of Projections in Approximate Symbolic Model Checking" in *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, 2000. (under review).
11. R.E. Bryant, P. Chauhan, E.M. Clarke, and A. Goel. "A Theory of Consistency for Modular Synchronous Systems" in *Proceedings of the Third International Conference, Formal Methods in Computer-Aided Design,* 2000. (under review).
12. E.M. Clarke, Y. Lu, H. Veith, D. Wang, and S. German. "Executable Protocol Specification in ESL" in *Proceedings of the Third International Conference, Formal Methods in Computer-Aided Design,* 2000. (under review).

6 Appendix A

A proof of the Theorem stated in section 4.

Proof: The monitor, M, can be characterized as follows.

Inputs : $I_M = I \times O$
Outputs : $O_M = correct_{me} \times correct_{env}$
Start State : $q_M^0 \in Q_M$
States : Q_M
Transition function, $\delta_M : I \times O \times Q_M \rightarrow Q_M$
Output function, $\lambda_M^{me} : I \times Q_M \rightarrow \{true, false\}$
Output function, $\lambda_M^{env} : O \times Q_M \rightarrow \{true, false\}$

The states are characterized by vectors of internal history variables. Thus, if $h_0, h_1, ..., h_n$ are the history variables, state q_M^i of Q_M is described by the values of $(h_0^i, h_1^i, ..., h_n^i)$.

1. *If the agent commits an illegal action while the environment is still correct, the monitor enters and stays in a state where no matter what the observed signals are, $correct_{me}$ is false and $correct_{env}$ is true. Furthermore, this state is entered only when this particular event happens. "Once the agent makes a mistake, the environment wins."*

For $\forall i \in I, \forall o \in O, \forall q \in Q$,
$\lambda_M^{me}(o, q) = false \wedge \lambda_M^{env}(i, q) = true$
$\iff \delta_M(i, o, q) = q_M^{meError}$

and for $q_M^{meError}$, for $\forall i \in I, \forall o \in O$,
$\lambda_M^{me}(o, q_M^{meError}) = false$,
$\lambda_M^{env}(i, q_M^{meError}) = true$

Thus, consequently, for $\forall i \in I, \forall o \in O$,
$\delta_M(i, o, q_M^{meError}) = q_M^{meError}$

2. *"Once the environment makes a mistake, the agent wins."*

For $\forall i \in I, \forall o \in O, \forall q \in Q$,
$\lambda_M^{env}(i, q) = false \wedge \lambda_M^{me}(o, q) = true$
$\iff \delta_M(i, o, q) = q_M^{envError}$

and for $q_M^{meError}$, for $\forall i \in I, \forall o \in O$,
$\lambda_M^{env}(i, q_M^{envError}) = false$,
$\lambda_M^{me}(o, q_M^{envError}) = true$

Thus, consequently, for $\forall i \in I, \forall o \in O$,
$\delta_M(i, o, q_M^{envError}) = q_M^{envError}$

3. *When both the agent and the environment make a mistake at the same time.*

For $\forall i \in I, \forall o \in O, \forall q \in Q$,
$\lambda_M^{env}(i, q) = false \wedge \lambda_M^{me}(o, q) = false$
$\iff \delta_M(i, o, q) = q_M^{bothError}$

and for $q_M^{meError}$, for $\forall i \in I, \forall o \in O$,
$\lambda_M^{env}(i, q_M^{bothError}) = false$,
$\lambda_M^{me}(o, q_M^{bothError}) = false$

Thus, consequently, for $\forall i \in I, \forall o \in O$,
$\delta_M(i, o, q_M^{bothError}) = q_M^{bothError}$

4. *The monitor does not cause "dead states" for the agent. "As long as the agent has been correct so far, there always exists a legal action for the agent.*

For $\forall q \in Q - \{q_M^{meError}, q_M^{bothError}\}$,
$\exists o \in O such\ that\ \lambda_M^{me}(o, q) = true$

Given such a specification M, the construction of the implementation, K, can be done as follows. Let the characterization of Moore machine K be

Inputs : I Outputs : O
Start State : q^0 States : Q
Transition function, $\delta : I \times Q \rightarrow Q$
State labeling function, $\lambda : Q \rightarrow O$

Let Q be of the same type as Q_M. Thus, if h_0, h_1, ..., h_n are the history variables of the monitor, K has the same history variables and state q^i of Q is described by the values of $(h_0^i, h_1^i, ..., h_n^i)$.

Construction

Basic Idea : Start with the same initial state as the monitor. Use the monitor's correct functions, λ_M^{me} to determine the outputs at each state. The transitions are determined by the inputs, outputs, and the current state using the monitor's transition function.

0. Initially, Q is empty. $Q = \emptyset$
1. Let initial state $q^0 = q_M^0$. Add q^0 to Q.
$Q = \{q^0\}$.
2. For q^0, $\exists o_0$ such that, $\lambda_M^{me}(o_0, q^0) = \lambda_M^{me}(o_0, q_M^0) = true$. (because of property 4 of the monitor, step 1 in the construction, and $q_M^0 \neq q_M^{meError}$, $q_M^{bothError}$.) Let $\lambda(q^0) = o_0$ and define transitions for all possible inputs. $\delta(i, q^0) = \delta_M(i, o_0, q^0)$ for $\forall i \in I$. If $\delta(i, q^0)$ is not in Q, add to Q.

Any new addition to Q, q', is guaranteed to be a member of Q_M because
i) if $q \in Q_M$ and $q' = \delta(i, q)$, for some $i \in I$, then $q' \in Q_M$ because $q' = \delta_M(i, o, q)$ for some $o \in O$ and $\delta_M(i, o, q)$ is defined for all $i \in I$ and $o \in O$ for $q \in Q_M$.
ii) $q^0 \in Q_M$.

Thus, by induction on i) and ii), for $\forall q, q \in Q \Rightarrow q \in Q_M$. [property a]

Furthermore, because $\delta_M(i, o, q) = q_M^{meError}$, $q_M^{bothError}$ only for o such that $\lambda_M^{me}(o, q) = false$ (due to property 1 and 3 of the monitor), for any new addition to Q, q', $q' \neq q_M^{meError}, q_M^{bothError}$ [property b]

3. For every newly added $q \in Q$, pick $o \in O$ such that $\lambda_M^{me}(o, q) = true$ and let $\lambda(q) = o$. (because of property 4, property a, and property b, it is guaranteed that such a o exists.) and for every $i \in I$, define $\delta(i, q)$ such that $\delta(i, q) = \delta_M(i, o, q)$ and if $\delta(i, q)$ is not in Q, add to Q.
4. Iterate back to step 3 and stop when no new q is added to Q.

Thus,

- $Q \subset Q_M$ • $q_M^{meError}, q_M^{bothError} \notin Q$
- $\forall q \in Q, \lambda(q) = o \Rightarrow \lambda_M^{me}(o, q) = true$

And so, K implements the specification M.