A Specification Methodology by a Collection of Compact Properties as Applied to the Intel® Itanium™ Processor Bus Protocol

Kanna Shimizu[1], David L. Dill[1], and Ching-Tsun Chou[2]

[1] Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA
kannas@stanford.edu, dill@cs.stanford.edu
[2] Intel Corporation, 3600 Juliette Lane, SC12-401, Santa Clara, CA 95052, USA
ching-tsun.chou@intel.com

Abstract. In practice, formal specifications are often considered too costly for the benefits they promise. Specifically, interface specifications such as standard bus protocol descriptions are still documented informally, and although many admit formal versions would be useful, they are dissuaded by the time and effort needed for development.
We champion a formal specification methodology that attacks this cost-value problem from two angles. First, the framework allows formal specifications to be feasible for signal-level bus protocols with minimal effort, lowering costs. And second, a specification written in this style has many different uses, other than as a precise specification document, resulting in increased value over cost. This methodology allows the specification to be easily transformed into an executable checker or an simulation environment, for example.
In an earlier paper, we demonstrated the methodology on a widely-used bus protocol. Now, we show that the generalized methodology can be applied to more advanced bus protocols, in particular, the Intel® Itanium™ Processor bus protocol. In addition, the paper outlines how writing and checking such a specification revealed interesting issues, such as deadlock and missed data phases, during the development of the protocol.

1 Introduction

As digital circuits become larger, designs are broken up into more and more pieces, increasing the importance of design integration. The popularity of IP (Intellectual Property) cores, and the increased awareness that their interfaces must be clearly defined, is an affirmation of this trend. This development necessitates that functional interface specifications be correct and precise because they serve a pivotal role when integrating designs. However, specifications are still written in natural languages and not formal languages, forfeiting an opportunity for analysis, automated checks, and preciseness. In many cases, specifications widely in use are wordy, ambiguous, and contradictory; all problems that can be resolved by formal specification development. A subtle point missed in

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 340–354, 2001.

practice is that an informal specification may have inconsistencies that a human reader will not notice, or may be missing rules that a human reader may infer automatically, but, because it is inconsistent and incomplete, any correctness reasoning is impossible. A practical consequence of this is that a good protocol compliance checker can not be created from such a document.

Despite these arguments, in practice, formal specifications are rarely used. The reason seems to lie with the perceived high cost of formal development: mainly, the lengthy development time and the investment needed for formal verification training. For many, the value of a correct specification does not justify these costs. This paper is one step in a broader effort to reduce the cost by making the specification process more reasonable, and to increase the value of formal documents by developing direct applications for them. Methodology, as opposed to tool or language development, is the key to achieving this goal.

Until recently, formal specification research has focused mostly on developing tools or languages. Very little work has been done on how to develop complete specifications. Tools assume the existence of them, or are just used with an ad hoc list of properties. Our goal is to develop a methodology that produces a self-contained, complete, behavioral specification, while adhering to the cost-value goal. Currently, we are focusing on signal-level bus protocol descriptions, since they are both important and challenging to specify.

There has been a few other bus protocol specification projects. In 1999, Chauhan, Clarke, Lu and Wang [CCLW99] specified PCI (Peripheral Component Interconnect) protocol [SIG95] using CTL. Our specification has advantages that a CTL one does not have. In 1998, Mokkedem, Hosabettu, and Gopalakrishnan formalized higher-level properties of PCI involving communication over bus bridges [MHG98]. Their specification is almost unrelated to the one here, which focuses on the low-level behavior of individual signals. In fact, the methodology is most closely related to a 1998 paper by Kaufmann, Martin, and Pixley [KMP98], which proposed using logical constraints for environment modeling. However, they do not give guidance on how the constraints should be written, which is the focus of this project.

In [SDH00], we present a specification style that was developed using the core subset of the PCI protocol as an example. We also describe two debugging methods based on model checking, that were found to be effective with the style. As a second paper in this series, the primary contributions of this paper are the following. First, the work shows that the generalized methodology can be applied to more advanced bus protocols, in particular, the Intel® Itanium™ Processor bus protocol[Cor]. Second, it demonstrates how writing and checking such a specification reveals interesting issues during the development of the protocol. This point is illustrated with a few examples from the specification effort.

It has been found that the expressiveness of the style is not a problem despite the protocol's pipeline feature, that the increased complexity did not affect model checker performance, and the method's debugging strategy is still effective. The technique uncovered several issues in a development version of the protocol which allow deadlock of the bus and missed data phases. The protocol engineers have

recognized the significance of these discoveries and subsequently made changes to the protocol to disallow such scenarios. Furthermore, with this method, the total time to write and check the description was 2 man-months.

The Specification Style

The specification style addresses the cost problem in two ways. First, formal verification expertise is not required to write the specification. In contrast, many specification frameworks require knowledge of LTL (linear time temporal logic) or CTL (computation tree logic)[CE81] as evidenced by numerous projects such as CMU's PCI specification [CCLW99] and IBM Haifa's FoCs software [ABG+00]. Because it does not require the complex constructs of these languages, the formal specification can be and has been written in a hardware description language such as Verilog, a language familiar to many engineers. This feature partly counters the training cost argument against formal specifications. More generally, the style is language-independent (indeed, the specification can still be written in LTL or CTL if desired). The methodology can be applied to any of the existing tool frameworks and languages such as SMV[McM], FoCs[ABG+00] or LUSTRE [HCRP91].

Second, because the style is based on writing many small protocol rules and purposely avoids writing large state machines, the specifications have been found to be easier to write and maintain. Many descriptions comprise of one large state machine with actions specified for each state. Designing such a state machine correctly is a complex, error-prone task. This method instead relies on many small state machines, but the bulk of the specification is done using compact rules.

To maximize its value, the style allows the specification to have multiple functions. For a formal verification framework using a model checker, *with no extra work,* the specification can be used to constrain the inputs to the design under test, and at the same time, it can be used as the list of properties to check for. For example, this was done by Govindaraju and Dill; using the PCI specification written in this style, they were immediately able to formally verify a PCI driver implementation [GD00]. The specification constrained the state space, and also gave them properties to check for. This assume-guarantee approach is not new, but the style easily allows it while an *ad hoc* specification probably would not. For designs that are too complex to be formally verified and can only be simulated, the specification can be directly used as a conventional checker monitor during simulation runs. This is possible because the specification is executable and can be written in a language such as Verilog. For example, at Intel®, the specification that was developed for the Itanium™ processor bus has been translated into a C++ simulation checker.

Specification Writing. At the foundation of the specification style is the concept of monitors. Monitors observe the output signals of interacting agents, flag illegal behavior, and assign blame to the erring agent. The monitor concept allows executable specifications to be written for non-deterministic behavior, and

allows specifications to be written as a conjunction of simple properties (in these respects, monitors have many of the advantages of temporal logic). The concept is not new; numerous specification frameworks are based on monitors.

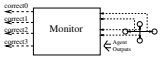


Fig. 1. A Monitor

Within this monitor framework, the style is based on using multiple constraints to collectively define the correctness of an agent's signalling behavior. The constraints are simple propositional formulas with a time construct. The basic form is,

$$prev(signal_0 \wedge ... \wedge signal_1 \vee (statemachine = value)) \rightarrow signal_i \vee ... \wedge signal_n$$

where "$\rightarrow$" is the logical symbol for "implies". The antecedent is an expression containing signal values and state machine values, and the consequent is an expression of signal values. The *prev* construct allows the state of the signal a cycle before to be expressed, and it can be nested although in the examples attempted, only up to a doubly nested *prev* was needed. For readability and to facilitate debugging and implementation, the constraints are written as an implication with the past condition as the antecedent and the current condition as the consequent. For example, the protocol constraint,

$$prev(trdy \wedge stop) \rightarrow stop$$

means "if the signals *trdy* and *stop* were true in the previous cycle, then *stop* must be true in the current cycle" where a "true" signal is asserted and a "false" signal is deasserted.

Due to various desired characteristics on the specification, such as guaranteeing the existence of an implementation, there is one important restriction placed on the constraints. *Each constraint can only constrain the behavior of one agent.* This leads to the ability to blame just one agent when a specification constraint is violated during agent interaction. If signals o_a and q_a are outputs of the same agent and r_b is an output of another agent, the first constraint obeys this restriction, while the second doesn't.

$$correct : previous(r_b) \rightarrow o_a \wedge q_a$$

$$incorrect : previous(r_b) \rightarrow o_a \wedge r_b$$

The PCI protocol was specified with this restriction, and the targeted subset of the Itanium^{TM} processor bus protocol was also successfully specified with this restriction.

The *prev* construct is not enough to retain state information. State machines are needed for this. But instead of relying on large state machines, the style

relies on many, small, standard state machines which each track one thread of information. An example is a 2-state, set-and-reset machine which becomes set when a certain event happens and stays set until it is no longer needed. It is used to record certain information such as whether the transaction is a read or a write. Another example is a counter which counts the number of cycles from a certain event, or counts the number of occurrences of a special event. Only these two types of state machines were needed. A key point is that a protocol can be written with a few standard state machines, even if it is as complex as a pipelined protocol.

In addition to the benefits already described, there are numerous advantages to writing a specification as a "collection of small constraints and state machines." Since most existing natural-language specifications, such as the PCI specification, are already written as a list of rules, the translation to this type of specification requires less manipulation and results in fewer opportunities for error. Furthermore, not only is such a specification easier to write, it is easier to read and understand than a complicated, large state machine description. And finally, on a theoretical note, this specification style together with a simple-to-check property guarantees the existence of a implementation (and more precisely, a stronger property called *receptiveness*). Although many of these qualities are hard to quantify, is it noted that our PCI specification has been used by others working on related projects such as Clarke *et al.* [CGY$^+$00] and Aloul *et al.* [AS00].

Specification Checking. Two debugging methods were found to be effective for this style when developing the PCI specification. Their main purpose is to check if the specification is overly restrictive, or under-restrictive, and whether it agrees with the protocol designer's intent.

The *dead state check* finds contradictions in the specification. It searches for a state where all the specification constraints have been true so far, but there is no next state where all of the constraints are true. This indicates that the specification places contradictory requirements on the system at some point in its execution. To determine this, it is sufficient to model check the specification with the following property,

$$AG(all\_correct \rightarrow EX(all\_correct))$$

where *all_correct* true indicates that none of the constraints have been violated so far. This property guarantees that the system can execute infinitely obeying the protocol no matter which path it chooses, as long as at each step, an *all_correct* next state is chosen.

The *characteristic check* checks whether the specification guarantees certain properties. These characteristics are logical statements about agent events and are expressed in CTL.[1] They are checked against the specification using a CTL

[1] It must be emphasized that the *specification* constraints are simple, bounded, linear time properties and these *checking* properties are more complex, unbounded, CTL formulas.

model checker and by the counterexamples provided by the model checker, the check finds bad scenarios allowed by the specification. This technique requires knowledge of the protocol because the characteristics cannot be deduced automatically from the specification.

2 Specification

A core subset of the Itanium™ Processor bus protocol was specified; the request phase, the snoop phase, the response phase, the data phase, and the deferred phase are all covered, but, for example, the arbitration phase is not. Higher level properties of deferred transactions, such as the assurance of completion, are not specified or checked because they are probably better treated in a different specification.

In this section, it is illustrated how the specification style specifies advanced features that were not part of the simpler PCI protocol. There is a description of pipeline specification, the treatment of the protocol's time-unbounded rules, and an explanation of a reaction timing issue.

Definitions Here are some terms that are used throughout this paper.

constraints	The small, propositional formulas in the *formal* protocol specification describing agent behavior
rules	Specification properties in the *informal*, English specification provided by the protocol designer. There is no one-to-one relationship between constraints and rules. In most cases, one rule corresponds to multiple constraints.
history variables	Variables that retain information from the past. This can be as simple as the value of a single bus signal from one cycle ago, or as complicated as a counter.
agents	The entities communicating with one another using the interface, or in this case, the bus.

2.1 Pipelining

First, an un-pipelined version of an example constraint will be described. The protocol has six phases. Transactions must go through the arbitration phase, request phase, error phase, snoop phase, response phase, and data phase. The type of transaction, whether it is a write or a read, is determined during the request phase. An important signaling event, the assertion of *trdy*, happens during the response phase. Thus, the example rule, "*trdy* must be asserted for a write transaction," requires the transaction type (write) to be stored from the request phase until the response phase. Also, the end of the response phase is defined by the assertion of *rs* and so the rule requires that *trdy* be asserted before *rs*.

Consequently, to specify the constraint, two auxiliary variables are sufficient. First, there is a history variable *write* which becomes true during the request phase if the transaction is a write, and stays so until the transaction is completed.

The second history variable, *trdy_happened*, becomes true when *trdy* is asserted and stays so for the duration of the transaction.

Using these variables, the constraint becomes, "if a transaction is a write, then if *trdy* assertion hasn't happened yet, *rs* cannot be asserted (i.e. the response phase can't complete)."

$$prev(write) \rightarrow (prev(\neg trdy\_happened) \rightarrow \neg rs)$$

For a pipelined version, while transaction i is in the response phase for example, transaction $i-1$ can be processed concurrently, say in its request phase. Thus, there needs to be multiple *write* variables and *trdy_happened* variables for each outstanding transaction. Assume that there is a mechanism to tag each transaction with an ID number. This same number can be used as a subscript on the history variables to create separate variables for each transaction. Thus, the history variables become $write_i$ instead of *write*, and $trdy\_happened_i$ instead of *trdy_happened*. Also, the constraints are activated only when a particular transaction reaches a particular pipeline phase. Consequently, each constraint developed in the un-pipelined version is (indirectly) indexed by a transaction ID in the pipelined version. The example constraint now becomes,

$$prev(write_i \wedge (response\_phase = i)) \rightarrow (prev(\neg trdy\_happened_i) \rightarrow \neg rs)$$

which is, "if transaction i is a write and is undergoing the response phase, then if *trdy* assertion hasn't happened yet, *rs* cannot be asserted." Thus, to create a pipelined version from an un-pipelined specification, the constraints and history variables are replicated and indexed by transaction IDs.

The transaction ID assigning process is implemented by counters. In this scheme, the transaction ID corresponds to the order in which the transaction started. The first transaction that undergoes the arbitration phase (which is the first phase in the pipeline) is assigned the ID of 0. The ith transaction that undergoes the arbitration phase is assigned the ID of $i-1$. Since there can only be eight outstanding transactions, the IDs are assigned modulo 8. This ID scheme works because most phases for a transaction happen in-order;[2] the ith transaction to undergo the request phase is the the ith transaction to go through the response phase.

For each phase, there is a signal expression which indicates that the phase has completed. For a particular phase, if the number of occurrences of these "complete" signaling is known, the number of transactions that have undergone this phase so far is also known. In this way, the specification can keep track of which transaction is being processed in each phase. For example, the request phase's completion event is the assertion of *ads*. The request phase counter increments at every occurrence of this, and the counter value indicates the ID of the transaction currently in the request phase. As this transaction moves onto subsequent phases, each phase counter will have this same value.

[2] Data phases can be deferred and completed later so they do not necessarily happen in order.

Thus, the general form for constraints is, "if transaction i is a write (or read) and it is currently undergoing the response (or request or snoop) phase, then p must hold."

$$prev(transaction\_type_i \wedge (some\_phase = i)) \rightarrow p$$

(where for example, $transaction\_type_i$ is a "write" and some_phase is a "response phase" counter)

Thus, using these counters and appropriate history variables, a pipelined protocol can be easily specified with small constraints. The one drawback of this methodology is the linear increase in the number of state variables, which may be a problem when model checking. However, this is only a problem with verification and not the specification. The specification scales well; the constraints only have to be duplicated with the counter values and variable subscripts changed. In fact, the methodology specifies a certain intricate Itanium™ bus feature effectively and simply. Although there are only five main pipeline stages, the protocol supports eight outstanding transactions at any time. The extra transactions are buffered at each stage. With the monitor-style constraints, *the buffers do not have to be explicitly modeled* to specify the agent-bus behavior. There are simply eight copies of the constraints for the eight outstanding transactions.

2.2 Time-Unbounded Rules

Compared to the tightly timed PCI protocol, the Itanium™ Processor bus has a less constrained timing relationship among the different signal events. With PCI, most rules fall into the category, "exactly n cycles after event a happens, event b *must* happen." The PCI protocol is a *time-bounded* protocol where most events are guaranteed to happen within a certain time span.

In comparison, there are no such guarantees with the Itanium™ Processor bus protocol. Most of the rules follow the form, "any time after n cycles have elapsed since event a, event b *may* happen." The environment must expect that the event can happen at n, or $n + 1$, or $n + 2$, and so on, and be designed accordingly. An example from the protocol is "*trdy* may be deasserted a minimum of 3 cycles after the deassertion of the previous *trdy*".

Furthermore, there are no rules stating that an event must eventually happen (the so-called *liveness* property). In essence, the protocol is a *time-unbounded* protocol. Hence, the protocol allows the bus agents to have more freedom in ordering events, and optimizing bus performance. However, this extra degree of freedom leads to more corner cases in the specification that need to be checked.

Constraint Style. In the prior section, it is explained that the constraints for this protocol are written in the form,

$$prev(transaction\_type_i \wedge (some\_phase = i)) \rightarrow p$$

The form of p will now be described. The most natural form, considering the "time-unbounded" characteristics of the protocol, is

$$p : \neg prev(q) \rightarrow \neg r$$

"If q is true in the previous cycle, then r *may* be true in the current cycle." The expression for q is a *trigger* condition which enables a certain exchange or a change of state. A trigger condition which signals that an agent is ready to receive data is one example. Another example is a trigger condition which indicates the completion of an event so that a bus signal can be deasserted.

One consequence of this time-unbounded feature, is that the dead state check, in its original form, does not catch contradictions. This is due to the fact that *because most actions are not required to happen at any given time (the time-unbounded characteristic), the bus can always choose to loop in the current state and "do nothing."* If the check searches for a legal (all the specification constraints are true) current state which has no legal next state, with a time-unbounded protocol, the current state is always a legal next state and so the check is vacuous. Therefore, the dead state definition needs to be expanded so that the test check for the following desired property,

$$AG(all\_correct \rightarrow EX(all\_correct \wedge \neg same))$$

where *same* is true if all the state variables, except the timer-like variables which increment at each clock, have the same value as in the previous state. Thus, this check ensures that, at every legal state, there is at least one possible legal next state where some change happens and the bus is not forced to stay in the current state. And so, a check that was effective for PCI is modified for the Itanium^{TM} protocol so that a wider class of anomalies are detected.

2.3 2-Clocks or 1-Clock Reaction Time

Unlike the PCI protocol, the Itanium^{TM} process bus is a latched protocol where there is a 1 cycle delay from when the bus agent asserts (or deasserts) a signal to when the action appears on the bus. Thus, when observing events on the bus, a reaction to a trigger event happens (at earliest) in two cycles instead of one. On cycle n, a trigger condition becomes true on the bus; on cycle $n+1$, the agent asserts a signal in response; and on cycle $n+2$, the assertion appears on the bus. And so most time-bounded constraints are in the form, $prev(prev(input)) \rightarrow output$ where *input* and *output* are bus signal expressions.

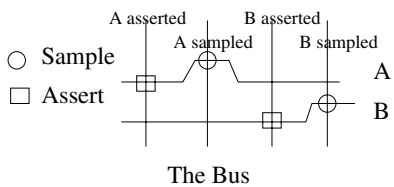


Fig. 2. The Latched Bus Protocol - 2 Cycle Response Time

However, the response to a trigger *from the same agent* may happen in one cycle. There is no one-cycle delay through the latched bus in this case. For

example, there is a rule, "*ids* cannot be asserted in cycle N if *trdy* is sampled asserted on clock N." As a constraint, this becomes $prev(trdy) \rightarrow \neg ids$. This requirement is possible only because both *trdy* and *ids* are controlled by the same agent at all times. The agent does not need to wait for the trigger (*trdy*) to appear on the bus to react ($\neg ids$) to it.

The difficulty arises when the trigger condition of a rule is a *mix* of external and internal signals. An example is "$input \wedge output_0$ true requires $output_1$ to become true." If an external signal causes the trigger expression to become true ($output_0$ was already true and *input* just became true), then a two cycle reaction time is needed. However, if the agent's own signal causes the trigger condition to become true (*input* was already true and $output_0$ just became true), it can react in one cycle. Thus, the rule needs to be separated into two constraints depending on the situation. However, the English specification fails to distinguish between the two cases, and states a blanket requirement allowing a one cycle reaction timing at all times. This is problematic because if a particular agent asserts a reaction earlier (one cycle) than expected by other agents (two cycles), it may lead to an undefined state. In fact, the modified dead state check detected a scenario where this misstated rule led to a contradiction.

3 Debugging

Several issues were found with the development version of the Itanium™ processor bus protocol using the methodology's debugging procedures. Some are omission of rules that are arguably implicit in the official specification, but violate the completeness concept. Others are serious enough to cause data phases to be missed unnoticed, or cause a deadlock. These were resolved by revising the Intel® protocol specification.

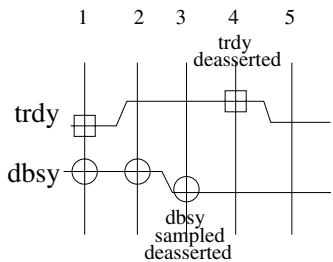
3.1 Found by Dead State Check

The dead state check mainly found cases of missing rules. The informal specification tends to state the sufficiency conditions of an action, while leaving necessary conditions implicit. For example, a sufficient condition for the assertion of *trdy*, "if the transaction is a write, then a *trdy* assertion must happen," is stated in the specification. Logically, it specifies $write\_transaction \rightarrow trdy\_happens$. However, a necessary condition that *trdy* can only be asserted at certain times, is missing. The specification should state that "*trdy* can be asserted *only if* the transaction is a write or has a snoop-initiated data transfer." Else, the system will reach an undefined state because the agents do not expect *trdy* to be asserted during a read, for example. By adding such a rule, the specification is made more complete so that a simulation checker can catch erroneous behavior at the earliest time. Overall, there were five cases of such omissions where the specification does not state that a particular event can happen *only if* certain conditions are true.

3.2 Found by Characteristic Check

Missing Trigger Condition and Resulting Deadlock. There is a pair of communicating agents: a data sending agent, the *Sender*, and a receiving agent, the *Receiver*. When the bus signal *trdy* is true, the Receiver is signaling that it is ready to receive data. When *dbsy* is true, the Sender is sending data. So when $\neg dbsy$ is true, the Sender is idle, and is ready to start the next data transfer. Thus, when $trdy \wedge \neg dbsy$ is true, both agents are ready; consequently, $trdy \wedge \neg dbsy$ is a trigger condition that allows a new data phase to start.

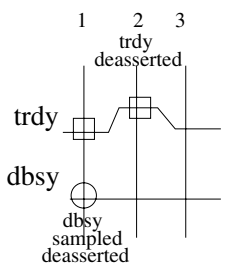
The protocol is designed so that the Receiver keeps *trdy* true until the data sending agent is idle and *dbsy* is deasserted ($\neg dbsy$). Thus, the normal sequence of events is as shown in Fig. 3. Note the one cycle delay between a signal change and its appearance on the bus because of the latched property.



Time	Receiver	Sender
1	asserts *trdy* and observes that *dbsy* is active	driving *dbsy*
2	keeps *trdy* asserted	deasserts *dbsy*
3	observes that *dbsy* is low	idle
4	in response, deasserts *trdy*	idle
5	idle	idle

Fig. 3. *trdy* and *dbsy* relation

What makes the protocol tricky is that if *dbsy* is already deasserted, as an optimization, *trdy* can assert and deassert right away (Fig. 4). Note that, unlike the normal sequence, the trigger ($trdy \wedge \neg dbsy$) is true for only one cycle in this case. Thus, this scenario lets the idle state, $\neg trdy \wedge \neg dbsy$, happen a cycle earlier (that is why it's an optimization).



Time	Sender	Receiver
1	asserts *trdy* and observes that *dbsy* is inactive	idle
2	deasserts *trdy* as an optimization	idle
3	idle	idle

Fig. 4. When Optimized : *trdy* and *dbsy* relation

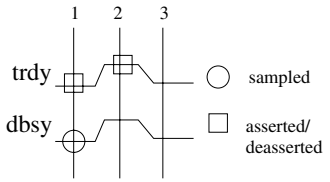


Fig. 5. Missed Trigger Condition : $trdy \wedge \neg dbsy$ is missed

It is important that *trdy* and *dbsy* never become true on the same clock because of the following possibility (Fig. 5). On cycle 1, the Receiver samples *dbsy* low so it does the optimization, but because in cycle 2, *dbsy* is high, *trdy* is inadvertently deasserted before the trigger condition becomes true. So the awaited trigger condition never happens and a data phase cannot start.

In most cases, the protocol is designed so that *trdy* and *dbsy* cannot become true on the same clock, but, in a special case, a loophole in the protocol allows this. Normally, because *trdy* and *dbsy* are handshake signals between two communicating agents, they are not asserted at the same time. However, there is a case where a third agent can assert *dbsy* and thereby break this rule. This special case is when an agent which previously deferred to complete a data phase (the *Deferrer*), takes advantage of the apparently idle bus to complete the deferred data phase. Meanwhile, there is a separate ongoing transaction where an Receiver agent is about to assert *trdy* to communicate to a Sender agent. The Deferrer asserts *dbsy* to start the data transfer at the same time the Receiver asserts *trdy*, and, as a result, *trdy* and *dbsy* are asserted at the same time. Note that, in this scenario, the *trdy* and *dbsy* assertions are not for the same data transfer. This overlooked case causes the normal data transfer to wait forever for the (missed) trigger condition of $trdy \wedge \neg dbsy$.

In a similar case,[3] a deadlock occurs because a transaction cannot proceed unless the data phase completes.[4] But the data phase can not happen because the trigger condition did not become true.

Since this trigger condition is crucial for data phases, and optimizations often lead to unexpected scenarios, the existence of the sequence shown in Fig. 5 was one of the first checked properties. Although the model checking properties cannot be automatically deduced, after the specification process, properties testing for suspicious sequences can be developed with little difficulty.

[3] For the protocol experts: this happens when 1. the responding agent and the requesting agent are the same and 2. it is a write with a snoop-initiated data transfer, where the second *trdy*, which is for the snoop-initiated transfer, happens exactly when the data phase (which was allowed to be indefinitely delayed because of 1.) for the first *trdy* starts.

[4] For the protocol experts: this requirement is because the second data phase is snoop-initiated, and it must happen together with the response.

Dropped Data Phase. Under certain circumstances, a write data phase can be delayed indefinitely.[5] The immediate danger of this is that the data phase never happens, and the system proceeds without any trace of the phantom data phase. The basic signalling mechanism of a write transaction is,

1. The Receiver indicates a "ready" state by asserting *trdy* true.
2. The Receiver may deassert *trdy* false before the Sender starts the data phase.
3. The Sender, acknowledging the "ready" signal, starts the data transfer by asserting *dbsy*.

Thus, the normal sequence of events is, "*trdy* is asserted and then deasserted, data is transmitted, *trdy* is asserted and then deasserted, data is transmitted, ..." Now, consider the sequence "*trdy* is asserted and then deasserted, *trdy* is asserted again, data is transmitted, ... (Fig. 6)." The one-to-one correspondence between a *trdy* assertion and a data transfer breaks down. The second *trdy* assertion should not have happened before the start of the first data transfer. Consequently, the data phase for the first *trdy* misses its window to start the transfer. This happens only in the case where a data phase can be delayed indefinitely.

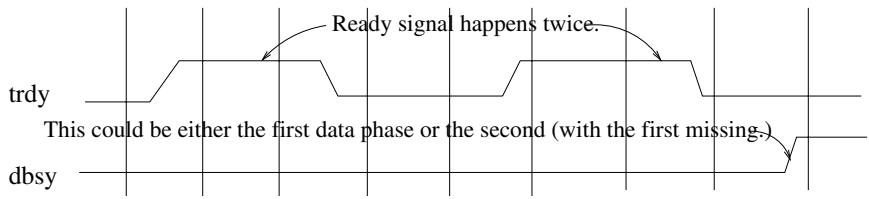


Fig. 6. Early second *trdy* assertion

This was found by model checking whether the specification allows a second *trdy* assertion before the start of a pending data phase. Coincidently, this problem was also discovered using simulation by the testing team at Intel, but since our methodology does not require an implementation, it found the problem in a shorter time with less effort.

4 Conclusion

The formal specification for the core subset of the Itanium Processor bus protocol consists of 46 independent constraints which can be replicated eight times for the pipeline depth of eight. To minimize model checking complications, a pipeline depth of two was used for debugging the specification. Again, this is not a limitation of the specification methodology, which scales well. The description

[5] For the protocol experts: this happens when the responding agent and the requesting agent are the same.

file for a pipeline depth of two, written in Cadence SMV [McM], is 650 lines long, excluding the variable declarations. The formal specification was debugged using the techniques described; the current specification has no dead states and all characteristics checked for, hold. Model checking was done using Cadence SMV [McM] and all characteristics checks can be completed within two minutes on a Pentium Pro system with 128Mb of memory.

Acknowledgement. This research was supported by GSR contract SA2206-23106PG-2. Many thanks to Mani Azimi and Sridhar Lakshmanamurthy at Intel®, and Chris Wilson at Stanford University.

References

[ABG+00] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs - Automatic Generation of Simulation Checkers from Formal Specification. In *International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[AS00] F. Aloul and K. Sakallah. Efficient Verification of the PCI Local Bus using Boolean Satisfiability. In *International Workshop on Logic Synthesis (IWLS)*, 2000.

[CCLW99] 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.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, May 1981.

[CGY+00] E. Clarke, S. German, Y. Lu, H. Veith, and D. Wang. Executable Protocol Specification in ESL. In *Proceedings of the Third International Conference of Formal Methods in Computer-Aided Design*, November 2000.

[Cor] Intel Corporation. Itanium Processor Bus Protocol Specification. Internal document.

[GD00] Shankar G. Govindaraju and David L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of International Conference on Computer-Aided Design*, November 2000. San Jose, CA.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[KMP98] M. Kaufmann, A. Martin, and C. Pixley. Design Constraints in Symbolic Model Checking. In *International Conference on Computer-Aided Verification*, 1998.

[McM] Kenneth McMillan. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[MHG98] 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*, volume 1522 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[SDH00] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-Based Formal Specification of PCI. In *Proceedings of the Third International Conference of Formal Methods in Computer-Aided Design*, November 2000.

[SIG95] PCI SIG. PCI Local Bus Specification, Revision 2.2, 12 1995.