Formalization and Analysis of a Solution to the PCI 2.1 Bus Transaction Ordering Problem *

Abdel Mokkedem, Ravi M. Hosabettu, Michael D. Jones and Ganesh C. Gopalakrishnan
Department of Computer Science, University of Utah, Salt Lake City, UT 84112-9205
(mokkedem,hosabett,mjones,ganesh@cs.utah.edu)

Abstract. The transaction ordering problem of the original PCI 2.1 standard bus specification violates the desired correctness property of maintaining the so called 'Producer/Consumer' relationship between writers and readers of data. This violation stems mainly from the so called *completion stealing* problem, first identified and solved by Corella et al [4], and supported by a formal paper and pencil argument. In this paper, we develop a flexible graph theory library in PVS for modeling computer bus structures, formalize the PCI 2.1 protocol containing the solution of [4] in it, and mechanically prove the absence of completion stealing. Next, we define the Producer/Consumer property in PVS and sketch its mechanical proof. Noting the complexity of this proof effort (unfinished as yet), we explore a combination of theorem proving and model-checking in which the model used for model-checking is made tractable by exploiting the formal theorems established during theorem-proving as well as several intuitively justified assumptions. The theorem-proving infrastructure we have built for modeling CPU interconnect structures is highly reusable. Our work is one example of a natural division of labor between theorem-proving and model-checking in tackling system-level verification problems under realistic time budgets.

1. Introduction

Most modern computer I/O subsystems are based on industry-wide open standards. Manufacturers of bus peripherals are interested in conforming to the standards while at the same time choosing an implementation method that best suits their needs. To ensure compatibility between the peripherals built by various manufacturers, two things must be ensured: (i) it must be possible to establish that a given implementation conforms to the standard; (ii) the standard must be (internally) consistent and defined in an unambiguous manner. This paper contributes towards the second concern by studying a modern I/O bus standard, namely PCI [14, 15]. We primarily employ the theorem-proving technology embodied in the PVS [12] theorem prover. We

* Supported in part by DARPA under contract #DABT6396C0094 (Utah Verifier), and NSF grant CCR-9800928. The second author was also supported by a University of Utah Graduate Fellowship.

also illustrate a natural division of labor between theorem-proving and model-checking to accomplish verification under realistic time budgets, by using the SPIN [8] model-checker.

Motivation for our work stems from several sources. Interconnect standards are typically long-lived. The volume of products built based on standards is staggering. The sheer number of *new* bus standards being proposed as well as the functional complexity of each are equally staggering. Interconnect standards will also play a major role in supporting new industries such as *intellectual property* (IP) core based design of integrated circuits [16]. Hence it is crucial that every avenue be explored to ensure the fidelity of each such standard, typically characterized by its consistency, the set of crucial properties stated of it, and the ease with which these properties can be established either formally or semi-formally with an acceptable measure of confidence under realistic time budgets. Towards this end, our work elaborated in this paper provides two key insights:

- Formal methods based on theorem proving can play a crucial role in the development of these standards by providing the precise and expressive notation of higher order logic to characterize entire families of devices, allowing formal statements and proofs of crucial correctness properties, and forming a solid framework for reliable future extensions to standards.

- Formalization using a theorem prover can also facilitate the application of 'light-weight' formal methods such as model-checking by significantly limiting the number and/or complexity of the cases to be modeled using the light-weight methods.

In this paper, we demonstrate how a modern theorem prover, namely PVS [12], can be used to carry out such a formalization for a significant portion of the PCI 2.1 standard. Our main contribution is a PVS-based formalization of a solution to the PCI 2.1 transaction ordering problem proposed, as well as informally proved, in [4, 3]. Briefly, there are two problems with the PCI 2.1 standard: *deadlock*, arising from inadvertently disallowing certain transaction re-orderings; and *inconsistency*, arising from the so called "completion stealing". More specifically, [4, 5] show that the PCI 2.1 standard allows implementations that deadlock, by not allowing certain transactions to bypass others in bus channels. They also point out that the PCI 2.1 standard allows a crucial correctness property—namely Producer/Consumer—to be violated. Producer/Consumer requires that if a writer writes data in one location and then writes a flag announcing the availability of the data, a reader which reads the flag true can subsequently read the data location and be guaranteed that it receives the data that was written. [5, 4] show that this criterion is violated because PCI 2.1 does not maintain master IDs, and consequently third-party bus masters can inadvertently steal a "completion token" intended for another requester. The solution proposed in [5, 4] involves using reserved pins in the PCI 2.1 interface to implement *Local Master IDs.* This solution has been widely discussed in the PCI mailing list `pci-sig-request@znyx.com` as "the Hewlett-Packard solution". In [4, 3], an intuitive manual proof is given for the fact that this solution guarantees Producer/Consumer by avoiding completion stealing.

Our main contributions are as follows. First, we use a more detailed model of the PCI bus which is closer to the actual standard. Next, we build a reusable theory hierarchy which will allow transaction oriented interconnect structures and busses to be easily specified in future projects of this kind (we have invested considerable effort in building this theory hierarchy for instance, by extending recently proposed graph libraries [1]). We identify the necessary conditions which characterize well-formed PCI networks and mechanically prove crucial properties pertaining to the PCI bus, including providing a formal specification of the *three* graph configurations to be considered in proving Producer/Consumer. We mechanically prove the absence of completion stealing. As far as the Producer/Consumer property goes, we sketch its unfinished mechanical proof. We then explore a complementary use of theorem proving and model-checking in which the model-checker significantly limits the number of cases considered by capitalizing on results established by the theorem prover.

Related Work: Modeling modern coherent memory busses [7], as well as the verification of coherence and memory models [13] has been done in the context of PVS. Although I/O busses have been studied [10], the examination of entire I/O bus standards in the context of modern theorem provers has not been done. While model-checking tools have been used to debug many bus protocols (e.g., [2]), formal examination of entire bus standards cannot adequately be performed in their context. Our work is one of the few that addresses, in the context of a powerful theorem prover, a modern as well as widely used I/O standard.

2. The PCI Local Bus

The PCI Local Bus has been defined with the primary goal of establishing an industry open standard for system I/O buses that addresses the performance issues associated with peripheral functions with high bandwidth requirements. PCI is a high performance, 32-bit or 64-bit

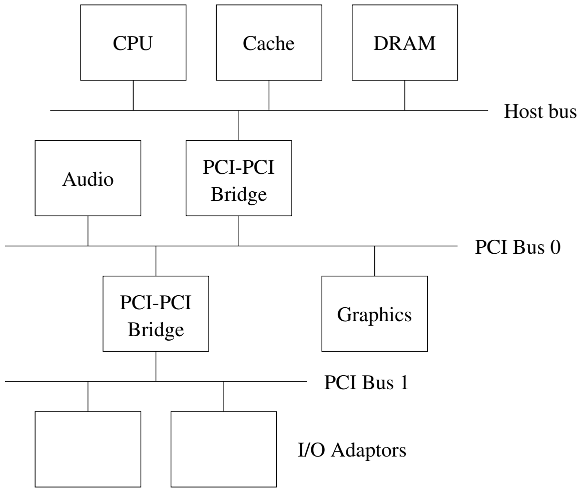

Figure 1. A PCI acyclic network

bus with multiplexed address and data lines. It is intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in boards, and processor/memory systems. Beginning with the next section, we present an informal description of the PCI bus.

2.1. Topology

Figure 1 shows an example of an acyclic network connecting several PCI busses and a Host bus through *bridges*. Primarily, a bridge interfaces one bus to another and limits the loading of each bus. A bridge between two busses B and B' supports two opposing channels corresponding to the two direction of the traffic. One of the channels has B as its *in-bus* and B' as its *out-bus*, while the other has B' as its *in-bus* and B as its *out-bus*. Similarly, every agent has two opposing channels, called the *master channel* and the *target channel*. The hypergraph defined by the PCI network topology, which has bridges and agents as vertices and busses as hyper-edges gives rise to a directed graph that has the bridge channels and agent channels as nodes. There is an edge from a channel N_1 to a channel N_2, written $N_1 \rightarrow N_2$, if the out-bus B of N_1 is the same as the in-bus of N_2. We call this directed graph the *channel graph*. Note that $N_1 \rightarrow N_2$ implies $N_2' \rightarrow N_1'$, where N_1' and N_2' are the opposite channels of N_1 and N_2 respectively.

From the acyclicity of the hypergraph defined by the PCI network topology, it follows that the channel graph is a *directed acyclic graph*. Moreover, the fact that the hypergraph is connected means that for every pair (V_x, V_y), where each $V_{x/y}$ is an agent or a bridge, there exists exactly one pair (N_x, N_y), where each $N_{x/y}$ is one of the two channels of $V_{x/y}$, such that there is a path from N_x to N_y in the channel graph. If V_x is an agent then N_x is the master channel, and if V_y is an agent, then N_y is the target channel.

2.2. The PCI 2.1 protocol

A good summary of the PCI protocol is presented by Corella et al in [5]. We reproduce some material from this summary here; see [5, 14, 15] for more details. Revision 2.1 of the PCI protocol [14] became the official production version in 1995. It introduced a new transaction mechanism, *delayed transaction*, that allows much higher performance. *Posted* transactions are memory writes, while *delayed* transactions are (memory and I/O) reads and I/O writes. A delayed transaction completes on the destination bus before completing on the originating bus, while posted transactions complete on the originating bus before completing on the destination bus. One advantage of a delayed transaction is that the bus is not held in wait states while completing an access to a slow device.

Transaction propagation: A transaction is issued by an agent, the *master* of the transaction, and specifies an address which determines another agent, the *target* of the transaction. A *posted* transaction propagates from the *originating bus* of the transaction to the *destination bus*. A *delayed* transaction propagates from the originating bus to the destination bus, and then the *completion* of the transaction travels back from the destination to the originating bus. The completion carries the data, in the case of a delayed read, or the termination status (normal or abnormal) in the case of a delayed write. The address of the transaction determines the target of the transaction and hence the path that the transaction must follow.

As a (*global*) transaction propagates it causes one or more (*local*) subtransactions to be issued on one or more buses that separate the master from the target of the global transaction. Each of those subtransactions has a local master, which may be the master of the global transaction or a bridge acting on its behalf, and a local target, which may be the target of the global transaction or a bridge acting on its behalf.

For a *posted* subtransaction, the local target may either *complete* it or request the local master to retry. In the former case, the subtransaction propagates to the next level of bus interconnect. In the latter case, the local master must reissue the posted subtransaction. The local target then latches the transaction information, creating a Posted Memory Write ("P entry"). For a *delayed* subtransaction, the local target may either *ignore* it, or latch it creating a Delayed Read Request entry or a Delayed Write Request entry. In the latter case, an "R entry" (referring both to DRR and DWR) specifying the address of the transaction is created. In the case of a write, the data value written is also specified. The local master of a delayed transaction must reissue the transaction until it is completed. The local target meanwhile tries to obtain a completion corresponding to every R entry. When the completion is obtained, a "C entry" (a Delayed Read Completion, DRC, or a Delayed Write Completion, DWC) is created at the local target. The C entry specifies the address and the data in case of DRC and the address and the termination status in case of DWC. The termination status may be normal or abnormal. Once a C entry is created at the local target, the delayed subtransaction completes when *issued again* by the local master.

Ordering rules: PCI bridge channels must *disallow* certain re-orderings between P, R, and C entries in order to preserve desired semantics such as the Producer/Consumer relationship, while at the same time *permitting* certain re-ordering in order to avoid deadlocks. For instance, not allowing C entries to pass R entries can cause deadlocks [5]. Figure 2 specifies these orderings (this incorporates the fix proposed in [5] and is different from the PCI standard specification). In [5] a DWC entry is not allowed to pass a P entry because both DRC and DWC entries are treated as one common class called "C entries". As we maintain the distinction between DRC and DWC (as per the PCI standard), we can actually specify the detail that DWCs can pass P, thus achieving our formalization with respect to a more accurate bus model.

Row pass Column?	P	R	C
P	No	Yes	Yes
R	No	Y/N	Y/N
DRC	No	Yes	Y/N
DWC	Y/N	Yes	Y/N

Figure 2. The ordering rules in a bridge. Yes means "Must be allowed to pass", No means "Can not be allowed to pass" , Y/N means "the designer may choose either way".

R and C discarding rules: From the point of view of forward progress, the PCI protocol allows R entries to be discarded under certain circumstances. A bridge may discard an R entry only between enqueueing the entry and trying it on the destination bus for the first time. This is because an attempt on the destination bus *may* or *may not* result in the latching of the request. Taking the conservative position that it *does* get latched, a request is *committed* once it is attempted *at least once on the bus.* Thus, committed R entries may not be discarded; they must be repeated until they complete.

A bridge is allowed to discard a C entry in two cases: (i) when the completion corresponds to a read to a pre-fetchable address or to a Memory Read, (ii) when the master has not repeated the corresponding Delayed Request within 2^{15} clock cycles. In our PVS model of the protocol we model a weaker rule which allows a C entry to be discarded if it is not the oldest C entry in the channel.

States, events and transitions: At any time t, each channel contains a collection of *entries* $(E_i)_{0 \leq i \leq n}, n \geq 0$. An entry is a tuple (ϵ, α, τ), where ϵ is the transaction type (P, R, or C), α is the transaction address, and τ is the data (in the case of a read) or the status (in the case of a write) carried by the completion. As said earlier, a channel functions as a FIFO buffer except for the re-orderings permitted by Figure 2. The committed R entries present in a bridge channel N at time t comprise the *retry set* of N at time t. At the level of abstraction we consider, state transitions are caused by the following *events* [5]:

- A *P event* that *occurs on bus B*; A *P* event is triggered by a P entry E in a bridge channel or master channel N with out-bus B. If N_t is the target channel specified by the address α of E, there exists a path $N = N_0, N_1, \ldots, N_n = N_t$ from N to N_t in the channel graph. A P event is an abstraction of a posted bus transaction on bus B. There are *two kinds* of P events:

- A *P_retry* event, which has no effect other than marking the P entry as *committed.*

- A *P_completion* event, which deletes the entry E from N_0, and creates a P entry E' in N_1 with the same parameters. In the case where $N_1 = N_t$, the corresponding global posted transaction completes and the entry E' is created in the target channel of the addressed agent. We specify the target channel of an agent to store all the transactions completed, thus building up a transition history used in proofs.

- A *D event* that occurs on bus B; A D event is *triggered* by an R entry E in a channel N with out-bus B. If N is a bridge channel, such an event may happen only if there are no P entries older than E in N in accordance with the orderings specified in Figure 2. (This is to abide by the orderings in Figure 2.) If N is a bridge channel and E is not in the *retry set* of N, then E is added to the retry set of N. The event is then called an *R_commit event.* A D event is an abstraction of a delayed transaction on bus B. There are *two kinds* of D events:

- A *D_retry* event which can in turn be a *D_noop* event which has no effect, or a *D_latch* event, which creates an R entry E' in N_1 with the same parameters as E. A *D_latch* event can only happen if $N_1 \neq N_t$, E' is not already latched in N_1 and the channel N_1' opposite to N_1 does not contain a C entry with the same parameters as E.

- A *D_completion* event, which is enabled only if (i) $N_1 = N_t$, or (ii) $N_1 \neq N_t$, the channel N_1' opposite to N_1 contains a C entry E'' with the same parameters as E and there is no P entry in N_1' older than E'' (in case C is a DRC entry). Its effect on the state is to remove E'' from N_1' in case (ii) and to create a C entry E''' with the same parameters in the channel N_0' opposite to N_0 and to remove E from N_0. This is the mechanism by which completions for transactions travel from the target of the transaction back to the master. Also, we record, as a history of all completed transactions, all R entries that reach the final destination and all C entries that reach the original master for use in specifying and proving transaction ordering properties.

3. Transaction ordering problem and the HP proposal

Overview of the transaction ordering problem: In the PCI protocol, a C entry is matched against an R entry based on the Address and Byte Enable fields. It is not possible to use the identity of the master that issued the transaction in the comparison, because the Master ID information is not maintained in the PCI 2.1 standard. Thus, the local target does not know which master issued the R entry that resulted in a C entry. A completion of a delayed transaction may thus be given to a transaction initiated by a different master, resulting in completion stealing, causing violation of the Producer/Consumer relationship [4, 3]. To re-cap, the Producer/Consumer relationship says that if a master, called the Producer, writes to a location called Data, and then to a location called Flag, and if a second master, called the Consumer, sees the value written to Flag by the Producer and then reads Data, the Consumer is guaranteed to see the value written to Data by the Producer no matter where Producer, Consumer, Data and Flag are placed in an arbitrary PCI bus network.

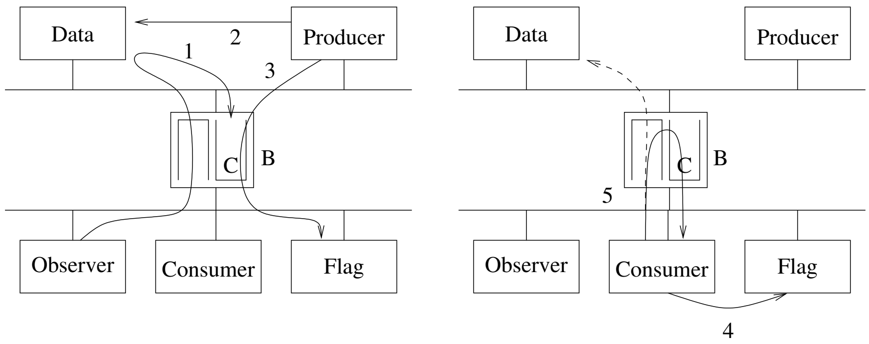

Figure 3. Completion stealing in a PCI network

Figure 3 shows a scenario which violates this rule. The left side of the figure shows the events leading up to the violation and the right side shows the violation. First, a third master called the Observer reads Data and the completion C containing the Data value meant for this read appears in the bridge B. Second, Producer writes a new value to Data, and then writes to Flag. The violation occurs after Consumer reads Flag, and attempts to read Data. The Consumer's read to Data matches the pending completion intended for Observer, so B returns completion C to Consumer. Since C contains the old value of Data, Consumer receives the old Data value instead of the new Data value written by Producer—thus violating the Producer/Consumer relationship. Many variants of this scenario are described in `pci-sig-request@znyx.com` [3], but they all relate to completion stealing. The problem is also documented in Section 3.11 of the PCI Local Bus Specification Revision 2.1, pages 116-117; however the severity of the problem is underestimated there. These complexities amply illustrate the need for formalization and analysis in this domain.

HP proposal: The solution proposed by HP [3] has two aspects: (i) using uncommitted hardware resources, *local master-ids are implemented*, and (ii) the requirement that bridges and multi-functions devices have only *one* outstanding transaction at a time for a given address. The reasons for the second requirement are to prevent R entries bearing local master-IDs from being *aliased* after one hop through a common bridge. (If such aliasing were to occur for some reason, completion stealing can happen in the subsequent bus.)

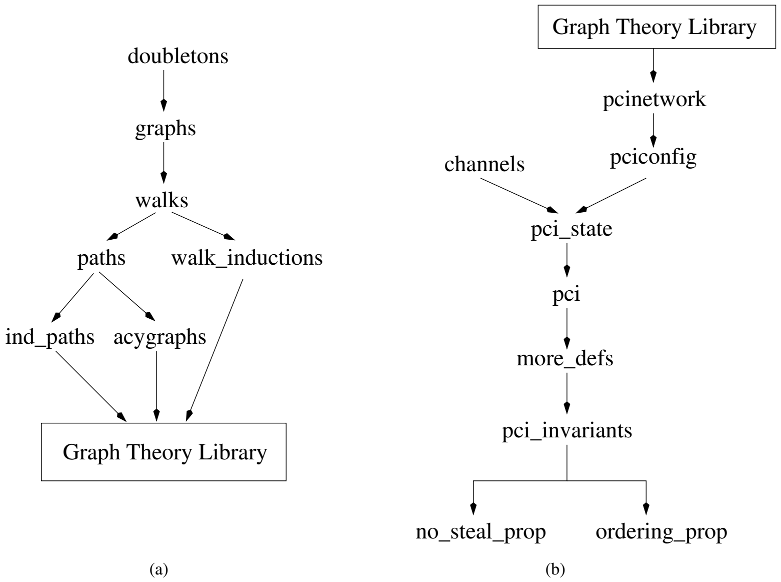

Figure 4. The theory hierarchy

Our PVS model of PCI protocol includes these two components. Our contribution is towards formally proving that the HP proposal is a complete solution to the transaction ordering problem for any acyclic network of PCI buses.

4. Formalization of the PCI Protocol in PVS

We now describe, in stages, how our PVS specification was constructed. Details not presented here, including the proofs completed so far, are available on the web [9].

4.1. The Model of PCI Network

The graph theory we used to model an arbitrary acyclic network of PCI busses is based on a modification of "A PVS Graph Theory Library" from NASA [1]. Figure 4(a) shows the graph theory hierarchy. Since the channel graph of a PCI network is a directed acyclic graph, we modified `doubleton` to be an ordered pair (used to define an edge) and then defined `graph` as directed graphs. A `walk` is a sequence of vertices connected by edges and a `path` is walk where no vertex appears twice. `ind_paths` provides the definition of independent paths used later in the proofs and `walk_inductions` defines induction schemes for walks. Finally an acyclic graph is defined as a graph where every walk is a path (in acygraphs). While most of the properties proved in the NASA library remained true, some needed modifications and we added some of our own.

1

```
pcinetwork: THEORY
BEGIN
   Agent_count, Bridge_count: posnat
   MasterId: TYPE = nat
   PCI_Device: TYPE = {n:MasterId | n < Agent_count+Bridge_count}
   Chanls: TYPE = { MASTER, TARGET }
   Nodes: TYPE = [# name: PCI_Device, chan: Chanls #]
   acynet: TYPE = acygraph[Nodes]
   opposite(nd:Nodes): Nodes =
       (# name:= name(nd),
          chan:= IF (chan(nd)=MASTER) THEN TARGET
                 ELSE MASTER ENDIF #)
```

The initial fragment of pcinetwork theory is shown in 1. A PCI device (an agent or a bridge) is represented by a natural number and Nodes defines the type of the channels of these devices. The type acynet defines acyclic graphs over Nodes. Given a node, opposite returns the node corresponding to its opposite channel.

However, not every acyclic graph over Nodes represents a valid network of PCI busses. For example, there cannot be any edge coming into the master channel of an agent. We have identified six conditions on a *well-formed* network of PCI busses:

1) To every edge, there is an opposite edge:

2

```
% edge? tests if the given 2 nodes are connected by an edge
% in the given network.
all_opposite_exists(net:acynet):bool = FORALL(nd1,nd2:Nodes):
  edge?(net)(nd1,nd2) IMPLIES edge?(net)(opposite(nd2),opposite(nd1))
```

2) For all agents, the master channels are only sources (i.e., only outgoing edges) and target channels are only destinations (i.e., only incoming edges):

3

```
% is_agent? tests if a given node is an agent.
agents_edges_ok(net:acynet):bool = FORALL(nd1:Nodes):
   is_agent?(nd1) IMPLIES
   (IF chan(nd1) = MASTER THEN
       (NOT (EXISTS (nd2:Nodes): edge?(net)(nd2,nd1)))
    ELSE  NOT (EXISTS (nd2:Nodes): edge?(net)(nd1,nd2)) ENDIF)
```

3) There is a path from the master channel of every agent to the target channel of every other agent:

```
% path_from? is true when the given sequence of vertices 'w'        4
% starts at 'nd1', ends at 'nd2' and is a path in 'net'.
master_target_connected(net:acynet):bool = FORALL(nd1,nd2:Nodes):
   (is_agent?(nd1) AND is_agent?(nd2) AND chan(nd1) = MASTER
        AND chan(nd2)=TARGET) IMPLIES
   (EXISTS (w:prewalk[Nodes]): path_from?(net,w,nd1,nd2))
```

4) The two channels of a bridge are not connected:

```
no_self_loop(net:acynet):bool =  FORALL(nd1,nd2:Nodes):              5
   (edge?(net)(nd1,nd2) and name(nd1) = name(nd2))
       IMPLIES is_agent?(nd1)
```

5) All the channels driving a PCI bus are connected "properly":

```
bus_connections_ok(net:acynet):bool = FORALL(nd1,nd2,nd3:Nodes):     6
   (edge?(net)(nd2, nd1) AND edge?(net)(nd3, nd1) AND nd2 /= nd3)
        IMPLIES
   (edge?(net)(nd2, opposite(nd3)) AND edge?(net)(nd3,opposite(nd2)))
```

6) Every channel has a local master id and for any two channels driving the same PCI bus, the local master id's are different:

```
lmid_map : [Nodes -> nat]                                            7

lmid_ok(net:acynet):bool = FORALL(nd1,nd2:Nodes):
   (edge?(net)(nd1,nd2) and name(nd1) /= name(nd2))
       IMPLIES lmid_map(nd1) /= lmid_map(opposite(nd2))
```

Finally, a well-formed PCI network is an acyclic graph over `Nodes` that satisfies all the above predicates.

4.2. Some Properties of PCI Network

We have proved many interesting properties about our PCI network model. The three very crucial properties that will be used in the proofs later are discussed below:

Property 1. For every path p in the network, there exists another path p' in the opposite direction, with the nodes in p' being the opposite nodes of the corresponding nodes in p. The proof of this property uses the well-formedness conditions stated before and the induction scheme for walks.

Property 2. There is at most one path between any two nodes in the network. Its proof uses the acyclicity property of the network, the induction scheme for walks and the well-formedness conditions stated before.

Property 3. The third property captures all the possible configurations of the four nodes corresponding to the Producer *Prod*, the Consumer *Cons*, the Data *Data* and the Flag *Flag*. In the following paragraph, we use the notation $a \to b$ to denote the path from node a to node b. Recall that *Prod* and *Cons* are master channels (of some two agents), and *Data* and *Flag* are target channels (of some two agents). Now all the four paths $Prod \to Data$, $Cons \to Data$, $Prod \to Flag$ and $Cons \to Flag$ are guaranteed to exist by the third well-formedness condition. Assume *Prod* and *Cons* are different nodes and *Data* and *Flag* are different nodes. (The last paragraph in this section discusses the case when these assumptions are not true.) Consider the path $Prod \to Data$. The point where the path $Cons \to Data$ hits $Prod \to Data$ is called the *hit* (h) point. And the point where the path $Prod \to Flag$ leaves $Prod \to Data$ is called the *take-off* (t) point. It is proved that the hit point exists in all cases, the segments $Prod \to h$ and $Cons \to h$ are disjoint (i.e., no internal node is shared between these segments), and the segment $h \to Data$ is common to both the paths $Prod \to Data$ and $Cons \to Data$. Similarly it is proved that the take-off point exists in all cases, the segments $t \to Data$ and $t \to Flag$ are disjoint, and the segment $Prod \to t$ is common to both the paths $Prod \to Data$ and $Prod \to Flag$. Figure 5 illustrates the different configurations possible for these three paths and show how to construct the path $Cons \to Flag$ in each of these configurations.

Figure 5(a) illustrates the case when t is ahead of or same as h. From the acyclicity of the channel graph, it follows that the segments $Cons \to h$ and $t \to Flag$ are disjoint. And the segments $Cons \to h$, $h \to t$ and $t \to Flag$ constitute a path from *Cons* to *Flag* and this path is unique from Property 2 above.

Figure 5(b) illustrates the case when t is before h by at least two edges. The segments $Cons \to h$ and $t \to Flag$ are disjoint, otherwise there would be two distinct paths between t and h. The figure also shows the path $Flag' \to Cons'$ (detailed in Figure 5(d). $Flag'$ is the opposite node of *Flag* i.e., `opposite(`*Flag*`)`.) This path is made up of the segment $Flag' \to a'$, the edge between a' and c, the segment $c \to d$, the edge between d and b', and the segment $b' \to Cons'$. Now $Flag' \to a'$ exists because $a \to Flag$ exists and from Property 1. Similarly $b' \to Cons'$ exists. The edge between d and b' exists from the fifth well-formedness condition. Similarly it is proved that the edge between a' and c exists. From Property 2, it follows that this path $Flag' \to Cons'$ is unique. (And from Property 1, the opposite path $Cons \to Flag$ exists; this isn't shown explicitly in the figure.)

Figure 5(c) illustrates the case when t is before h, and is directly connected by an edge. Again, the segments $Cons \to h$ and $t \to Flag$

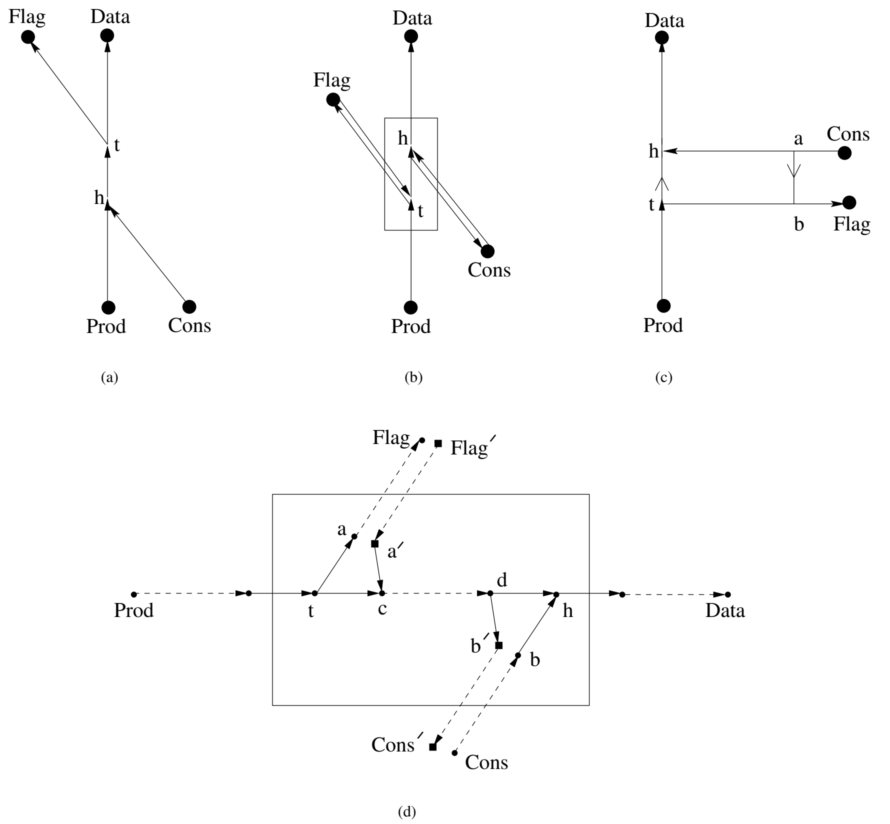

Figure 5. The possible configurations of *Prod*, *Cons*, *Data* and *Flag* are shown in (a)–(c). The portion shown inside the box in (b) is shown blown up in (d).

are disjoint. In this case, it is proved that there exist points a and b connected by an edge and the path from *Cons* to *Flag* is made up of the segment $Cons \to a$, the edge a to b and the segment $b \to Flag$.

The configurations when *Prod* and *Cons* are the same or *Data* and *Flag* are the same or both are similar to Figure 5(a) (with the respective nodes merged). We have mechanically proved that these cases *completely cover* all possible configurations of *Prod*, *Cons*, *Data* and *Flag* and the paths between them (in the `pciconfig` theory).

4.3. The PCI Protocol Formalization

The `pci_state` and the `pci` theories contain the formal specification of the PCI protocol including all the transitions discussed in Section 2. Recall that each node in our network model corresponds to a channel of a PCI device. The theory `channels` parameterized by the type of all PCI entries (`Entries`, shown in $\boxed{8}$) provides some operations on channels like addition/deletion of entries. An entry in our model includes all the transaction fields discussed in Section 2 augmented by (i) a local master ID '`lmid`' introduced in the HP proposal and (ii) `auxmid` and `tid` introduced by us and (iii) a `committed` field which indicates if the entry has been tried on the out-bus at least once. Each agent has a unique global master ID (`auxmid`) associated with it and every transaction issued by it has a transaction ID (`tid`) associated. These two fields are never used by the protocol and are introduced since they simplify the specification of Producer/Consumer. `addr_map` maps (associates) an address to an agent. The state of the network (shown in $\boxed{8}$) specifies the contents of every channel, the value of the next transaction ID for every agent and a record of the committed and completed transactions at every agent (discussed later).

8

```
Entries : TYPE = [# trans : Trans, addr: Addr, data: Data,
                    lmid: nat, auxmid: nat, tid: nat,
                    committed: bool #]

State : TYPE = [# channel: [Nodes -> channel[Entries]],
                  nextTid: [Agent_nodes -> nat],
                  trans_history: [Agent_nodes -> channel[CC_pair]] #]
```

We now give the definition of one of the transactions described in Section 2.2. The D_completion transaction (shown in $\boxed{9}$) obtains the completion for a R entry present in a node `nd`. The R entry is selected such that there is no older P entries and no other committed R entries for the same address in the channel (`choose_DR` predicate on line (a)). Depending on the address specified in the selected R entry, the final target as well as the successor of `nd` along the path to that target is determined (lines (b) and (c)). Now there are two cases: (i) the successor of `nd` is the final target or (ii) there is a completion ready for the R entry in the opposite channel of the successor.

In case (i) a new completion entry is created (line (d)) and inserted into `opposite(nd)` (line (f)). In case (ii) the local master ID of `nd` is used in determining if a completion is ready for its R entry (predicate `C_ready?` on line (j). See the definition at the beginning of $\boxed{9}$). Also, this completion is allowed to pass a P entry in case it is a Delayed Write Completion. This completion is deleted from the opposite channel of the successor (line (m)), the local master ID in it is replaced with the local master ID of the R entry (line (k)) and inserted into `opposite(nd)` (line (n)). In both cases, the R entry is removed from `nd` (lines (e) and (l)). When the original master obtains the completion of a R entry, that global delayed transaction is complete.

```
C_ready?(e:Entries,mid:nat,ch:channel[Entries]):bool =                       9
(trans(e)=DRR AND EXISTS (i:below[length(ch)]): (trans(nth(ch,i))=DRC
 AND addr(nth(ch,i))=addr(e) AND mid(nth(ch,i))= mid AND
 (FORALL (j:below[i]): NOT P_Entry?(nth(ch,j)))))
                  OR
(trans(e)=DWR AND EXISTS (i:below[length(ch)]): (trans(nth(ch,i))=DWC
 AND addr(nth(ch,i))=addr(e) AND mid(nth(ch,i))=mid))

D_completion(nd:Nodes,s:State) : State =
  IF NOT (is_agent?(nd) AND chan(nd) = TARGET) THEN
    IF DR_exists?(channel(s)(nd)) THEN
(a) LET i= choose_DR(channel(s)(nd)), e=nth(channel(s)(nd),i),
(b)     target = (# name:=addr_map(addr(e)), chan:=TARGET #),
(c)     succ = path_thru(net, nd, target) IN
     IF (succ = target) THEN
(d)  LET new_ce = complete(e) IN
      IF (is_agent?(nd) AND chan(nd) = MASTER) THEN
      LET cc_ent= (# entry:=e , flag:=Completed #),
          cc_pair= (# request:=cc_ent, completion:=new_ce #) IN
      s WITH [(channel):=channel(s) WITH
(e)            [(nd):=del(channel(s)(nd),i),
(f)             (opposite(nd)):=enq(channel(s)(opposite(nd)),new_ce),
(g)             (target):=enq(channel(s)(target),
                                 e WITH [(mid):= lmid(nd)])],
(h)           (trans_history):=trans_history(s) WITH
               [(nd):= enq(trans_history(s)(nd),cc_pair)]]
      ELSE s WITH [(channel):= "Exactly same changes as above"] ENDIF
(j)  ELSIF C_ready?(e,lmid(nd),channel(s)(opposite(succ))) THEN
      LET c_index = C_ready(e,lmid(nd),channel(s)(opposite(succ))),
          c_entry = nth(channel(s)(opposite(succ)),c_index),
(k)       new_ce = c_entry WITH [(mid) := mid(e)] IN
      IF (is_agent?(nd) AND chan(nd) = MASTER) THEN
      LET cc_ent= (# entry:=e , flag:=Completed #),
          cc_pair= (# request:=cc_ent, completion:=new_ce #) IN
      s WITH [(channel):=channel(s) WITH
(l)            [(nd):=del(channel(s)(nd),i),
(m)             (opposite(succ)):=del(channel(s)(opposite(succ)),
                                          c_index),
(n)             (opposite(nd)):=enq(channel(s)(opposite(nd)),new_ce)],
(o)           (trans_history):=trans_history(s) WITH
               [(nd):=enq(trans_history(s)(nd),cc_pair)]]
      ELSE s WITH [(channel):= "Exactly same changes as above"] ENDIF
     ELSE s ENDIF
    ELSE s ENDIF
  ELSE s ENDIF
```

As pointed out in Section 2, we record all the transaction entries that have reached an agent in its target channel. Accordingly, the D_completion event records the R entry in the target channel of the final destination (line (g)) and the C entry in the target channel of the original master (lines (f) and (n)). In addition, every agent records a history of all transactions committing and completing on its out-bus (`trans_history` field of the state). In the case of a delayed transaction completing, the completion obtained is recorded too (lines (h) and (o)). As explained in subsequent sections, these are used for specification purposes only.

5. Formalization of Producer/Consumer

Our formulation of Producer/Consumer relies on: (i) the absence of completion stealing, and (ii) the existence of a partial order which constrains read and write transactions to complete in the right order. Producer/Consumer requires that the "data" read by the consumer be the same as the "new data" written by the producer. While (ii) only guarantees that the transactions complete in the expected order, (i) is necessary to meet the "new data" requirement.

Definition: *A PCI based architecture provides the Producer/Consumer memory model if (i) there is no completion stealing and (ii) for every two agents $Prod$ and $Cons$ (which may or may not be identical), for every two addresses $Data$ and $Flag$ (which may or may not be identical): (1), (2), (3), (4) and (5)* **implies** *(6), where:*

1. *the agent $Prod$ issues two write transactions: W_{Data} (a PMW or DWR transaction to the address $Data$) followed by W_{Flag} (a PMW or DWR transaction to the address $Flag$),*

2. *the agent $Cons$ issues two Delayed Read Request transactions R_{Flag} followed by R_{Data},*

3. *W_{Flag} is committed on the originating bus after the completion of W_{Data} on the originating bus,*

4. *R_{Data} is committed on the originating bus after the completion of R_{Flag} on the originating bus,*

5. *R_{Flag} is completed on the destination bus after W_{Flag} completes on the destination bus, and*

6. *R_{Data} is completed on the destination bus after W_{Data} completes on the destination bus.*

It follows from (i) and (ii) that the value read by R_{Data} is the same value written by W_{Data} if there is no intervening write to $Data$ between W_{Data} and R_{Data}, even when multiple readers are allowed.

Notice that our definition has no other constraints on the environment in which transactions W_{Data}, W_{Flag}, R_{Flag} and R_{Data} are executed. In particular, both W_{Data} and W_{Flag} can be Delayed Writes too. Hence, this is a more general definition of Producer/Consumer.

To prove the above memory model for PCI in PVS, we specified two properties `No_steal_prop` and `Ordering_prop` encoding the conditions (i) and (ii) of the above definition respectively. `No_steal_prop` states that when a delayed transaction is completed by a master, the R entry and the C entry corresponding to the completion obtained should satisfy the `trans_match` predicate shown in 10. If a delayed request is serviced either with a completion meant for another master's transaction or with a completion meant for another transaction of the same master, this property is violated:

```
trans_match(e1:R_Entries,e2:C_Entries): bool =                          10
   ((trans(e1)=DRR and trans(e2)=DRC) OR
    (trans(e1)=DWR and trans(e2)=DWC)) AND
   addr(e1)=addr(e2) AND auxmid(e1)=auxmid(e2) AND tid(e1)=tid(e2)

No_steal_prop(s:State): bool =
  FORALL(nd:Agent_master,cc_pair:CC_pair):
    (member(cc_pair,trans_history(s)(nd)) AND
     R_Entry?(entry(request(cc_pair))) AND
     flag(request(cc_pair))=Completed) IMPLIES
    trans_match(entry(request(cc_pair)),completion(cc_pair))
```

The formal definition of `Ordering_prop`, shown in 11, mirrors part (ii) of the definition given earlier.

```
Ordering_prop(s: State): bool = FORALL(Prod:Agent_master,               11
         Cons:Agent_master,(WD,WF:W_Entries),(RF,RD:DRR_Entries)):
 LET Flag = (# name:= addr_map(addr(RF)), chan:= TARGET #),
     Data = (# name:= addr_map(addr(RD)), chan:= TARGET #)
 IN (addr(WD) = addr(RD)  AND addr(WF) = addr(RF) AND
      sameAgent(WD,WF,Prod) AND tid(WD) < tid(WF) AND
      sameAgent(RD,RF,Cons) AND tid(RF) < tid(RD) AND
(a)   (orig_commit_cmp?(WF,trans_history(s)(Prod))
       IMPLIES (orig_completed?(WD,trans_history(s)(Prod))
              AND In_ch_order(WD,WF,trans_history(s)(Prod)))) AND
(b)   (orig_commit_cmp?(RD,trans_history(s)(Cons))
       IMPLIES (orig_completed?(RF,trans_history(s)(Cons))
              AND In_ch_order(RF,RD,trans_history(s)(Cons)))) AND
(c)   (cmp_in(RF,channel(s)(Flag))
       IMPLIES (cmp_in(WF,channel(s)(Flag))
              AND In_ch_order(WF,RF,channel(s)(Flag)))))
    IMPLIES (cmp_in(RD,channel(s)(Data))
           IMPLIES (cmp_in(WD,channel(s)(Data))
                   AND In_ch_order(WD,RD,channel(s)(Data))))
```

The following predicates (defined in `more_defs` theory. See Figure 4(b).) are used in the specification of `Ordering_prop`. Two entries are issued from the same agent (`sameAgent` predicate) if their `auxmid` is the same as the global mid assigned to the agent. The predicate `orig_completed?` tests if a request is completed on the originating bus, `orig_committed?` tests if it is committed on the originating bus and `orig_commit_cmp?` tests if either of these is true. The predicate `cmp_in` tests if an entry is in the specified target channel (i.e., completed on the destination bus). Finally, the predicate `In_ch_ord` tells if entry `e1` is "before" entry `e2` in a given channel (according to the queue order.)

Note that the definitions in $\boxed{10}$ and $\boxed{11}$ use the two auxiliary fields `auxmid` and `tid` introduced by us as part of the definition of an entry, the "history" of the completed transactions maintained in the target channels of the agents and the "history" of the committed/completed transactions maintained at the agent nodes.

6. Proof of Producer/Consumer

We now sketch the proofs of `No_steal_prop` and `Ordering_prop` properties and also list some of the invariants and lemmas that were needed during the proofs.

6.1. Some of the Invariants Needed

A property is shown to be an invariant by proving that it is true in the initial state and is preserved by all transitions. The invariants listed here establish some basic properties about the PCI protocol (`pci_invariants` theory in Figure 4(b)) and were needed for the proofs of the main properties. All the invariants listed below have been established correct mechanically in PVS.

Invariant 1: For any P or R entry present in a channel, there exits a path in the network leading to the correct destination specified by the address of that entry.

Invariant 2: This invariant is concerned with the propagation of a delayed transaction in the network (depicted in Figure 6). The figure shows two paths: a path from the master channel `M` of an agent to the target channel `T` of another agent in which an R entry of a delayed transaction travels and an opposite path from `T'` to `M'` in which the C entry corresponding to the transaction travels back. The entries are shown as pairs—the type of the entry and the `lmid` field of the entry. The local master id assigned to the nodes shown in the figure is shown inside the boxes. Consider an R entry in a node `nd` as shown in the figure. Then along the path from the node issuing this transaction (M in the figure) to nd, there exist R entries of the same transaction (i.e., with the same addr, tid and auxmid), all of which are committed (shown underlined in the figure) and with their lmid field equal to the local master id assigned to the preceding node in the path. The R entry in nd may or may not be committed. Along the opposite path, there do not exist C entries (shown by an "X") corresponding to the R entries as shown in the figure.

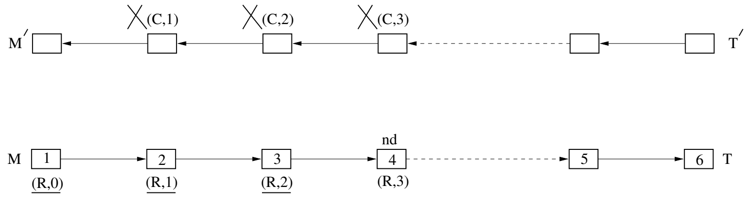

Figure 6. The figure showing how a delayed transaction propagates in the network.

Invariant 3: Consider a node with many R entries having the same address. Then only one of them may be committed. Also, if there exists a committed R entry in a node, then choose_DR (the function shown on line (a) in $\boxed{9}$, that is used to select an R entry for one of the delayed transaction transitions) always picks that entry.
Invariant 4: Assume that a C entry is present in a channel, say nd1 (refer to Figure 7). Then it is unique as determined by addr and lmid fields. Also, consider the channel that drives nd1′ and that has the same local master id as the lmid field of the C entry (nd2 in the figure; this is unique by well-formed condition 6). There exists a committed R entry of the same transaction (i.e., with the same addr, tid and auxmid) in that channel.

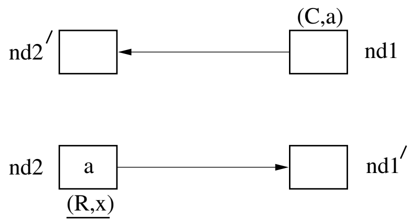

Figure 7. The figure illustrating the second part of *Invariant 4*.

Invariant 5: If a delayed transaction is completed on the originating bus, it is completed on the destination bus too.
Invariant 6: All entries in the master channel of an agent have the same auxmid as the global mid assigned to that agent. And the tid of all the entries issued by an agent is less than the next transaction id nextTid of that agent.

In addition to these invariants, we needed several lemmas specifying the correctness of the channel operations like addition and deletion of entries. The proofs of all these lemmas were straightforward.

6.2. Proof of No_steal_prop

The proof of this property essentially requires us to prove that R and C entries are "correctly" matched by a D_completion event. This fact is captured by the predicate `RC_match` shown in (12):

```
RC_match(s:State): bool = FORALL (nd:Nodes):                         12
   LET re = nth(channel(s)(nd),choose_DR(channel(s)(nd))),
       succ = path_thru(net,nd,
                  (# name := addr_map(addr(re)),chan := TARGET #)),
       ce = nth(channel(s)(opposite(succ)),
            C_ready(re,lmid(nd),channel(s)(opposite(succ)))) IN
   (NOT (is_agent?(nd) AND chan(nd) = TARGET) AND
    DR_exists?(channel(s)(nd)) AND
    NOT (succ = (# name := addr_map(addr(re)),chan := TARGET #)) AND
    C_ready?(re,lmid(nd),channel(s)(opposite(succ))))
       IMPLIES
   (auxmid(re)=auxmid(ce) AND tid(re)=tid(ce))
```

The D_completion event chooses an R entry `re` in a node `nd` and matches it against a C entry `ce` present in the opposite channel of its successor node `succ` (along the path to the final destination). The matched C entry `ce` contains the same `lmid` as the local mid assigned to `nd`. We are required to prove that R and C entries thus matched belong to the same transaction *i.e.,* have the same `auxmid` and `tid`. The proof of this theorem is based on the invariants mentioned earlier: from *Invariant 4*, we conclude that there exists an R entry `r` in `nd` belonging to the same transaction as `ce` and that `r` is committed. From *Invariant 3*, we conclude that `choose_DR` picks this `r` and hence is the same as `re`, and so `re` and `ce` belong to the same transaction. The proof of No_steal_prop is mechanically checked in PVS.

6.3. Proof of Ordering_prop

We sketch the proof of this property for the case when `WD` is a posted transaction(refer to 11). The proof for the case when `WD` is a Delayed Write Request is trivial since the premise (on line (a), which corresponds to clause 3 of the Producer/Consumer definition) ensures that `WF` is not even committed on the originating bus before `WD` completes on the originating bus, which guarantees that `WD` has already reached the target. (A delayed transaction completes on the destination bus before completing on the originating bus).

Assume RD has reached its destination Data. It must, then, have committed on the originating bus (at node *Cons*). From the condition on line (b) in $\boxed{11}$ (which corresponds to clause 4 of the Producer/Consumer definition), RF must have completed on the originating bus. Since RF is a delayed transaction, RF must have completed at its destination bus too and the corresponding completion CF must have reached the node that issued RF (i.e., *Cons*). From line (c) in $\boxed{11}$ (which corresponds to clause 5 in the definition of Producer/Consumer), it follows that WF has reached its destination *Flag*. In this scenario, we prove that WD must have reached its destination *Data* by a case analysis on the possible configurations (See Figure 5):

Case (a) Since WF has reached the *Flag*, WD entry must be ahead of t. This is because WF can not pass the posted transaction WD (because of the ordering rules of the PCI protocol) and hence must push it past t. Now RD (which is issued after CF has reached *Cons* which in turn implies that it is issued after WF has reached *Flag*) will push WD to its destination *Data*.

Case (b) This is the most complex case where all three entries—WF, CF, RD—"push" WD. Before WF reaches *Flag*, it would have pushed WD past t (hence into the segment $c \rightarrow Data$; refer to Figure 5(d)), before CF reaches *Cons*, it would have pushed WD past h (the segment $c \rightarrow d$ overlaps with the path CF takes to get to $Cons'$) and finally before RD reaches *Data*, it would have pushed WD to its destination.

Case (c) Similar to case (a). When WF reaches *Flag*, it would have pushed WD past t which means that WD is in the segment $h \rightarrow Data$ (since $t \rightarrow h$ is an edge). Now RD will push WD to its final destination.

Note that limit cases when *Prod* and *Cons* are the same nodes or when *Data* and *Flag* are the same nodes or both are special cases of case (a).

7. Model-Checking PCI

While No_steal_prop has been mechanically proven, we did not achieve a full mechanical proof of Ordering_prop despite three person-months of effort by experienced PVS users. Ordering_prop was difficult to prove mechanically using our model of PCI in PVS because the theory infrastructure required to reason about sequences of valid PCI states

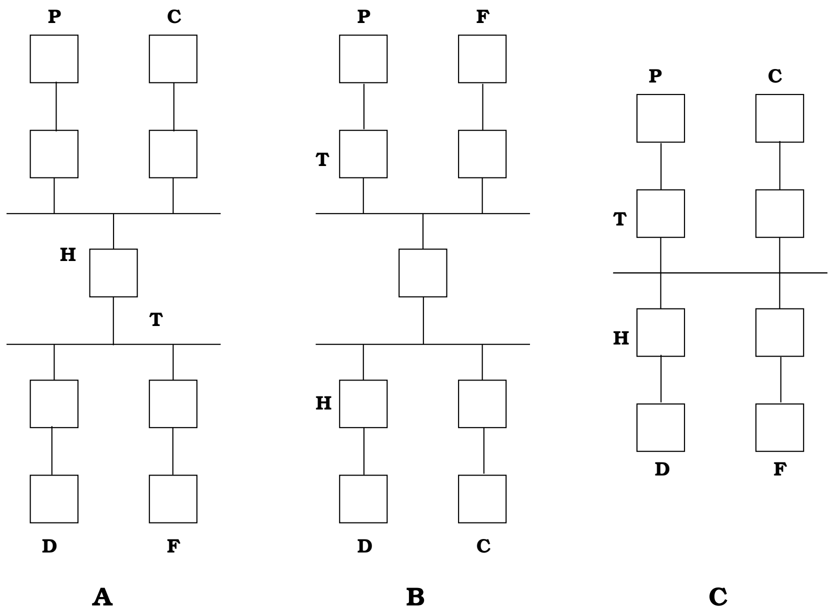

Figure 8. Producer/Consumer configurations used in model-checking

without repeatedly referring to graph or channel level details is large and does not yet exist in its entirety. While not conceptually difficult, building the required remaining infrastructure requires careful planning to identify useful reusable properties, and time to create the libraries of required theorems.

The goal of model-checking PCI was to leverage off the existing theorem proving results to verify (albeit incompletely) a significant portion of the Producer/Consumer property in a short amount of time. We also checked limited cases of `No_steal_prop` to validate our model. Due to capacity limitations and the finite model used by the model-checker, several abstractions were made during the verification of `No_steal_prop` and `Ordering_prop` in the model-checker. In this section, we discuss these abstractions, informally argue why they are reasonable and give a precise statements of what was, and was not, shown in our model-checking results.

7.1. Relation to Theorem Proving

The original statement of `Ordering_prop` is quantified over the infinite domains of PCI networks, states, and executions, making it unsuitable for model-checking directly. However, two key theorem proving results suggest a model-checking strategy. First, from Property 3 on Page 13, we know that only three configuration classes need to be considered in proving the Producer/Consumer property; these are repeated in Figure 8 in a fashion conducive to explaining our model-checking work better. Second, the proof outlined in Section 6.2 was completed, thus establishing the absence of completion stealing. Given these, our model-checking strategy has two steps:

1. Choose a single representative network from each of the three classes of network configurations to be considered in the proof of Producer/Consumer property.

2. For each representative network, reduce the cases of `Ordering_prop` checked until the task becomes tractable.

The first step reduces the infinite class of networks to be checked to the ones in Figure 8. In the figure, boxes represent bridges or agents and lines are buses. Producer, Consumer, Data and Flag agents are labeled P,C,D and F. The hit and takeoff points, labeled H and T, are also shown. The hit point is where messages from the Producer and the Consumer first hit en route to the Data agent. The take-off point is where the messages from the Producer to the Flag and Data agents separate.

The second step addresses the capacity issues. Initial experiments with the model-checker demonstrated that checking even a single network for `Ordering_prop` was beyond the capacity of available model-checkers. A series of reductions to the states and execution sequences considered were made to bring each case within the capacity of the model-checker.

After reducing the cases of `Ordering_prop` checked, we are left with incomplete coverage of `Ordering_prop` for the complete PCI model. The reductions were designed to carefully balance capacity limitations, while allowing reasonable informal justification given the theorem proving results.

7.2. Abstractions

The property reductions can be precisely characterized by the following statements. The first statement characterizes which transactions were ignored.

Reduction 1. Ignore all transactions unrelated to the four transactions mentioned in the definition of the Producer/Consumer property.

For example, the read transaction from the Observer agent in the completion stealing example from Figure 3 is ignored. The second statement describes which networks were considered.

Reduction 2. Include only configurations that include the four agents (Producer, Consumer, Flag and Data) mentioned in definition of the Producer/Consumer property.

Returning again to the completion stealing example, Reduction 2 removes the Observer agent from the network.

Reduction 1 ignores potentially damaging affects of incidental transactions to or from the Data,Flag,Producer or Consumer agents. Incidental transactions are transactions other than the four transactions (Data and Flag write, Data and Flag read) mentioned in the Producer/Consumer property. Reduction 2 ignores the potentially damaging affects of other agents in the network. However, the reductions are not as severe as they might seem. We next present a series of assumptions, each followed with a brief informal argument, that mitigate the coverage lost due to Reductions 1 and 2.

The first set of assumptions address the effects of ignoring incidental transactions. Incidental transactions can affect the presence of deadlock, completion of delayed messages, the observed values of Flag and Data, and the ordering of messages in bridges.

Assumption 1. Incidental transactions never cause deadlock.

In this work, we have not pursued the problem of proving the absence of deadlocks. While this issue has been treated in [5], we list it as an assumption because we have not formally related our model to their model.

Assumption 2. Incidental transactions do not affect the correct completion of delayed transactions.

The proof of `No_steal_prop` already demonstrates that completion stealing never happens in any PCI network and execution sequence. Because delayed transactions always complete correctly, we are not concerned with the completion affects of incidental delayed transactions. Once again we list this assumption because we have not formally related the model-checking model to the model used in PVS.

Assumption 3. Incidental transactions never affect the value of Data and Flag.

There are two classes of transactions to consider: those the are addressed to Data or Flag and those that are not. Transactions not addressed to Data and Flag do not affect their observed values. Incidental transactions to Data or Flag are ignored in our definition of Producer/Consumer.

Assumption 4. Incidental transactions either reduce, or do not change, the number of transactions that can pass each other.

If the transactions from the Producer or the Consumer pass incorrectly, Producer/Consumer property could be violated. Assumption 4 claims that incidental transactions never increase possibility for message passing. Two transactions in the same queue can pass only if they are related by the transitive closure of the ordering rules shown in Figure 2. More specifically, adding transaction M_1 between transactions M_0 and M_2 still allows M_0 to pass M_2 only if M_0 can pass M_2 without M_1. For example, adding an R transaction between an R and and a P transaction does not allow the R transaction to pass the P transaction. Also, if M_0 can pass M_2 then adding M_1 might disallow M_0 passing M_2. For example, adding a P transaction between adjacent R transactions disallows the R transactions passing each other.

The final assumption addresses the remaining affect of ignoring incidental agents. Incidental agents generate incidental transactions which we have already chosen to ignore. Incidental agents also make paths longer.

Assumption 5. If a transaction completes across a path of length 1 then it will complete across a path of length n

If the path includes no branches, then the Conjecture 5 is trivial. If the path does include branches then Assumption 5 holds if the routing is done correctly. By ignoring path lengths, we ignore routing issues. Since PCI uses a static routing scheme, it is an uninteresting problem.

7.3. Results

Using a model of PCI with the coverage reductions described above, we checked four initial states for three networks for the Producer/Consumer property and 1728 cases of the no stealing property.

For the Producer/Consumer property, there were four initial states for each of the three configurations in Figure 8, one initial configuration for each pair of delayed or posted flag or data writes. Clauses 1, 2, 3, and 4 of the definition of Producer/Consumer on Page 17 were forced to be satisfied by the initial configuration of each model used in each case. The twelve models were generated and managed using a Perl script. Clause 5 was stated as an antecedent to the assertion of Clause 6 in each model. Clause 6 was checked by asserting

$$flag\ write\ before\ flag\ read \implies complete\ data\ write$$

when the data read completed on the destination bus, as shown in Promela code fragment below:

```
                                                                  13
     [code for an agent]
     if
       [tests and code for other events]

       /* D_completion with agent */
(a)    ::(route_to_end &&
(b)      (inq_hd.committed) &&
(c)      (! opp_in_full))
       ->
(d)      assert (!((inq_hd.mid == consumer) &&
(e)                (inq_hd.adr == data) &&
(f)                fw_before_fr
                  ) ||
(g)               complete_dw));
         [remaining code to execute a D_completion with an agent]
     fi
```

Comments included in brackets describe code which has been omitted for brevity. The code in this fragment checks clause six when the data read was completed by a D_completion event, a similar assertion checks clause 6 for data reads completed by P_completion events. The guards in lines (a),(b) and (c) check that the conditions for a D_completion with an agent hold. These conditions are: (a) the oldest C entry in the bridge routes to the agent, (b) the oldest R entry at the agent is committed and (c) the agent input queue is not full. The assertion in lines (d) through (g) check clause 6. Lines (d) and (e) check that the current D_completion is indeed the completion of the data read at the consumer. Clause 5 is checked in line (f) by the value of the variable `fw_before_fr`. The value of `fw_before_fr` is set when the Flag is read by the Consumer. If the Flag has not been written by the Producer when the Consumer reads the Flag, then `fw_before_fr` is set to false otherwise `fw_before_fr` is set to true. Line (g) checks that if clause 5 was satisfied and the Data read is complete at the Consumer agent, then Data write by the Producer is also complete. If the Data write is not complete when the Data read completes, then Producer/Consumer property was violated and the assertion fails. Otherwise, the Producer/Consumer property was satisfied and the assertion passes.

The execution times and memory requirements for checking each of the twelve cases are summarized in Table II. Cases are identified by the configuration class followed by the transaction type for the Data and Flag writes. For example, A(P,D) represents a posted write to the Data and a delayed write to the Flag in configuration A (see Figure 8). The time and space requirements were collected on a SUN Ultra-Sparc with 512MB of memory using SPIN v.3.1.4 with the -DCOLLAPSE

Table I. Model-checking time and space requirements.

Case	Time (minutes)	Space (MB)
A (P,P)	3:32.0	2.7 (124.1)
A (P,D)	23:10.7	12.1 (707.7)
A (D,P)	2:17.2	2.3 (88.6)
A (D,D)	102:38.8	47.2 (2995.2)
B (P,P)	0:07.3	0.4 (4.5)
B (P,D)	2:13.5	2.0 (70.0)
B (D,P)	159:19:0	69.8 (4553.1)
B (D,D)	52:42.1	25.7 (1653.7)
C (P,P)	0:07.9	0.4 (5.9)
C (P,D)	3:04.4	2.3 (116.2)
C (D,P)	66:35.2	36.3 (2142.1)
C (D,D)	15:19.7	10.5 (533.8)

option to collapse stored state vectors. The reported execution time includes only model-checking and does not include compilation time which is approximately one additional minute. The reported memory usage includes only memory used to store state vectors and does not include the depth-first search stack which is an additional 30 MB in each case. Estimated memory required to store the uncompressed state vectors is shown in parenthesis following the actual memory used.

Checking all execution sequences required twelve hours of CPU time and found no errors for the PCI model extended with local master IDs. Removing local master IDs caused the property to fail in under one minute. All these experiments could be performed by a novice user of SPIN in less than 2 weeks, thus under-scoring the importance of deploying model-checking once the ground-rules for selecting the finite cases have been established, as we did using PVS.

8. Conclusions

Using PVS we have formalized that the PCI 2.1 protocol extended by local master ID provides the Producer/Consumer memory model. While the `Ordering_prop` conjunct of the Producer/Consumer property is not yet (completely) formally established at the time of writing, we believe that we are well on our way. In fact, we plan to approach the proof bottom-up, capitalizing on the paths identified in Figure 8 to simplify each case. Our model-checking based verification of the Producer/Consumer property was, in comparison, achieved in much shorter a span of time, and may well be the path the industry adopts.

At present, informally stated assumptions used to facilitate model-checking must be listed by hand and built into the model manually. We plan to address this weakness by investigating the combination of theorem proving and model checking in a way that allows these assumptions to be formally included into both model checking and theorem proving. Traditional verification tools such as simulation and emulation may also play a major role in validating these assumptions. Documenting the assumptions formally will also allow the assumptions to be checked during later evolutions of the design.

Our PVS proofs are generic in the sense that they cover any acyclic network of a finite number of PCI buses. We have shown that theorem proving technology is mature enough today to be able to tackle such a challenging problem. The proofs make extensive use of induction and several fundamental theorems from the acyclic directed graph theory we have proved in PVS. Although the expertise we accumulated in both PVS and formal verification in the last few years [7, 11, 6] was crucial to realize this work in a reasonable time (7 person months so far) we believe that significant progress has been made in the theorem proving technology contributing to practical use of tools like PVS in the formal verification of commercial protocols. However, this progress should continue from different perspectives in order to make such tool accessible by a larger community. We believe that a graphical proof editor providing suitable functionalities for easy construction of proofs would be very helpful. These functionalities include: (i) the ability to zoom in on the relevant part of the proof tree, (ii) the ability to reuse the proof of a subgoal occurring in the proof tree on other subgoals that need the same proof. This would make the task of managing large proofs much easier. In addition, the proof manager in PVS should be smarter in naming the formulas. The current naming scheme is a serious impediment to automatic proof reuse, especially when proofs are very long. A local update of a proved property[1] should not affect the global proof of the original property: one should be able to easily reuse the proof corresponding to the un-modified part.

[1] For example it is very common to add a conjunct to the assumptions or the conclusions of a sequent.

References

1. Butler, R. W. and J. A. Sjogren: 1997, 'A PVS Graph Theory Library'. Technical Report Memorandum, NASA Langly Research Center. http://atb-www.larc.nasa.gov/ftp/larc/PVS-library.
2. Clarke, E. M., O. Grumberg, H. Haraishi, S. Jha, D. Long, K. L. McMillan, and L. Ness: 1992, 'Verification of the Futurebus+ Cache Coherence Protocol'. Technical Report CMU-CS-92-206, School of Computer Science, Carnegie Mellon University.
3. Corella, F.: 1996, 'Proposal to fix ordering problem in PCI 2.1'. http://www.pcisig.com/reflector/thrd8.html#00706.
4. Corella, F.: 1997, 'Verifying Memory Ordering Model of I/O Systems'. In: *Computer Hardware Description Languages, CHDL '97*. Toledo, Spain. Invited Talk.
5. Corella, F., R. Shaw, and C. Zhang: 1997, 'A formal proof of absence of deadlock for any acyclic network of PCI buses'. In: *Computer Hardware Description Languages, CHDL '97*. Toledo, Spain.
6. Ghughal, R., A. Mokkedem, R. Nalumasu, and G. Gopalakrishnan: 1998, 'Using "test model-checking" to verify the Runway-PA8000 memory model'. In: *Tenth ACM Symposium on Parallel Algorithms and Architectures*. Puerto Vallarta, Mexico, pp. 231–239.
7. Gopalakrishnan, G., R. Ghughal, R. Hosabettu, A. Mokkedem, and R. Nalumasu: 1997, 'Formal Modeling and Validation Applied to a Commercial Coherent Bus: A Case Study'. In: H. F. Li and D. K. Probst (eds.): *Correct Hardware Design and Verification Method, CHARME '97*. Montreal, Canada, pp. 48–62.
8. Holzmann, G.: 1991, *Design and Validation of Computer Protocols*. Prentice Hall.
9. Mokkedem, A., R. M. Hosabettu, M. D. Jones, and G. Gopalakrishnan: 1999, 'Formalization and Analysis of a Solution to the PCI 2.1 Bus Transaction Ordering Problem: PVS Files'. Technical Report UUCS-99-007.
10. Nagasamy, V., S. Rajan, and P. R. Panda: 1997, 'Fiber channel protocol: Formal specification and verification'. In: *Sixth Annual Silicon Valley Networking Conference*.
11. Nalumasu, R., R. Ghughal, A. Mokkedem, and G. Gopalakrishnan: 1998, 'The 'Test Model-Checking' Approach to the Verification of Formal Memory Models of Multiprocessors'. In: *Lecture Notes in Computer Science*, Vol. 1427 of *Lecture Notes in Computer Science*. Vancouver, BC, Canada, pp. 464–476.
12. Owre, S., J. Rushby, N. Shankar, and F. von Henke: 1995, 'Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS'. *IEEE Transactions on Software Engineering* **21**(2), 107–125.
13. Park, S. and D. L. Dill: 1996, 'Protocol Verification by Aggregation of Distributed Actions'. In: R. Alur and T. A. Henzinger (eds.): *Computer-Aided Verification, CAV '96*, Vol. 1102 of *Lecture Notes in Computer Science*. New Brunswick, NJ, pp. 300–310.
14. PCISIG: 1995, 'PCI Special Interest Group–PCI Local Bus Specification, Revision 2.1'.
15. Solari, E. and G. Willse: 1996, *PCI Hardwarde and Software Architecture & Design*. Annabooks, 3rd edition edition. ISBN 0-929392-32-9.
16. VSI Alliance, 'Interface Standards for Design Re-use of Virtual Components'. http://www.vsi.org/.