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

Abdel Mokkedem, Ravi Hosabettu, and Ganesh Gopalakrishnan

Department of Computer Science, University of Utah,
Salt Lake City, UT 84112-9205
mokkedem,hosabett,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. In [3], a correction to this ordering problem was proposed and informally proved (called the "HP solution" here). In this paper, we present a formalization of the PCI 2.1 protocol in PVS. We formalize the fact that with Local Master ID added to the protocol no completion stealing is possible and the Producer/Consumer property is provided even in the presence of multiple readers. The state of our proofs leading to this result, as well as some of the much needed enhancements to theorem-proving frameworks that will greatly facilitate similar proofs, are also elaborated.

1 Introduction

Modern system I/O busses such as the PCI are defined with the primary goal of establishing industry-wide open standards. Given the considerable complexity of these standards, only when developed with sufficient rigor can such standards allow peripheral VLSI chip manufacturers to *independently* develop *compatible* I/O devices and bus bridges. Current industrial practices are, however, such that rigor is not reflected either in the development or the documentation of these standards. As a result, open standards often contain significant inconsistencies, resulting in costly work-arounds and interfacing problems. 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 the precise statement and proof of crucial correctness properties, and forming a solid framework for reliable future extensions to standards.

In this paper, we demonstrate how a modern theorem prover, namely PVS [11], 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

* Supported in part by DARPA under contract #DABT6396C0094 (Utah Verifier), and NSF MIP MIP-9321836

proved, in [3, 12]. Briefly, there are two problems with the PCI 2.1 standard: *deadlock*, arising from inadvertently disallowing certain transaction re-orderings; and *incorrect behavior*, arising from the so called "completion stealing". More specifically, [4, 3] show that the PCI 2.1 standard allows implementations that deadlock, by not allowing certain transactions to bypass others in bus channels (as will be elaborated later). They also point out that the PCI 2.1 standard allows a crucial correctness property—namely Producer/Consumer—to be violated. Briefly, 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. [4, 3] 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 (as will be elaborated later). The solution proposed in [4, 3] 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 [3, 12], 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 *all the graph-configurations* to be considered in proving Producer/Consumer. We formulate and prove many interesting properties about the PCI protocol which are subsequently used in the proof of the absence of completion stealing and the proof of Producer/Consumer. Most of the mechanical proofs leading upto these results have been finished, with the remaining effort estimated to be one person month. We also report on some of the much needed enhancements to theorem-proving systems without which proofs of this nature will be too tedious to finish in reasonable times—as has been our experience.

Related Work: Modeling modern coherent memory busses [6], 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 [9], the examination of entire I/O bus standards in the context of modern theorem provers has, to the best of our knowledge, 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 believed to be 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

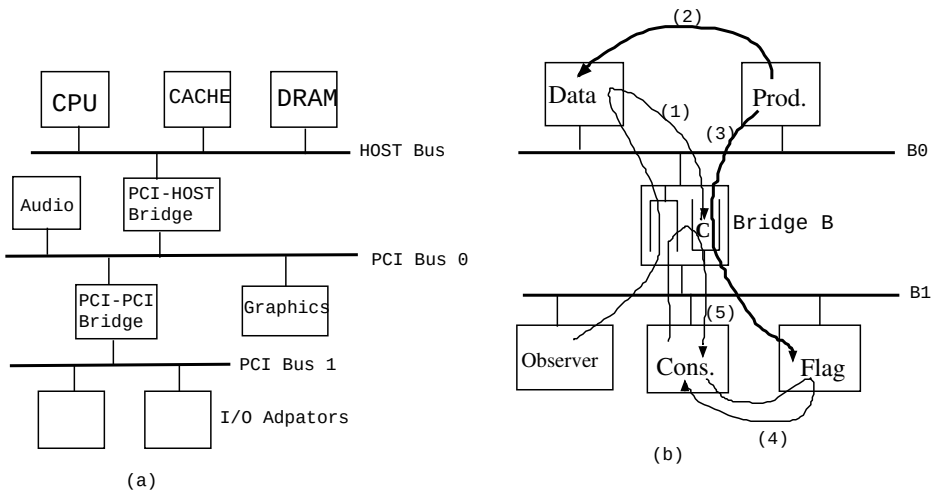

Fig. 1. (a) A PCI acyclic network and (b) Producer-Consumer violation.

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 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 (a) shows a typical 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' has two *opposite 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 opposite channels, called the *master channel* and the *target channel*. The hypergraph defined by the PCI network topology, which has bridges and agents as vertices, gives rise to a directed graph that has the *channels of bridges* and *channels of agents* 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.

The acyclicity of the hypergraph defined by the PCI network topology implies that the channel graph is a *directed acyclic graph*. Moreover, the fact that the hypergraph is connected implies 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 [4]. We reproduce some material from this summary here; see [4, 7, 14] for more details. Revision 2.1 (one of the latest, if not *the* latest revision) of the PCI protocol [7] became the official production version on June 1, 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 uniquely 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 uniquely 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 (in which case the subtransaction propagates to the next level of bus interconnect) or request the local master to retry. In this case, the local master must reissue the posted subtransaction until it is completed. The local target then latches

the transaction information, creating a Posted Memory Write (PWM, or "P entry"). For a *delayed* subtransaction, the local target may either *ignore* it or latch it creating a *DRR entry* (in case of a Delayed Read Request) or a *DWR entry* (in the case of a Delayed Write Request). In the latter case, an "R entry" (referring both to DRR and DWR) that specifies the address of the transaction (and in the case of a write, also the data) is created. The local master of a delayed transaction must reissue the transaction until it is completed. *Meanwhile*, the local target 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 (normal or abnormal) (in case of DWC). 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 reordering in order to avoid deadlocks. For instance, not allowing C entries to pass R enties can cause deadlocks [4]. Table 1 specifies these orderings (this incorporates the fix proposed in [4] and is different from the PCI standard specification). In [4] 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 Col.?	P	R	C
P	No	Yes	Yes
R	No	Y/N	Y/N
DRC	No	Yes	Y/N
DWC	Y/N	Yes	Y/N

Yes: Must be allowed to pass.
No: Can not be allowed pass.
Y/N: The bridge designer may choose either way.

Table 1. Ordering rules in a Bridge.

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 is allowed to discard an R entry *only* from the time it is enqueued till it has been attempted on the destination bus. This is because, as said earlier, 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 Table 1. 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* [4]:

- 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' is the target channel specified by the address α of E, there exists a path $N = N_0, N_1, \ldots, N_n = N'$ from N to N' 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 $N_1 = N'$, the corresponding global posted transaction completes and the entry E' is created in the target channel of the addressed agent (i.e. when N'). In fact, we use the target channel of an agent to store all the transactions completed in that agent for specification purposes (in other words, a "history" of all transactions completed is kept to assist in the specification and proof).

- 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. (This is to abide by the orderings in Table 1.) 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 and the event is said to be an *R_commit event* that commits the entry E. If N' is the target channel specified by the address α, there exists a path $N = N_0, N_1, \ldots, N_n = N'$ from N to N' in the channel graph. 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 $N_1 \neq N'$, 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) $N_1 = N'$, or (ii) $N_1 \neq N'$, 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 the case of DRC only). 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.

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. The identity of the master that issued the transaction cannot be used 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 [3, 12]. To recap, 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 1 (b) describes one possible scenario which violates this rule. A third master called the Observer reads Data and the completion C containing the Data value meant for this read is waiting in the bridge B. A Producer, then, writes a new value to Data followed by writing to Flag and a Consumer then reads the new Flag and attempts to read Data. Since the Master ID information is not maintained, the bridge returns the completion C meant for the Observer. Since C contains the old value of Data, Producer/Consumer is violated. Many variants of this scenario are described in `pci-sig-request@znyx.com` [12], 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 under-estimated there.

HP proposal: The solution proposed by HP [12] 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 a bit too involved, but basically allowing mutiple outstanding transactions again invites failure due to completion-stealing—essentially, R entries bearing local master-IDs can get "aliased" after one hop through a common bridge.

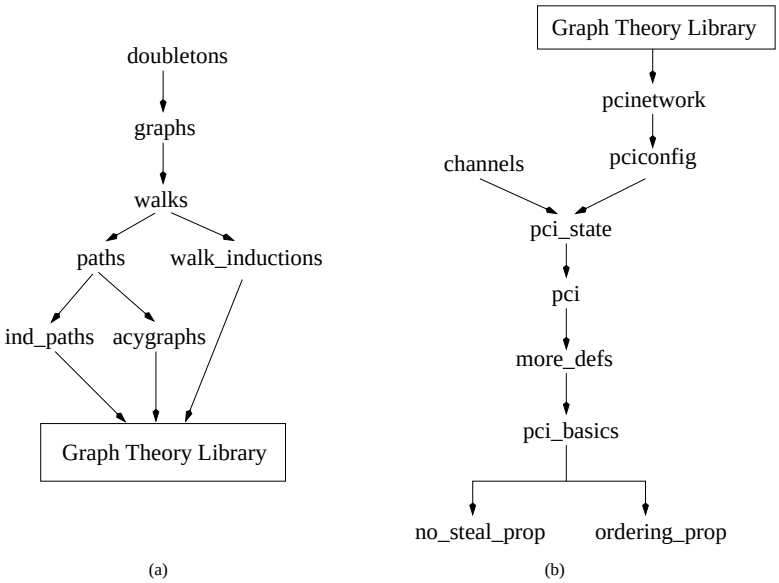

Fig. 2. The theory hierarchy

Our PVS model of PCI protocol includes these two components. Our contribution is towards formally proving that (i) and (ii) together provide 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 [8].

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 2(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.

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`.

```
pcinetwork: THEORY                                                    1
BEGIN
   Agent_count, Bridge_count: posnat % The number of agents and bridges
   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 #)
```

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:

```
  all_opposite_exists(net:acynet):bool = FORALL(nd1,nd2:Nodes):       2
     edge?(net)(nd1,nd2) IMPLIES edge?(net)(opposite(nd2),opposite(nd1))
```

2) For all agents, the master channels are only sources and target channels are only destinations:

```
  agents_edges_ok(net:acynet):bool = FORALL(nd1:Nodes):               3
     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:

```
  master_target_connected(net:acynet):bool = FORALL(nd1,nd2:Nodes):   4
     (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 The 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:

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.

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

3. The third property captures all the possible configurations of the four nodes corresponding to the Producer *Prod*, the Consumer *Cons*, Data *Data* and Flag *Flag*. The two *trivial* cases are when *Prod* and *Cons* are the same and when *Data* and *Flag* are the same. All non-trivial configurations showing the paths *Prod* $\rightarrow$ *Data*, *Cons* $\rightarrow$ *Data*, *Prod* $\rightarrow$ *Flag* and *Cons* $\rightarrow$ *Flag*—all of which are guaranteed to exist by the third well-formedness condition—are shown in Figure 3. In this figure, there are two significant points called the *hit* (`h`) and the *take-off* (`t`) points. The hit point is where the path from *Cons* to *Data* hits the path from *Prod* to *Data*, while the take-off point is where the path from *Prod* to *Flag* leaves the path from *Prod* to *Data*. It is proved that the hit point exists in all cases, the segments *Prod* $\rightarrow$ `h` and *Cons* $\rightarrow$ `h` are disjoint and the segment `h` $\rightarrow$ *Data* is common to both the paths *Prod* $\rightarrow$ *Data* and *Cons* $\rightarrow$ *Data*. (A similar remark applies for the takeoff point too). Figure 3(a) illustrates the case when `t` is ahead of or same as `h`. Figure 3(b) illustrates the case when `t` is before `h` by at least two edges. (The two dotted line segments show the paths in the opposite direction corresponding to the two paths shown by the solid line segments. See the first property discussed earlier.) In this case, the path from *Flag* to *Cons* is made up of the segment *Flag* $\rightarrow$ `t`, the segment `t` $\rightarrow$ `h`, and the segment `h` $\rightarrow$ *Cons*. Figure 3(c) illustrates the case when `t` is before `h`, and is directly connected (by an edge). In this case, there exist points `1` and `2` connected by an edge and the path from *Cons* to *Flag* is made up of the segment *Cons* $\rightarrow$ `1`, the edge `1` to `2` and the segment `2` $\rightarrow$ *Flag*. We have mechanically proved that these cases *completely cover* all possible configurations of *Prod*, *Cons*, *Data* and *Flag* (in the `pciconfig` theory).

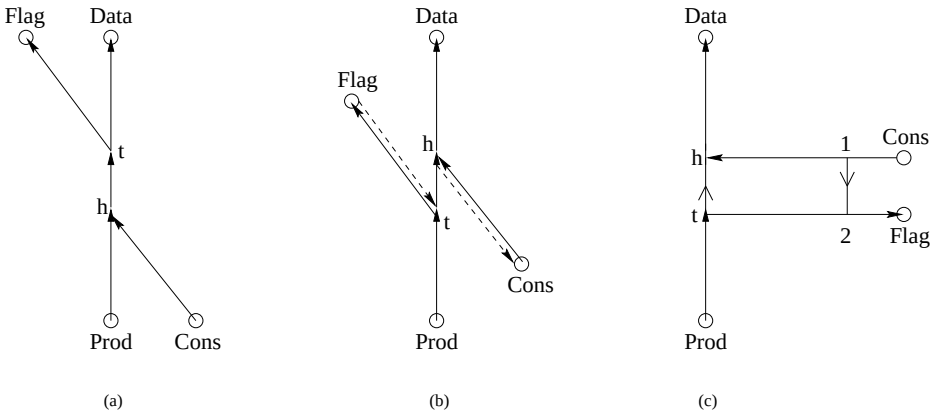

Fig. 3. The possible configurations of *Prod*, *Cons*, *Data* and *Flag*.

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` parametrized by the type of all PCI entries (`Entries`, shown in 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 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).

```
Entries : TYPE = [# trans : Trans, addr: Addr, data: Data, lmid: nat,   8
                    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.

```
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),
        target = (# name:=addr_map(addr(e)), chan:=TARGET #),
        succ = path_thru(net, nd, target) IN
      IF (succ = target) THEN
      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
                  [(nd):=del(channel(s)(nd),i),
                   (opposite(nd)):=enq(channel(s)(opposite(nd)),new_ce),
                   (target):=enq(channel(s)(target),
                                 e WITH [(mid):= lmid(nd)])],
                (trans_history):=trans_history(s) WITH
                   [(nd):= enq(trans_history(s)(nd),cc_pair)]]
        ELSE s WITH [(channel):= "Exactly same changes as above" ENDIF
(b)   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),
          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
                  [(nd):=del(channel(s)(nd),i),
                (opposite(succ)):=del(channel(s)(opposite(succ)),c_index),
                   (opposite(nd)):=enq(channel(s)(opposite(nd)),new_ce)],
                 (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
```

The D_completion transaction (shown in 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. Now there are two cases: (i) the successor of `nd` is the final target in which case a new completion entry is created or (ii) there is a completion ready for the R entry in the opposite channel of the successor. 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 (b)). Also, this completion is allowed to pass a P entry in case it is a Delayed Write Completion. In both cases, the R entry is removed from `nd` and the completion obtained is put in `opposite(nd)`. When the original master obtains the completion of a R entry, that global delayed transaction is complete.

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 and the C entry in the target channel of the original master. 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. 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 right order, (i) is necessary to meet the "new data" requirement.

Definition 1. *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 is quite simple and intuitive to conclude 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 1 given earlier.

```
Ordering_prop(s: State): bool = FORALL(Prod:Agent_master,                11
            Cons:Agent_target,(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
    (orig_commit_cmp?(WF,trans_history(s)(Prod))                          (i)
      IMPLIES (orig_completed?(WD,trans_history(s)(Prod))
               AND In_ch_order(WD,WF,trans_history(s)(Prod)))) AND
    (orig_commit_cmp?(RD,trans_history(s)(Cons))                          (ii)
      IMPLIES (orig_completed?(RF,trans_history(s)(Cons))
               AND In_ch_order(RF,RD,trans_history(s)(Cons)))) AND
    (cmp_in(RF,channel(s)(Flag))                                          (iii)
      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 2(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. Finally, the predicate `In_ch_ord` tells if entry `e1` is "before" entry `e2` in a given channel.

Note that the definitions in 10 and 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

In this section, we sketch the proofs of `No_steal_prop` and `Ordering_prop` properties and also list some of the auxiliary invariants and lemmas that were needed during the proofs. 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 proofs mentioned in this section are in progress.

6.1 Auxiliary Invariants

The auxiliary invariants listed here establish some basic properties about the PCI protocol (`pci_basics` theory in Figure 2(b)) and were needed for the proofs of the main properties. These include:

- All entries in the master channel of an agent have the same `auxmid` as the global mid assigned to that agent.
- The next transaction id of an agent is always greater than the id of all the transactions issued by it earlier.
- For any entry present in a channel, there exits a path in the network leading to the correct destination specified by the address of that entry.
- A transaction cannot be completed without being committed.
- If a C entry is present in a channel, then it is unique (as determined by `addr` and `lmid` fields) and the corresponding R entry is already removed from its opposite channel.
- At any node, only one R entry for a particular address may be committed.
- If a delayed transaction is completed on the originating bus, it is completed on the destination bus too.

Some of the other invariants needed will be discussed in later subsections. In addition to these invariants, we needed several lemmas specifying the correctness of the channel operations like addition/deletion of entries.

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 (nd1:Nodes):                          12
   LET re = nth(channel(s)(nd1), choose_DR(channel(s)(nd1))),
       succ = path_thru(net, nd1,
                        (# name := addr_map(addr(re)),chan := TARGET #)),
       ce = nth(channel(s)(opposite(succ)),
                C_ready(re, lmid(nd1), channel(s)(opposite(succ)))) IN
   DR_exists?(channel(s)(nd1)) AND
   NOT (succ = (# name := addr_map(addr(re)),chan := TARGET #)) AND
   C_ready?(re, lmid(nd1), channel(s)(opposite(succ)))
   IMPLIES (auxmid(re)=auxmid(ce) AND tid(re)=tid(ce) AND committed(re))
```

The D_completion event chooses an R entry `re` in a node `nd1` 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 `nd1`. 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`. This relies on the fact that only one delayed transaction for a particular address is committed in any node and that all nodes driving the same PCI bus have different local mids (well-formedness condition `lmid_ok`).

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 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 before `WF` is committed. (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 completed on the originating bus (at node `Cons`). From the invariants proved earlier and the condition on line (ii) in 11, `RF` must have completed at its destination bus and the corresponding completion `CF` must have reached the node that issued `RF` (*i.e.,* `Cons`). From (iii) in 11, 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 3):

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` 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`. These "pushing" arguments are formulated and proved as invariants and rely crucially on the the fact that there is a unique path between any two nodes in the channel graph, if any and the ordering rules of the PCI protocol.

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`, before `CF` reaches `Cons`, it would have pushed `WD` past `h` 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 (d) `Prod` and `Cons` are the same nodes and (e) `Data` and `Flag` are the same nodes are special cases of case (a).

7 Conclusions

Using PVS we have formalized that the PCI 2.1 protocol extended by local master ID provides the Producer/Consumer memory model. Due to technical difficulties, we have not established the Producer/Consumer relationship at the time of writing, but believe that we are well on our way. Our 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 [5, 6, 10] was crucial to realize this work in a reasonable time (5 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 user-interface providing suitable functionalities including basic graphical operations such that zoom, shrink, expand, etc. of nodes in the graphical representation of the proof would be very helpful. Also, it is unacceptable that the user has to cut-and-paste a thousand-line proof and apply it step by step to a similar subgoal in an adjacent node in the proof tree. A mouse-drag from the proved node to an unproved node that is similar should be enough. Such situations arose many times in our work. 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.

References

1. Ricky W. Butler and Jon A. Sjogren. A PVS Graph Theory Library. Technical Report Memorandum, NASA Langly Research Center, December 1997. http://atb-www.larc.nasa.gov/ftp/larc/PVS-library.

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

2. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In L. Claesen, editor, *Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.
3. Francisco Corella. Verifying memory ordering model of I/O systems. In *Invited talk at Computer Hardware Description Languages 1997, Toledo*, Spain, April 1997.
4. Francisco Corella, Robert Shaw, and Cui Zhang. A formal proof of absence of deadlock for any acyclic network of PCI buses. In *Computer Hardware Description Languages*, 1997.
5. Rajnish Ghughal, Abdel Mokkedem, Ratan Nalumasu, and Ganesh Gopalakrishnan. Using "test model-checking" to verify the runway-pa8000 memory model. In *Tenth Annual ACM Symposium On Parallel Algorithms And Architectures*, pages 231–239, Puerto Vallarta, Mexico, June 1998. ACM Press.
6. G. Gopalakrishnan, R. Ghughal, R. Hosabettu, A. Mokkedem, and R. Nalumasu. Formal modeling and validation applied to a commercial coherent bus: A case study. In Hon F. Li and David K. Probst, editors, *CHARME*, Montreal, Canada, 1997.
7. PCI Special Interest Group. PCI Local Bus Specification, Revision 2.1, June 1995.
8. A. Mokkedem. Verification of PCI 2.1 Local Bus in PVS. http://www.cs.utah.edu/-~mokkedem/pvs/pvs.html.
9. Vijay Nagasamy, Sreeranga Rajan, and Preeti R. Panda. Fiber channel protocol: Formal specification and verification. In *Sixth Annual Silicon Valley Networking Conference*, 1997.
10. Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkdem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 464–476, Vancouver, BC, Canada, June 1998. Springer-Verlag.
11. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
12. F. Corella. Hewlett Packard. Proposal to fix ordering problem in PCI 2.1, 1996. http://www.pcisig.com/reflector/thrd8.html#00706.
13. S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 300–309, New Brunswick, NJ, USA, July 1996. Springer-Verlag.
14. Edward Solari and George Willse. *PCI Hardwarde and Software Architecture & Design*. Annabooks, 3rd edition edition, December 1996. ISBN 0-929392-32-9.