

# Incremental Proof of the Producer/Consumer Property for the PCI Protocol

Dominique Cansell<sup>1</sup>, Ganesh Gopalakrishnan<sup>2</sup>, Mike Jones<sup>3</sup>, Dominique Méry<sup>1</sup>, and  
Airy Weinzoepflen<sup>1</sup> \*

<sup>1</sup> LORIA, Nancy France{Dominique.Cansell, Dominique.Mery,  
Airy.Weinzoepflen}@loria.fr

<sup>2</sup> University of Utah, Salt Lake City, Utah, USAganesh@cs.utah.edu

<sup>3</sup> Brigham Young University, Provo, Utah, USAjones@cs.byu.edu

**Abstract.** We present an incremental proof of the producer/consumer property for the PCI protocol. In the incremental proof, a corrected model of the multi-bus PCI 2.1 protocol is shown to be a refinement of the producer/consumer property. Multi-bus PCI must be corrected because the original PCI specification violates the producer/consumer property. The final model of PCI includes transaction types and reordering along with the completion mechanism for delayed PCI transactions. Verification results include multiple concurrent sessions of the producer/consumer property in a family of topologically isomorphic network configurations. The remaining configurations are identified and left for future work. In contrast to previous case studies involving this problem [13, 15], the incremental proof provides structure which simplifies otherwise difficult monolithic proof attempts.

## 1 Introduction

*Overview* Modelling complex systems can be improved by the use of refinement techniques. A refinement technique allows one to gradually develop a system step by step, or to tackle complex problem like the PCI Transaction Ordering Problem. The B event-based method provides a framework for deriving abstract systems satisfying the producer consumer property. The paper is an illustration of the effectiveness of refinement for those systems. The work described in this paper is a joint effort between researchers at LORIA, Nancy and the University of Utah and was facilitated by several exchange visits between institutions.

*Refining abstract systems* The essence of the refinement is to separate abstract systems with respect to safety properties and to simplify the view of the system under development. The development process begins with a very simple and abstract system which states properties expressed in requirements. The development process continues by adding more and more details through the refinement relationship. The refinement relationship expresses the enrichment of the initial abstract system and is supported by the Atelier B tool[20]. When a new refined system is analyzed, the tool generates proof

---

\* Supported in part by NSF grants CCR-9987516 and CCR-00814006 and in part by NSF/CNRS cooperation project 1998-2000 and PRST IL/QSL/DIXIT project

obligations which are proved either automatically, or using the Atelier B interactive prover. The refinement process is repeated for a chain of abstract systems and stops when the last model is suitably close to the implementation of system.

*Understanding systems by proofs* The documentation of a non-trivial system is very tedious and complex, since it is often expressed in terms similar to an automaton. The main aspects of our approach is the use of a general theorem prover which helps in the proof process understanding what is missing in the model under development; the refinement allows us to be very close to the concrete model described in the documentation of the system. Though the prover is interactive, it helps in deriving properties as invariant or safety properties. The prover plays a central role in the refinement process and allows us to get a complete proof of the development and hence of the final protocol. The verification is *uniform* in the sense that no assumptions are made about the number of nodes in the system. The incremental process is based on refinement and we have to add step by step details of the document. The main problem is to start from a very abstract, yet significant, system expressing the goal of the protocol (namely the producer/consumer system in this case study). The methodology allows us to address problems in small steps because of the incremental process. It allows us to think of the properties of the specified system and to communicate together using precise formal definitions.

*Outline* In our study, we build a model of the PCI protocol (*Peripheral Component Interconnect*, an I/O standard) using refinement, and prove that this protocol satisfies a crucial property, the Producer/Consumer property.

First, we briefly recall the PCI protocol, then present the B System methodology. Next, we use the B System to model a simple realization of the Producer/Consumer property, which we refine into an abstraction of the PCI protocol at the bus/bridge level. We conclude with some observations about refining a model of PCI from the Producer/Consumer property and compare the refinement verification with previous efforts to verify the same property using theorem proving and model checking.

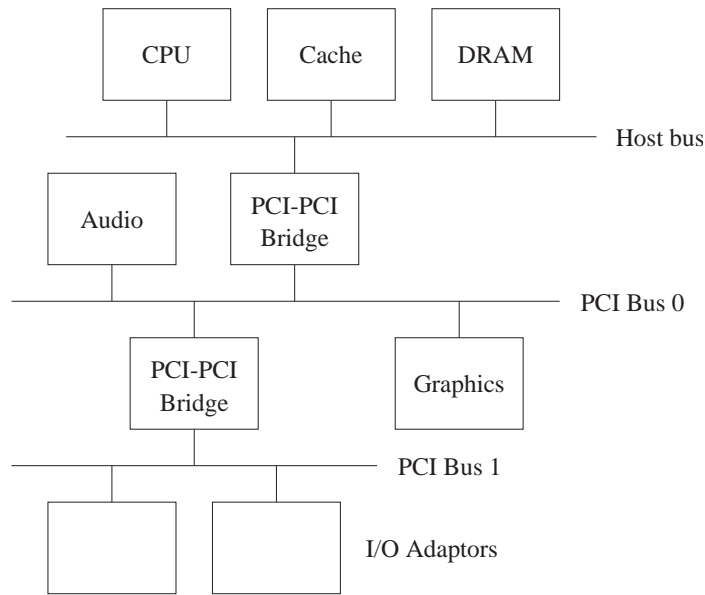
## 2 The PCI Transaction Ordering Problem

### 2.1 A Short Review of the PCI Protocol

A PCI network is a network of *agents* connected to *buses*, the *buses* being interconnected via *bridges*. The entire network forming an acyclic graph with exactly one path from any agent to every other agent as shown on figure 1. Routing is performed using a static routing table.

All agents have two waiting queues (or channels): one incoming (the target channel) and one outgoing (the master channel). Similarly, every bridge between buses b1 and b2 has two channels: one channel with b1 as its in-bus and b2 as its out-bus, and one channel with b2 as its in-bus and b1 as its out-bus. The PCI protocol (Revision 2.1)[16] uses two kinds of transactions: *Posted* transactions and *Delayed* transactions. For simplicity of explanation, consider a simple PCI network with two agents connected to two different buses linked via a bridge.

First, suppose an agent, the *Consumer* emits a delayed read (DR) transaction addressed to another agent *Data*. The entire mechanism is shown on figure 2.

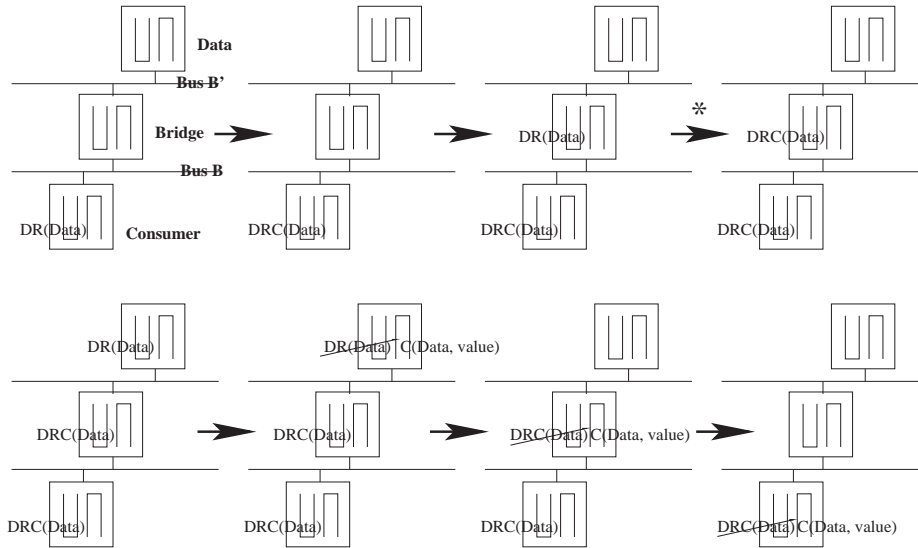


**Fig. 1.** a PCI acyclic network

A routing table indicates that messages addressed to Data pass through bus b1, so the Consumer attempts to enqueue its transaction in the bridge channel going from b1 to b2 (1). After the first attempt, the DR transaction is marked as committed by the Consumer (2), the Consumer re-tries the transaction until the target bridge channel receives this transaction (3). The Consumer retains a copy of the delayed transaction in its master channel. Next, the bridge channel tries to relay this transaction toward the target channel of Data, after one try the transaction is also marked committed by the bridge channel (4). The bridge channel continues to re-send the transaction until it is received by the target (5). The bridge also keeps a copy of the delayed transaction. The copies of delayed transactions are kept and deleted by the completion, which is described next.

Finally the DR transaction reaches the target channel of Data and the completion mechanism begins. First, the delayed read transaction is erased from the target channel of Data and is replaced by a Completion transaction in the Data master channel (6). The completion containing the return value of Data and the ID of target agent of the original DR transaction (Data in this case). The Completion transaction then travels backward from Data to Consumer, deleting copies of the DR transaction in every channel it crosses, until it reaches the Consumer (7). The Completion carries the target of the original transaction (Data in this case) so it is easy to match a completion transaction with its corresponding DR transaction copy. When the completion reaches the Consumer, the transaction is complete and there are no more DR transaction copies pending since they have all been erased on the way back (8).

The difference between Posted transactions and Delayed transactions is that a Posted transaction is not acknowledged by a completion. A posted transaction completes on the



**Fig. 2.** Delayed read mechanism

originating bus before completing on the destination bus, while it is the exact opposite for a Delayed transaction. One advantage of a delayed transaction is that the bus is not held in a wait state while completing an access to a slow device. In order to avoid deadlock in a PCI network, re-orderings in the channels are allowed. Legal re-orderings are shown in figure 3. Also, uncommitted transactions may be discarded.

## 2.2 A Bug in the Protocol

Despite the re-orderings allowed to avoid deadlock, the PCI protocol is intended to satisfy the Producer/Consumer transaction ordering property.

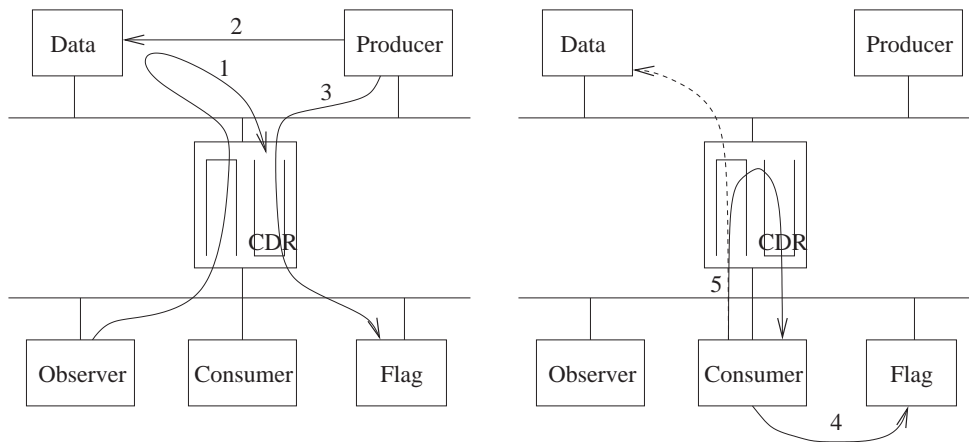
**The Producer/Consumer Property** The Producer/Consumer property (P/C) states that if an agent (the Producer) writes a value to another agent (the Data), and then sets a flag at a third agent (the Flag), then any other agent (the Consumer) which reads the value written in Flag is guaranteed to read the value written to Data by Producer when reading Data—assuming no other agents overwrite the value written by the Producer. This property is intended to hold in every PCI network for any choice of Producer, Consumer, Data and Flag. Unfortunately, PCI violates this property due to a phenomenon called *completion stealing*.

**Completion Stealing, Breach of P/C** Figure 4. shows a violation of Producer/Consumer involving completion stealing.

Suppose an agent, called Observer, is introduced, and that the Observer requests a value at agent Data. Next the transaction completion from Data, intended for the Observer,

Row pass Column?	P	DR DW	CDR/CDW
P	No	Yes	Yes
DR DW	No	Y/N	Y/N
CDR	No	Yes	Y/N
CDW	Y/N	Yes	Y/N

**Fig. 3.** The ordering rules in a bridge. C stands for *Completion*, P for *Posted*, D for *Delayed*, R for *Read* and W for *Write*. Yes means *Must be allowed to pass*, No means *Can not be allowed to pass*, Y/N means *the designer may choose either way*



**Fig. 4.** Completion stealing in a PCI network

waits in the bridge whilst Producer writes to Data and Flag. Then Consumer reads the value written in Flag and emits a Delayed Read transaction toward Data, which is then committed.

The completion from the Observer transaction is still waiting in the bridge. The completion *for* the Observer stores the address of the target (Data) but not the source (Observer).

The violation happens when the completion waiting on the bridge completes with the Consumer instead of the Observer and the Consumer reads the *old* value of Data.

### 2.3 A Revision Proposition

Corella made a proposal to fix this bug [11] :

- implement local-master identifiers so that a delayed transaction knows where it originated from.

- require bridges and multi-function devices to have only one outstanding transaction at a time for a given address (because of some aliasing problems that might occur and that might enable completion-stealing).

Previously, we have tried to verify the Producer/Consumer property with Corella’s solution using a combination of model checking and theorem proving [15].

### 3 Proof-Based Development

The B system [2, 3] approach is based on B notations of the B method [4] and extends the methodological scope of initial concepts as set-theoretical notations and generalized substitutions to take into account *abstract systems*. An *abstract system* is characterized by a (finite) list of variables possibly modified by a (finite) list of events; an invariant establishes properties satisfied by variables and maintained by the activation of events reacting to the environment, when guards are true. Abstract systems are close to actions systems of Back [5] and to UNITY programs [9]. We briefly recall definitions and principles of abstract systems and we explain how they can be managed by Atelier B [20].

#### Definition 1. (event)

An event has the following general form:  $yyy = \mathbf{any\ } x \mathbf{\ where\ } P(x, v) \mathbf{\ then\ } S(v, x) \mathbf{\ end}$  where  $yyy$  is the name of the event,  $x$  are local variables,  $v$  are state variables,  $P(x, v)$  is a condition, called a guard,  $S(v, x)$  is a (generalized) substitution expressing transformations of states and  $\exists x . P(x, v)$  is the guard of the event.

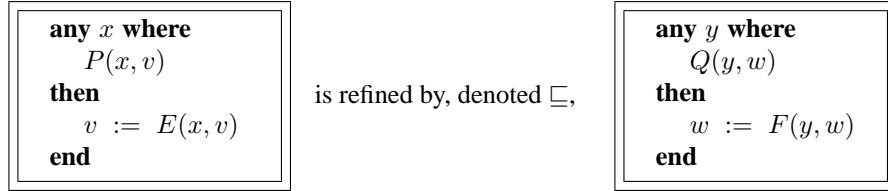
Proof obligations are produced from substitutions, whose semantics is defined using weakest preconditions. For instance, we give two rules for obtaining proof obligations from texts of events.

<p><b>any</b> <math>x</math> <b>where</b>  <math>P(x, v)</math>  <b>then</b>  <math>v : R(x, v, v_0)</math>  <b>end</b></p>	<p>The event preserves the invariant <math>I</math>, when the following proof obligation holds: <math>I(v_0) \wedge P(x, v_0) \wedge R(x, v, v_0) \Rightarrow I(v)</math>. <math>v_0</math> is the value of the variable <math>v</math>, when the event is triggered ; <math>v</math> is the value of the variable just after the activation of the event.</p>
---	--

<p><b>any</b> <math>x</math> <b>where</b>  <math>P(x, v)</math>  <b>then</b>  <math>v := E(x, v)</math>  <b>end</b></p>	<p>The event preserves the invariant <math>I</math>, when the following proof obligation holds: <math>I(v) \wedge P(x, v) \Rightarrow I(E(x, v))</math>; <math>E(x, v)</math> is the value of the variable <math>v</math> just after the activation of the event.</p>
---	---

The refinement of events defines a link between events and preserves properties of events; it maintains the invariant and there are several ways to refine events. We give

an example of refinement and a list of proof obligations is generated to validate the refinement.



- $I(v)$  is an invariant of an abstract system  $S$  and  $J(v, w)$  is an invariant of a refined abstract system  $S'$  for  $w$  and a gluing invariant linking  $v$  and  $w$ .
- $I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow \exists x.(P(x, v) \wedge J(E(x, v), F(y, w)))$  is the proof obligation for checking the refinement.

Proof obligations of an event refinement contain informations that help in modifying invariant or event, when one or more are remaining unproved because they are unprovable; the latter case leads to a reformulation of the event or a new statement of the refined invariant. The proof engine is the tool supporting the validation of the refinement. Refinement of events leads to define refinement of abstract systems; refining an abstract system by another one is based on the refinement of events in the abstract system according to the relation defined above:  $S \sqsubseteq S'$  (read  $S'$  refines  $S$ ) means that  $S$  and  $S'$  have the same set of events and every abstract event (of  $S$ ) is refined by the concrete event of  $S'$ . However, new events can be introduced in the new event system and refine the abstract *silent* event. A *silent* event is either a skip event (variables are not modified in the abstract model) or a keep event (modifying variables but maintaining abstract invariant). Rules for the refinement of abstract systems are simply defined as follows.

**Definition 2.** (*refinement of abstract systems*)

Let  $S$  and  $S'$  abstract systems

$S$  is refined by  $S' \cup U$  where  $U = \{U_k : k \in K\}$ , when

- $S \sqsubseteq S'$  and **silent**  $\sqsubseteq U_k, (k \in K)$ ,
- $I(v) \Rightarrow \text{grd}(S') \vee \text{grd}(U)$  ( $\text{grd}(U)$  (resp.  $\text{grd}(S')$ ) stands for the disjunction of  $U$ 's guards (resp. the disjunction of  $S'$ 's guards),
- $I(v) \Rightarrow V \in \mathbb{N}$  ( $V$  is the variant controlling the convergence),
- $I(v) \Rightarrow (V = n \Rightarrow [U_k](V < n)), (k \in K)$ .

The refinement of abstract systems allows us to enrich system models in a *step by step* approach and in a gradual way. Refinement provides a way to construct stronger invariants of the system and to introduce communications, if needed. No assumptions are made regarding the size of the system, in contrast to model checking. The case study will show that refinement helps in understanding the system since the complexity of the system is distributed and proofs are made easier. Finally, the *B system* approach is a part of the validation process. Proof obligations ensure that properties are preserved

when events are triggered, deadlocks are avoided and no new events take control of the entire machine, that is, the events of the previous machine still happen—but in less time. Atelier B is not yet updated to directly handle the *B system* approach, but it allows us to generate proofs obligations for abstract systems and refinements; the case study is developed to illustrate the adequacy of that approach when constructing distributed systems.

## 4 Modelling in Order to Prove

### 4.1 Refining From the Producer/Consumer Property Toward an Abstraction of PCI Revision 2.1

The goal of our study is to formally prove that the *revised PCI protocol* satisfies the Producer/Consumer property, using abstraction and refinement techniques in the B event-based (system) method.

We shall start from a simple model of a system of the Producer/Consumer property, refine it gradually into a system abstracting the revised PCI protocol. The refinement will be validated at every step, thus we will have proven that the revised PCI protocol satisfies the Producer/Consumer property. The Atelier B [20] software was used in this study and abstract systems have been completely checked by the tool.

### 4.2 Initial specification of the Producer/Consumer Property

The first abstract system describes the behavior of the Producer and Consumer. The Producer produces a series of time-stamped data: the event *produce* describes the production of the data. The Consumer consumes a datum in the set of available data by copying the datum from *prod* to *cons*: the *consume* event models this task.

In this abstraction, the Producer/Consumer property is expressed by the fact that all consumed datum has been produced previously ( $cons \subseteq prod$ ).

In our first model *pci1*, we define two sets *AD* and *VAL*. The variables of this model are *prod* and *cons* which are initialized to the empty set. The invariant of *pci1* is the following:

$$\begin{array}{l} prod \in AD \leftrightarrow VAL \wedge \\ cons \in AD \leftrightarrow VAL \wedge \\ cons \subseteq prod \end{array}$$

There are only two events in this abstract model which are :

```

produce = any ad, data where
    ad ∈ AD ∧
    data ∈ VAL ∧
    ad ∉ DOM(prod)
then
    prod := prod ∪ {ad ↦ data}
end;

```

```

consume = any ad where
    ad ∈ AD ∧
    ad ∈ DOM(prod) ∧
    ad ∉ DOM(cons)
then
    cons := cons ∪ {ad ↦ prod(ad)}
end;

```

### 4.3 Introduction of the Data and Flag Agent

The second model refines the first model. This model introduces agents Data (*mem*) and Flag (*flag*) which are intermediate agents between the Consumer and Producer. In the state of the Producer/Consumer property for the PCI protocol, the Consumer can not consume (read) the data before checking the value of the flag to determine if the data is ready to read. The value of the flag is updated by the Producer after producing a new data value (*produce*) and writing the value to the data agent (*datawrite*.) The new and refined observable events in this model are:

```

produce = any ad, data where
    ad ∈ AD ∧
    data ∈ VAL ∧
    ad ∉ DOM(prod)
then
    prod := prod ∪ {ad ↦ data}
end;

```

```

datawrite = any ad where
    ad ∈ AD ∧
    ad ∉ DOM(mem) ∧
    ad ∈ DOM(prod)
then
    mem := mem ∪ {ad ↦ prod(ad)}
end;

```

```

flagwrite = any ad where
    ad ∈ AD ∧
    ad ∉ DOM(flag) ∧
    ad ∈ DOM(mem)
then
    flag := flag ∪ {ad ↦ mem(ad)}
end;

```

<pre> consume = <b>any</b> <i>ad</i> <b>where</b>     <i>ad</i> ∈ <i>AD</i> ∧     <i>ad</i> ∉ DOM(<i>cons</i>) ∧     <i>ad</i> ∈ DOM(<i>flag</i>) <b>then</b>     <i>cons</i> := <i>cons</i> ∪ {<i>ad</i> ↦ <i>mem</i>(<i>ad</i>)} <b>end</b>; </pre>
---

The invariant still states that data is consumed after it is produced but adds requirements on when the Consumer may consume the data:

$cons \subseteq flag \subseteq mem \subseteq prod$
--

#### 4.4 Transaction Abstractions

In this model, abstractions of the transactions are added using new set-variables : *dw*, *fw*, *fr*, *cfr*, *dr*, *cdr*. In this abstraction, we keep only the address of the data produced (*ad*) and not the corresponding value (*data* in method *produce* of the previous model). The set-variable *dw* represents the set of completed *data write* transactions, *fw* represents the set of completed *flag write* transactions, *fr* represents the set of *flag read* transactions having reached the Flag agent, *cfr* represents the set of *flag read completion* transactions having completed at the Consumer agent, *dr* represents the set of *data read* transactions having reached the Data agent and *cdr* represents the set of *data read completion* transactions having completed at the Consumer agent.

We restrict our study to *posted* write transactions, and *delayed* read transactions because reads can not be posted and delayed writes allow only one write message to be in transit at a time for the Producer/Consumer property, while posted writes allow both write messages (*data write* and *flag write*) to be in transit at the same time. Thus the model with posted writes is of more interest.

In this model, transactions travel instantaneously. In later models, when topology, bridges and buses will be introduced (cf. 4.5), we will introduce a delay in transactions travel. The Producer/Consumer property is expressed as:  $fr \subseteq fw \wedge dr \subseteq dw$ . The invariant is now:

$ \begin{aligned} dr &\subseteq dw \wedge \\ cfr &\subseteq fr \cap dw \wedge \\ dr &\subseteq cfr \wedge \\ cdr &\subseteq dr \wedge \\ fr &\subseteq fw \wedge \\ dw &= \text{DOM}(mem) \wedge \\ fw &= \text{DOM}(flag) \wedge \\ \text{DOM}(cons) &\subseteq cdr \end{aligned} $
--

In order to abstract the transactions, we also need to update our old observable events (modify the guards) and add new intermediate observable events. We proved this refinement without much difficulty.(cf.figure 7)

```
consume = any ad where  
     $ad \in AD \wedge ad \in cdr \wedge ad \notin \text{DOM}(cons)$   
then  
     $cons := cons \cup \{ad \mapsto mem(ad)\}$   
end;
```

```
produce = any ad, data where  
     $ad \in AD \wedge data \in VAL \wedge ad \notin \text{DOM}(prod)$   
then  
     $prod := prod \cup \{ad \mapsto data\}$   
end;
```

```
datawrite = any ad where  
     $ad \in AD \wedge ad \notin \text{DOM}(mem) \wedge ad \in \text{DOM}(prod)$   
then  
     $mem := mem \cup \{ad \mapsto prod(ad)\} \parallel$   
     $dw := dw \cup \{ad\}$   
end;
```

```
flagwrite = any ad where  
     $ad \in AD \wedge ad \notin \text{DOM}(flag) \wedge ad \in \text{DOM}(mem)$   
then  
     $flag := flag \cup \{ad \mapsto mem(ad)\} \parallel$   
     $fw := fw \cup \{ad\}$   
end;
```

```
flagread = any ad where  
     $ad \in AD \wedge ad \notin fr \wedge ad \in fw$   
then  
     $fr := fr \cup \{ad\}$   
end;
```

```
compflagread = any ad where  
     $ad \in AD \wedge ad \notin cfr \wedge ad \in fr$   
then  
     $cfr := cfr \cup \{ad\}$   
end;
```

```

dataread = any ad where
    ad ∈ AD ∧ ad ∈ cfr ∧ ad ∉ dr
then
    dr := dr ∪ {ad}
end;

```

```

compdataread = any ad where
    ad ∈ AD ∧ ad ∈ dr ∧ ad ∉ cdr
then
    cdr := cdr ∪ {ad}
end;

```

#### 4.5 Concrete Transactions and Topologies

In this section, we abstract the PCI protocol at the bus/bridge level by introducing topologies (equivalence classes for PCI networks), concrete transactions, bridges, ... We still do not allow re-ordering at this level of abstraction, but we prepare the way by modelling bridge channels. We proved that this model is a refinement of the previous model, and thus still satisfies the Producer/Consumer property. This refinement proof was considerably more difficult than the previous ones (see the discussion in the next subsection).

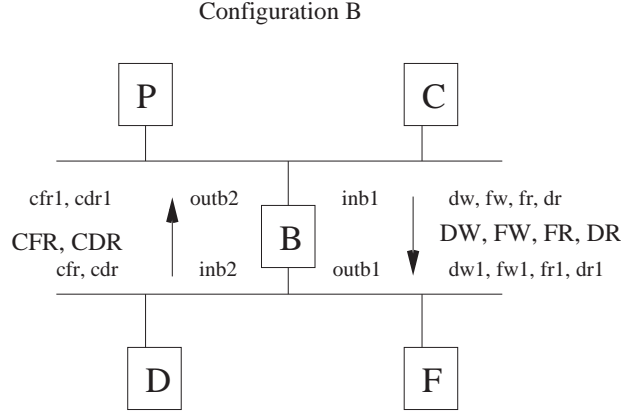
**Topologies** The Producer, Consumer, Data and Flag agents can be arranged into one of four distinct families of labeled configurations shown in figure 5. A fixed set of configuration families is obtained by distinguishing configurations only by their topology (or shape). This is done by ignoring the number of bridges in a sequence of bridges between agents or a common bridge (labeled B in the figure). As a result, each of the four families contain an unbounded number of unique configurations. For example, the box labeled P in figure 5 may include an unbounded number of bridges connecting the Producer to the common bridge labeled B.

In general, the set of topologies for configuration families over  $n$  agents can be enumerated by an algorithm based on full Steiner topologies [12]. A Steiner topology is a Steiner tree without the associated geometrical embedding. Steiner trees are used to connect a set of points with minimal edge length. New internal points, called Steiner points, can be added to a Steiner tree to minimize edge length. A full Steiner topology is a Steiner topology with the maximal number of Steiner points. The algorithm for enumerating all configuration families on  $n$  agents first enumerates all full Steiner topologies with  $n$  agents or less then considers all topologically unique ways to add the remaining agents to the full Steiner topology.

For properties defined on four agents (such as the producer/consumer property) the configuration enumeration algorithm generates two unique topologies. The remaining configuration families are generated by considering unique labelings of the topologies. Each configuration family must be verified separately.

-bb-error = =

**Fig. 5.** Topologies equivalence classes



**Fig. 6.** Abstraction of the bridge and concretization of transactions, in the B configuration

**PCI at the bus/bridge level** For the purpose of our study, we restricted ourselves to one of the four topology equivalence classes, and completed the proof for this class, rather than simultaneously develop a model for every topology.

We choose the *most interesting* topology, configuration B in figure 5. This topology is the most interesting, because it allows the most transactions to cross the common bridge, thus allowing reordering rules to interfere with the Producer/Consumer property. In the model of PCI at the bus-bridge level, transactions travel instantaneously on the bus, but not the bridge.

Thus, we modelled the two channels of the common bridge by two queues each :  $inb1$  and  $outb1$  for the channel from bus b1 to bus b2,  $inb2$  and  $outb2$  for the channel from bus b2 to bus b1 as shown in figure 6. We define this following set of states

$$STATES = \{DW, FW, FR, CFR, DR, CDR\}$$

Our queues are defined as partial injections  $\mathbb{N} \rightsquigarrow AD \times STATES$

Transactions going from bus b1 to bus b2, enter the queue  $inb1$  at the first available position ( $cinb1$ ), then exit the bridge by entering queue  $outb1$ , and finally they reach their targeted agent. One part of the gluing invariant is:

$$\begin{aligned} dw &= \text{DOM}(\text{RAN}(inb1) \triangleright \{DW\}) \wedge \\ dw1 &= \text{DOM}(\text{RAN}(outb1) \triangleright \{DW\}) \end{aligned}$$

We introduced new set-variables  $dw1, fw1, fr1, cfr1, dr1$  and  $cdr1$ , which are subsets of the previous  $dw, fw, fr, cfr, dr$  and  $cdr$  variables (cf.4.4) and indicate which transactions have already crossed the common bridge, and which have not.

In this model, the channel queues are strictly first in first out. (FIFO). That will not be the case later when we introduce re-ordering (cf.4.6).

Rather than giving the complete set of events, we give only the events which might be observable when a *flag write* transaction travels from the Producer to its corresponding Flag agent (and crosses the common bridge in the current configuration, cf. figure 6):

```

flagwrite = any ad where
    ad ∈ AD ∧
    ad ∉ DOM(flag) ∧
    ad ∈ DOM(mem) ∧
    (ad ↦ FW) ∉ RAN(inb1) ∧
    (ad ↦ DW) ∈ RAN(inb1)
then
    flag := flag ∪ {ad ↦ mem(ad)} ||
    inb1 := inb1 ∪ {cinb1 ↦ (ad ↦ FW) } ||
    fw := fw ∪ {ad} ||
    cinb1 := cinb1 + 1
end;

```

```

FWout = any ad where
    ad ∈ AD ∧
    coutb1 < cinb1 ∧
    coutb1 ↦ (ad ↦ FW) ∈ inb1 ∧
    (ad ↦ FW) ∉ RAN(outb1)
then
    coutb1 := coutb1 + 1 ||
    outb1 := outb1 ∪ {coutb1 ↦ (ad ↦ FW) } ||
    fw1 := fw1 ∪ {ad}
end;

```

**Example of a Very Hard PO:** *compflagread.17*

“Local hypotheses  $\cdot \cdot \wedge$

*ad* ∈ *AD* ∧  
*ad* ∈ *fw1*\$1 ∧  
*not*(*ad* ∈ *cfr*\$1) ∧  
*ad* ∈ *fr1*\$1 ∧  
*not*(*ad* ↦ *CFR* ∈ RAN(*inb2*\$1)) ∧

“Check that the invariant *cfr* ⊆ *dw1* is preserved by the operation – ref 4.4, 5.5.

$\cdot \wedge$

⇒

*ad* ∈ *dw1*\$1

This proof obligation was generated when refining method *compflagread* with concrete transactions. We were required to prove that if the *flag read* transaction is about to

complete for a given address ( $ad \in fr1\$1 \wedge ad \in fw1\$1 \wedge not(ad \in cfr\$1)$ ), then the corresponding *data write* transaction has previously exited the bridge ( $ad \in dw1\$1$ ).

This property seemed obvious when thinking about the protocol, yet it was difficult to prove (about one week of work). To do so, we had to introduce the following properties in the global invariant. Although these properties seemed obvious according to the protocol and our current model, we had to prove them for every observable event. The original PO, `compflagread.17`, helped us build a stronger invariant.

- $\forall ad.(ad \in AD \wedge ad \mapsto DW \in \text{RAN}(outb1))$   
 $\Rightarrow ad \mapsto DW \in \text{RAN}(inb1) \wedge$   
 $outb1^{-1}(ad \mapsto DW) \leq inb1^{-1}(ad \mapsto DW)$
- $\forall ad.(ad \in AD \wedge ad \mapsto FW \in \text{RAN}(outb1))$   
 $\Rightarrow ad \mapsto FW \in \text{RAN}(inb1) \wedge$   
 $outb1^{-1}(ad \mapsto FW) \leq inb1^{-1}(ad \mapsto FW)$
- $\forall ad.(ad \in AD \wedge ad \in fw)$   
 $\Rightarrow ad \mapsto DW \in \text{RAN}(inb1) \wedge$   
 $0 \leq -1 + inb1^{-1}(ad \mapsto FW) - inb1^{-1}(ad \mapsto DW) \wedge$   
 $inb1^{-1}(ad \mapsto DW) + 1 \leq inb1^{-1}(ad \mapsto FW)$
- $\forall ad.(ad \in dw \wedge inb1^{-1}(ad \mapsto DW) \in \text{DOM}(outb1))$   
 $\Rightarrow ad \mapsto DW \in \text{RAN}(outb1)$

Note that we used  $\leq$  instead of  $=$  in the first two properties. This is because later on, when re-ordering will be introduced (cf.4.6), posted transactions FW and DW might exit the bridge at a position preceding the position they have entered, since they will be able to bypass delayed transactions waiting in the bridge. If we had kept the equality sign, we wouldn't have been able to prove refinement in our last model.

These properties will remain true in further models, when we introduce re-ordering, since *data write* is a posted transaction, it must have exited the bridge before the FW transaction did (DW was emitted before FW, from the same agent and can NOT be bypassed), and FW has exited the bridge since  $ad \in fw1\$1$ .

#### 4.6 Last step Toward Abstract PCI

In the final step, we refine the previous models in an abstraction of PCI with re-ordering rules. We also add commitment and delays in transactions travel from the common bridge toward the target agent, to reach a complete PCI abstraction. After proving this refinement, we deduce that PCI satisfies the Producer/Consumer property.

**From the Bridge to the Agents** Adding a delay in transactions travelling from the common bridge toward the target agent is an easy but necessary refinement.

Easy because we just add new variable-sets ( $dw2, fw2, fr2, cfr2, dr2$  and  $cdr2$ ), subsets of the previous variable-sets ( $dw1, fw1, fr1, cfr1, dr1$  and  $cd1$ ) and new methods like :

```

FWbridge2agent = any ad where
    ad ∈ AD ∧
    ad ∈ fw1 ∧
    ad ∉ fw2
then
    fw2 := fw2 ∪ {ad}
end;

```

```

FRbridge2agent = any ad where
    ad ∈ AD ∧
    ad ∈ fr1 ∧
    ad ∈ fw2 ∧
    ad ∉ fr2
then
    fr2 := fr2 ∪ {ad}
end;

```

the guard  $ad \in fw2$  is very important, it means that the  $FR$ transaction won't bypass the corresponding  $FW$ transaction on its way between the bridge and the Flag

Necessary, because even if  $fw \subseteq dw$ , we don't have  $fw2 \subseteq dw2$ . The *data write* transaction is still emitted by the Producer before its corresponding *flag write* transaction, but it doesn't necessarily reach its target (Data) before the *flag write* transaction, since the path between the bridge and Data and the path between the bridge and Flag are different, independent, and might have different lengths. (remember our study is based on topology equivalence classes, cf.4.5)

**Commitment** We modelled commitment over the bridge channels, in fact only channel 1 (queues *inb1* and *outb1*) in our current configuration. Deletion was taken care of with completion transactions, crossing the common bridge through channel 2 in our configuration. Committed transactions are held in  $AD \times STATES$  set-variables *commitinb1* and *commitoutb1*.

The invariant gluing this abstraction to the previous one is :

$$\begin{aligned} \text{commitinb1} &\subseteq \text{RAN}(\text{inb1}) \triangleright \{FR, DR\} \wedge \\ \text{commitoutb1} &\subseteq \text{RAN}(\text{outb1}) \triangleright \{FR, DR\} \end{aligned}$$

Indeed, only delayed transactions are committed (so *not FW* and *DW*, cf. 4.4), and in the current configuration they both cross the common bridge through channel 1.

Deletion happens during completion transaction travels.

Here is an example of commitment, and an example of where deletion is added in this model :

```

flagread = any ad where ...
then
    ... ||
    commitinb1 := commitinb1 ∪ {ad ↦ FR}
end;

```

```

CFRout = any ad where ...
then
    ... ||
    commitinb1 := commitinb1 - {ad ↦ FR}
end;

```

The main invariant to verify on the commitment variables is :

$$\forall(ad). (ad \in AD \wedge ad \in cdr2 \Rightarrow \{ad\} \triangleleft commitinb1 = \emptyset \wedge \{ad\} \triangleleft commitoutb1 = \emptyset)$$

This invariant means that at the end of every Producer/Consumer cycle ( $ad \in cdr2$ ), all pending committed transactions relative to this cycle have been deleted.

Unfortunately, this invariant could not be proven directly. We had to decompose the Producer/Consumer cycle, and add an invariant for every step of the cycle. Each of these proofs were therefore made easier, and it also strengthened our invariant.

One such step-invariant was :

$$\forall(ad). (ad \in AD \wedge ad \in dr1 \wedge ad \notin dr2 \Rightarrow \{ad\} \triangleleft commitinb1 = \{(ad \mapsto DR)\} \wedge \{ad\} \triangleleft commitoutb1 = \{(ad \mapsto DR)\})$$

These two first refinements were quite easy to prove with Atelier B, that was not the case for the next one.

**Re-ordering Rules** In this final model, we introduced re-ordering rules.

In the current configuration, the only re-ordering rule to consider is the one where posted transactions are allowed to pass delayed read and delayed write transactions. For the Producer/Consumer property, other transaction types do not interact in the current configuration. Our study also does not include delayed write, so only these four re-ordering rules are to be considered :

- reordering1: *DW* is allowed to pass *FR*
- reordering2: *DW* is allowed to pass *DR*
- reordering3: *FW* is allowed to pass *FR*
- reordering4: *FW* is allowed to pass *DR*

We introduced them very easily in the model :

```

reordering1 =
  any ad1, ad2 where
    ad1 ∈ AD ∧ ad2 ∈ AD ∧
    ad1 ≠ ad2 ∧ (ad1 ↦ DW) ∈ RAN(inb1-outb1) ∧
    (ad2 ↦ FR) ∈ RAN(inb1-outb1) ∧
    inb1-1(ad2 ↦ FR) ≤ inb1-1(ad1 ↦ DW) ∧
    inb1-1(ad2 ↦ FR)+1 = inb1-1(ad1 ↦ DW)
  then
    inb1 := inb1 ⇐ ( {inb1-1(ad2 ↦ FR) ↦ (ad1 ↦ DW)}
                    ∪ {inb1-1(ad1 ↦ DW) ↦ (ad2 ↦ FR)} )
  end;

```

The rule is read : “if one *DW* transaction relative to one P/C cycle (*ad1*) has entered the common bridge and not yet exited it ( $(ad1 \mapsto DW) \in \text{RAN}(inb1-outb1)$ ), while a *FR* transaction relative to another P/C cycle ( $ad2 \neq ad1$ ) has also entered the common bridge, before this *DW* transaction ( $inb1^{-1}(ad2 \mapsto FR) \leq inb1^{-1}(ad1 \mapsto DW)$ ), and has not yet exited the common bridge either ( $(ad2 \mapsto FR) \in \text{RAN}(inb1-outb1)$ ), and if these two transactions are next to each other in the queue ( $inb1^{-1}(ad2 \mapsto FR)+1 = inb1^{-1}(ad1 \mapsto DW)$ ), then, the re-ordering event reordering1 is likely to be observed. reordering1 consists of a simple position exchange in the queue inb1.

It took us almost one week, to prove this refinement, so we only introduced re-ordering rules reordering2, reordering3 and reordering4, checked the PO generated, and claim that the refinement proofs should be symmetrically equivalent. So far we did not manage to do them.

## 5 Conclusion and Related Works

This study aimed at showing possibilities offered by the B-event method to prove a simple property on a very complex system, namely the PCI protocol.

The interest of the method is that we proved it gradually, thus lessening the complexity of the intermediate proofs. Most of the non obvious PO generated by Atelier B were proved automatically (about 2/3 of the nPO column on figure 7), the others had to be proved interactively but Atelier B is especially designed for this purpose.

The final model we made is very close to the PCI protocol, even if it’s still an abstraction. Having proved that it refines a model of the Producer/Consumer property enables us to say that PCI satisfies this property, at least in the configuration studied (out of 4 possible topology equivalence classes).

In this case study, parameterized system verification is performed using incremental proof. The parameterized verification problem has been widely studied in the context of model checking [8, 14, 1, 17] and, more recently, logic program transformations [18] and theorem proving combined with model checking [7, 13]. Incremental refinement proof requires interactive proof, unlike model checking, but avoids the state explosion and non-termination problems encountered by model checking solutions. While all interactive proof methods avoid state explosion and non-termination problems, incremen-

tal proofs provide a convenient framework for breaking an otherwise difficult proof into manageable pieces.

Previously, the co-authors from the University of Utah (in collaboration with Mokkdem and Hosabettu) attempted to prove the same property directly from a higher-order logic theory of PCI transitions [15]. Creating the proof monolithically in one step from property to protocol was prohibitively difficult. The incremental proof effort using Atelier B divided the proof into manageable steps that could be completed with comparatively minimal user effort. The one-step proof was abandoned, unfinished, after 18 months of effort. In contrast, the incremental proof was completed in less than three months. The PCI protocol has been considered in other case studies [19, 10]. In each of the other case studies, the bus signalling properties of PCI, rather than multi-bus transaction ordering properties, were of interest. Both case studies rely on forms of model checking.

Project status

COMPONENT	TC	POG	Obv	nPO	nUn	%Pr
pci1	OK	OK	77	6	0	100
pci2	OK	OK	238	16	0	100
pci3	OK	OK	650	37	0	100
pci4B	OK	OK	1981	332	0	100
pci5B	OK	OK	3003	657	0	100
TOTAL	OK	OK	5949	1048	0	100

**Fig. 7.** Summary of proofs, all the Proof Obligations generated (Obv and nPO) have been proved (%Pr = 100).

## References

1. P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parameterized system verification. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, Trento, Italy, July 1999. Springer-Verlag.
2. J.-R. Abrial. Extending b without changing it (for developing distributed systems). In H. Habrias, editor, *1<sup>st</sup> Conference on the B method*, pages 169–190, November 1996.
3. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98 :Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
4. Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.
5. R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

6. R. J. R. Back. Correctness preserving programs refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam, 1980.
7. Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Routing information protocol in HOL/SPIN. In *Theorem Provers in Higher-Order Logics 2000: TPHOLs00*, August 2000.
8. M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, April 1989.
9. K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.
10. Edmund Clarke, Somesh Jha, Yuan Lu, and Dong Wang. Abstract BDDs: A technique for using abstraction in model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1999. Springer-Verlag.
11. Francisco Corella. Proposal to fix ordering problem in PCI 2.1, 1996. Accessed June 2001 [www.pcisig.com/reflector/thrd8.html#00704](http://www.pcisig.com/reflector/thrd8.html#00704).
12. F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, Netherlands, 1992.
13. Michael Jones and Ganesh Gopalakrishnan. Verifying transaction ordering properties in unbounded bus networks through combined deductive/algorithmic methods. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: FMCAD'00*, number 1954 in LNCS, page 505, Austin, Texas, November 2000.
14. Y. Kesten, O. Maler, M. Marcus, A Pnueli, and E. Shahar. symbolic model checking with rich assertional languages. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.
15. Abdel Mokkedem, Ravi Hosabettu, Michael D. Jones, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. *Formal Methods in Systems Design*, 16(1):93–119, January 2000.
16. PCISIG. PCI Special Interest Group–PCI Local Bus Specification, Revision 2.1, June 1995.
17. Amir Pnueli and Elad Shahar. Liveness and acceleration in parameterized verification. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer-Aided Verification, CAV '00*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343, Chicago, IL, July 2000. Springer-Verlag.
18. A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, I.V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 172–187. Springer-Verlag, 2000.
19. Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In Warran A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: FMCAD'00*, number 1954 in LNCS, page 335, Austin, Texas, November 2000.
20. STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 3.5 edition, 1998.