Verification of the Futurebus+ Cache Coherence protocol: A case study in model checking

Kylie Williams & Robert Esser
School of Computer Science
University of Adelaide
Adelaide 5005, Australia
{kewillia, esser}@cs.adelaide.edu.au

Abstract

This paper presents a case study for automatic verification using the Communicating Sequential Processes formalism. The case study concerns the Futurebus+ cache coherency standard; we develop a formal model of the protocol and perform some verification tasks upon it. In the process of doing so, we extend the previous solution by developing a formal specification of cache coherence that is suitable for the verification of both directory and snooping based cache coherence protocols.

1 Introduction

The consistent growth of 'systems-on-a-chip' and the rapid increase in complexity of these systems has lead to an increased need for formal modelling and verification tools for system-level design. An important aspect of such systems is their heterogeneous nature; that is, they often contain mixed digital/analogue and mixed technology inputs and outputs, RF transmitters and possibly embedded software systems.

In order to develop a heterogenous model checker it is necessary to first understand the different approaches to model checking offered by both signal based verification formalisms such as Communicating Sequential Processes (CSP) and temporal logic based tools such as Symbolic Model Verifier (SMV). To this end, the aim of this case study was to consider a non-trivial verification task using both approaches in order to motivate a comparison between them.

The work by (Clarke Jr, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L. & Ness, L. A. 1993) on the verification of the Futurebus+ cache coherency protocol is thought by its authors to be the first attempt at formally verifying an IEEE standard. This work emphasized the importance of formal verification as it highlighted several unknown bugs in the protocol (which are more fully discussed in Section 6.1). The work also demonstrated that formal verification techniques are applicable to real world examples such as the Futurebus+ protocol. Further work (McMillan, K. L. 1992) demonstrated another error in the protocol, relating to a possible livelock in a particular bus configuration which was not checked in the original work.

This research was supported by ARC Linkage International 2002-2003

2 Communicating Sequential Processes

Communicating Sequential Processes (CSP) (Hoare, C. A. R. 1978) and (Roscoe, A. W. 1998) is both a formalism for describing concurrent systems and a theory of the behavior of such systems. The language is action-based, meaning that components of the system interact using direct communication (called *events*) rather than indirectly (for example, using manipulation of shared variables). The communication events are synchronous and the language has the capacity for parallel composition, non-determinism and hiding of events. FDR is an automatic verification system for CSP.

The simplest process in CSP is the process *Stop* which can do nothing and never communicates. For example, given the events *up* and *down* in Σ (the set of all possible events) the process $up \rightarrow down \rightarrow STOP$ will perform the event *up*, then *down* and then cease to communicate. The operator $\rightarrow$ is called the *prefixing* operator. The concept of prefixing leads us quickly to the concept of *recursion*. For example, the process $P = a \rightarrow P$ will continually be willing to participate in the 'a' event.

The process $P \ \Box \ Q$ (the machine readable version of this operator is []) is a process that offers the environment the first events of P and Q and then continue to behave as the process P or Q accordingly, the choice of route is made *externally* to the process. For example, the process $(a \rightarrow P) \ \Box \ (b \rightarrow Q)$ will *either* participate in an 'a' event and then behave as process P *or* participate in a 'b' event and then behave as process Q.

Consider the process $(a \rightarrow a \rightarrow P) \ \Box \ (a \rightarrow b \rightarrow Q)$. In this case, after the initial 'a' event, the process is free to offer 'a' or 'b' as alternatives (but not both). It is assumed that since we have written the process in this way, it is not important which route is taken. For example, the process $a \rightarrow ((a \rightarrow P) \ \Box \ (b \rightarrow Q))$ participates in the initial 'a' event and then is free to offer 'a' *and* 'b' as alternative events.

A *deterministic* process is one where the range of events offered to the environment depends only on things it has seen (i.e. the sequence of communications so far). In other words, it is formally nondeterministic when some internal decision can lead to uncertainty about what will be offered.

The process $(a \rightarrow a \rightarrow P) \ \Box \ (a \rightarrow b \rightarrow Q)$ presents the concept of non-determinism. While nondeterminism may not be a particularly useful real world behavior of a system, it is a very useful modelling tool. CSP uses the operator $\sqcap$ (the machine readable version of this operator is |~|) to indicate a non-deterministic choice. For example, the process

$P \sqcap Q$ will behave either like P or Q.

It is important to distinguish between $(a \rightarrow P) \Box (b \rightarrow Q)$ and $(a \rightarrow P) \sqcap (b \rightarrow Q)$. In the first process, the choice between a and b is external, the environment decides which route will be taken. In the second process, the choice is an internal decision and can not be influenced by the environment.

There are a number of ways to consider the behaviour of a CSP process; (Roscoe, A. W. 1998) defines three levels of semantics for processes: operational, denotational and algebraic. The operational semantics defines the way in which a process may be viewed as a state-transition diagram, where the transitions between states are the events defined the in the process definition. The denotational semantics describe a notion of process equivalence; a property in CSP is expressed as a CSP process, *Sys*, and a system (represented by the process *Spec*) satisfies the property if *Sys refines Spec*. There are three levels of denotational semantics in CSP - traces, failures and failures/divergences. Trace semantics deal with safety properties, failures semantics deals with both safety and liveness properties and failures/divergences semantics deals with safety, liveness and divergence properties. Finally, the algebraic semantics define a set of laws that extend the notion of equivalence provided by the denotational semantics.

In CSP a *trace* is a sequence of visible events that occur in an execution of a process. A trace may be finite (for example, if the process terminates or deadlocks) or infinite. The set of all traces of a process is called its *trace semantics* (denoted $traces(P)$). Note that this set is non-empty since it always contains the trace $\langle\rangle$, which denotes the empty trace (the process has performed no events). In general the events that a process participates in represents the behavior of a system defined by the process, and so it is common for the requirements of a system to be specified in terms of its traces. If *Spec* is a property, then the system *Sys* refines *Spec* if and only if $traces(Sys) \subseteq traces(Spec)$. From this we can see that *Sys* cannot engage in 'illegal' activities (i.e. those events not defined by *Spec*) and thus the traces model is used to verify safety properties.

Unfortunately, in the traces model it is true that STOP (the deadlock process) refines P for *every* process P. That is, it is impossible to distinguish between a deadlocking and non-deadlocking process in the traces model. For this reason, the *failures model* of CSP was developed. A *refusal set* is the set of events that a processes *Sys* will fail to accept forever. A *failure* (of a process *Sys*) is a pair (s, X) such that $s \in traces(Sys)$ and $X \in refusals(Sys/s)$. Note that Sys/s represents the process *Sys after* performing the trace s. The set *failures(Sys)* is the set of all failures of *Sys*. A process *Sys* failures-refines a process *Spec* if and only if $failures(Sys) \subseteq failures(Spec)$ (It should be noted that $failures(Sys) \subseteq failures(Spec)$ implies $traces(Sys) \subseteq traces(Spec)$). That is, *Sys* can neither accept nor refuse an event unless *Spec* also accepts or refuses the event and thus the failures model is often used for verifying liveness properties.

A process is said to *diverge* if it can enter a situation where it is infinitely performing some internal action and does not interact with the 'outside world'. We talk about a process diverging after a particular trace and the set *divergences(Sys)* containing the traces s on which *Sys* can diverge and also all extensions $s.t$ of those traces. A process that diverges after some trace, s, is equivalent (regardless of other behavior) to one that diverges immediately. Thus, a process *Sys* is said to failures/divergence refine a process *Spec* if and only if $failures(Sys) \subseteq failures(Spec)$ and $divergences(Sys) \subseteq divergences(Spec)$. If *Sys* failures/divergence refines *Spec* and visa versa, we say that the two processes are equivalent.

2.1 The operation of FDR

The model developed in this case study was verified using the tool FDR (Failures Divergences Refinement). Naturally, the refinement checking carried out by the FDR tool is based in the denotational semantics of process equivalence described previously. Given that a simple process can generate an infinite trace set, the direct calculation of the set of traces, failures and divergences for a given process would be exceedingly inefficient. To overcome this problem the FDR tool uses the operational semantics of the language to develop a (normalized) state-transition diagram for the system; the process of normalization is based in the algebraic laws governing the equivalence of processes.

To improve efficiency the FDR tool uses a two-level approach to the development of the state-transition diagram. The low level fully computes the state-transition tree while the high level computes a series of rules for the combination of the low level components. FDR analyzes the structure of a given process to determine an appropriate division between low and high levels. In order to undertake refinement checking of two processes the process representing the Specification needs to be converted to *normal form*. An important feature of the normal form is that each trace corresponds to a unique state. Each state in the normal form must adhere to the following basic rules:

- There are no invisible actions (called τ actions);
- Each node has a unique successor on each visible initial action it can perform.

Once the process representing the specification has been normalized a failures/divergence refinement can be verified by first establishing which states of the implementation system are divergent and marking them as such. The marked implementation can then be model-checked against the normal form of the specification. Finally, the FDR tool provides a number of predefined functions for automatic hierarchial compression of large process networks. These compression techniques allow the verification of systems whose state spaces exceed 10^7 states. For more information regarding the operation of FDR and the theory of CSP consult (Roscoe, A. W. 1998), (FSE 1997) or (Hoare, C. A. R. 1978).

3 The Futurebus+ protocol standard

The IEEE Futurebus+ Logical Protocol standard is a technology-independent protocol for single and multiple buses multi-processor architectures. Part of the standard specifies a protocol for maintaining cache coherency in a hierarchical multi-bus system and it is this part of the standard that we are interested in modelling.

Since the protocol is large, for brevity we have omitted discussion of certain sections of the protocol. In particular, we will limit our discussion to that part of the protocol that deals with a single bus segment that allows for (particular) transactions to be split, rather than the whole multi-segment hierarchy that the standard allows. Also, the IEEE Futurebus+ standard describes a number of protocols other than the cache coherence protocol, for example, power-up, reset and configuration protocols. These sections were omitted from our model. Similarly, the protocol specifies how a module should respond in exceptional circumstances, such as when a transmission

error occurred. We omitted this from the model by assuming that no transmission errors occurred. Since the properties of cache coherence is only concerned with modifications to a single location (see Section 4), transactions in the model referred only to a single cache line. Similarly we only needed to detect a change in the value of a cache line, and thus the data in the line was modelled using only two distinct values rather than the full 64 bytes specified by the standard. Finally, the protocol standard specifies a number of transactions that are designed for IO optimization. These transactions do not maintain cache coherence (as stated in the protocol) and thus were omitted from our model.

The protocol allows for a number of *modules* to be part of each bus segment. A module represents an agent that is interested in the transactions occurring on a cache line. In particular, we are interested in *cache modules* and *shared memory modules*. A cache module is a module that represents a cache/processor pair and a shared memory module represents the shared memory of the system. Note that in theory a number of shared memory modules can be present in a single system, each responsible for different cache lines. However, since the verification properties only refer to the changes in a single line we have restricted our model to those configurations with only one shared memory.

In the Futurebus+ protocol a data line held in a cache is either *clean* or *dirty*. A dirty line is one that has been modified by the cache; a clean line is unchanged. Each cache module in the system is required to keep an attribute for the cache line; the attribute represents the read and write access the cache has to the line. The attributes specified by the Futurebus+ protocol are: *invalid*, *shared unmodified*, *exclusive unmodified* and *exclusive modified*. A cache line with the invalid attribute will present the processor with both read and write misses. A cache line with the shared unmodified attribute will present the processor with read hits; however, a bus transaction will have to be executed to obtain write access to the cache line. A cache line with the exclusive unmodified attribute allows the processor to read and write the line without a bus transaction. If this line is evicted from the cache a copy back transaction is unnecessary since the line is clean. A cache line with the exclusive modified attribute allows read and write access by the process, however, eviction from the cache will require a copyback transaction since the line is dirty.

Within individual bus segments coherence is maintained by having all modules *snoop* each transaction and update their line status accordingly. A cache with the exclusive modified status on the particular line will *intervene* in a transaction and supply the data in place of the shared memory. A cache (or shared memory) may choose to *split* a transaction, meaning that the copy of the data line is not provided immediately and so the requesting cache is forced to wait; a module that splits a transactions is required to eventually respond with the data line. We are restricting ourselves in this discussion to the section of the protocol dealing with a single bus segment that allows split transactions and are not interested how the split transactions are extended to maintain coherency across a hierarchy of bus segments. The standard defines a number of transactions that relate to the movement of the data lines around the bus segment. The transactions defined in the protocol standard are:

Read Shared This transaction is initiated by a cache which wishes to obtain *read* access to the data line (in response to a read request by its processor).

Read Modified This transaction is initiated by a cache who wishes to obtain *read/write* access to the data line (in response to a write request by its processor).

Invalidate This transaction is initiated by a cache who has read access to the data line and wishes to obtain *write* access to the line (in response to a write request by its processor).

Copyback This transaction is initiated by a cache who has modified the data line and wishes to evict the line from its memory.

Shared Response This transaction is initiated by a cache who has forced another module to go into a requester state (through a split or wait response). This response is sharable, others may snarf it.

Modified Response This transaction is initiated by a cache who has forced another module to go into a requester state (through a split or wait response). This response is not sharable.

The transition between line attributes in response to transactions is shown in Figure 1. The possible transactions that have caused the state changes in the diagram is given by:

1. The module completed a read shared transaction that was snarfed by another module, or it has snarfed the completed read shared transaction of another module.
2. The module completed a read shared transaction that was not snarfed by another module.
3. The module completed a read modified transaction.
4. The module may voluntarily clear the cache of a line, or the module did not snarf a read shared transaction belonging to another module, or another module initiated a read modified or invalidate transaction.
5. The module completed an invalidate transaction
6. The module may change an exclusive unmodified line to exclusive modified at any time without a bus transaction.
7. The module may change the line state to Shared Unmodified without a bus transaction, or the module snarfed the read shared transaction of another module.
8. The module removed the line from the cache (after performing a copyback transaction).
9. The module performed a copyback transaction and kept a copy of the line.
10. The module removed the line from the cache, or the module did not snarf the read share transaction of another module, or another module initiated a read modified transaction.

4 Formal Specification of Cache Coherence

In order to verify that the Futurebus+ protocol adequately maintains cache coherence we need to develop a formal model of the requirements for cache coherence. Informally, a system is coherent if each read request results in the most recently written value. As described in Section 3 the standard defines an access attribute for each cache line, which is used to determine if a cache has the most recent value of the line. The properties developed by Clarke *et al* used these

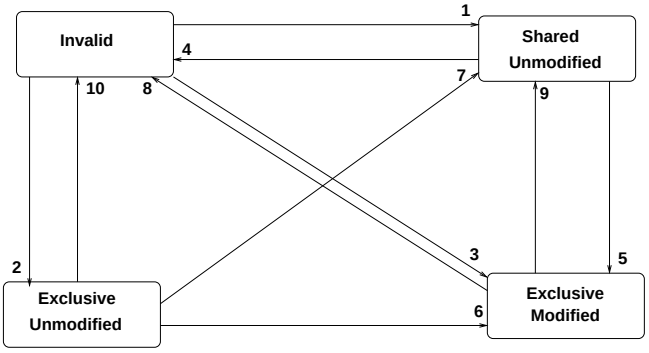

Figure 1: Each cache is required to keep an attribute related to each data line in the system. Attributes dictate the read and write permission that the cache module has to the particular data line. The attribute is updated in accordance to read and write requests originating from the processor attached to the cache and other cache modules in the system. Note that these transactions are discussed in the text.

access attributes to describe cache coherence. We have chosen a more direct approach which is based upon the requirements for cache coherence as specified by (Hennessy, J. L. & Patterson, D. A. 1996):

- A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
- A read by a processor to a location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated and no other writes to X occur between the two accesses.
- Writes to the same location are serialized: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The Futurebus+ protocol document states it "guarantees that all modules observe the modifications of a given cache line in the same order" which corresponds to the third property specified above.

5 Constructing a formal model

The protocol document specifies the cache coherence protocol both informally, in English language, and more formally using a set of boolean *attributes*. The specification of the setting and resetting of particular attributes (such as the 'shared unmodified' attribute) is very detailed and complex. Construction of an abstract model using this definition would be extremely difficult. For this reason, our model was constructed primarily from the informal English language specification with reference to the attributes when ambiguities arose. Since we were primarily concerned with the maintenance of cache coherence and also in the overall simplicity of our model, several aspects of the specification were abstracted and in some cases omitted. Furthermore, for brevity we limit our discussion to a small section of the protocol and the developed model.

The definitions and terminology used in the standard immediately suggest a number of events that can

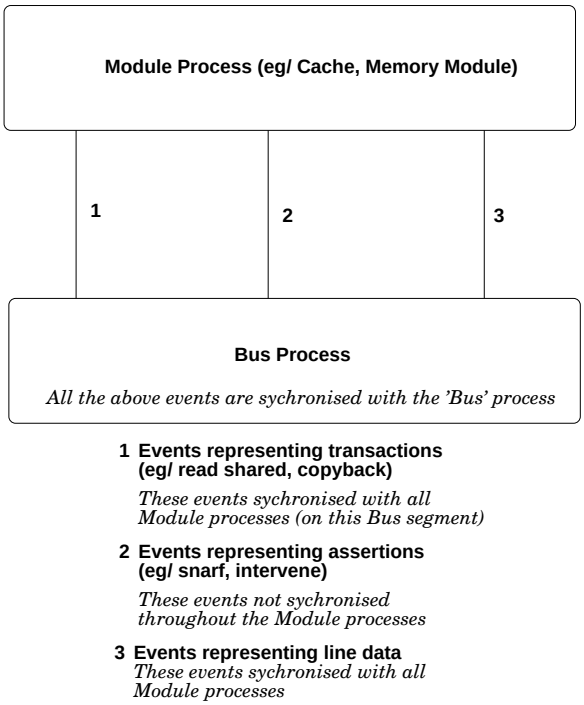

Figure 2: A Module (such as a cache) conceptually connected to a bus segment. In the CSP model of the protocol, each bus segment is modelled using a Bus process and each Module is modelled using a separate process. Event synchronization is used to simulate communication, with the Bus process helping to act as a broadcast medium.

be used to model the protocol. Each bus segment can have a number of cache modules and possibly a shared memory module. Naturally, read and write requests to a cache were modelled using events. Also, appropriate corresponding events were used to signify that a request was completed and to provide the current value of the cache line (for example, *read* and *read response*. These events were either 'generated' by a controlling process (such as a 'Processor') or by the CSP environment. They were also be used to select particular cache modules to perform a read or write request, as required in Section 4.

Since only transactions related to a single line were considered (and hence each cache needed only to be of size 1), the cache could never become 'full'. In order to abstract over the various considerations of a full cache, an event, called flush, was introduced that allowed a quiescent cache to change the status of the cache line to invalid (after performing a copy back transaction if required) at any time.

Clearly, each cache process must have access to the value and access status of the cache line it holds. This was achieved through the use of process variables. Each cache and shared memory module on the (single) bus segment had a unique identification number, which helped it distinguish bus transactions originating from other modules on the segment. These identification numbers were also accessed using process variables. An example topology is shown in Figure 2.

A cache that is either quiescent or waiting to perform a bus transaction must listen for other transactions and respond appropriately. In order to achieve this 'snooping' of the (conceptual) bus a combination of event synchronization and a monitor process (called Bus) was used. At times when the cache is required to be snooping the bus, the cache process included an external choice allowing it to accept a bus

transaction event from any other module in the system. The cache could then go into a 'response' mode and provide an appropriate response to the particular transaction. For example, a cache with a dirty copy of the cache line would be required to provide an *intervene* response.

It is unnecessary for every module in the system to synchronize on all the response events. Segments were built up by synchronizing the modules and the bus process on a set of events that represented the transactions that could be passed along the bus (for example, 'read shared'). That is, if one module wished to participate in a 'read shared' event, then all the modules and the bus had to participate in that event (thus forming a snooping mechanism). The standard specifies a number of assertions that indicate the actions of a particular cache in response to various bus requests. Since it is unknown by the originating cache how many of these assertions will be generated (or which module will be asserting them) global synchronization on these events is inappropriate. In the model developed, each module synchronized with the bus on the assertion events and the bus process monitored the assertions. Finally, the list of assertions and the transmission of the data value were synchronized globally. An example of this can be seen in Listings 1 and 2.

```
SnoopBT(i, v, s) = (
  readshared?k:others(i) -> SnoopRS(i, v, s)
  []
  readmodified?k:others(i) -> SnoopRM(i, v, s)
  []
  copyback?k:others(i) -> SnoopCB(i, v, s)
  []
  invalidate?k:others(i) -> SnoopIV(i, v, s)
  []
  sharedresponse?k:others(i) -> SnoopSR(i, v, s)
  []
  modresponse?k:others(i) -> SnoopMR(i, v, s)
)

SnoopIV(i, v, s) =
(
  assertions?wt?sp?iv?tf ->
  if (wt or sp) then Cache(i, v, s)
  else value?newvalue -> Cache(i, v, invalid)
)
```

Listing 1: A cache with no requester/responder attributes snooping the bus transactions. If an 'Invalidate' transaction is detected, the cache waits for the result of the transaction. The function *others* returns the set of identification numbers indicating the other modules on this bus segment.

```
Bus =
(
  readshared?i -> BusRS(false, false, false)
  []
  readmodified?i -> BusRM(false, false)
  []
  invalidate?i -> BusInv(false)
  []
  copyback?i -> BusCB(false, false)
  []
  sharedresponse?i -> BusSR(false)
  []
  modresponse?i -> BusMR
)

BusInv(wt) =
(
  wait -> BusInv(true)
  []
  assertions!wt!false!false!false ->
    (if wt then Bus else value?v -> Bus)
)
```

Listing 2: We see the Bus waiting for a transaction to occur. If an 'Invalidate' transaction occurs, the Bus collects the valid responses to that transaction and then 'broadcasts' the results.

The standard contains a detailed specification of a bus arbitration protocol. We abstracted over this by using the CSP choice operation $\Box$ to choose a bus master. A module that was waiting to become the bus master either accepted an event signalling the transmission over the bus of its own transaction or accepted an event signalling the transmission over the bus of a transaction originating from another module. This behavior is illustrated in Listing 3.

```
CacheIVPending(i, v, s) =
(
  invalidate.i -> PerformIV(i, v, s)
  []
  readshared?k:others(i) ->
    SnoopRSIVPending(i, v, s)
  []
  -- other possible transactions
)
```

Listing 3: Selection of a bus master is left to the CSP environment through the use of an external choice over all possible master's

6 Verification of the CSP Model

As detailed in Section 4 a cache coherency protocol has to adhere to the three conditions outlined by (Hennessy, J. L. & Patterson, D. A. 1996). Each of these conditions can be expressed elegantly in CSP and the model checked for errors by ensuring that process representing the model *refines* the process representing the property.

The first property deals with the interaction between a *single* module and a cache line. The property can be expressed in CSP as in Listing 4.

```
P1 =
(
  |~| v : VALUES @
  (|~| i: CACHES @ write.i!v -> wresponse.i.v ->
                   read.i -> rresponse.i.v -> P1
  )
)

P1_Constraint =
(
  |~| v : VALUES @
  (|~| i: CACHES @ write.i!v -> wresponse.i.v ->
                   read.i -> rresponse.i.v ->
                   P1_Constraint
  )
)
```

Listing 4: CSP expression of the first property regarding cache coherence

This property is concerned only with the modification of the cache line by a *single* cache module. The consequence of this is that we need to prevent other modules from interleaving their requests for the cache line. In order to achieve this we synchronize our model of the protocol (which allows for full interleaving) with a 'controller' process that selects a single cache module at a time to modify the cache line. Note that once the read is completed the controller may select a different cache to access the line.

The second property considers the interaction between the read and writes of pairs of modules in the system. The property states that the reads and writes need to be 'sufficiently separated'; we use the read and write response events to ensure that this is the case (since the response event is generated when the originating processor receives the data). This property can be expressed as in Listing 5.

```
P2 =
(
  |~| v :VALUES @
  (|~| i :CACHES @ write.i.v -> wresponse.i.v ->
  (|~| k :another(i) @
       read.k -> rresponse.k.v -> P2)
  )
)
```

```
P2_Constraint = (
  |~| v :VALUES @
  (|~| i :CACHES @ write.i.v -> wresponse.i.v ->
  (|~| k :another(i) @
       read.k -> rresponse.k.v -> P2_Constraint)
  )
)
```

Listing 5: CSP expression of the second property regarding cache coherence

The function *another* returns the set of modules whose identity is not equal to i, this ensures that the same processor is not selected twice (since the property explicitly states that the read is performed by a separate module than the write). Similarly from first property we are concerned here with the interaction between *pairs* of cache modules and thus we need to restrict our system from interleaving of other requests for the line that do not originate from the selected pair.

The final property (Listing 6 considers the impact of the interleaving of read and writes requests of all the modules in the system. It captures the informal specification that a read by *any* module in the system will result in the most recently written value. Since interleaving is allowed to occur within this property, no controlling process is required, and the system as a whole is verified against the property (again, the response events are used to ensure that read and writes are sufficiently separated).

```
P3 =
(
  (|~| i : CACHES @ rresponse.i?v -> P3)
  |~|
  (|~| i : CACHES @ wresponse.i?v -> P3'(v))
)

P3'(v) =
(
  (|~| i : CACHES @ rresponse.i!v -> P3'(v))
  |~|
  (|~| i : CACHES @ wresponse.i?nv -> P3'(nv))
)
```

Listing 6: CSP expression of the final property regarding cache coherence

Naturally, there a several other properties that are of interest. Clearly deadlock and livelock need to be avoided and these can be verified easily using FDR without the need to define explicit properties. The Futurebus standard also defines cases where a module may detect an 'error'. The two classes are defined 'bus error' and 'error'. In order to check that such errors do not occur we defined two events *buserror* and *error* that the cache module would perform when it detected the appropriate error. The absence of these events can easily be checked using trace semantics.

6.1 Verification Results

The errors that we detected were the same ones described by Clarke *et al*. The first error in the protocol is a violation of the third property outlined by Hennessey and Patterson (PropertyHP3 in the previous section). The error manifests itself in the Single bus segment portion of the protocol, with two cache modules. Cache A firstly obtains an exclusive unmodified copy of the cache line. Cache B then performs a read shared transaction that the memory module splits. Cache A snoops the transaction, and retains the exclusive modified copy. At this time, cache A can perform a write request and transitions to an exclusive modified cache line. Finally, memory performs a shared response transaction and cache B obtains a now stale copy of the line. The error can be fixed by forcing cache A to transition to shared unmodified when it sees that the read request was split by memory. This error can be detected in the CSP model of the protocol in only 36 states. The fix to this error suggested by Clarke *et al* satisfies this property and does not break the other requirements.

It should be noted that the properties verified in this case study differ slightly to those used by Clarke *et al*. In particular, the SMV solution used properties that related very specifically to the protocol definition, and are unsuitable to verify a non-snooping based protocol. For example, one of the properties used was: $\mathbf{AG}(p1.writable- > not(p2.readable))$ (for every cache pair p1 and p2); that is, if one cache has writable access then no other cache has a valid copy. The approach taken by this case study highlights the distinction between the formal specification of cache coherency and the specification of a cache coherency protocol. Further, it is encouraging that both approaches led to the detection of the same errors.

7 Comparison of modelling formalisms

Automatic design verification is a technique for automatically detecting errors in models of systems. The designer describes the system in some modelling formalism and then uses a verification tool upon it to verify certain properties of the system. Two main paradigms of automatic formal verification are *temporal logic model checking* and *refinement checking*. Both techniques perform a state-space exploration of the model, either on-the-fly or exhaustive. The property to be verified can be expressed in one of two ways; either as a temporal logic formula or as an abstract design expressed in the same formalism as the model. If the requirement is specified as an abstract design the verification problem is termed refinement checking.

The CSP formalism, and the tool FDR, is an example of a system that uses refinement checking; as described in Section 2 there can be a number of different types of refinement depending on the type of properties that need to be verified. SMV ("Symbolic Model Verifier") is a temporal logic model checking tool. It uses a BDD-based symbolic model checking algorithm and makes use of the CTL flavor of temporal logic. As with all automated model checking tools, when the correctness property is found to be violated, both tools offer a *counter example* facility which provides an example execution trace that illustrates why the property is false.

In order to ensure that a model is correct, it is necessary to be able to accurately express all the correctness properties. Thus, the expressiveness of the property formalism is an important consideration. There are a number of temporal logic formalisms. The temporal logic LTL (propositional linear temporal logic) requires that a formula is developed from a combination of atomic propositions, propositional operators and temporal operators. The most common temporal logic operators: "x is always true", "x will eventually be true", "x is true until y becomes true" and "x is true in the next state". Invisible infinite repetition of the last state ensures that the next state operator has a well-defined meaning in the final state. The temporal logic CTL is a temporal logic that uses the same temporal operators as LTL but requires that every temporal operator be preceded by a *path quantifier*. A path quantifier is either "x is true in all possible paths starting from the state s" or "x is true in at least one possible path starting from the state s". Unfortunately, as stated by (Valmari, A. 1998) not all LTL formulae can be expressed in CTL and visa versa. However, this drawback is counteracted by the fact that CTL is both expressive enough for most verification tasks and has an efficient model checking algorithm. Good references for the use of CTL in

model checking are (Clarke Jr, E. M., Grumberg, O. & Peled, D. A. 2000) and (Burch, J. R, Clarke Jr, E. M, McMillan, K. L, Dill, D. L & Hwang, L. J 1992).

Refinement checking problems are of the form $Sys > Spec$ where *Sys* is a model of the system and *Spec* is the requirement. The operator $>$ is a preorder over *Spec* and *Sys*. Depending on the definition of this preorder, we can specify different properties about a system. For example, in CSP the traces model specifies a particular preorder and the failures model specifies another. Further, temporal logic formulae in general do not differentiate between livelock and deadlock; CSP has an advantage in this area since failures refinement can distinguish between the two.

The CSP formalism is especially well suited to communication protocols. The Futurebus+ standard defines the bus transactions and processor commands in terms of 'signals'. This approach lends itself very easily to the definition of each signal as an event. The resulting model thus has the advantage of being very aesthetically similar to the formal specification.

8 Conclusion

We presented an alternative verification case study of the Futurebus+ cache coherency protocol. We found that the CSP formalism is well suited to modelling such protocols; the direct relation between events in the model and the definitions in the protocol allowed the model and protocol standard to be aesthetically very similar. We also found the use of parameterizations an excellent tool in the quick development of bus configurations for verification.

The CSP formalism is also well suited to the expression of the cache coherence properties specified by Hennessey and Patterson. The properties used for verification by Clarke *et al* were of a more protocol specific nature and while they are equivalent to the properties described in Section 4 they are unsuitable to be used with different types of cache coherency protocols.

Finally, the Futurebus+ protocol is a good candidate for a case study to learn about a formal verification tool. The protocol is relatively simple, yet is complicated enough to be non-trivial. A particular advantage of the protocol is the known errors that can be easily reproduced and corrected, allowing a user to develop a working knowledge of a particular verification tool in order to verify future systems.

References

Burch, J. R, Clarke Jr, E. M, McMillan, K. L, Dill, D. L & Hwang, L. J (1992), 'Symbolic model checking: 10^{20} states and beyond', *Information and Computation* **98**(2), 142–170.

Clarke Jr, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L. & Ness, L. A. (1993), Verification of the futurebus+ cache coherence protocol, *in* D. Agnew, L. Claesen & R. Camposano, eds, 'The Eleventh International Symposium on Computer Hardware Description Langauges and their Applications', Elsevier Science Publishers, Ottawa, Canada.

Clarke Jr, E. M., Grumberg, O. & Peled, D. A. (2000), *Model Checking*, The MIT Press, Cambridge, Massachusetts. London, England.

FSE (1997), *Failures-Divergence Refinement: FDR2 Manual*.

Hennessy, J. L. & Patterson, D. A. (1996), *Computer Architecture: A Quantative Approach*, second edn, Morgan Kaufmann.

Hoare, C. A. R. (1978), Communicating sequential processes, *in* 'Communications of the ACM', Vol. 21, pp. 666 – 677.

McMillan, K. L. (1992), Symbolic Model Checking: An Approach to the State Explosion Problem, PhD thesis, Carneige Mellon University.

Rajamani, S. K. (1999), New directions in refinement checking, PhD thesis, Universtiy of California at Berkley.

Roscoe, A. W. (1998), *The Theory and Practice of Concurrency*, Prentice Hall.

Valmari, A. (1998), The state explosion problem, *in* W. Reisig & G. Rozenberg, eds, 'Lectures on Petri Nets I: Basic Models', Lecture Notes in Computer Science 1491, LNCS Tutorials, Springer-Verlag.