Automatic Generation of Verifiable Cache Coherence Simulation Models from High-level Specifications

A.J. Field, P.G. Harrison, K. Kanani
{ajf,pgh,kk2}@doc.ic.ac.uk
Department of Computing
Imperial College
London SW7 2BZ U.K.
Fax: +(44) 171 581 8024

October 27, 1997

Abstract

Performance modelling and verification are vital steps in the development cycle of any cache coherency protocol. Two separate models are usually required to perform each analysis step and as protocols become increasingly complex each can become correspondingly unwieldy. We examine how stochastic process algebra can be used to describe cache coherency protocols in such a way as to allow both the correctness and performance of the protocol to be investigated together.

We reintroduce a formalism called "Spade" and show how a simulation model can be generated from a Spade description of a memory system. We also show how the coherency protocol underlying the memory system can be verified as correct by showing observational equivalence between the system and a specification for the memory system based on a memory consistency model. We demonstrate the approach by applying it to a model of the write-through coherency protocol for bus-based shared-memory multiprocessors. We validate simulation results from the model by comparing against a hand coded simulation.

Keywords: Stochastic process algebra, Verification, Performance modelling, Cache coherency, Synthesis

1 Introduction

Caching in computer systems is one of the most potent architectural techniques employed in reducing latencies associated with main memory references. In multi-processor systems where there is the possibility of having several cached copies of a memory block (sometimes referred to as a *line*), a protocol is required to ensure that the cache contents are coherent.

In this paper we consider two aspects of the design of coherency protocols, namely performance analysis and verification, using Stochastic Process Algebra (SPA) as a common framework for protocol specification. The SPA used is called "Spade" and is described in detail in [3].

We introduce a Spade compiler which generates a module containing C++ classes, one for each user defined Spade agent. This generated module can then be linked with a simulation harness to produce the final executable simulator. The intermediate stage of producing C++ code not only imparts a significant execution speed gain on the final simulator (when compared with other interpreted process algebra simulators) but also allows for user defined C/C++ functions to be incorporated as part of the final simulator. It is by this mechanism that we are able to use a precomputed or dynamically generated memory address trace to drive a workload model for the protocol.

The underlying theory of Spade also gives us the ability to be able to analyse Spade programs for properties such as equivalence with other Spade specifications. In this way we are able to verify the coherency protocol of specified memory system correct with respect to a specification for the memory model of that architecture.

This gives a more integrated approach to design of protocol since only a single model needs to be used to encompass both the verification and performance analysis stages of design. This methodology is

demonstrated by applying it to a model of the write-through coherency protocol for bus-based shared-memory multiprocessors.

The two main approaches to performance analysis, namely discrete-event simulation and analytical modelling, have been applied extensively to the analysis of cache coherency protocols. Each approach has its own benefits and drawbacks. Simulation of protocols usually takes place at a low level of abstraction. The protocol is implemented for a particular architecture and it is this system that is executed in software as the simulation. Simulation has the benefit of being conceptually simple. However, being a generic method for performance modelling, it is not easy to make rapid changes to a protocol and get feedback without detailed alteration to, and re-exeuction of, the simulation code.

This Spade formalism used here is expressive enough to model a large class of simulations, including all of those that can be described as a generalised semi-markov process. With this expressive power, however, comes the drawback that most of the models built upon this formalism are analytically intractable for the purposes of performance modelling. However, a model can still be simulated and performance results gathered that way. Approximations can be made that will allow the formulation of tractable models.

Verification of cache coherency protocols requires ensuring that the system of caches employing the protocol exhibit some specified property based on the *memory consistency model* [14] the architecture is supposed to present. At the highest level of abstraction, a cache coherency protocol can be described in terms of a finite state machine that describes the changes in state associated with a cache line as the result of some predefined events. The specification of the protocol is then a statement of the invariant that is to be maintained in terms of the global machine state defined across all of the caches. The invariant is dictated by the memory consistency model that the protocol has to satisfy. Verification is an exercise in ensuring that all reachable states satisfy the invariant.

Established verification methods tend to fall into the categories of simulation studies, reachability analysis or logical reasoning. Each method usually has its roots in the verification of communication protocols but can be specifically tailored for cache coherency.

The objective here is to use the Spade formalism to establish correctness against a high-level description of (strong) coherency. Verification is carried out by establishing a bisimulation between the protocol model and a Spade specification for the memory model. The paper is essentially a case study, showing how the techniques proposed can be applied to a simple, yet realistic, system.

The rest of the paper is organised as follows: Section 2 introduces the Spade formalism. Section 3 goes on to describe how we generate a simulation from a Spade specification. Section 4 discusses how we go about verification of Spade models. We bring these ideas together in section 5 by applying them to a bus based shared memory multiprocessor architecture employing a write through invalidation based protocol. The conclusions from the paper are laid out in Section 6.

2 SPA and the "Spade" Formalism

2.1 Stochastic process algebras

The well known process algebra (PA), CCS, [6] is the basis for the Spade SPA. An SPA is a PA that has constructs that allows the stochastic as well as the functional behavior of a system to be captured [13, 8]. The extensions to ordinary process algebras which permit this invariably include the ability to associate some sort of temporal property to the firing of actions, be it the notion that actions are able to synchronize at a certain rate (e.g PEPA) [8] , or an explicit delay between action availability (e.g Spade). Another crucial requirement is the ability to resolve non-determinism stochastically. For instance, if we take a process, CPU, which is able to offer the actions, *read*, and, *write*, to the environment, in normal process algebra (e.g. CCS) we can do no better than specify the agent as such:

$$CPU = \tau.read.CPU + \tau.write.CPU$$

The agent non-deterministically offers either a *read* or a *write* action to the environment in which it is placed. Nowhere in this specification are we able to express the desire that these two actions are offered with a certain likelihood. SPAs allow us to capture this notion either implicitly (such as both actions being offered at some rate) or explicitly (a probability is assigned to each possible transition). Thus the same example in Spade would be:

$$CPU = [p_r].read.CPU + [p_w].write.CPU$$

and in PEPA would be

$$CPU = (cp_r, read).CPU + (cp_w, write).CPU$$

where c, p_r and p_w are the memory reference rate, probability of a read and probability of a write request respectively. These two extensions to a process algebra are sufficient to allow the designer to model the performance properties of a system.

2.2 The language of Spade

The syntax of Spade is very similar to CCS and therefore we shall only point out the stochastic extensions pertinent to our model here.

First, we need a method to associate a delay between the firing of actions. This is accomplished by the *delay prefixing* construct, $(t).A$. Here, simulation time is incremented by t units (t being a real number). Such an *evolution* transition is denoted:

$$(t).A \quad {}^{t} \quad A$$

We can therefore, for example, define an agent:

$$A_2 = start.(2).stop.\mathbf{nil}$$

which will offer the action *start* and then delay for two units before offering the action *stop* for synchronization before terminating.

The normal *choice* operator, '+' also has an n-ary stochastic counterpart, *probabilistic* choice. Here, an agent itself resolves the choice according to the defined probabilities. Thus the agent

$$A_4 = [\frac{1}{3}].one.\mathbf{nil} + [\frac{1}{3}].two.\mathbf{nil} + [\frac{1}{3}].three.\mathbf{nil}$$

will 'spontaneously' resolve to offer only the actions *one*, *two* or *three* with equal likelihood. These *probabilistic* transitions are denoted:

$$A_4 \overset{\frac{1}{3}}{\hookrightarrow} one.\mathbf{nil}$$
$$A_4 \overset{\frac{1}{3}}{\hookrightarrow} two.\mathbf{nil}$$
$$A_4 \overset{\frac{1}{3}}{\hookrightarrow} three.\mathbf{nil}$$

It is used to make stochastic non-deterministic choice. Note the proviso that the total probability must sum to 1 (an action is certain to be offered). The notion of a probability associated with a transition firing is discussed in detail in [3].

The final construct that comes of use to us is the *sample* action. Transitions involving this action are *probabilistic* transitions (and like evolution transitions do not resolve sum choices). The purpose of these actions is to allow function values to be bound to variables which may be used in the definition of the agent. Thus

$$Arrival\_Process = \mathcal{R}[X \leftarrow exp(\lambda)].(X).arrival.Arrival\_Process$$

would define an agent that would offer an *arrival* action every X units of time where X is assigned from an exponential distribution parameterized by the value λ. Our main purpose for the use of this construct is to be able to call functions which have side effects in our simulator. Such side effects can be used to keep track of the simulation state or read in another memory reference from a file when the function is called, for example. These side effects do not and should not affect the process algebra code itself.

The complete formalism has other constructs which are not required in this context and will therefore not be discussed. A full account of the syntax and semantics for Spade can be found in [3].

As with the language described as 'bare' CCS in [6] we extend Spade with constructs that allow us to describe partial systems more succinctly. In particular we will make use of *agent parameterization* (e.g. $Queue(n) = arrival.Queue(n+1) + departure.Queue(n-1)$), *value passing* (e.g. $Adder = in(n).\overline{out}(n+1).Adder$), 'if-then' constructs and data-storing structures.

We will thus be able to construct agents such as:

$$\begin{aligned} HASH\_TABLE(data) \quad = \quad & in(index).(if\ data[index] \neq NULL\ then \\ & \overline{out}(data[index]).HASH\_TABLE(data) \\ & else\ \overline{error}.HASH\_TABLE(data) \\ &) \end{aligned}$$

which we would otherwise not be able to do.

These extensions have to be formulated so as not to make the theory of the formalism inconsistent. Each is usually no more than 'syntactic sugar' for a more verbose definition for the same system. A more detailed account as to the validity of these extensions can be found in [7].

3 Compiling Spade programs for simulation

The scheme we have adopted involves a two phased approach to generating the final simulator. In the first phase, the Spade source program is compiled to produce a C++ module. This module contains the C++ definitions for classes derived from the agent and function definitions in the Spade source. During this stage we are also able to check the Spade program for semantic errors and use simple transformation techniques to improve the execution speed of the final simulator.

The second stage of simulator generation involves the compilation and linking of this module with a set of predefined libraries which act as the simulation harness. The simulation harness is responsible for the construction of the data structure which represents the Spade program and application of the transition rules upon this structure. It is the repeated application of the transition rules that 'drives' the simulation. At this stage we are also in the position to link in any other user defined C++ function which we may wish to call from within our Spade code.

The result of this stage is an executable file which is the simulator. Execution of this file can generate a trace dump of transitions as they are fired and a record of the current value of performance metrics that are being monitored. Either of these outputs is optional but obviously the simulator would not be very useful if neither were produced. The relationship between these components of the simulator are shown in Figure 1.

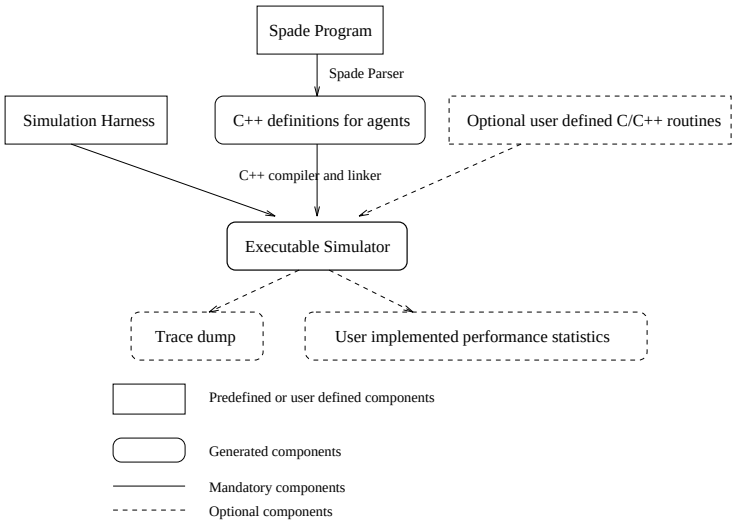

Figure 1: Organization of a Spade simulation components

Spade transitions can be categorized into three groups: *labelled* transitions ($\xrightarrow{\alpha}$) which can be thought of as mimicking internal simulation computation, *evolution* transitions (^{t}) which mimic the progression of simulation time and *probabilistic* transitions ($\overset{p}{\hookrightarrow}$) which introduce the stochastic element into the simulation.

Examining the transitional semantics of Spade, the process of simulation of a model is a cycle of resolving any probabilistic transitions, followed by firing of all possible labelled transitions at which time the system is evolved (and simulation time updated) until another probabilistic (or labelled) transition becomes possible. That is :

```
loop forever{
1. resolve probabilistic transitions.
2. resolve all synchronizations between visible actions.
3. update simulation time according to smallest time evolution.
}
```

We wish to simulate only closed systems (the simulation harness is not responsible for providing a context/environment for our system) and therefore require all labelled transitions to be a result of invisible actions (resulting from internal synchronizations).

Performance modelling of the system involves resolving how much time a system spends in a given state, and to this end we have to insert actions into our model which signal to the simulation harness the

points at which the system enters and exits these states. The state holding time is then just the difference in the global simulation time between the firing of these actions. Note that this is an invasive method of performance modelling processes as suggested in [3] but other non-invasive methods (ones which do not require modification of the model code) can be employed; these are, however, less efficient in terms of execution time [7].

The process of compilation is therefore to produce C++ classes that hold the structure representing the agent definition which can be interrogated for available transitions and evolved accordingly.

4 Verification

Verification of a model takes place in three stages. The first stage is the systematic removal of all stochastic information in the model. Since we are interested purely in the functional behaviour of our model, our notion of equivalence (which is essentially bisimulation) is too discriminating if we include such information and use the definition for bisimulation between Spade processes. We wish to equate two systems if and only if they present the environment with the same action (c.f. equivalence in the process algebra sense). We achieve this by transforming Spade (a stochastic process algebra) into CCS (a non-stochastic process algebra). We define a mapping function between Spade and CCS expressions, γ::$\mathrm{P}_{\mathrm{spade}} \longrightarrow \mathrm{P}_{\mathrm{CCS}}$, which syntactically replaces stochastic delay with non-deterministic delay and probabilistic choice with non-deterministic choice. Care has to be taken since we may have inadvertently increased the functionality of our model. Take, as an example, a process which is the composition of a process which delays for 1 time unit before evolving further ((1).$a.nil$) and one which delays for 3 time units before evolving further ((3).$b.nil$)

$$P = (1).a.nil + (3).b.nil$$

During the translation phase this process becomes

$$\gamma(P) = \tau.a.0 + \tau.b.0$$

Here our CCS process can possibly offer the actions a or b whereas our Spade process will never offer the action b due to the timeout; this is due to the fact that the 1 unit delay evolves to resolve the choice before the 3 unit delay.

$$P \overset{1}{} (0).a.nil + (2).b.nil \overset{a}{\rightarrow} nil$$

To overcome this we deal only with Spade processes that do not use stochastic behaviour in such a way. We therefore define γ such that (for the 'sum' operator)

$$\gamma(P + Q) = \gamma(P) + \gamma(Q)$$
such that if P |=<a>T and Q |=<b>T then (P + Q) |=<a>T $\wedge$ <b>T

Thus if any constituent of a sum can offer an action then the sum itself must be able to offer those actions (for some trace). This is easily decidable since only a deterministic timeout would consistently preclude the possibility of one of the actions. We can see that the P defined above does not satisfy the new constraint.

The second stage in verification is to reduce the size of the model as much as possible. The simplest way to achieve this is by abstracting out all internal actions (which we hope are implementation specific) using the τ law,

$$a.\tau.P = a.P$$

other more sophisticated methods such as *process pruning* can be used [7] but are not discussed here. The reduced model should then ideally contain only the protocol state transition information, i.e. be a description of the FSM defining the protocol. This stage does no more than assert that our implementation of the protocol satisfies the specification of the protocol.

The final stage is verification of the protocol itself; does the protocol satisfy the constraints of the memory model the architecture presents? Note that in practice this stage has to be carried out only once. If our model reduction stage produced a system that was isomorphic to the FSM machine describing the protocol then we can turn to the established method of state space enumeration for verification of such a system [4].

This may not be possible since our model may possibly describe an actual implementation of the protocol, whereas the FSM machine description is an abstract specification. In general we may not be

able to identify and eliminate all implementation specific detail. Therefore we will show observational equivalence between the reduced system and a specification system for the memory model. Here we use the knowledge of the invariant that is to be maintained in the enumeration approach in order to propose a relation between the protocol model and the memory specification model. We are then left with the task of checking to see if the relation we have defined is indeed a bisimulation. This method has the advantage of both being able to handle models which have not been fully reduced, but implicitly uses symmetries in the underlying system to reduce the cost of analysis. The analysis can also be carried out symbolically which does not restrain us by having to state the system size, something which we would have to do for the enumeration approach.

5 Case Study: a Bus-based SMMP

5.1 The architecture

We are now in a position to be able to write a complete Spade specification of a parallel computer memory architecure; our example will be for a bus-based SMMP. The memory system shall use a *write-through invalidation* based protocol to ensure that the caches are kept coherent [12].

Parallel programs are compiled into N processes, one for each processor. The caches hold copies of recently addressed items from main memory. The unit of storage in the cache is a *line* which is a collection of one or more memory *words*. Each cache has a fixed capacity of lines. The capacity of a cache is much less than that of main memory. Main memory itself is accessed in units of *words* across the *bus*.

On each memory read/write, the issuing processor stalls and the cache is inspected to see if it has a valid copy of the addressed word. This lookup takes t_{cache} units of time. If the issued request was a read and a valid copy of the word is present, then a *read hit* occurs. The word is made available to the processor and the processor resumes immediately.

On a *read miss* however, the address of the requested word is broadcast on the bus and a *snoop* takes place on each of the other caches. The snoop takes t_{snoop} units of time. If a copy of the word is present in any of the caches then a cache to cache transfer of the data is initiated. The complete line is transmitted along the bus with associated delay, t_{cc}, for each word. If the word is not present in any cache, then it must be supplied from main memory with (a larger) associated delay, t_{cm}, for each word. Only one transaction may take place along the bus at any time, thus the possibility exists for the cache having to queue for the bus until it is available.

On a *write hit*, the local cache is updated as part of the cache lookup and there is no additional delay. The address of the word is broadcast on the bus and all other caches *invalidate* their copy of the line containing the word before main memory is updated with the new value. The processor can then resume.

In the case of a *write miss*, the same events occur as that above apart from the fact that the updated line is transfered from main memory into the requesting cache before the processor is allowed to resume.

For our particular case study we use a workload defined by a memory address trace file, although is possible to link directly into a suitably augmented parallel program that generates the workload as it executes (in the this particular instance the trace was generated from the *mp3d* application [9]).

5.2 The Spade model

To build the Spade model for the system, we first decide on how to split the system into agents which we shall define. The chosen partitioning is shown in Figure 2. The system consists of a BUS agent which is responsible for serialization of accesses to main memory and broadcasting of signals, a number of $CACHE$ agents which are responsible for holding the lines with their associated validity tags and driving the protocol and a number of CPU agents which are responsible for providing the workload.

Let us begin by defining the CPU agent. This agent does no more than delay by $delay_r(n)$ or $delay_w(n)$ units of time before issuing a memory request ($\overline{Read}(address(n))$ or $\overline{Write}(address(n))$). It will stall until the request is completed (denoted by the $Resume_r/Resume_w$ actions) before resuming. Note that if $delay_r(n) < delay_w(n)$ then the issued request is a read and visa-versa. This method of using a *timeout* in choice resolution is a powerful one. The CPU agent is parameterized by its identifier.

$$\begin{aligned} CPU(n) &= (delay_r(n)).\overline{Read}(address(n)).Resume_r(d).CPU(n) \\ &+ (delay_w(n)).\overline{Write}(address(n)).Resume_w.CPU(n) \end{aligned}$$

the values for $address(n)$, $delay_r(n)$ and $delay_w(n)$ are derived from the workload trace and thus require

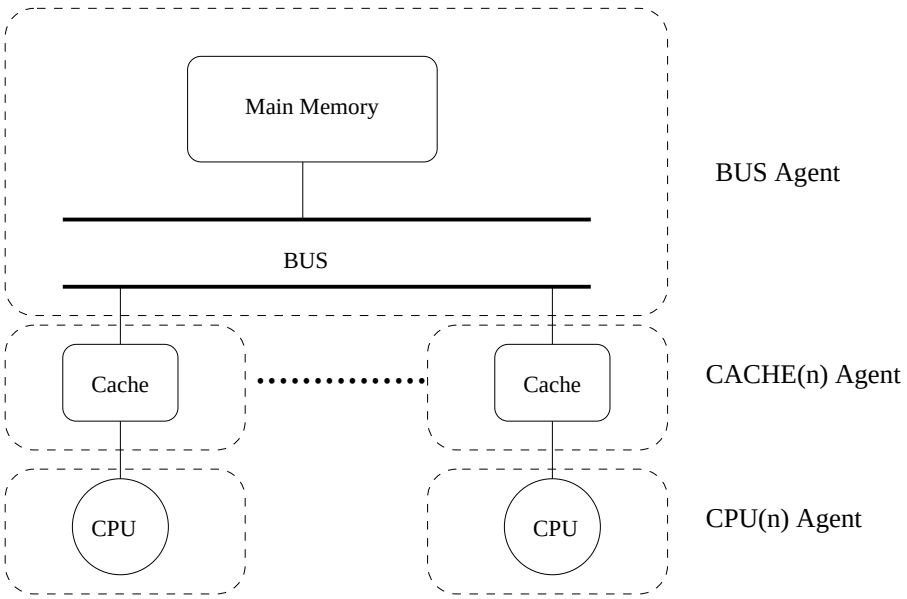

Figure 2: Organisation of Spade agents

the identity (number) of the processor as a parameter.

We define the $CACHE$ agent in terms of three other agents, $DATA$, $SNOOP$ and $PROTOCOL$, each responsible for an aspect of the caches' behaviour. The $DATA$ agent is just an agent that stores the data of the cache. This agent can be interrogated for the values associated with a given line:

$$\begin{aligned} DATA(data) &= SetVals(l,d,tg,v).DATA(data[l \mapsto (d,tg,v)]) \\ &+ GetVals(l).\overline{Vals}((data\ l)\#1,(data\ l)\#2,(data\ l)\#3).DATA(data) \end{aligned}$$

values for a given line, l, for the data contents, d, the tag associated with the line, tg and the validity of the line, v can be set with the $SetVals(l,d,tg,v)$ action and retrieved with the $GetVals(l)$ and $\overline{Vals}$ actions.

The $SNOOP$ agent is responsible for monitoring bus transactions and acting upon them if need be:

$$\begin{aligned} SNOOP(n) &= Snoop(n,a).\overline{GetVals}(find(a)).Vals(d,tg,v). \\ &\quad \overline{Vote}((v=1)and(tag(a)=tg),d).SNOOP(n) \\ &+ Inv(n,a).\overline{GetVals}(find(a)).Vals(d,tg,v). \\ &\quad (if\ tg=tag(a)\ then\ \overline{SetVals}(find(a),d,tg,0).SNOOP(n) \\ &\quad else\ SNOOP(n)) \end{aligned}$$

where the function $find$ maps an address to a cache line and the function tag maps an address to a tag for that address.

This agent will either respond to a snoop request from the bus for a particular memory address ($Snoop(n,a)$), replying with the fact that a valid value is present or not and the actual data ($\overline{Vote}$...). The agent will also accept an invalidation request from the bus for a particular address ($Inv(n,a)$). The agent will invalidate the line contents if the invalidation request was appropriate, i.e. the tag for the address to be invalidated is the same as those of the appropriate cache line.

The $PROTOCOL$ agent is responsible for driving the actual coherency protocol. It is responsible for carrying out the cache lookup and responding with the appropriate actions if a read/write hit/miss occurs. This agent employs the use of two further agents, $READING$ and $WRITING$ in order to improve the overall clarity of the code.

$$\begin{aligned} PROTOCOL(n) &= Read(a).(\mathrm{t}_{cache}).\overline{GetVals}(find(a)).Vals(d,tg,v).READING(n,a,d,tg,v) \\ &+ Write(a).(\mathrm{t}_{cache}).\overline{GetVals}(find(a)).Vals(d,tg,v).WRITING(n,a,d+1,tg,v) \end{aligned}$$

Upon a read or write request (denoted by the actions $Read$ and $Write$ respectively) the pertinent values are passed on to the $READING$ or $WRITING$ agent (as parameters) which will handle each case

separately. In the case of a read request for an invalid line (spawning the agent $READING(n,a,d,tg,0)$) the bus is first aquired via the *Lock* action. Once exclusive access to the bus has been granted a memory read request is issued ($\overline{MReadReq}(a)$) and the reply ($MRead(d1)$), the version of the block, used to update the contents of the cache. The agent then signals the fact that the CPU may restart by offering the $\overline{Resume}$ action (note that during this period the CPU attached to this cache has been waiting, offering the *Resume* action). In the case of the line being valid for the read request (spawning the agent $READING(n,a,d,tg,1)$) a tag check is carried out before either 'resuming' the CPU immediately or behaving as if a read miss had occured.

$$\begin{aligned}
READING(n,a,d,tg,0) &= Lock.\overline{MReadReq}(a).MRead(d1).\\
&\quad \overline{SetVals}(find(a),d1,tag(a),1).\overline{Resume_r(d1)}.Free.PROTOCOL(n)\\
READING(n,a,d,tg,1) &= if\ tag(a)=tg\ then\ \overline{Resume_r(d)}.PROTOCOL(n)\\
&\quad else\ READING(n,a,d,tg,0)
\end{aligned}$$

For a write request to an invalid line (spawning $WRITING(n,a,d,tg,0)$) the bus is first aquired (via *Lock*) and then the appropriate line read before an invalidation request issued. The line is then updated and the CPU signaled to restart. If the write is to a valid line (and the tag is appropriate) the block does not have to first be fetched.

$$\begin{aligned}
WRITING(n,a,d,tg,0) &= Lock.\overline{MReadReq}(a).MRead(d1).\overline{Invalidate}(a,d).\\
&\quad \overline{SetVals}(find(a),d,tag(a),1).\overline{Resume_w}.Free.PROTOCOL(n)\\
WRITING(n,a,d,tg,1) &= if\ tag(a)=tg\ then\\
&\quad Lock.\overline{Invalidate}(a,d).\overline{SetVals}(find(a),d,tag(a),1).\\
&\quad \overline{Resume_w}.Free.PROTOCOL(n)\\
&\quad else\ WRITING(n,a,d,tg,0)
\end{aligned}$$

The actual $CACHE$ agent is just a parallel composition of these three agents (with appropriate restrictions to ensure that certain communications are along private channels and do not interfere with the workings of other agents).

$$CACHE(n) = (DATA(data)|PROTOCOL(n)|SNOOP(n))\backslash\{GetVals, SetVals, Vals\}$$

The final agent that defines the system is BUS. The purpose of this agent is to mimic the actions of the main memory bus and contents of main memory. It serializes access to it by providing a semaphore type locking mechanism for access to its 'functions'. The action *Lock* must be fired by a cache before other actions are made available to it.

Once 'locked' the bus is available to satisfy a memory request, invalidation request or be 'freed' (unlocked so that another cache may acquire it).

$$\begin{aligned}
BUSF &= \overline{Lock}.BUSL\\
BUSL &= MReadReq(a).BUSSNP(0,a,0,0) + Invalidate(a,d).BUSB(0,a,d)\\
&\quad + \overline{Free}.BUSF
\end{aligned}$$

When a memory request or an invalidation request is issued to the bus, an appropriate signal has to be broadcast to all of the other caches on the bus. Due to the fact that synchronizations between Spade agents are pairwise only (unlike, for example, in CSP [10] where multiple agents can synchronize in a single transition), this broadcasting has to be emulated using a series of pairwise synchronizations. This functionality is implemented in the $BUSSNP$ and $BUSB$ agents. The $BUSSNP$ agent is responsible for gathering 'snooping' information from each cache on a snooping event and returning the data for that memory address.

$$\begin{aligned}
BUSSNP(n,a,d,x) &= \overline{Snoop}(n,a).Vote(v,data).BUSSNP(n+1,a,data,x\ or\ v)\\
BUSSNP(7,a,d,0) &= \overline{Snoop}(7,a).(Vote(0,data).(\mathrm{t}_{snoop}+\mathrm{t}_{cm}).\overline{MRead}(getMem(a)).BUSL\\
&\quad + Vote(1,data).(\mathrm{t}_{snoop}+\mathrm{t}_{cc}).\overline{MRead}(data).BUSL)\\
BUSSNP(7,a,d,1) &= \overline{Snoop}(7,a).Vote(v,data).(\mathrm{t}_{snoop}+\mathrm{t}_{cc}).\overline{MRead}(data).BUSL
\end{aligned}$$

The delay associated with the memory request will depend on whether a valid copy of the data was found during snooping (a cache-cache transfer will take place) or not (a cache-memory will take place). This

is checked by an accumulated parameter, x, part of the $BUSSNP$ agent. During the broadcast phase of the bus (when $0 \leq n < 7$) this value is logically *or*'d with the validity state of the line in the current snooping cache (cache number n). The correct delay, according to whether the transfer is cache-cache or cache-mem, is introduced into the system by $BUSSNP(7, \cdots$, the final stage of broadcast.

The $BUSB$ agent is responsible for issuing an invalidation request ($\overline{Inv}(n, a)$) to each cache and updating main memory ($\mathcal{R}[tmp \leftarrow setMem(a, d)]$).

$$\begin{aligned} BUSB(n, a, d) &= \overline{Inv}(n, a).BUSB(n+1, a, d) \\ BUSB(7, a, d) &= \overline{Inv}(7, a).(\mathrm{t}_{snoop} + \mathrm{t}_{cm}).\mathcal{R}[tmp \leftarrow setMem(a, d)].BUSL \end{aligned}$$

The system as a whole, consisting of 8 nodes, can now be defined as

$$\begin{aligned} NODE(n) &= (CPU(n) \mid CACHE(n)) \backslash \{Read, Write, Resume\} \\ SYSTEM &= NODE(0) \mid NODE(1) \mid NODE(2) \mid NODE(3) \mid NODE(4) \mid NODE(5) \mid \\ &\quad NODE(6) \mid NODE(7) \mid BUSF \end{aligned}$$

the agent $SYSTEM$ is the *top level* agent and is thus evolved/simulated by the simulation harness.

5.3 Model verification

We wish to verify that the protocol we have described has the properties required for a sequentially consistent memory model. We begin by giving a CCS specification for sequentially coherent system [14].

In order to meet the requirements for sequential consistency the memory model has to maintain the number of reads or writes pending to a particular memory location [15] and the current version of the datum in the location [4]. A completed write to the memory location will increase the version number associated with that location by one and decrease the number of outstanding writes to that location by one. A completed read to a location decreases the number of outstanding reads to that location by one but leaves the version number of that location unchanged.

For the sake of simplicity define a specification system comprising two memory locations, **a** and **b**, (each of which map on to the same cache line). Since the cache lines are independent of one another as far as the protocol is concerned and that a conflict for a particular cache line occurs between two location only at any one time, we do not need a system larger than this to capture the essence of the system pertaining to verification.

Thus the parameterisation of the process has the following interpretation:

roa, *rob* is the number of outstanding reads to location **a**, **b** respectively.
woa, *wob* is the number of outstanding writes to location **a**, **b** respectively.
va, *vb* is the latest version of **a**, **b** respectively..

For sequential consistency the system has to behave as follows:

$$\begin{aligned} &SYSTEM[roa, rob, woa, wob, va, vb] = \\ &Read(a).SYSTEM[roa+1, rob, woa, wob, va, vb] \\ &+Read(b).SYSTEM[roa, rob+1, woa, wob, va, vb] \\ &+Write(a).SYSTEM[roa, rob, woa+1, wob, va, vb] \\ &+Write(b).SYSTEM[roa, rob, woa, wob+1, va, vb] \\ &+Resume_r(va).SYSTEM[roa-1, rob, woa, wob, va, vb] \\ &+Resume_r(vb).SYSTEM[roa, rob-1, woa, wob, va, vb] \\ &+Resume_w.SYSTEM[roa, rob, woa-1, wob, va+1, vb] \\ &+Resume_w.SYSTEM[roa, rob, woa, wob-1, va, vb+1] \end{aligned}$$

For verification we aim to present a (weak) bisimulation containing the specified system and the implemented one. We will use the required invariant condition that only lines that are in the state *Valid* will contain fresh data which is identical to the contents of main memory (va or vb). Lines that are in the *Invalid* state contain obsolete data.

If we define, for the sake of clarity, the process $C_v(n, s, buf, < Ca, v, t, st >)$,
where $s \in \{Idle, ReadMiss, WriteHit, WriteMiss\}$ and $st \in \{Valid, Invalid\}$ and $< \cdots >$ represents a

group of tupled values, as :

$$\begin{aligned}
C_v(n, Idle, buf, < ca, v, t, st >) &= DATA(data)|SNOOP(n)|PROTOCOL(n) \\
C_v(n, ReadMiss, buf, < ca, v, t, st >) &= DATA(data)|SNOOP(n)|READING(n, buf, d, t, st) \\
C_v(n, WriteMiss, buf, < ca, v, t, st >) &= DATA(data)|SNOOP(n)|WRITING(n, buf, d, t, 0) \\
C_v(n, WriteHit, buf, < ca, v, t, st >) &= DATA(data)|SNOOP(n)|WRITING(n, buf, d, t, 1)
\end{aligned}$$

where $data = \{ca \mapsto (v, t, st)\}$.

We can begin to define a possible bisimulation relation. The full bisimulation contains the intermediate processes that are present during a bus transaction. Since we have implemented bus transactions as atomic with respect to one an other, we have omitted them as part of the bisimulation for the sake of clarity. Note that this technique of removing transient (as far as verification is concerned) processes can be used to reduce model size. We first have to be able to identify them systematically.

$$\begin{aligned}
&C(n_1 \cdots n_{21}, va, vb) = \\
&(C_v[*, Idle, 0, < *, *, Invalid >]^{n_1} \\
&|C_v[*, Idle, 0, < va, 0, Valid >]^{n_2} \\
&|C_v[*, Idle, 0, < vb, 1, Valid >]^{n_3} \\
&\\
&|C_v[*, ReadMiss, a, < *, *, Invalid >]^{n_4} \\
&|C_v[*, WriteMiss, a, < *, *, Invalid >]^{n_5} \\
&|C_v[*, WriteHit, a, < *, *, Invalid >]^{n_6} \\
&\\
&|C_v[*, ReadMiss, b, < *, *, Invalid >]^{n_7} \\
&|C_v[*, WriteMiss, b, < *, *, Invalid >]^{n_8} \\
&|C_v[*, WriteHit, b, < *, *, Invalid >]^{n_9} \\
&\\
&|C_v[*, ReadMiss, a, < va, 0, Valid >]^{n_{10}} \\
&|C_v[*, WriteMiss, a, < va, 0, Valid >]^{n_{11}} \\
&|C_v[*, WriteHit, a, < va, 0, Valid >]^{n_{12}} \\
&\\
&|C_v[*, ReadMiss, b, < va, 0, Valid >]^{n_{13}} \\
&|C_v[*, WriteMiss, b, < va, 0, Valid >]^{n_{14}} \\
&|C_v[*, WriteHit, b, < va, 0, Valid >]^{n_{15}} \\
&\\
&|C_v[*, ReadMiss, a, < vb, 1, Valid >]^{n_{16}} \\
&|C_v[*, WriteMiss, a, < vb, 1, Valid >]^{n_{17}} \\
&|C_v[*, WriteHit, a, < vb, 1, Valid >]^{n_{18}} \\
&\\
&|C_v[*, ReadMiss, b, < vb, 1, Valid >]^{n_{19}} \\
&|C_v[*, WriteMiss, b, < vb, 1, Valid >]^{n_{20}} \\
&|C_v[*, WriteHit, b, < vb, 1, Valid >]^{n_{21}}) \\
&\\
&|BUSF
\end{aligned}$$

where P^n means $P \mid \cdots \text{n times} \cdots \mid P$ and $*$ matches any symbol.

We can then define a relation which we hope to be a bisimulation as:

$$\begin{aligned}
&\{(C(n_1 \cdots n_{21}, va, vb), \\
&SYSTEM[n_4 + n_{10} + n_{16}, n_7 + n_{13} + n_{19}, \\
&n_5 + n_6 + n_{11} + n_{12} + n_{17} + n_{18}, n_8 + n_9 + n_{14} + n_{15} + n_{20} + n_{21}, \\
&va, vb])\}
\end{aligned}$$

Note that this relation is actually defined symbolically and to be able to enumerate it we would require the total number of caches.

This relation has to be verified as being indeed a (weak) bisimulation. Since the processes are stable, we can then assert that they are equivalent and that the protocol meets the necessary requirements for strong consistency.

If we recall the property a relation has to hold to be a weak bisimulation [6]: A binary relation $S \subseteq P_{CCS} \times P_{CCS}$ over agents is a *(weak) bisimulation* if $(P, Q) \in S$ implies for all $\alpha \in Act$

1. Whenever $P \stackrel{\alpha}{\rightarrow} P'$ then, for some Q', $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $(P', Q') \in S$

2. Whenever $Q \stackrel{\alpha}{\rightarrow} Q'$ then, for some P', $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ and $(P', Q') \in S$

all we have to do is check to see if this property holds for our defined relation. We demonstrate the principle for only one action here; the others follow similarly.
We see that

$$C(n_1 \cdots n_{21}, va, vb) \stackrel{Read(a)}{\longrightarrow} C((n_1 - 1) \cdots (n_4 + 1) \cdots n_{21}, va, vb)$$

and

$$\begin{aligned} &SYSTEM[\ n_4 + n_{10} + n_{16}, n_7 + n_{13} + n_{19}, \\ &\qquad n_5 + n_6 + n_{11} + n_{12} + n_{17} + n_{18}, \\ &\qquad n_8 + n_9 + n_{14} + n_{15} + n_{20} + n_{21}, va, vb] \\ &\stackrel{Read(a)}{\longrightarrow} \\ &SYSTEM[(n_4 + 1) + n_{10} + n_{16}, n_7 + n_{13} + n_{19}, \\ &\qquad n_5 + n_6 + n_{11} + n_{12} + n_{17} + n_{18}, n_8 + n_9 + n_{14} + n_{15} + n_{20} + n_{21}, va, vb] \end{aligned}$$

with $(C((n_1 - 1) \cdots (n_4 + 1) \cdots n_{21}, va, vb), SYSTEM[(n_4 + 1) + n_{10} + n_{16}, n_7 + n_{13} + n_{19}, n_5 + n_6 + n_{11} + n_{12} + n_{17} + n_{18}, n_8 + n_9 + n_{14} + n_{15} + n_{20} + n_{21}, va, vb])$
in the relation.

This process of checking the derivations can be carried out for all of the enabled actions of $C \ldots$ and $SYSTEM \ldots$ which define our candidate bisimulation.

5.4 Simulation

The Spade program was used to generate an executable simulator using the approach described in section 3. The model was executed and performance results (cache hit rate and cache response time) were gathered. These results were validated by comparison with a hand-coded C version of the system. It is worth mentioning that the C version of the simulator consisted of 301 lines of code (excluding library code) and the Spade version only 34. The execution speed of the Spade simulator was slower than the C version (as could be expected), taking 22 seconds to complete as opposed to 3, but compares very well with other PA simulators such as CWB [11] in terms of the number of transitions fired every second (several thousand v. a few dozen).

The issue of simulator speed is not deemed important at this stage since the simulator harness and Spade compiler are in the early stages of development and as such can be thought of as being a prototype. Many powerful facilities can be employed to improve execution efficiency.

This problem can be tackled from two directions: improving the Spade code and improving the implementation of the simulation harness. Improvement of the Spade code should result in fewer transitions needing to be fired in order to mimick a particular simulation event. Such techiniques such as static code analysis, program transformation and partial evaluation can be employed to reduce the size of the Spade agent (which also aids the verification process). The optimisation of the simulation harness itself will improve the number of Spade transitions that can be fired per unit time and therefore effectively decrease the time in which a particular simulation run will last. Such optimisations involve either the improvement of the alogrithms which implement agent simulation (finding-choosing-firing transitions [7]) such as the prevention of recalculating unnecessary transitions each time around the simulation cycle, or more efficient implementation of the actual algorithms, e.g. improving representations of agents as datastructures, memory allocation and garbage collection.

6 Summary and Conclusions

We have demonstrated how a simple cache coherence protocol can be modelled using the stochastic process algebra Spade. We used a Spade description of a bus based SMMP employing a write through invalidation based coherency protocol to generate a simulation for the system. This simulation was executed and the performance results generated validated against a hand coded C simulation of the same system. The protocol was also shown to maintain strongy coherency by generating a bisimulation relation between it and a specification for the memory model using knowledge of global invariants. This methodology can, and has been, extended to cover other cache coherency protocols as reported in [12].

The notion of equivalence (weak bisimulation congruence) used in this example may not be the only notion of equivalence that maybe be valid. Work has still to be done in identifying perhaps weaker notions of equivalence which may still be sound for equating coherency models. Progress is also being made in implementing a more efficient optimising compiler for Spade and the development of a high level specification language for coherency protocols as an aid to their systematic development.

References

[1] E. Ametistova, I. Mitrani. *Modelling and evaluation of cache coherency protocols in multi- processor systems.* 9th UK performance engineering workshop, 1993

[2] A. Bennett, A. Field, P. Harrison. *Modelling and validation of shared memory coherency protocols.* Performance Evaluation 27&28 541-563, 1996

[3] B.Strulo. *Process algebra for discrete event simulation.* PhD Thesis Imperial college October 1993

[4] F. Pong, M. Dubois. *The verification of cache coherence protocols.* 5th Annual Symp. on Parallel algorithms and architectures 1993

[5] A.K. Nanda and L.N. Bhuyan . *A formal specification and verification technique for cache coherency protocol*s. Proc of the 1992 Int. Conf. On Par. Proc. pp I22-I26

[6] R. Milner. *Communication and concurrency.* Prentice hall, 1989

[7] K. Kanani. *Performance modelling and verification of cache coherency protocols using stochastic process algebra.* PhD Thesis Imperial college. To Appear.

[8] J. Hillston, *A Compositional approach to performance modelling.* PhD Thesis, University of Edinburgh, 1994

[9] S.C. Woo, M. Ohara, Et. al. *the SPLASH-2 programs: charecterization and methodological consideration.* 22nd Annual International Symposium on Computer Architecture, June 1995.

[10] C.A.R.Hoare *Communicating Sequential Processes.* Prentice-Hall, 1985.

[11] F.Moller, P.Stevens, *The Edinburgh Concurrency Workbench* Dept. of Computer Science, University of Edinburgh, 1994.

[12] P. Stenström, *A Survey of cache coherence schemes for multiprocessors* IEEE Computer, June 1990.

[13] N. Götz, U. Herzog and M. Rettlebach. *Multiprocessors and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras.* Proc. of 16th Int'l Symposium on Computer Performance Modelling. Measurement and Evaluation, PERFORMANCE '93. LNCS 729.

[14] S.V. Adve, K. Gharachorloo. *Shared memory consistency models: A tutorial.* IEEE Computer. Dec. 1996

[15] J.R.Goodman *Cache consistency and sequential consistency.* IEEE SCI work group, Tech. report 61, March 1989.