SIMULATION OnlineFirst, published on October 22, 2009 as doi:10.1177/0037549709349843

Specification-based Verification in a Distributed Shared Memory Simulation Model

Worawan Marurngsith
Department of Computer Science,
Faculty of Science and Technology,
Thammasat University,
Pathum Thani, 12121,
Thailand
wmrs@cs.tu.ac.th

Roland N. Ibbett
Edinburgh Parallel Computing Centre,
University of Edinburgh,
Edinburgh, UK
R.N.Ibbett@ed.ac.uk

The emergence of chip multiprocessors is leading to rapid advances in hardware and software systems to provide distributed shared memory (DSM) programming models, so-called DSM systems. A DSM system provides programming advantages within a scalable and cost-effective hardware solution. This benefit derives from the fact that a DSM system creates a shared-memory abstraction on top of a distributed-memory machine by caching data replicas locally. In this respect, a coherence protocol is a vital component responsible for assuring data consistency across all replicas. The design of coherence protocols impacts a DSM system in terms of both performance and accuracy. Performance is often measured via simulation and various verification techniques have been proposed to deal with protocol accuracy. Nevertheless, integrating accuracy verification into a DSM cluster simulation to ensure correct simulation results is still an open issue.

In this paper, we address three properties of a coherence protocol (safety, liveness, and inclusion) without which errors may occur in the simulation results. We propose a specification-based parameter–model interaction (SPMI) technique to detect these cases in a particular DSM cluster simulator called DSiMCluster. Our experimental results demonstrate that with SPMI, DSiMCluster can ensure the coherence protocol properties and provides a correct reflection of memory characteristics in shared-memory and DSM multiprocessors.

Keywords: distributed shared memory, DSM cluster, coherence protocol, verification technique.

1. Introduction

Recent advances in providing distributed shared memory (DSM) systems on clusters of multi-cores have shifted towards performance optimization using innovative coherence protocols [1,2]. This phenomenon epitomizes the rapid changes in the design and complexity of architectural alternatives. It also imposes a constraint on simulation methodologies not only to cover new designs but also to ensure the correctness of emulating these designs. This is because, in order to have confidence in the simulation results, it is often more time consuming to verify the correctness of behavioral emulations than to design them.

Over the years, a number of techniques have been proposed to verify that coherence protocols will behave in accord with their specifications [3–5] and will maintain two crucial properties, safety and liveness [6]. The *safety* (or *soundness*) property means that the protocols always guarantee data consistency, while the *liveness* property assures

SIMULATION, Vol. 00, Issue 0, Xxxxxxx 2009 000–000

DOI: 10.1177/0037549709349843
Figures 1–14 appear in color online: http://sim.sagepub.com

that there is no deadlock or livelock during the protocol's state transitions. *Deadlock* is a situation in which two or more caches are indefinitely blocked while each of them waits for resources or acknowledgments to be released from another cache. Similarly, *livelock* is a situation in which one or more caches are prevented from proceeding further, yet each stays indefinitely in a state with no exit. A survey of coherence-protocol verification techniques based on three widely accepted approaches (i.e. state enumeration, (symbolic) model checking, and symbolic state models) has shown the maturity in methodology to verify that a protocol specification has inherent safety and liveness [6]. Moreover, some recent approaches [5, 7–10] have extended one of these methodologies to verify more complex protocols, e.g. adaptive or hierarchical protocol or directory-based protocols [11, 12]. In effect, this work allows the proof of correctness of a protocol specification by focusing on the protocol semantics without any consideration of architectural behavior.

Nevertheless, architectural characteristics, such as the inclusion property, also play an important role in the correctness of execution results [13]. In a multiprocessor system, a multiple-level cache hierarchy has an *inclusion* or *multi-level inclusion (MLI)* property if 'the contents of a cache at level $i + 1$, C_{i+1}, is a superset of the contents of all its child caches, C_i, at level i' [14]. Therefore, when a coherence protocol *invalidates* the content of C_{i+1}, the corresponding content in C_i should also be invalidated. Subsequently, this content should not be seen by the processor.

A technique to ensure that the interactions of simulation components are coordinated around specific collaboration constraints in coupled discrete event simulation (DEVS) models showed the possibility of verifying model behavior on the fly [15]. Simulation-based research in coherence protocol characterization [16] compared simulation results against measurements on a real system in order to confirm that its simulation result was correct. Although there are ways to include, in a simulation model an automatic verification technique (in [17,18] for example), a technique to verify all three properties (safety, liveness, and MLI) in a protocol specification and also to direct a simulation based on the specification semantics, is not yet available.

The analysis of related works has shown the feasibility of developing a mechanism to verify a simulated component during a simulation run. This on-the-fly verification requires two components: (a) a well-formed specification of a simulated component and (b) a mechanism to obtain its semantics from the specification. The research described here addresses the verification technique of a particular component in the DSM simulation, which emulates different bus-based coherence protocols. From the related works discussed above, some of the safety and liveness properties can be verified by checking the specification. However, the full extent of the emulated behavior has to be verified during a simulation run by testing the simulated results against some verification rules. To allow this runtime testing, the functionalities of a protocol and the assertions of the safety, liveness, and inclusion properties have to be explicitly defined.

We propose a specification-based parameter–model interaction (SPMI) technique to detect these cases in a particular DSM cluster simulator called DSiMCluster [19]. The rest of the paper is organized as follows. The next section introduces the most widely used bus-based protocol, Illinois, as an example to be used in the rest of the paper. Section 3 presents a text-based description language to identify the coherence protocol specification used in DSiMCluster. In Section 4 we describe the techniques used early in the design process to verify the specification of the coherence protocols. Section 5 describes the ways to apply the verified specification to drive a simulation. Section 6 presents experimental results.

2. Illinois Protocol

The Illinois protocol (also known as the MESI protocol) is the most common protocol that supports write-back cache; it is widely used in multi-core processors [20]. The protocol is a write-invalidate protocol with four states, namely: Invalid, Exclusive[1], Private Dirty[2], and Shared[3]. A cache line in either the Exclusive or Private Dirty state owns the data. Cache lines in the Shared state do not have a specific owner. In an ownership-based protocol, the owner forwards its data to a requester. This reduces the time to access data in comparison to transferring it from the main memory.

The state machine of the Illinois protocol is shown in Figure 1. Each state in the state diagram refers to the state of a physical cache line (or cache entry) implemented in the Cache Entity simulation in DSiMCluster. All cache lines are initially marked as Invalid. When an access miss occurs to an Invalid line, data can be supplied either from a remote cache or the main memory. If the data is supplied from a remote cache, a read access causes the cache line to be marked as Shared, while a write causes the line to be marked as Private Dirty. In either case, the previous owner changes its state to Shared or Invalid, respectively. If the previous owner was in the Private Dirty state, the data is also written back to memory at the same time as the data is supplied to the requester. If data is supplied from the main memory, the requesting cache line becomes the owner, while the state is marked as Exclusive for a read and as Private Dirty for a write.

For a write hit to a cache line in one of the owner states (either in the Exclusive or Private Dirty state), the update proceeds without delay. However, if the line is shared (in Shared state), the update must be performed after the other caches have marked their lines Invalid (i.e. it waits for the

1. Also called *Valid Exclusive* or *Read Private.*
2. Also called *Modified.*
3. Also called *Read Shared.*

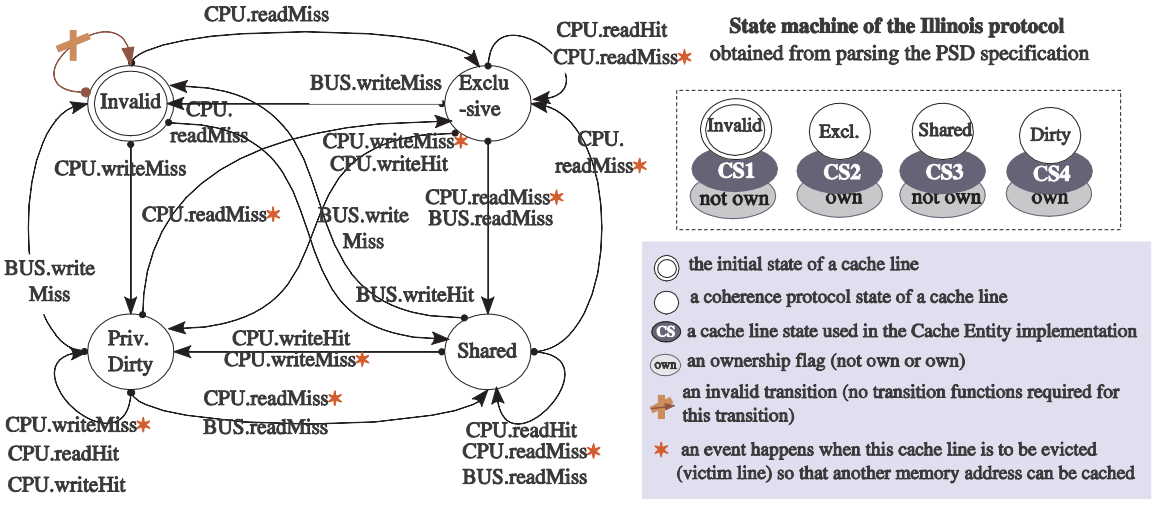


Figure 1. Cache state transitions for Illinois cache coherency protocol.

positive acknowledgments). The state of the updated line is Private Dirty and the line becomes the owner.

From the definition of the Illinois protocol, there are four possible sources of data inconsistency that can be observed from the state of replica lines [7]. Firstly, if a Private Dirty cache line co-exists with a Shared line, one of them can hold incorrect data. Secondly, if two or more cache lines are in the Exclusive state at the same time, the replicas can be written concurrently causing an inconsistent view of the value. Thirdly, if two or more cache lines are in the Private Dirty states at the same time, those cache lines will have been written concurrently yet may hold different data values. Lastly, if an Exclusive cache line co-exists with a Shared line, one cache can be written while another is reading, causing inconsistent values.

3. Protocol Specification

Fundamental to protocol verification is a symbolic or formal representation to describe a protocol specification which can be used for semantic comparison against the actual behavior. Recent research on coherence protocol verification has employed various specification languages and tools relating to the techniques exploited, for example, Murφ [21], the *Spade* formalism [4], Symbolic State Models (SSM) [8], *TLA+* [10], a table-based specification [5], the extended finite state machine (EFSMs) [7], and symbolic model verifier (SMV)-based languages [11]. In general, a specification describes a coherence protocol as a composition of three components: a finite set of states, a finite set of actions or events, and a transition relation. In EFSMs, the global conditions and a description of global conditional actions are included. The conditional action is used to express the actions in some protocols in which different sources of a new cache line (i.e. from main memory or from a remote cache) cause a transition to different outcome states. Moreover, the global condition is used to describe the permissible *global states* that facilitate protocol verification. A protocol global state, i.e. the collection of individual cache states [7], denotes the reachable state of all cache replicas as the outcome of a state transition. Thus, a record of global states can identify possible sources of data inconsistency that can negate the safety property.

Existing representations are sufficient to represent the state machine of a protocol, thus allowing the possible state transitions to be verified. However, as this work aims to find a way to use the verified specification as a script (or a mapping function) to directly drive a simulation, some practical aspects of the protocol should also be included. The attributes essential to describing an implementation of a coherence protocol in a simulation model are (a) the coherence protocol actions that are based on a priority test of cache line ownership; (b) the state when a protocol transition is halted, waiting for bus arbitration or held while waiting for an acknowledgment; and (c) the mapping functions needed to map the specification term to the implementation term and direct a simulation using these functions.

3.1 Description Language

To bridge this gap, a *protocol state-transition description* (PSD), a text-based description language designed for describing both the pragmatic and semantic attributes of a coherence protocol, has been developed. In semantic terms, the PSD language describes a state machine of a coherence protocol as a composition of a finite set of states, a finite set of actions or events, and a transition relation, similar to the existing work described above. The

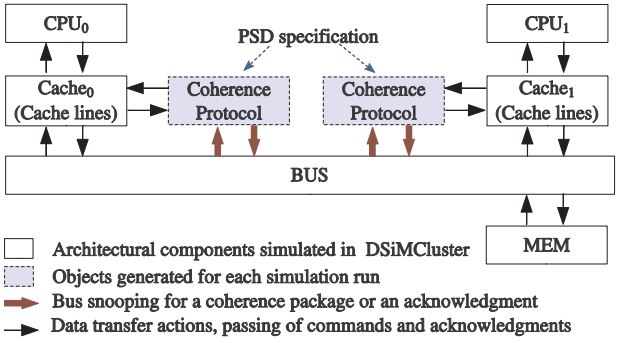


Figure 2. Organization of coherence protocols and components in a 2-node SMP.

global conditions and conditional actions as presented in EFSMs are also included in PSD. In pragmatic terms, the PSD language can express the actions between a coherence protocol and the interconnected components. Figure 2 shows the interconnection between two coherence protocols and the framework components in a two-node Symmetric Multiprocessors (SMP). Each coherence protocol is connected to a cache and the shared bus. To maintain cache coherence, a protocol snoops for read/write events from the shared bus and also receives read/write events from local caches.

In PSD, there are four groups of reserved words describing: the (architectural) components (e.g. `BUS`, `CPU`, and `MEM`); the predefined states of a cache line; the accepted events; and the transition-function identifiers. The component reserved words are used to specify the source or destination of an event, e.g. `CPU` identities that the events are initiated from the CPU that is connected to the local cache. Five predefined states of a cache line are used as the generic states to map with any user-defined protocol states. Read and write accesses to a cache line cause eight possible events to be received by a coherence protocol (Figure 3, b.1). These events identify the location of the access (i.e. `CPU` for local caches and `BUS` for remote caches). The transition-function identifiers are the predefined internal transition functions. Each of these function identifiers is used to describe a non-atomic action (or a partially executed action) to be performed in order to transit from one protocol state to another. These functions are then mapped to the implementation of a coherence protocol in a simulation (see Section 4.1).

A unit of the PSD called a *protocol definition* is a text file describing one particular protocol (Figure 3). Structurally speaking, a protocol definition comprises three sections, namely, a header, a list of protocol states, and a verification definition. As highlighted in Figure 3, each of these sections begins with a corresponding tag, followed by its body, indicated by curly braces[4]. The complete lexical and grammar rules of the PSD language are presented in [22].

The first section, *Header*, is a declaration part that introduces names into the protocol definition. It comprises four statements declaring the protocol name, state names, ownership definition, and cache-to-protocol state-mapping definition. Conceptually, the components of a protocol obtained from a Header section are presented in Figure 4. This figure shows that a Header section introduces the protocol states (S1–S4). It also maps each protocol state to the corresponding cache state (CS1–CS4) and defines whether the state implies ownership of a cache line. A cache state, `gb_INVALID`, (the CS1state in the figure) identifies the initial state of all cache lines; this is also used as the initial state of a protocol. The Header section of the Illinois protocol shown in Figure 3, a, lists the four states of the protocol. The ownership flag (`yes/no`) is mapped in the order defined in the protocol state list.

The second section, *State*, describes the actions or events of each coherence state using Rule, Ignored, and Priority statements. Conceptually, the State section is used to construct the state machine of a protocol, as shown in Figure 5. In this figure, the initial state *S1* can transit to states *S2*, *S3*, or *S4* when *Rule1* or *Rule2* is invoked. For example, when an event *EV1* arrives at a cache line in state *S1*, according to *Rule1*, the transition function *F3* must be performed first. The subsequent transition functions and the outcome state depend on whether the predicate test of the conditional function *F3* is true or false. If the predicate test is true, the protocol will perform functions *F4* and *F11* prior to changing the state of the cache line from *S1* to *S2*. Otherwise, the protocol will perform functions *F12* and *F11* before changing the cache line state from *S1* to *S3*.

Figure 3, b.1–b.3, shows a State section describing the 'Shared' state of the Illinois protocol. This State section describes the state machine shown conceptually in Figure 5. A Rule statement defines a selection of transition rules based on an incoming event from a particular component. The body of a Rule indicates the names and number of transition functions a protocol has to perform before reaching an outcome state. Transition function names are reserved words that can be declared using either a conditional or unconditional form. A conditional transition function must be followed by round braces embracing a predicate declaration, the name of functions to be chosen according to the predicate test result. An unconditional function, on the other hand, can omit this part. Short descriptions of a conditional transition function (the Predicate statement and the True/False functions) and a priority declaration (the Priority function and Priority statement) are also given.

The last section (Figure 3, c), *Verification*, is used as a reference for the verification of the coherence protocol both during the parsing steps and also during a simulation run. This part comprises two statements declaring a set of invalid global states and a set of invalid transitions. The

4. A double slash, //, begins a comment that extends to the end of the line.

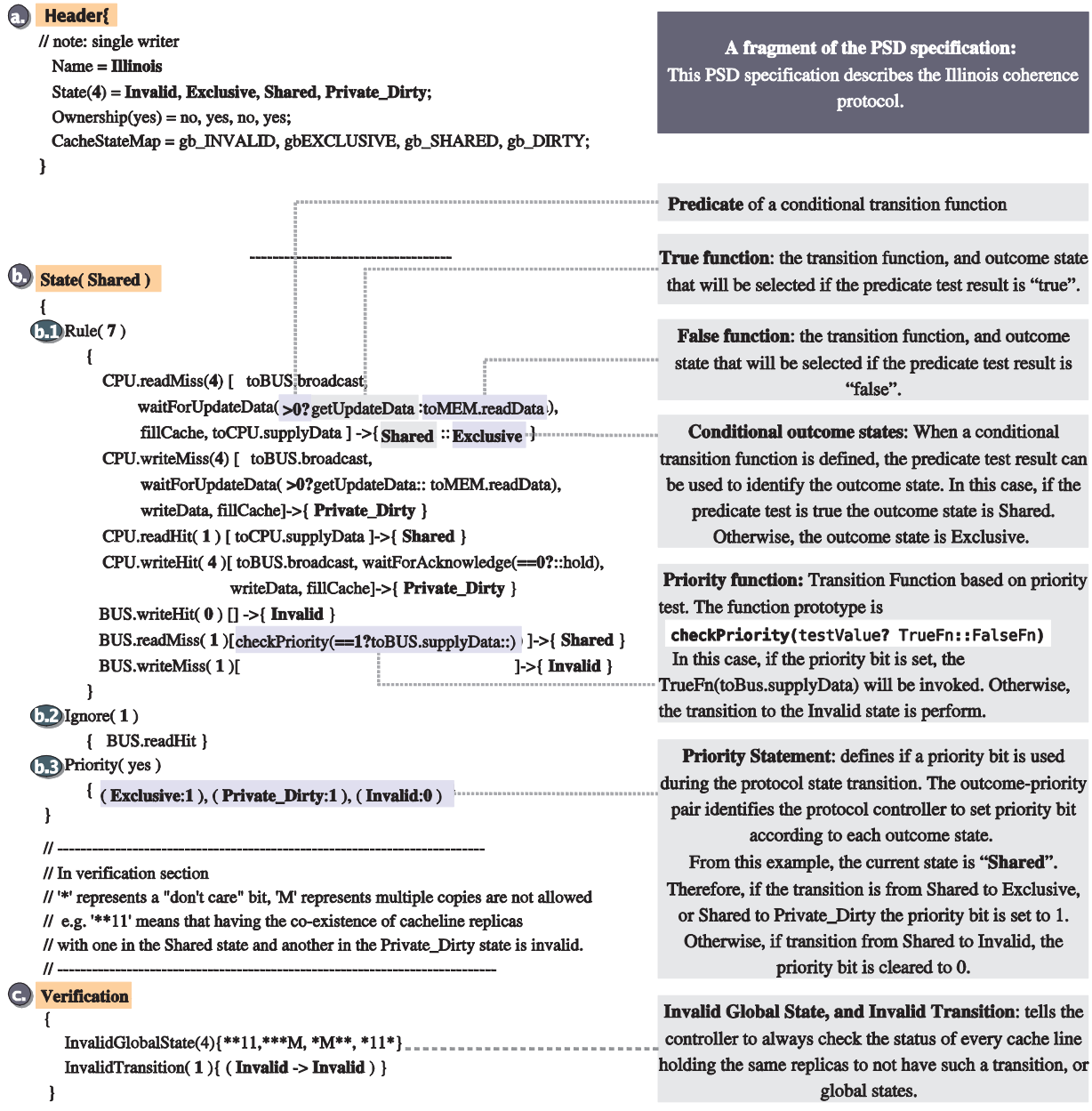


Figure 3. Specification of the Illinois coherence protocol.

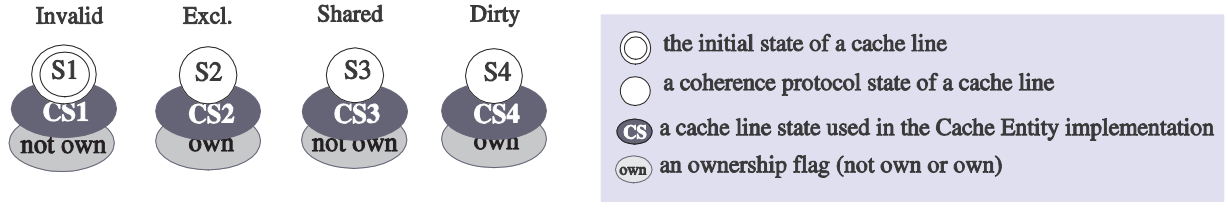


Figure 4. A conceptual view of the PSD header section.

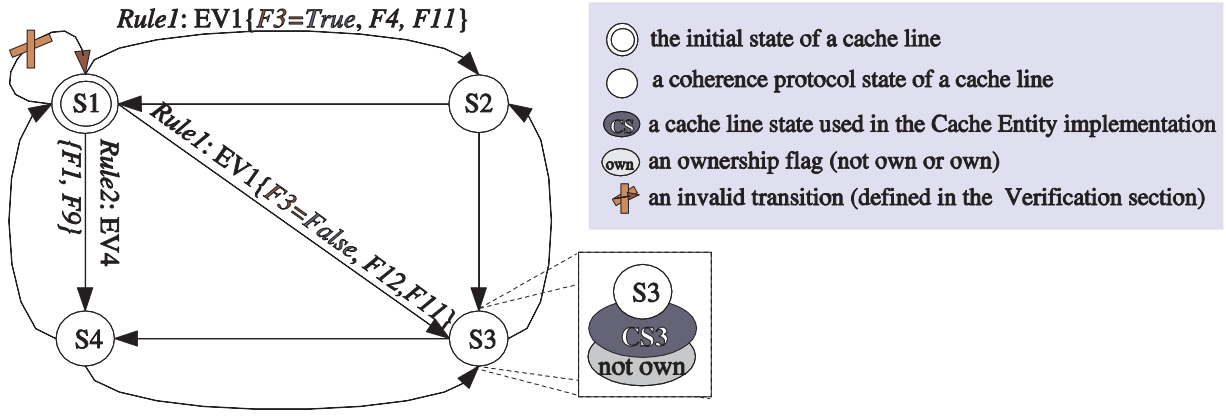


Figure 5. A conceptual views of the PSD State and Verification sections.

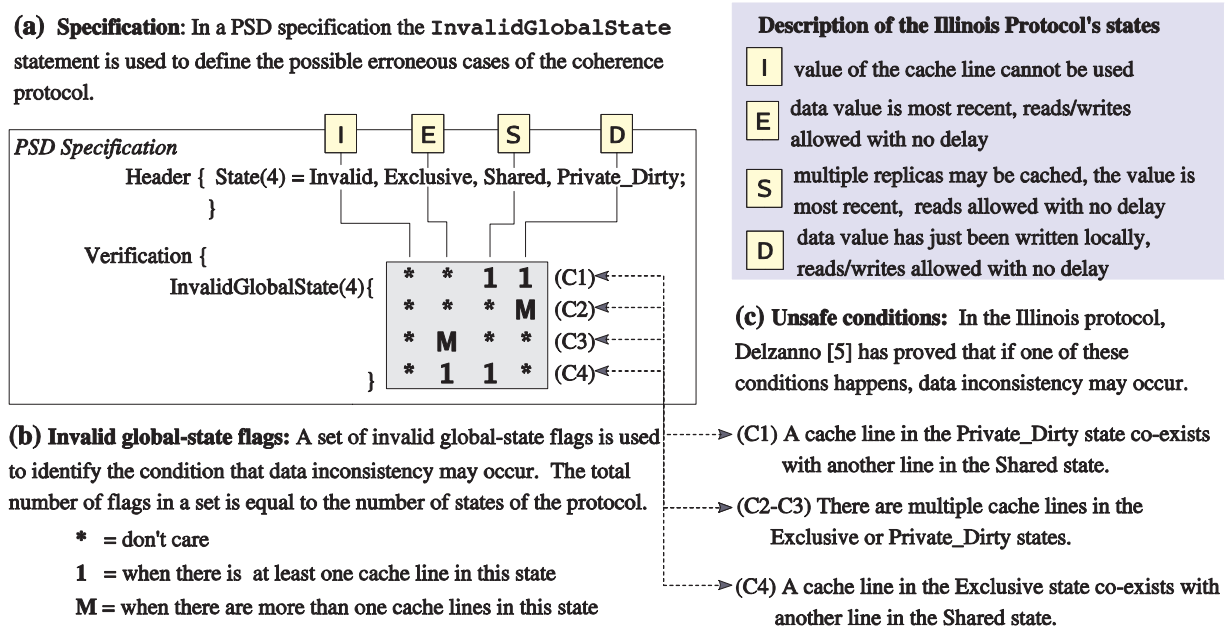


Figure 6. Invalid global states of the Illinois protocol.

invalid global state has been shown to be useful to indicate the conditions where data inconsistency may occur [7]. In PSD, each condition, a so-called *unsafe condition,* is represented by a set of invalid global-state flags. The total number of flags in a set is equal to the total number of states of the protocol. Each flag position is matched to the order of the states of the State list defined in the Header section.

Figure 6 shows how the *invalid global-state* flags are defined in the Illinois protocol. Figure 6(a) depicts the matching of the flag position to each protocol state defined in the Header section. Figure 6(b) shows the description of the invalid global-state flag (i.e. '`*`' means that this state is not considered, '`1`' refers to when there is at least one cache line in this state, and '`M`' refers to when there is more than one cache line in this state). Figure 6(c) describes the four unsafe conditions of the Illinois protocol, each of which is mapped to a set of invalid global-state flags of the PSD specification. The set of invalid global-state flags is used in the PSD specification for two purposes: (1) to check the soundness property during a simulation run and (2) to test the liveness property in the PSD parser.

The last statement of the Verification section is the `InvalidTransition` statement. The invalid transition statement lists the transitions that are not permissible in the protocol. As presented in the example protocol, the

Illinois protocol performs transition functions corresponding to the accepted events. However, after the transition functions are performed, the transition from the Invalid state to the Invalid state is not possible. The definition of this invalid transition is shown in Figure 3, c.

3.2 The PSD Parser

The semantic values of a protocol specification are recognized by a PSD parser. The central role of the PSD parser is to produce a verified state machine from a PSD specification. The PSD parser processes a specification in four steps. The first two steps comprise the lexical and syntactic analysis of a PSD specification. If no syntax errors are found, the result of these steps is a well-formed specification, i.e. the state machine of the coherence protocol. A well-formed specification must satisfy the entire set of PSD test conditions (PC) listed below.

PC1 Every state has been mapped to a cache line state and at least one protocol state is mapped to a `gb INVALID` cache state.

PC2 The ownership flag must be defined.

PC3 When the ownership action is used, at least one of the protocol states is set as the owner of a cache line.

PC4 Every state must have an associated 'State' section.

PC5 A PSD specification must comprise a Header section, one or more State sections, and a Verification section.

Once a protocol specification has passed the syntactic analysis step and has satisfied the five test conditions listed above, the specification is considered as *well formed*. The last two steps of the PSD parsing process aim to verify the correctness of the well-formed specification. Firstly, the specification is checked to ensure the soundness or safety property. In summary, the test of soundness is to check for any unsafe conditions that can cause an inconsistent view of a memory value. If the specification has successfully passed the test of soundness, the last step is to test its liveness property. The liveness test ensures that the specification does not cause a deadlock or livelock. Thus, the state transition can proceed and eventually will produce a result.

4. Verification

4.1 Verification of Soundness

During the PSD parsing steps, the conditions that might lead to some errors in a simulation run are checked in two steps, the test of soundness and the test of liveness properties. A soundness property ensures that the implementation cannot take an action that is inconsistent with the specification [23]. There are two aspects to the verification of soundness. The first is to ensure that the implementation of the simulation works in accordance with the state-transition specification. The second is to prevent known conditions that might cause inconsistent values of data to be seen by the processors.

4.1.1 State Machine Mapping

The state-machine mapping technique has been used to ensure that the implementation of the simulation works in accordance with the state-transition specification. As described in the previous section, the state machine obtained from a PSD specification includes the set of states and rules that describe the transition functions to be performed during a state transition. Figure 7(a) shows an example of a state machine obtained from a three-state coherence protocol. When a `CPU.readMiss` event arrives at an Invalid cache line, according to *Rule 1* of the Invalid state, the protocol must perform functions *A*, *B*, and *C* to maintain data coherence before transit to the Valid state.

To ensure that the DSiMCluster simulator works in accordance with this *Rule*, functions *A*, *B*, and *C* are used to map the specification term into the implementation term. As shown in Figure 7(b), after the DSiMCluster simulator executes function *A*, the coherence protocol stays in a temporary state *TS1*. A temporary state is used to provide channels through which a transition operation can be carried out using multiple steps. The protocol stays in a temporary state until all of the partially executed operations have been finished.

In a PSD specification, three PSD test conditions are checked to ensure the correctness of the state machine. The first condition (PC6) is to ensure that each protocol state has been defined to respond to all possible events received. The second condition (PC7) is to ensure that the requested data will be provided for every read access. Moreover, the third condition (PC8) is to check that the updated data will be written to the cache line for every write access.

PC6 At each State section, all eight protocol events must be defined in either the Rule or the Ignore statements.

PC7 The Rules of both the `CPU.readHit` and the `CPU.readMiss` events must have the `toCPU.supplyData` function defined.

PC8 The Rules of both `CPU.writeHit` and `CPU.writeMiss` events must have the `toBus.broadcast` and `writeData` functions defined.

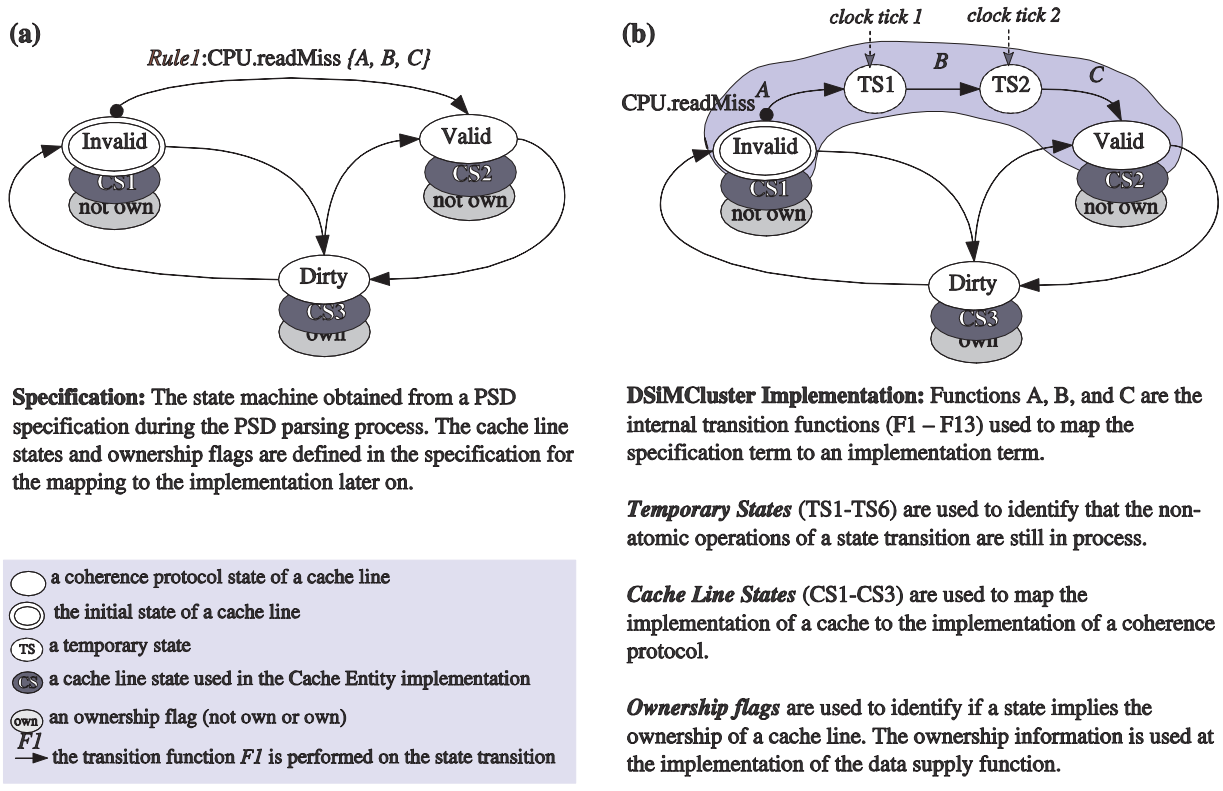


Figure 7. Mapping of specification term to implementation term.

4.1.2 Prevention of Unsafe Conditions

The second aspect of the verification of soundness is to prevent known conditions that might cause inconsistent values of data to be seen by processors. Two possible unsafe conditions described in [8] and [7] have been checked in the PSD parser. Firstly, the protocol should not perform a state transition when it receives any unexpected events. Secondly, the global state of a coherence protocol must be permissible.

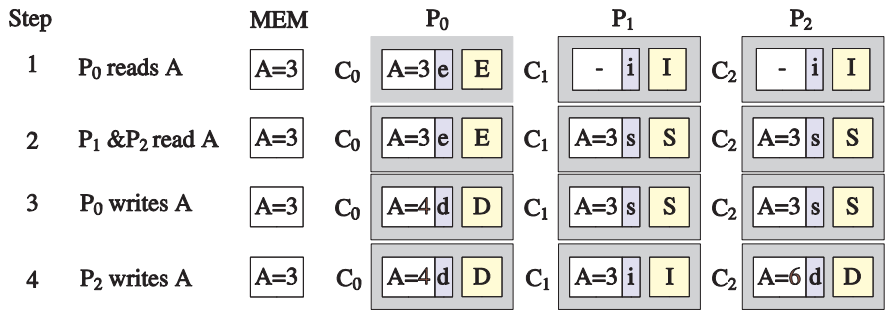
Prevention of unexpected events. The eight possible events that can be received by a coherence protocol are defined as reserved words in PSD. A PSD test condition, PC6, is checked to ensure that each State section recognizes all eight events either through a Rule or an Ignored statement. Following this, another PSD test condition is checked (PC9) in order to ensure that the events defined in the Rule statement (i.e. events causing a state transition) are different from the events defined in the Ignore statement (i.e. events causing no transition).

PC9 Corresponding to PC6, for each State section, each event must be defined only once, by either a Rule or an Ignore statement.

Declaration of invalid global states. When there are multiple replicas of data, processors may see different data values. Therefore, a coherence protocol maintains a coherence state for each replica to tell the processors whether their data is the most up-to-date one. However if a simulation has implemented a protocol wrongly, or if the mapping functions of the PSD specification has been defined wrongly, the results obtained from such a simulation cannot be used to represent the real characteristics of memory accesses. Figure 8 shows an example of such a case.

To ensure that DSiMCluster will detect the situations where data inconsistency may be seen by processors, the set of invalid global states is used. Global states that are not permissible by the protocol definition are normally classified as *erroneous states* [19]. As mentioned at the end of Section 3.1, a purpose of defining unsafe conditions (using the set of invalid global-state flags in the PSD) is to check for soundness during a simulation run. There are four erroneous states in the Illinois protocol [7] as listed in Figure 6. (The definitions of Illinois states and the invalid global states used in Figures 8 and 9 are shown in Figure 6.)

Figure 9 shows the process of soundness checking during a simulation run in DSiMCluster. The checking is performed in three steps. Firstly, the current global state of each cache line is recorded during a memory access (Figure 9(a)). Secondly, when the memory access has



An example scenario when a simulation implements the Illinois protocol wrongly.
(a) If in step 2 the cache line state at Processor P_0 has not been updated, in step 3 the processor P_0 can write to the cache line with no delay (as it holds an Exclusive cache line). Thus, the inconsistent values of A will be seen by P0, P1, and P2.
(b) In step 4, if P2 can write to a Shared cache line without any updates for the most recent value first, the inconsistency of data are seen from both P0 and P2 as they hold the Private_Dirty cache lines with different value.
If these errors are not detected by the simulation, the simulation results cannot represent the correct characteristics of memory accesses.

Figure 8. An example of a simulation error.

been completed, a set of test flags is produced using the recorded global state (Figure 9(b)). Finally, before the simulation can continue, the set of produced test flags must not match with the invalid global state (i.e. it is not invalid). The algorithm of the global state checking is as follows:

```
for all test_flag
   for all global_state_flag
   {
    Producing the test_flags
    if (test_flag = global_state_flag)
    then
     {
      /* Unsafe condition detected */
        stop simulation
     }
   }
```

If the test flags match with the invalid global state, the simulation is stopped as the unsafe condition has been detected. Figure 9(c) illustrates this testing step. Note that if any of the errors shown in this figure occur, DSiMCluster will stop the simulation once the first unsafe condition is detected (after step 2).

Three flags are used when recording the global state of a cache line: `0`, `1`, and `M`. The flag '`0`' means that there are no replicas in this state. Flags '`1`' and '`M`' show that there is only one replica or there are multiple replicas of the cache line in this state, respectively. The order of the states defined in the protocol state list (in the PSD Header section) is mapped to the position of a flag in a global state. At step 1 in Figure 9(a), the global state of variable *A* is `0100`. The recorded global state means that there are no replicas of *A* cached in the Invalid, Shared, and Private Dirty states, and there is one replica of *A* cached in the Exclusive state.

In the PSD parser, three PSD test conditions are included to check the definition of the invalid global states. The first condition, PC10, is to ensure that at least one invalid global state has been defined in a PSD specification. The second condition, PC11, is to test that when a local cache line enters a state that allows only one replica, all remote cache lines must exit the state. The third condition, PC12, is to ensure that when two states must not co-exist, then if a local cache line enters one of these states, remote cache lines must not enter the prohibited state.

PC10 At least one invalid global state must be defined in a PSD specification.

PC11 When a state is defined to have only one replica, all `CPU` events that cause the transition to the state must have corresponding `BUS` events that exit the state.

PC12 When two states must not co-exist, all `CPU` events that enter one of the two states must not have corresponding `BUS` events that enter the other state.

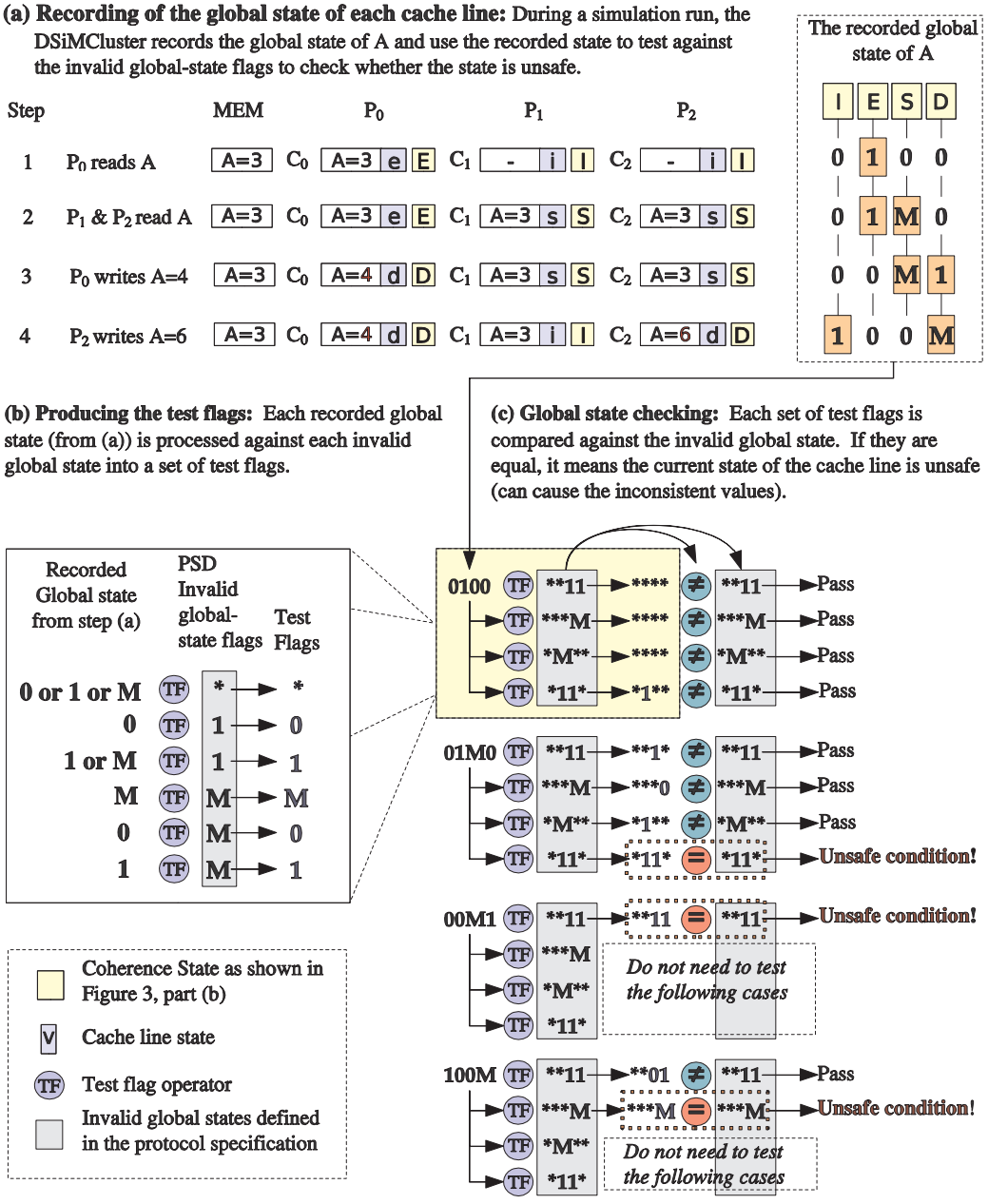
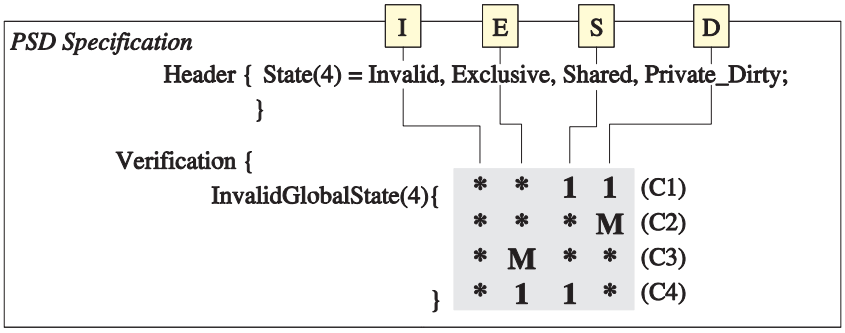


Figure 9. Process of unsafe condition checking during a simulation run.

If a well-formed specification satisfies the seven PSD test conditions (PC6–PC12), the specification has passed the test for soundness. In the last step of the PSD parser, the specification is then verified for its liveness property.

4.2 Verification of Liveness

The previous section shows the usefulness of the invalid global-state definitions for testing soundness. As mentioned at the end of Section 3.1, another purpose of defining the invalid global states is to obtain all co-existing state pairs that are valid, so that a liveness test can be done on these pairs. Figure 10 shows the list of eight valid co-existence state pairs of the Illinois protocol. In the figure, a *State1,State2* pair identifies that if a local cache line is in *State1*, another remote cache can have its replica in *State2*. The PSD parser uses each of these pairs to check for the liveness properties in both the deadlock and livelock testing steps.



Valid co-existence state pairs: Each pair, *(State1, State2),* shows that if the local cache line is in *State1*, then a remote cache can have the replica of this line in *State2*. The valid co-existence state pairs of the Illinois protocol are listed below.

(Invalid, Invalid), (Invalid, Exclusive), (Invalid, Shared), (Invalid, Private_Dirty)

(Exclusive, Invalid)

(Shared, Invalid), (Shared, Shared)

(Private_Dirty, Invalid)

Figure 10. The valid co-existence state pairs of the Illinois protocol.

4.2.1 Deadlock Prevention

A deadlock may occur if, during a transition function (involving partially executed operations), at least two caches are waiting for resources or acknowledgments to be released from one of the others. Once the specification has passed the test of soundness, it is guaranteed that these caches must stay in the valid co-existence states. Therefore, to prevent a deadlock, all possible co-existence states are checked against two PSD test conditions. Firstly, for all valid co-existence state pairs, the waits and supplies of resources must be matched (PC13). Secondly, for all valid coexistence state pairs, every broadcast operation must have the packet acknowledgment sent from the cache in the co-existence state (PC14).

PC13 For all co-existence states, every `waitForUpdateData` function has a matched function, `toBUS.supplyData`.

PC14 For all co-existence states, every `toBUS.broadcast` and `waitForAcknowledgment` function has a matched `sendAcknowledgment`.

4.2.2 Livelock Prevention

A livelock occurs when one or more caches stays indefinitely in a state with no exit after performing a transition function in response to a valid event. To prevent livelock, two PSD test conditions are checked. Firstly, at least one event of the local cache accesses will cause a transition which exits the current state (PC15). This condition is to guarantee that there are no trapped states in a protocol specification. Secondly, to prevent a livelock, all transitions must be valid (PC16).

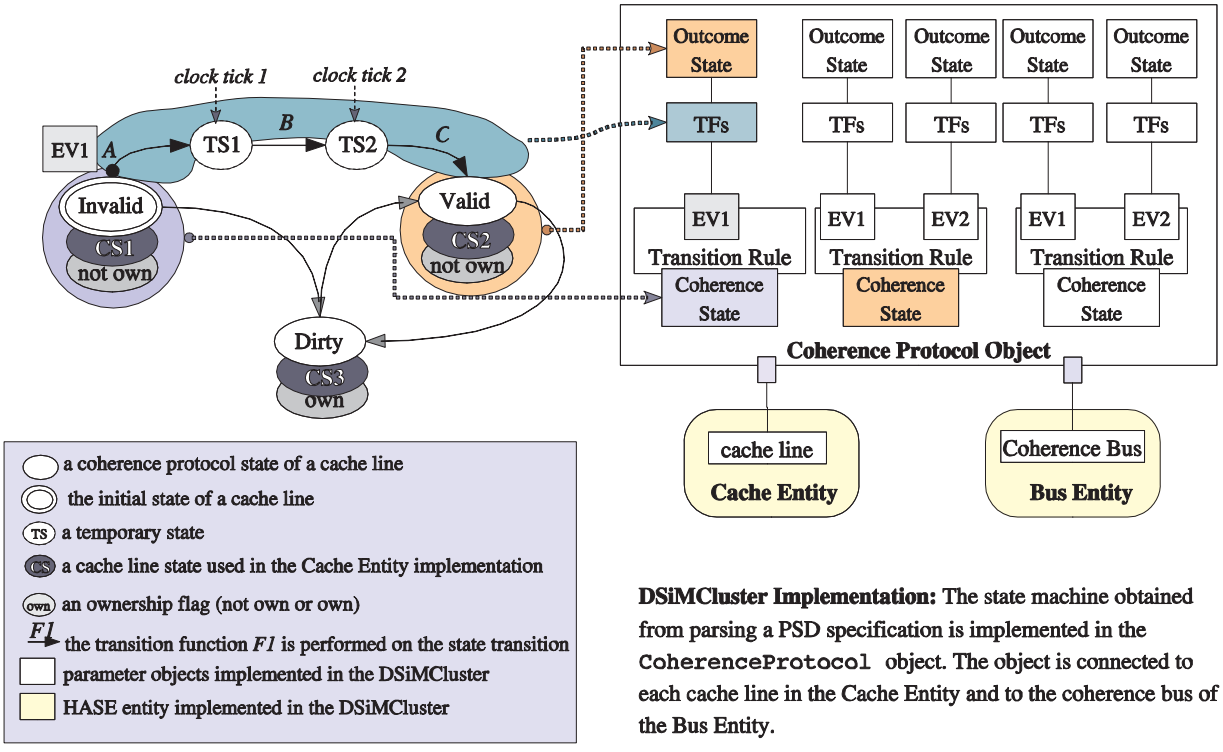
PC15 For each State section, there is a Rule defining one of the four local-access events (`CPU.readHit`, `CPU.readMiss`, `CPU.writeHit` or `CPU.writeMiss`) in which the outcome state must exit to another state.

PC16 For each Rule defined in a State section, the state transition must not violate the `invalidStateTransition` defined in the Verification section.

4.3 Verification of Multi-level Inclusion

In a multiprocessor system, a multiple-level cache hierarchy has an *inclusion* or *MLI* property if 'the contents of a cache at level $i+1$, C_{i+1}, is a superset of the contents of all its child caches, C_i, at level i' [14]. Therefore, when a coherence protocol *invalidates* a content of C_{i+1}, the corresponding content in C_i should also be invalidated. Subsequently, this content should not be seen by the processor.

In a system with multiple-level caches, the updating of a cache state must be finished before the next access to the upper level cache. Thus, the cache access is blocked until the actions of the coherence protocol have been completed. In DSiMCluster, the state `Coherence Stall`



DSiMCluster Implementation: The state machine obtained from parsing a PSD specification is implemented in the `CoherenceProtocol` object. The object is connected to each cache line in the Cache Entity and to the coherence bus of the Bus Entity.

Figure 11. The state machine represented using the `CoherenceProtocol` object.

is used to prevent the upper level cache from being accessed by the processor prior to the completion of the coherence actions. Once the coherence actions are finished, the cache lines that have been invalidated by the coherence actions are marked as invalidated by inclusion. The following access misses to these cache lines are recorded separately as inclusion misses. The record of inclusion misses shows that if a simulation does not consider the inclusion property (i.e. it allows a processor to access the upper-level caches during coherence actions on the lower-level cache), memory access characteristics obtained from such a simulation will be incorrect. The percentages of inclusion misses show the extent of the errors that would occur in a faulty simulation system.

5. Specification-based Parameter–Model Interaction

An SPMI is proposed as a technique to verify and co-simulate a coherence protocol within the DSiMCluster framework [24]. This technique includes protocol specification using a PSD language, and ways to direct the model's behavior using the semantics obtained from this specification. This section provides the implementation details showing how the PSD specification has been included and mapped to the DSiMCluster simulator.

The SPMI technique is a systematic means to add a specific feature of a parameter to a simulation model using the semantics obtained from a well-formed specification. It comprises four major operations: (a) creating an object which represents the PSD specification; (b) building the specification semantics into the object using the PSD parser; (c) connecting the object to a simulation model; and (d) emulating the protocol behavior (described in Figure 13).

5.1 Coherence Protocol Object

The first operation in SPMI is to create an object representing the PSD specification. The `CoherenceProtocol` class is used as the central channel to interconnect between the protocol behavioral emulation and the semantics obtained from a protocol specification. The `CoherenceProtocol` represents the state machine obtained from parsing a PSD specification. Figure 11 shows the representation of a state machine using the CoherenceProtocol object and the connection of the object to the Cache and Bus framework components. This `CoherenceProtocol` class contains two important member classes, namely, the `TransitionRule` object and the `CoherenceController` object. Figure 12, b and c, illustrates the two objects and their interconnection.

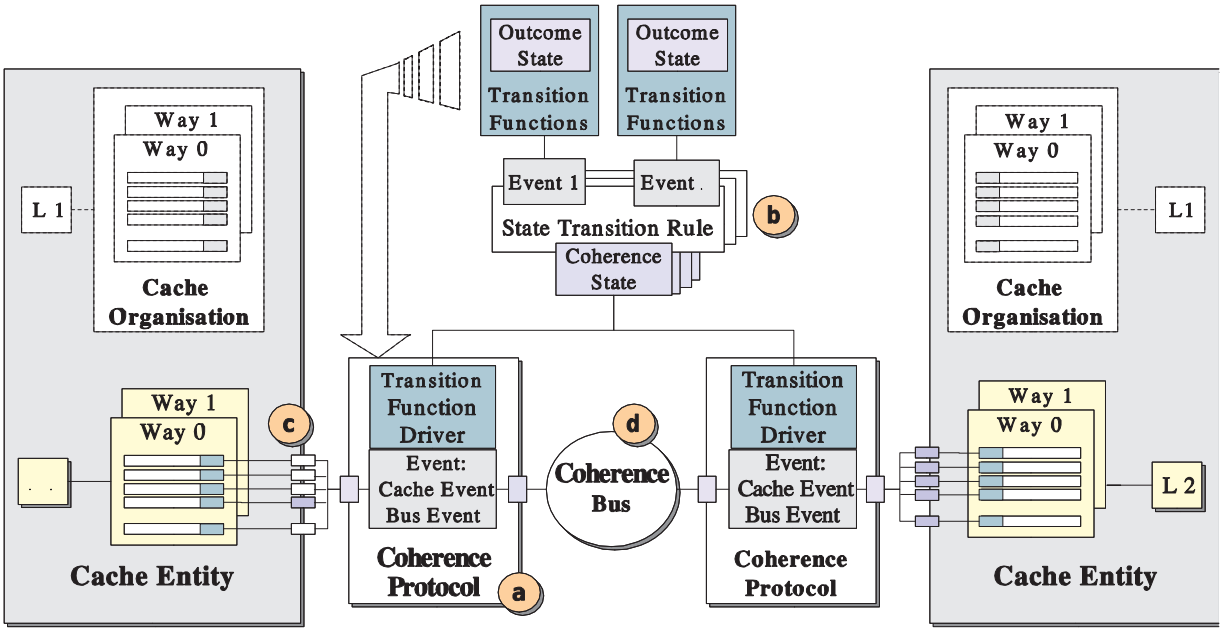


Figure 12. Protocol objects and their connections.

5.2 Building the Specification Semantics

The semantic values of a protocol specification are recognized by a PSD parser. The PSD parser has been implemented as an external object to be plugged into the simulation component called `Cache Entity`. The parser takes a PSD specification and generates an object representing the protocol (the `CoherenceProtocol` object) which is returned to the DSiMCluster kernel. The parser works in collaboration with a lexical scanner. The lexical scanner performs lexical analysis by recognizing tokens and passing these accepted tokens to the parser. The parser then performs the syntactic analysis by matching the sequence of tokens against the PSD grammar rules. When the grammar rules of transition functions are recognized, the pointer to the corresponding transition function is allocated to the protocol object. After the specification is compiled, the protocol object will contain pointers to its relevant transition functions each of which implements the simulated behavior. During the compilation process, the PSD parser verifies the soundness and liveness properties of the specification using the steps explained in Sections 4.1 and 4.2. If the specification has passed all 16 of the PSD test conditions, the specification is ready to be plugged into the simulation kernel.

5.3 Interconnection between Simulation Components and the Protocol Objects

Once a specification semantic is successfully parsed (the state-transition information has been stored in a `CoherenceProtocol` object), the connection process begins. Each simulation component (Cache Entity) makes a copy of the `CoherenceProtocol` object, creates a handle to its copy, and then passes this handle to the constructor of a `CacheOrganisation` object. A `CacheOrganisation` object[5] creates `CacheWays` and `CacheLines` according to the configuration, and also creates a `CoherenceController` (Figure 12, c) linked to each `CacheLine`. The handle of `CoherenceProtocol` is then attached to every `CoherenceController` (Figure 12, a). Once all Cache Entities have finished these steps, the Bus Entity[6] communicates with each of the Caches to get their `CoherenceProtocol` handles. The Bus Entity then creates an instance of a `CoherenceBus` object and links this object with the obtained `CoherenceProtocol` handles (Figure 12, d). Figure 12 depicts the complete connection of two protocol objects to the cache hierarchy of a 1 × 2 DSM model which maintains cache coherence at level 2 caches.

5.4 Emulation of the Coherence Protocol during a Simulation Run

During a simulation run, Cache Entities play an important role in emulating the behavior of the coherence protocols

5. I.e. an object representing the structure of cache, its contents, and associativity according to the configuration.

6. A Bus Entity in the DSiMCluster represents a bus and its behavior.

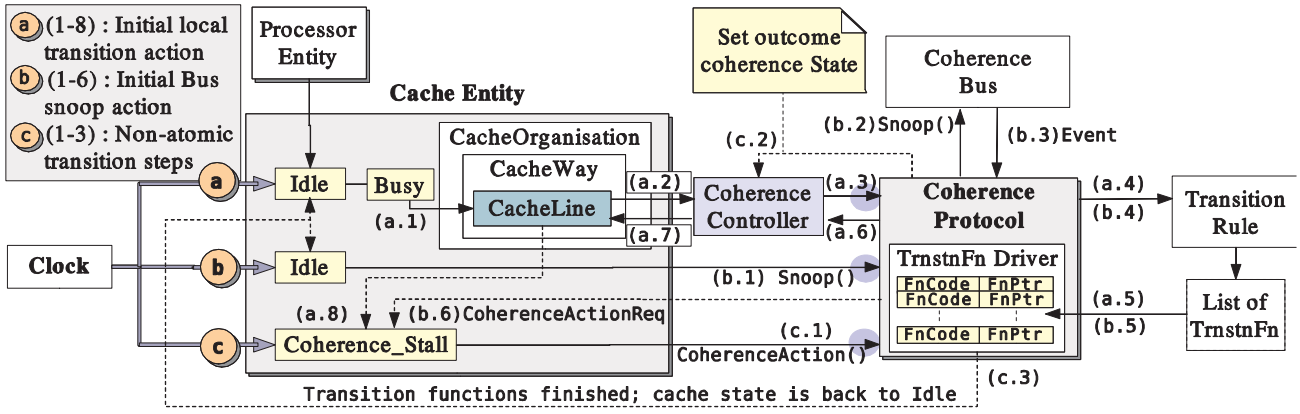


Figure 13. Block diagram shows the steps of the protocol behavior emulation.

using the `CoherenceProtocol` objects. As shown in Figure 13, the emulation process is activated by the simulation clock signal sent to a Cache Entity. Coherence actions can be activated in two ways: (a) by a CPU access causing a hit/miss action at a cache line (Figure 13, a), or (b) by a BUS notified event snooped from the coherence bus (Figure 13, b).

The `Coherence Stall` state of a Cache Entity is used to direct a sequence of protocol actions that require more than one clock cycle to finish, a so-called *non-atomic action* (Figure 13, c). The `Coherence Stall` state is used as a temporary state to provide channels through which a transition operation can be carried out using multiple steps. It is important to note that DSiMCluster uses a two-phase clock to model a Cache Entity. In this figure only the action of clock phase 0 has been illustrated. In clock phase 1, each Cache Entity directs its `CoherenceProtocol` object to snoop on the bus and to send an acknowledgment to a bus transaction if required. This is to prevent the case of *deadlock*, in which other caches could be waiting for a bus acknowledgment forever.

6. Model Verification Data

The specifications of eight bus-based coherence protocols have been developed and included in DSiMCluster. Verification using the PSD parser has been presented in [22]. The result of the parsing verification step has shown that the PSD language is able to describe a state machine of a complicated protocol such as MOESI. Some unsafe conditions have been declared and excluded from the specification using the PSD test conditions during the test of soundness and liveness properties.

However, these test conditions do not guarantee that the protocols will produce the correct result, since errors may creep into the definition of the transition functions. Therefore, an experiment was run, firstly to validate the result of one coherence protocol against measurement, and secondly to use the results obtained from the validated protocol to verify the result of the other protocols.

The experiment was performed in five steps. In the first step, a workload obtained from the NAS NPB 2.3 benchmark was run on a SunFire 15K machine to select the function which dominates the runtime. In the second step, the benchmark program was modified in order to obtain the input and output of the selected function. After the input and output of the selected function had been obtained, in the third step, the function was implemented as a DSiMCluster workload format (DWF) file. In the fourth step, the modified benchmark program (obtained from step 2) was executed on the SunFire 5K machine and its cache profiles were measured. The last step was to perform the simulation experiment. In this step, the DWF workload file (obtained from step 3) was simulated by DSiMCluster. In all simulations, the DSiMCluster simulator was customized to model a SunFire 15K configuration.

6.1 Experimental Design

A 1 × 4 DSM model with the SPMI technique was chosen for preliminary model verification. In this test, three different coherence protocols were used with the same workload and compared with the results for one protocol against measurements obtained from a real system. To carry out these tests, three protocols with different numbers of states and behavioral characteristics were selected: Synapse, Illinois, and MOESI. As the SunFire maintains cache coherence by using the MOESI protocol, the first step is to validate the results of the MOESI specification. The validated result is then used to verify the Synapse and Illinois specifications. Each of these protocols was examined using the OpenMP LU-decomposition workload obtained from the NAS NPB 2.3 benchmark [25]. All experiment runs used the class A benchmark, i.e. a small-scale workload with a three-dimensional matrix size

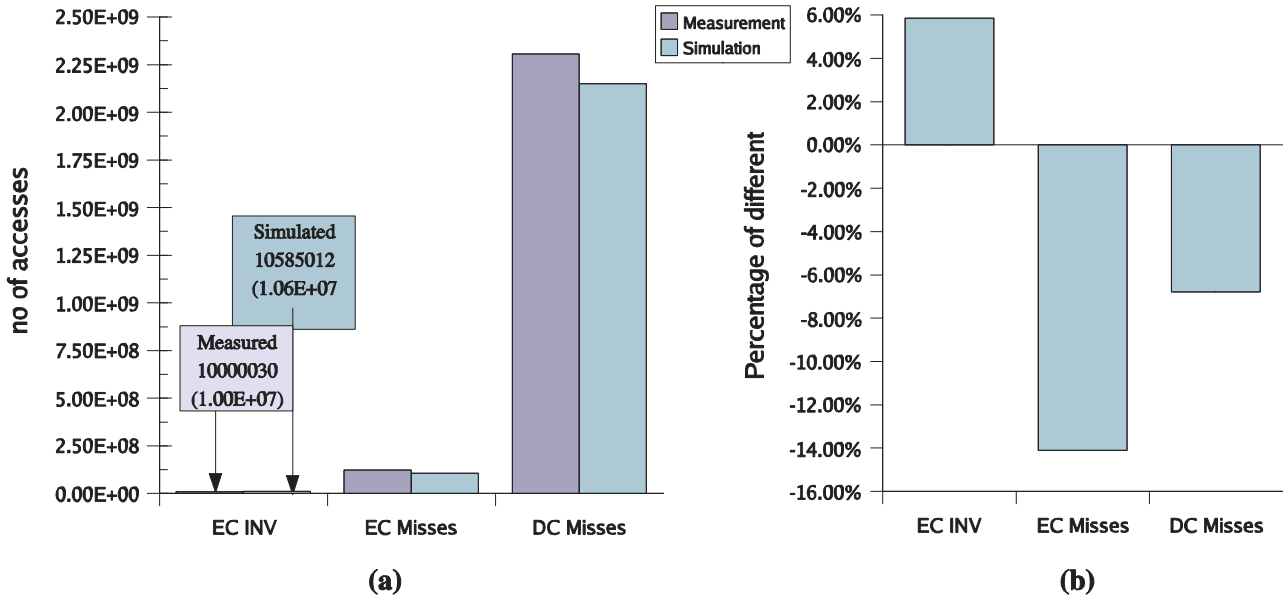


Figure 14. Comparison of simulation-measurement results.

of 64 × 64 × 64. The values of hardware counters of the SunFire 15K machine were collected using the same workload configuration, i.e. using four OpenMP parallel threads running on four processors, one thread per processor. The results obtained from measurement were compared against the simulation results.

Model configuration. The simulation is based on a 1 × 4 DSM model which has been configured after the node architecture of the SunFire 15K server and Sun Fireplane system interconnect [26]. In the target machine, each processing node has a two-level cache. The level 1 Data Cache (DC) is a 64 KB on-chip cache with a line size of 32 bytes and a write-through, no-write-allocate policy. The DC is indexed by virtual address and tagged by physical address. The level 2 or External Cache (EC) is an 8 MB external cache with a coherence control at a granularity of 64 bytes. The level 2 cache is indexed and tagged by physical address with a write-back, write-allocate policy. The coherence policy of the SunFire system is MOESI, maintained at level 2 cache using the bus provided by the Sun Fireplane system interconnects [26].

DSiMCluster was configured to represent the SunFire 15K configurations described above. However, due to the limitation of address length, half of the virtual and physical address space was simulated, i.e. a 32-bit virtual address and 22-bit physical address.

6.2 Verification Results

Figure 14 shows a comparison of simulation results against the results of measurement when running a particular function inside the LU program, called BUTS, which dominates the runtime. Note that the measurement has been scoped at a particular function to make the comparison of results feasible. We have verified the results of the program running on DSiMCluster against the results from running the program on the SunFire machine. Both runs produce the same results, thus confirming that DSiMCluster generates the correct execution sequences. Figure 14(a) shows the total number of invalidation and cache miss events occurring at the EC and level 1 DC. Figure 14(b) shows the percentage difference between the simulation results and the measurement results. Both the EC invalidation and DC misses results show similarity between the simulation and the measurement results (±5–6% in both cases). However, the EC misses show a noticeable difference (14.50 percentage points different). This is because, in the SunFire 5K machine, EC is a unified cache (i.e. it also accommodates instructions) while the simulation did not emulate instruction caching. Therefore, the number of EC misses obtained during the simulation is far smaller. Nevertheless, as processors only read the content of instructions, they produced no invalidation events based on violation of cache updates.

Table 1 summarizes the results obtained corresponding to the assessment used in the verification process for the safety, liveness, and inclusion properties. The protocol specifications have been examined to *prevent* erroneous cases for the safety and liveness properties as described earlier. Moreover, in this experiment, the inclusion property is checked during each simulation run. To do so, the total number of events that invalidate a content of all the level 2 caches (or EC) are counted and confirmed with

Table 1. Verification results of three protocols.

Protocol	Safety		Liveness		Inclusion	
	No of INV global state	Critical state	Trapped state	Wait-Ack matching pairs	Percentage of EC INV	Percentage of MLI INV
Synapse	2	Dirty	None	14	13.02%	4.57%
Illinois	4	Dirty, VldExcl	None	18	12.84%	4.87%
MOESI	7	Mod, Excl	None	29	8.75%	5.17%

INV: invalidations; VldExcl: Vlaid Exclusive; Mod: Modified; Excl: Exclusive

the total number of subsequent invalidations that occur in level 1 (as the percentage shown in the last two columns in the table). All the simulation runs terminated successfully, and produced the same results as the measurements.

These two figures highlight the fact that the SPMI technique allows DSiMCluster to reflect a correct projection of the simulation results with a very small fraction of time spent on specification parsing and erroneous cases detection. These results correspond to the measurement results obtained from a real machine with the difference of less than 10% in the data cache accesses.

7. Conclusion

The PSD-based objects represent different bus-based cache coherence protocols. This latter group has been designed to apply automatic verification to the DSiMCluster model through a well-formed specification. The specification of each of these objects has been defined using the DEVS *passive* states. These objects stay indefinitely in a state until they receive an external request from the attached entity (i.e. the framework component to which they are attached). Once a request has arrived, the objects perform the corresponding state transition functions and change their state. Consequently, behavior emulation occurring during the state transition of these objects shares the timing information with the attached entity. During normal execution cycles, an implementation of a SPMI verification technique is used to simulate the bus-based coherence protocols, thus keeping the cache coherent.

The experimental results have demonstrated how possible errors in protocol specifications that may impact the *soundness* and *liveness* properties of the coherence protocols can be detected early while parsing the PSD specifications. A verification experiment using the DSiMCluster simulation model against a SunFire 15K machine has been presented. A workload of a particular function, BUTS, obtained from the LU decomposition program of the NPB 2.3 benchmark (class A) has been run on both the real machine and the DSiMCluster model with similar configurations. The numbers of cache misses obtained from both machines have been compared, resulting in a difference of less than ±5–6% on average, and 14.5% in the worst case at the level 2 external cache. This result has confirmed that the model is working correctly, so thus the model is ready to be used for some further performance evaluation experiments on a wider range of design parameters.

8. Acknowledgments

Worawan Marurngsith was a PhD student at the University of Edinburgh supported by a Thammasat University scholarship. HASE has been supported by the UK EPSRC under grants GR/R27129 and GR/S28143. We wish to thank the Edinburgh Parallel Computing Centre (EPCC) for providing time on the SunFire15K machine (lomond) used in the experiments. We also wish to thank all members of the HASE group for their support.

9. References

[1] Marathe, J., F. Mueller and B.R. de Supinski. 2006. Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. ACM Transactions on Architecture and Code Optimization, 3(4): 390–423.

[2] Moga, A. and M. Dubois. 2009. A comparative evaluation of hybrid distributed shared-memory systems. *Journal of Systems Architecture*, 55(1): 43–52.

[3] Bennett, A.J., T. Field and P. Harrison. 1996. Modelling and validation of shared memory coherency protocols. *Performance Evaluation*, 27–28: 541–563.

[4] Field, A., P. Harrison and K. Kanani. 1998. Automatic generation of verifiable cache coherence simulation models from high-level specifications. In *Australian Computer Science Communications*, 20: 261–275.

[5] Sorin, D.J., M. Plakal, A.E. Condon, M.D. Hill, M.M.K. Martin and D.A. Wood. 2002. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6): 556–578.

[6] Pong, F. and M. Dubois. 1997. Verification techniques for cache coherence protocols. ACM Computing Surveys, 29(1): 82–126.

[7] Delzanno, G. 2003. Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design, 23(3):257–301.

[8] Pong, F. and M. Dubois. 2000. Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Transactions on Parallel and Distributed Systems*, 11(9): 989–1006.

[9] Stoy, J.E., X. Shen and Arvind. 2001. Proofs of correctness of cache coherence protocols. In FME 2001: Proceedings of Formal Methods Europe 2001 on Formal Methods for Increasing Software Productivity, Berlin, Germany, volume 2021 of Lecture Notes in Computer Science, Springer Verlag, pp. 43–71.

[10] Tasiran, S., Y. Yu and B. Batson. 2003. Using a formal specification and a model checker to monitor and direct simulation. In DAC'03: Proceedings of the 40th conference on Design automation, Anaheim, CA, USA, ACM Press, pp. 356–361.

[11] Lv, Y., H. Lin and H. Pan. 2007. Computing invariants for parameter abstraction. In *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, Washington, DC, USA, IEEE Computer Society, pp. 29–38.

[12] Qu,W., Y. Guo, Z. Pang and X. Yang. 2008. Efficient verification of parameterized cache coherence protocols. In ICYCS 2008: Proceedings of the 9th International Conference for Young Computer Scientists 2008, IEEE Computer Society, pp. 154–159.

[13] Chame, J. and M. Dubois. 1993. Cache inclusion and processor sampling in multiprocessor simulations. In SIGMETRICS'93: Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, San Diego, CA, USA, ACM Press, pp. 36–47.

[14] Baer, J.-L. and W.-H. Wang. 1988. On the inclusion properties for multi-level cache hierarchies. In ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture, Honolulu, Hawaii, IEEE Computer Society Press, pp. 73–80.

[15] Yilmaz, L. 2004 Verifying collaborative behavior in component-based DEVS models. *Simulation: Transactions of the Society for Modeling and Simulation International*, 80(7): 399–415.

[16] Marathe, J., A. Nagarajan and F. Mueller. 2004. Detailed cache coherence characterization for OpenMP benchmarks. In ICS'04: Proceedings of the 18th Annual ACM International Conference on Supercomputing, Saint-Malo, France, ACM Press, pp. 287–297.

[17] Martin, M.M.K., D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill and D.A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. ACM SIGARCH Computer Architecture News, 33(4): 92–99.

[18] Wainer, G., L. Morihama and V. Passuello. 2002. Automatic verification of DEVS models. In SIW 2002: Proceedings of SISO Spring Interoperability Workshop, Orlando, FL, USA, Simulation Interoperability Standards Organization's (SISO).

[19] Marurngsith, W. and R.N. Ibbett. 2009. DSiMCluster: a simulation model for efficient memory analysis experiments of DSM clusters. *Simulation Transactions of the Society for Modeling and Simulation International,* 85(6): 355–374.

[20] Borodin, D. and B. Juurlink. 2008. A low-cost cache coherence verification method for snooping systems. In DSD '08: Proceedings of the 11th Euromicro Conference on Digital System Design Architectures, Methods and Tools, Parma, Italy, IEEE Computer Society Press, pp. 219–227.

[21] Dill, D.L., A.J. Drexler, A.J. Hu and C.H. Yang. 1992. Protocol verification as a hardware design aid. In ICCD '92: Proceedings of the 1992 IEEE International Conference on Computer Design on VLSI in Computer & Processors, Cambridge, MA, USA, IEEE Computer Society, pp.522–525.

[22] Marurngsith, W. 2006. *Simulation Modelling of Distributed-Shared Memory Multiprocessors,* PhD Thesis, Institute of Computing Systems Architecture, School of Informatics, University of Edinburgh, Edinburgh, UK.

[23] Shen, X. 2000. *Design and Verification of Adaptive Cache Coherence Protocols,* PhD Thesis, Massachusetts Institute of Technology, USA.

[24] Marurngsith, W. and R.N. Ibbett. 2005. Specification-based parameter-model interaction: towards a correct reflection of memory characteristics in a DSM cluster simulation. In SCSC'05: proceedings of The 2005 Summer Computer Simulation Conference, Philadelphia, USA, The Society for Modeling and Simulation International (SCS), pp.18–25.

[25] Jin, H., M. Frumkin and J. Yan. 1999. *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance,* NAS Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[26] Charlesworth, A. 2001. The sun fireplane system interconnect. In Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), Denver, Colorado, USA, ACM Press, pp. 7–7.

Worawan Marurngsith *received a PhD in Informatics and MSc in Computer Science from the University of Edinburgh. She is currently a full-time lecturer at the Department of Computer Science, Thammasat University, Thailand.*

Roland Ibbett *is an Emeritus Professor of Computer Science of the University of Edinburgh and a founding member of the Edinburgh Parallel Computing Centre. Before moving to Edinburgh in 1985 he was a member of the Computer Science Department at the University of Manchester where he worked on the MU5 project.*