Modeling a Cache Coherence Protocol with the Guarded Action Language

Quentin L. Meunier, Yann Thierry Mieg, Emmanuelle Encrenaz

Modeling a Cache Coherence Protocol with the Guarded Action Language

Quentin L. Meunier Yann Thierry-Mieg Emmanuelle Encrenaz
Sorbonne Université, CNRS
Laboratoire d'Informatique de Paris 6
LIP6, F-75005 Paris, France
Quentin.Meunier@lip6.fr Yann.Thierry-Mieg@lip6.fr Emmanuelle.Encrenaz@lip6.fr

We present a formal model built for verification of the hardware Tera-Scale ARchitecture (TSAR), focusing on its Distributed Hybrid Cache Coherence Protocol (DHCCP). This protocol is by nature asynchronous, concurrent and distributed, which makes classical validation of the design (e.g. through testing) difficult. We therefore applied formal methods to prove essential properties of the protocol, such as absence of deadlocks, eventual consensus, and fairness.

1 Introduction

Testing and simulation are unfortunately not sufficient to provide strong correctness guarantees expected from hardware designs, where patching is not an option. However, proving a design correct is a difficult process despite the improvement of verification tools, as complexity of hardware designs has also grown along with Moore's law.

The TSAR (Tera-Scale ARchitecture) shared memory architecture [1] studied in this paper is a general-purpose multicore architecture, in which cache-coherence is entirely supported by the hardware. The main technical challenge of this architecture is scalability, as it is intended to integrate up to 1024 cores. Its embedded cache-coherence protocol is a key architectural point, and has been designed scale well when the number of cores grows.

To formally prove properties of the designed protocol, named Distributed Hybrid Cache Coherence Protocol (DHCCP), we built several formal models over a number of student internships, designed to investigate how different model-checking tools could address our need. This case study is the result of a collaboration between the experts in systems-on-chip design that developed TSAR, and experts in formal verification that provide model-checking tools.

We first present the DHCCP protocol and the relevant characteristics of the hardware that must be captured by the semantics of the model. We then present the formal models that we built over time in Promela, Divine and GAL, together with the results we were able to obtain.

All these models are made available as part of this submission process and will be accessible in the MARS repository as well as on github[1].

2 TSAR Hardware Architecture and Coherence Protocol

Memory Layout. The architecture is clustered and has a 2D-mesh topology, as represented in Figure 1. Each cluster typically contains 4 processor cores with their L1 caches, a local interconnect, one L2-cache bank, and some peripherals.

[1] https://github.com/lip6/TSAR-DHCCP

Submitted to:
MARS 2018

The set of all L2-cache banks form a distributed L2 cache: the physical address space is statically split into fixed-size segments, and each L2 cache is responsible for one segment, defined by the physical address MSB bits. As such, the L2 cache is logically shared and physically distributed, making the architecture NUCA (Non Uniform Cache Access): all processors can access all L2-cache banks, but the access time and power consumption depend on the distance between the processor and the cache bank.

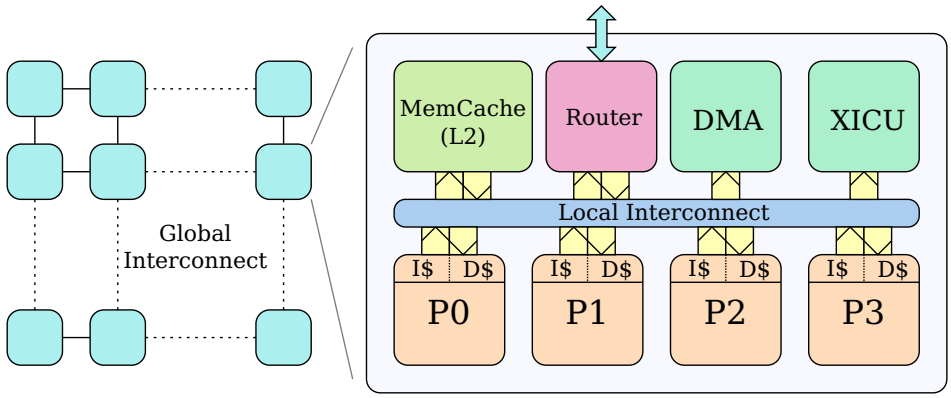

Figure 1: **Tsar Architecture Overview: Mesh Topology and Cluster Details**

L1–L2 Communication. For L1-L2 communication, TSAR uses a *L1-L2 Interconnect* composed of two hierarchical levels: a *Local Interconnect* for intra-cluster communication, and a *Global Interconnect* for inter-cluster communication, implemented as a network-on-chip with a 2D-mesh topology. For a L1 cache, accessing a L2 located in another local interconnect results in a longer latency, but both levels implement a logically flat address space. The L1-L2 interconnect provides a built-in broadcast service used by the coherence protocol to efficiently broadcast invalidation messages and contains five independent networks.

Hardware Coherence Mechanism. For scalability purpose, TSAR implements a directory-based cache-coherence policy. From a conceptual point of view, the coherence protocol is supported by a global directory located in the L2 controller: this global directory stores the status of each cache line replicated in at least one L1 cache of the architecture. The policy between L1 and L2 caches is write-through, meaning that the L2 cache always contains the most recent value of a cache line, and there is no exclusive ownership state for a L1 cache. This global directory is physically distributed in the corresponding L2 banks.

The basic coherence mechanism is the following: when the L2 controller receives a WRITE request for a given cache line, it sends an UPDATE or INVAL request to all L1 caches containing a copy except the writer. The write request is acknowledged only when all UPDATE or INVAL transactions are completed.

The L2 cache is inclusive: a cache line present in at least one L1 cache must be present in the L2 cache. Thus for any evicted line, the corresponding copies must be invalidated. When a shared piece of data is modified, the DHCCP protocol uses two different strategies depending on the number of copies:

- MULTICASTUPDATE: when the number of copies is smaller than a certain threshold, called *DHCCP threshold*, the L2-cache controller registers the locations of all the copies and sends an UPDATE request to each concerned L1 cache.
- BROADCASTINVAL: when the number of copies is larger than the *DHCCP threshold* or if there is no room left to register the copies locations, the L2-cache controller registers only the number

of copies without their location, and sends an INVAL request to all L1 caches. Only the L1 caches which own a copy of the line must respond to this request, thus reducing the traffic.

The list of sharers of a given cache line is stored in the L2 directory with a per bank hardware heap shared between lines. A counter of sharers is also maintained in the directory entry. When the threshold of copies is exceeded or when the heap is full, the sharers list is freed and only the counter of copies is used. In this case, broadcast invalidates are used to maintain cache-coherence.

Types of Transactions. Two types of transactions are defined for L1-L2 communication:

Direct transactions, containing the messages `read`, `write`, `ll` (Load-Linked), `sc` (Store Conditional), `cas` (Compare-and-Swap), along with their responses. These transactions use two separated networks for commands and responses. They are initiated by L1-cache controllers. For these transactions, the target can be any L2-cache controller.

Coherence transactions, containing the messages `cleanup` (eviction from L1 or invalidation acknowledgement), `clack` (CLeanup ACK), `multicast_update`, `multicast_inval`, `broadcast_inval`, and `multi_ack`. There are four types of coherence transactions, each requiring two or three steps.

Coherence Transactions.

- A **`local cleanup`** is a two step transaction initiated by a L1-cache controller when it makes a cache line replacement, usually following a miss. The L1 cache sends a `cleanup` request indicating it is invalidating the cache line, and the L2-cache controller returns a `clack` response to acknowledge the line invalidation.
- A **`multicast update`** is a two step transaction initiated by the L2-cache controller when it receives a `write` request to a replicated cache line, for which the number of copies does not exceed the DHCCP threshold. It sends as many `multi_updt` requests as the number of registered copies, but the writer. The expected response is a `multi_ack` sent by each involved L1 cache. The L2-cache controller counts the number of responses to detect the completion of the `multicast update` transaction.
- A **`multicast invalidate`** transaction is a three step transaction initiated by a L2-cache controller when it makes a cache line replacement (following a miss) and the victim line has a number of copies smaller than the DHCCP threshold. It sends as many `multi_inval` requests as the number of registered copies; each L1 cache returns a `cleanup` response, and the L2 cache acknowledges the invalidation with a `clack`. The L2-cache controller counts the number of responses to detect the completion of the `multicast invalidate` transaction.
- A **`broadcast invalidate`** is a three step transaction initiated by a L2-cache controller when it either replaces a line, or receives a `Write` request to a replicated cache line, and this cache line has a number of copies larger than the DHCCP threshold. The L2 cache sends a single message `broadcast_inval`, which is dynamically replicated by the network to all L1 caches. L1 caches which have a copy must respond by a `cleanup` message. All these `cleanup` responses are counted by the L2 cache to detect the completion of the transaction. Finally, the L2-cache controller acknowledges each invalidation with a `clack`.

The `multi_updt`, `multi_inval`, `broadcast_inval` and `clack` messages use the direction *L2 → L1 Cache*, while the `cleanup` and `multi_ack` messages use the direction *L1 → L2 Cache*.

A L1-cache controller must be sure that a sent `cleanup` request has arrived to the L2 cache before sending a `read` request for the same line; otherwise, there would be a risk of inconsistency if the latter request passes the cleanup. To enforce this, L2 caches must respond to each `cleanup` message with a

`clack`. In order to avoid deadlocks, `clack` responses must use a physically separated network. Therefore, the coherence transactions require three separated networks:

- `cleanup` and `multi_ack` share the first network.
- `multi_updt`, `multi_inval` and `broadcast_inval` share the second network.
- `clack` are conveyed by the third network.

Direct transactions use two further networks: one for requests and one for responses, for a total of five independent networks.

3 Modeled Architecture

Verification Objectives. If intensive testing is mandatory, formal verification can help detect subtle bugs due to some uncommon interleavings of messages on the different networks. In our case, one of the main challenges in the DHCCP protocol consists in counting the correct number of responses for each coherence transaction, since an incorrect count will eventually result in a deadlock.

Our goals were first to model formally the DHCCP protocol so as to prove the absence of deadlocks. We also wanted to prove simple functional properties, such as every request receives a response, or a shared copy modification eventually leads to an invalidation or an update of the other copies. Proving these properties would greatly strengthen our confidence in the protocol.

We did not expect verification to scale up to the full 1024 core design, but that is not truly necessary due to the symmetry of DHCCP. Our main goal was then to target a platform configuration that would exhibit all the characteristics of DHCCP by working up progressively from smaller configurations. To exhibit all relevant properties we determined that we needed a threshold for DHCCP that is strictly less than the number of possible copies of a data, so all types of coherence transactions can occur, so we need to scale to a design with at least three processors. Another parameter we identified is that at least two addresses should be represented to show that no problem arises from the sharing of the channels between the different L1 caches and L2 banks.

Architecture Parameters. In order to be able to verify properties, the components in the architecture need to be abstracted, and some restrictions have to be made. The modeled components are processor, L1 cache, L2 cache, memory and interconnect channels. All of these components except channels are modeled as reactive communicating automata, they have a control location or `state` that defines which messages they can send or receive.

As seen in the section on the interconnect, the topology is logically “flat” , and is modeled as such. Channels have the capability to serialize requests coming from different sources, and to route a request to different destinations.

The architecture parameters are the following:

- Number of processors and their L1 caches: `NB_PROC`.
- Number of cache lines in the L2 cache: `NB_L2`. Each L2-cache bank contains only one line in the model, and thus several cache lines translate into several L2-cache banks.
- DHCCP threshold: `CACHE_TH`.

This parametrized description should be preserved during the modelling to be able to easily consider configurations of increasing complexity.

There are three significant restrictions in the models we built: 1. the single line L2-cache bank, 2. the assumption that the L2 is large enough to contain all memory lines, and 3. the absence of data modelling.

The first restriction prevents a L2-cache bank from receiving on its port requests at different addresses. This is a reasonable simplification, as there is globally not much interaction between requests targeting different cache lines in the L2 cache. The second restriction prevents several memory addresses from being in the same L2 bank; lifting this restriction would require to modify the L2 state machine and add several variables into it (thus yielding even quicker combinatorial explosion). The third restriction means that we only model the control part of the protocol, and abstract away the content of the data in the lines of cache. Adding data is easy from a modelling point of view, and would enable verification of some coherency properties but these data values do not impact correctness of the protocol (e.g. deadlock freedom) and they would participate in state space explosion.

Table 1 describes the variables we considered relevant inside each component. The processor model is very simple and sends random read or write requests at a random address. The L1 cache contains only one line, and the line validity is defined by the cache state. Each L1 cache is defined with a unique identifier. Each L2 cache contains the line address it can hold, along with arrays for the explicit list of copies, and a variable to store the copies count. A channel interconnects two components and is modeled as a one-place buffer. There is one channel per network, i.e. 5 between L1 caches and L2 banks for the case of DHCCP.

Figure 2 shows a system instance with `NB_PROC = 2`, `NB_L2 = 2`, and `CACHE_TH = 2`, and with some of the components variables values. Appendices A and B show the finite state machines for the L1 and L2 caches. All states are represented, but some transition actions are omitted for clarity. Appendice C describes the different messages modeled and the channels on which they are conveyed. These messages types are directly obtained from the DHCCP description in section 2. Figure 3 shows the dynamics, through a sequence diagram describing a cache miss.

Table 1: Components modeled with their variables

Processor		**Channel**	
`state`	FSM state	`address`:	target address of the request
`addr`:	address of the last request emitted	`id` (optional):	identifier of the request sender
		`type`:	request type
L1 cache			
`state`		FSM state	
`id`:		identifier of the cache (different for all caches)	
`v_addr`:		address contained in the cache when it is valid (validity is encoded in the sate)	
`addr_save`:		address of the last request sent	
L2 cache			
`state`		FSM state	
`line_addr`:		address of the line mapped in this L2-cache bank	
`n_copies`:		number of copies for this line	
`dirty`:		true if the line has been modified w.r.t. the memory	
`src_save` and `src_save_clnup`:		used to save a request source id when a coherence request is needed	
`cpt`, `cpt_clnup` and `rsp_cpt`:		counters used for the sending of multicast or broadcast requests	
`v_c_id[]` and `c_id[]`:		arrays of size CACHE_TH storing the explicit list of copies; if `v_c_id[i]`== 1, then entry `i` in `c_id` is valid and `c_id[i]` contains the corresponding cache identifier.	

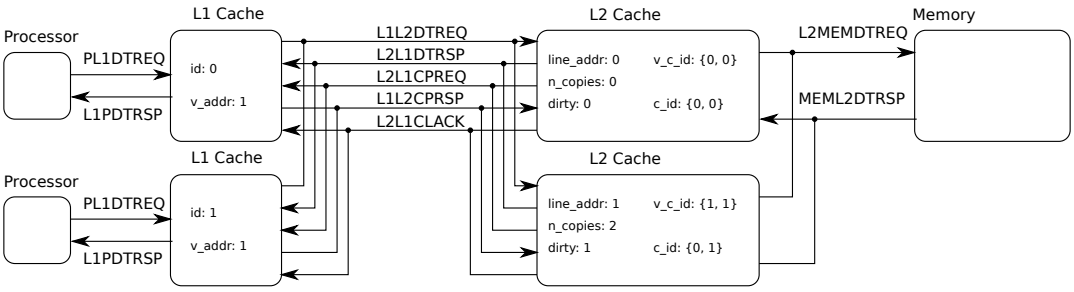

Figure 2: Example of an architectural state modelisation, in which the line with address 1 has a copy in both L1 caches

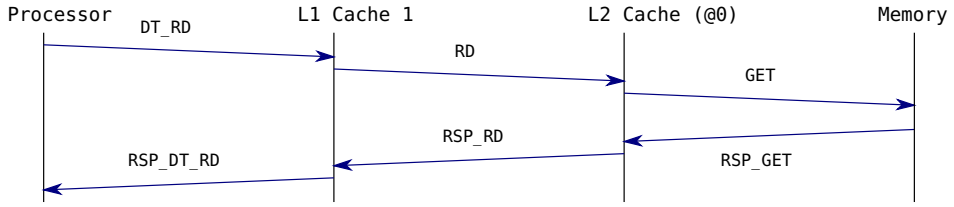

Figure 3: Message Sequence in case of misses in the L1 and L2 caches. The messages are the following:

(1) The processor connected to the L1 cache sends a read request (message `DT_RD`) for the line with address 0 (`PL1DTREQ` channel).

(2) The L1 cache does not own a valid copy of the line and sends a `RD` message for the line at address 0 (`L1L2DTREQ` channel).

(3) The L2 cache in charge of address 0 receives the request and, as it misses, sends a `GET` message for address 0 to the memory (`L2MEMDTREQ` channel).

(4) The memory responds with a `RSP_GET` message to the requesting L2 (`MEML2DTRSP` channel).

(5) The L2 cache updates it state for the line and responds to the L1 with a `RSP_RD` message (`L2L1DTRSP` channel).

(6) Upon receiving the response, the L1 cache updates its state and responds to the processor with a `RSP_DT_RD` message (`L1PDTRSP` channel).

4 Modeling using Communicating Process

Promela Models. To model DHCCP we first used the Promela language and its supporting tool Spin [10]. It offers as modeling framework asynchronous process communicating over channels. The language itself is relatively comfortable, each component is described using code-like control structures (case, loop). The tool can exhibit traces as sequence diagrams, which are particularly valuable to develop and debug the model.

We first built [8] up automata of the behavior of the L1 cache and the memory controller, and abstracted the activity of processors to arbitrary read and write requests. These automata were then encoded into Promela using labels and goto.

To validate the models, we used properties encoded into "observation" automata, synchronized with the system. In some cases, adding these automata proved to be a problem, as they can incorrectly block the system if not well designed. Also, to observe channels we had to duplicate channels and messages (one for the true channel, one for the observer) which is quite intrusive. Overall, this observation mechanism was quite cumbersome, and participated in state space explosion.

We separately analyzed the components in simple configurations before assembling them. The simple configurations helped validate that the Promela code correctly reflected the automata. Full model-checking was however only possible on the simplest instances, with a single processor and two addresses. For the full system, we were only able to use simulation and bounded model-checking (up to roughly 10^8 states).

This Promela model was then extended and refined with the same goals in mind [7]. We simplified and abstracted the data manipulation, and removed the observers. We were still unable to explore the full state space for three processors, reaching 270 steps in depth for 10^9 states. We explored instead two configurations with two processors, while varying the threshold variable. We were also able to include the LL/SC (Load-link/Store-conditional) support in the TSAR architecture, leading unfortunately to much more complex automata, as more control messages were introduced.

While partial-order reduction was activated, we could not activate d_step as every set of actions contains at least one channel interaction. The channels themselves are shared for both writing and reading (precluding use of xr/xs keywords) since it models a bus.

In conclusion, despite some aggressive simplifications (e.g. full data abstraction), we were unable to fully verify deadlock freedom for Promela models for even the smallest truly relevant configuration, i.e. at least three processors, two addresses, and a threshold at 2 so that both multicasts and broadcasts can occur. The language and simulator were however quite comfortable to use.

Divine Models. We then built a second set of models [5], but this time using the Divine [2] language. Similarly to Promela, Divine is a language to express systems as (asynchronous) processes communicating over channels. However, it is a much simpler language with less features. Each process is described using an automaton with "local" variables ; transitions have a source and target state, a guard or enabling condition over variables, may send or receive messages from channels and update variables.

We chose to use Divine mostly because we wanted to try other tools than Spin. Thanks to having a relatively simple syntax and semantics, and also due to the existence of a nice set of benchmark models [9], several tools provided support for the language. The Divine tool coming with the language handled LTL in multi-core and distributed settings, LTSmin [3] offered support for both explicit and symbolic exploration, and building upon new results [4], a prototype path to input Divine models to our symbolic model-checker ITS-Tools had been recently built.

Building the model in Divine was harder than in Promela however. On the positive side, we have good control over the atomicity of statements; in the Promela models, due to interaction of send/receive actions with the `atomic` keyword, some interleavings of independent progression of different process were still observable. For instance, a (passive) component that consumes a message and immediately sends an answer on a different channel could not be modeled without the state between receive and send being materialized.

On the other hand, we fell into some difficulties to model channels; Promela let us peek at the content of a channel without consuming it, enabling a routing mechanism where an entity only consumes messages which are addressed to it. Without this mechanism, we were in fact unable to model the semantics we needed; we had to resort to explicit modeling of the channel as shared global variables, and simulating read and write operations with instructions. Divine's support for parametric modeling

was also poor, since it basically recommends the use of the macro expansion tool *m4*, which is not comfortable.

During this internship, we again built separate components, then progressively assembled them to build more complex configurations. We used LTL (instead of observation automata) to express expected properties of the system, under fairness constraints that force all of the processes to progress. Without fairness, most of the properties were not valid, which was expected, but unfortunately fairness is not supported by all the tools.

With these models, we were able to reproduce a real deadlock found on a previous version of the DHCCP protocol. On the configuration with two processors, two addresses and a threshold set at 1, the scenario exhibited by the sequence diagram in Figure 4 could occur. It leads to an incorrect counter value, that ultimately leads to a deadlock by propagation as the head message in the FIFO channel cannot be consumed and communications lock down.

This bug was detected in the cycle-accurate TSAR prototype by its designers, using testing and simulation, but it took one year to detect this issue and correctly diagnose it. Building a solution to correct the problem was non trivial, and took another six months. In contrast, once the formal models were built, detecting it was easy, even for a relatively non expert Master 2 student, and model-checking could exhibit readable diagnostic traces. We were then able to modify our model to match the next (correct) version of the DHCCP protocol described in section 2, and we checked that the deadlock issue was indeed resolved.

However, we were still unable to fully verify larger configurations with three or more processors. The use of ITS-tools was possible only for deadlock detection, as fairness constraints were unavailable to check the more complex LTL properties. The input from Divine to ITS-tools also involves several automated transformation steps, that yielded a relatively complex model due in part to channel communications modeled as shared variables, and to the loss of structural information in the transformation (i.e. from a set of processes to a single large specification). Experimentation with LTSmin was not extensive, but we measured performance similar to that of the Divine tool; again chains of transformations and relatively poor tool integration made this path less comfortable than just using Divine natively.

In conclusion, using Divine models, we were able to successfully reproduce a critical bug in the TSAR coherence protocol, and to prove that the patch did solve the problem. However, we were still unable to perform verification for larger instances with three or more processors.

5 Modeling with GAL

The third set of models were built using the Guarded Action Language (GAL) [6] during a Master Thesis [11]. GAL is a language offering very fine control over the expression of concurrent semantics, with no assumptions on the existence of higher level constructs such as processes or channels and very few keywords.

Guarded Action Language. GAL is a formalism supporting hierarchical descriptions of components; terminal or leaf components are specified as GAL type declarations, while composite type definitions allow to instantiate existing types (of GAL or composite nature) and synchronize these instances. A GAL specification is then composed of a set of type declarations, and a specific instance `main` which is designated as the full system. These characteristics of the language, borrowed from architectural description languages, help reuse model elements in various configurations easily.

A GAL type declaration defines a set of integer variables and fixed-size arrays of integers as variables. A state of a GAL is then a complete assignment of an integer value to each of these variables. Transitions

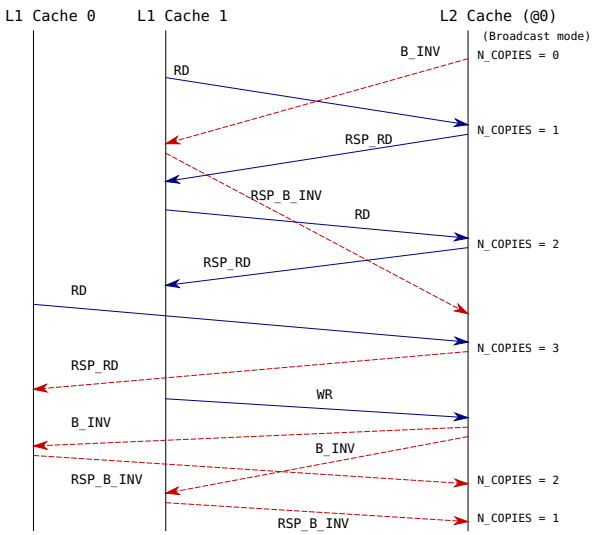

Figure 4: Part of a deadlock sequence in a previous version of the DHCCP protocol. Blue arrows represent direct messages while red arrows represent coherence messages. Messages can be received out of order due to the existence of several channels between the memory controller and the L1 cache. The `RSP_B_INV` message was exclusive to that version of DHCCP. At the end of the sequence, the line count in the L2 is one whereas there are no copies left.

are defined as a triplet $\langle g, l, a \rangle$, where g is a boolean enabling condition or guard, l is a label chosen from a finite set, and a is a sequence of statements or actions that must be executed atomically. A statement can be an assignment to a variable of an expression computed over variables, or a call to a label. The call statement is resolved by finding a transition bearing the target label, whose guard is enabled, and executing it.

A composite type declaration defines a set of instances and fixed-size arrays of instances as variables. A state of a composite is then a complete assignment of a subcomponent state to each of these instances. Synchronizations are defined as a pair $\langle l, a \rangle$, where l is a label chosen from a finite set, and a is a sequence of call statements that must be executed atomically. A call statement has a target, which can be either the enclosing instance itself, or any nested instance. The target of the call must then evolve through a transition (or synchronization in composite case) that bears the target label.

GAL also offers parametric modeling features, in the form of parameters defined over a discrete range of values[2]. These parameters let us define parametric transitions, that correspond to a set of non parametric transitions, one per possible value of the parameters. Parameters can be used to define parametric labels, in order to model communications over discrete data types as calls to labels.

GAL is mostly designed to be the target in a model transformation process, where the specification is typically expressed using a domain specific notation such as Promela or Divine, and automatically translated to GAL for analysis purposes. Going for direct modeling in GAL however, gives us proximity to the symbolic solution engine, enabling the use of advanced features that are not built into the general purpose transformation, e.g. from Divine.

In particular, the automatic transformation loses structural information, yielding a single GAL com-

[2]In the syntax, parameters are distinguished from variables by a $ sign.

```
1  typedef addr_t = 0 .. $NB_L2 - 1 ;
2  typedef type_t = 0 .. 19 ;
3  typedef id_t = 0 .. $NB_PROC - 1 ;
4
5  gal ChannelAddrType {
6   int isFull = 0 ;
7   int addr = 0 ;
8   int type = 0 ;
9   transition read
10   (addr_t $addr, type_t $rtype)
11   [isFull == 1 && addr == $addr
12    && type == $rtype]
13   label "read" ($addr, $rtype) {
14    isFull = 0 ;
15    addr = 0 ;
16    type = 0 ;
17  }
18  transition write
19   (addr_t $addr, type_t $wtype)
20   [isFull == 0]
21   label "write" ($addr, $wtype) {
22    isFull = 1 ;
23    addr = $addr ;
24    type = $wtype ;
25  }
26 }
27 gal CacheL1 {
28  int state = $INIT ; // state in the automaton
29  int v_addr = 0 ; // address in cache if VALID
30  int addr_save = 0 ; // saves the address
31                      // of a sent request
32  int id; // fixed identifier of this L1 cache
33
34  transition t_init (id_t $id)
35   [state == $INIT]
36   label "init" ($id) {
37    state = $L1_EMPTY;
38    id = $id;
39  }
40  transition t_Empty_WriteWaitEmpty
41   (id_t $id, addr_t $addr)
42   [state == $L1_EMPTY && id == $id ]
43   label "read_PL1DTREQ_write_L1L2DTREQ"
44      ($id, $addr, $DT_WR, $WR) {
45     state = $L1_WRITE_WAIT_EMPTY ;
46     addr_save = $addr;
47  }
48 ...
```

Figure 5: GAL encoding of a channel carrying messages (5–26) featuring a type (8) and an address field (7). Part of the GAL modeling the L1 cache (27–46).

ponent instead of a composition of process. The encoding of channels as global variables also prevents the transformation from automatically building a representation using synchronizations on labels.

Elementary Components. We separately developed the models of the communication channels and of the various components. For normal components, we model automata by defining a `state` variable, then adding a GAL transition for every transition of the automata.

For channels, we built a GAL with enough variables to store the message, and a boolean flag to test if the channel is full. For each possible message going through the channel (which must be a finite domain) we generate two transitions *read* and *write* with a distinct label for each of them. In this example (see Figure 5) the channel messages carry a target address (among possible memory slots) and a message type (among 20 possible values, e.g. update request). The read operation also flushes the state of the channel, to prevent this unreadable state information from participating in the state space explosion.

The processor is modeled as a three state automaton, alternating between an idle state and a state where a read (resp. write) request is sent to the L1 cache on an arbitrary address. Then, it awaits the reply to go back to idle. The Memory cells are even simpler: since data values have been abstracted away, they feature a single state and two transitions that simply acknowledge and reply appropriately to `PUT` and `GET` requests.

The L1 and L2 cache are much larger with respectively 14 and 16 states. They also feature variables that increase the state space size significantly. When transitions of an automaton read or write on channels, we add labels to those actions, indicating which channel is used. The message data is filtered by specifying values in the target label. For instance, the transition from `EMPTY` to `WRITE_WAIT_EMPTY` shown in Figure 5 reads a `DT_WR` request from the channel `PL1DTREQ` and writes a `WR` to the channel

```
composite ProcessorCacheL1 {
 Processor p;
 CacheL1 c;
 ChannelAddrType chan_PL1DTREQ;
 ChannelAddrType chan_L1PDTACK;

 synchronization init (id_t $id)
  label "init" ($id) {
   c."init" ($id);
 }
 // local communications are unlabeled
 synchronization s_write_PL1DTREQ
  (addr_t $addr, type_t $type) {
   p."write_PL1DTREQ" ($addr, $type) ;
   chan_PL1DTREQ."write" ($addr, $type) ;
 }
 // exposing the ports of the L1 cache
 synchronization s_read_L2L1CPREQ
  (id_t $id, addr_t $addr, type_t $type)
  label "c_read_L2L1CPREQ"($id, $addr, $type) {
   c."read_L2L1CPREQ"($id,$addr, $type);
 }
...
```

Figure 6: Composite encoding a component nesting a Processor, a L1 cache and two channels.

`L1L2DTREQ`. Some components such as the L1 cache have a set identifier defined at initialization, that is used to tag outgoing messages, or to filter messages according to their target address. Transitions without communications on channels are left unlabeled, and can thus occur at any time.

Assembling a Configuration. From these models of channels and components we hierarchically build a representation of the full system. A first composite type `CompositeCacheL1` is defined containing an instance of a Processor, an instance of a L1 cache, and two instances of channels connecting them together.

Channels are connected to appropriate endpoints using synchronizations. Unlabeled synchronizations are used to label local communications within the composite. The synchronization `s_write_PL1DTREQ` shown in Figure 6 is an example of this, and models the processor `p` sending a write request. The contents of the message (address and type) could be anything at this synchronization level. The labels that correspond to communications between the L1 cache and the L2 cache are however reexposed as labels of the composite, which simply forwards the request to the L1 cache instance.

We then build a top level composite acting as `main` that contains an array of instances of `CompositeCacheL1`, an array of instances of L2 caches and all the channel instances connecting them. The whole description is parametric and controlled by the three parameters set at the top of the file. Several other configurations were built to test each component in isolation.

Experiments and Measurements. On these models we were able to prove absence of deadlock and some logical properties expressed in CTL. Overall we considered all configurations with up to `NB_PROC` = 6 processors, up to `NB_L2` = 3 L2 addresses, and a `CACHE_TH` between 1 and 3. For each configuration, we ran the verification on an Intel Xeon 2.6 GHz machine with a limit set to 8 hours to complete the simulation, and a maximum of 192 GB of memory. Table 2 reports on the experiments that finished within the time and memory constraints, and gives their time, the number of accessible states and the memory used.

The state space size is relatively modest, compared to the models crafted in other languages, thanks to the fine control over the atomicity of steps offered in GAL. We were indeed able to scale up to the configurations of interest, i.e. 3 processors, 2 L2 cache, and a threshold of 2.

A set of 16 properties were expressed in CTL, covering request response scenarios such as ”any time two processors share an address and one writes on it, the other one eventually gets an invalidate request.”

Table 2: Runtime (in seconds) and memory (in MB) required for the configurations of the parameters NB_PROC,NB_L2,CACHE_TH which could be explored.

PROC	L2	TH	States	Time	Mem
1	1	1	51	0.05	5
1	1	2	52	0.05	5
1	1	3	53	0.06	5
1	2	1	555	0.13	7
1	2	2	565	0.12	7
1	2	3	575	0.18	8
1	3	1	4503	0.47	15
1	3	2	4572	0.52	15
1	3	3	4641	0.44	15
2	1	1	7070	0.54	14
2	1	2	1892	0.46	15
2	1	3	2160	0.47	15
2	2	1	681471	14	232
2	2	2	68401	5	98
2	2	3	77449	6	99
2	3	1	2.76e+07	640.14	3482
2	3	2	1.13e+06	44	568
2	3	3	1.27e+06	46	599
3	1	1	175234	3	57
3	1	2	226329	35	326
3	1	3	130450	51	625
3	2	1	4.32e+07	860.58	3871
3	**2**	**2**	6.77e+07	3692	7547
3	2	3	1.54e+07	1396	6315
3	3	3	5.14e+08	18759	66733
4	1	1	3.36e+06	44	662
4	1	2	5.53e+06	184	1645
4	1	3	1.04e+07	574	3285
4	2	2	4.49e+09	149471	171457
5	1	1	6.06e+07	254	1389
5	1	2	1.08e+08	9315	9439
6	1	1	1.05e+09	831	3710
6	1	2	1.87e+09	12324	23118

6 Conclusion

We presented a case-study focusing on the modeling of a cache-coherence protocol in GAL, and discussed previous implementations using other languages and tools for the same protocol. We noticed a certain difficulty, for the students who worked on the subject, to assimilate the underlying formalism of each tool. We also noticed how small changes in the language semantics can result in big changes, either in the model description or in the state space size. In particular, the communication and synchronisation primitives offered by a language are of high importance for getting clean and efficient models.

The parametric and compositional features of GAL proved adequate to write models directly by hand. The parametric features are useful both for writing a model with several instances of a same module (e.g. cache) and for exploiting internally the similarities between these modules to improve verification efficiency.

Overall, this case study showed that model-checking tools could highlight real bugs and could run on problems of substantial size, provided appropriate formal models can be built accurately.

Acknowledgements

This work would not have been possible without the contributions of the students working on the project Mohamad Najem, Akli Mansour, Zahia Gharbi and Di Zhao.

References

[1] *TSAR: Tera-Scale Multiprocessor ARchitecture home page*. Available at https://www-soc.lip6.fr/trac/tsar.

[2] Jirí Barnat, Lubo Brim & Petr Rockai (2009): *DiVinE 2.0: High-Performance Model Checking*. In: *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, IEEE Computer Society Press, pp. 31–32, doi:10.1109/HiBi.2009.10.

[3] Stefan Blom, Jaco van de Pol & Michael Weber (2010): *LTSmin: Distributed and symbolic reachability*. In: *Computer Aided Verification (CAV)*, Springer, pp. 354–359, doi:10.1007/978-3-642-14295-6_31.

[4] Maximilien Colange, Souheib Baarir, Fabrice Kordon & Yann Thierry-Mieg (2013): *Towards Distributed Software Model-Checking using Decision Diagrams*. In: *Computer Aided Verification (CAV)*, *LNCS 8044* , Springer Verlag, pp. 830–845, doi:10.1007/978-3-642-39799-8_58.

[5] Zahia Gharbi (2013): *Vérification compositionnelle du Protocole de Cohérence de Cache de la Machine Multiprocesseur TSAR (in French)*. Master's thesis, Université Pierre et Marie Curie.

[6] *ITS-tools model checker and GAL language home page.* Available at `http://ddd.lip6.fr/`.

[7] Akli Mansour (2012): *Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseur TSAR : Absence de deadlocks (in French)*. First Year Master Student Project, Université Pierre et Marie Curie.

[8] Mohamad Najem (2011): *Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseur TSAR (in French)*. First Year Master Student Project, Université Pierre et Marie Curie.

[9] Radek Pelánek (2007): *BEEM: Benchmarks for Explicit Model Checkers*. In: *Model Checking Software, 14th Int'l SPIN Workshop*, *LNCS* 4595, Springer, pp. 263–267, doi:10.1007/978-3-540-73370-6_17.

[10] *Spin model checker home page.* Available at `http://spinroot.com/`.

[11] Di Zhao (2015): *Vérification de protocole de cohérence de cache hybride multicast/broadcast avec les techniques de model-checking (in French)*. Master's thesis, Université Pierre et Marie Curie.

A L1 Cache Finite State Machine

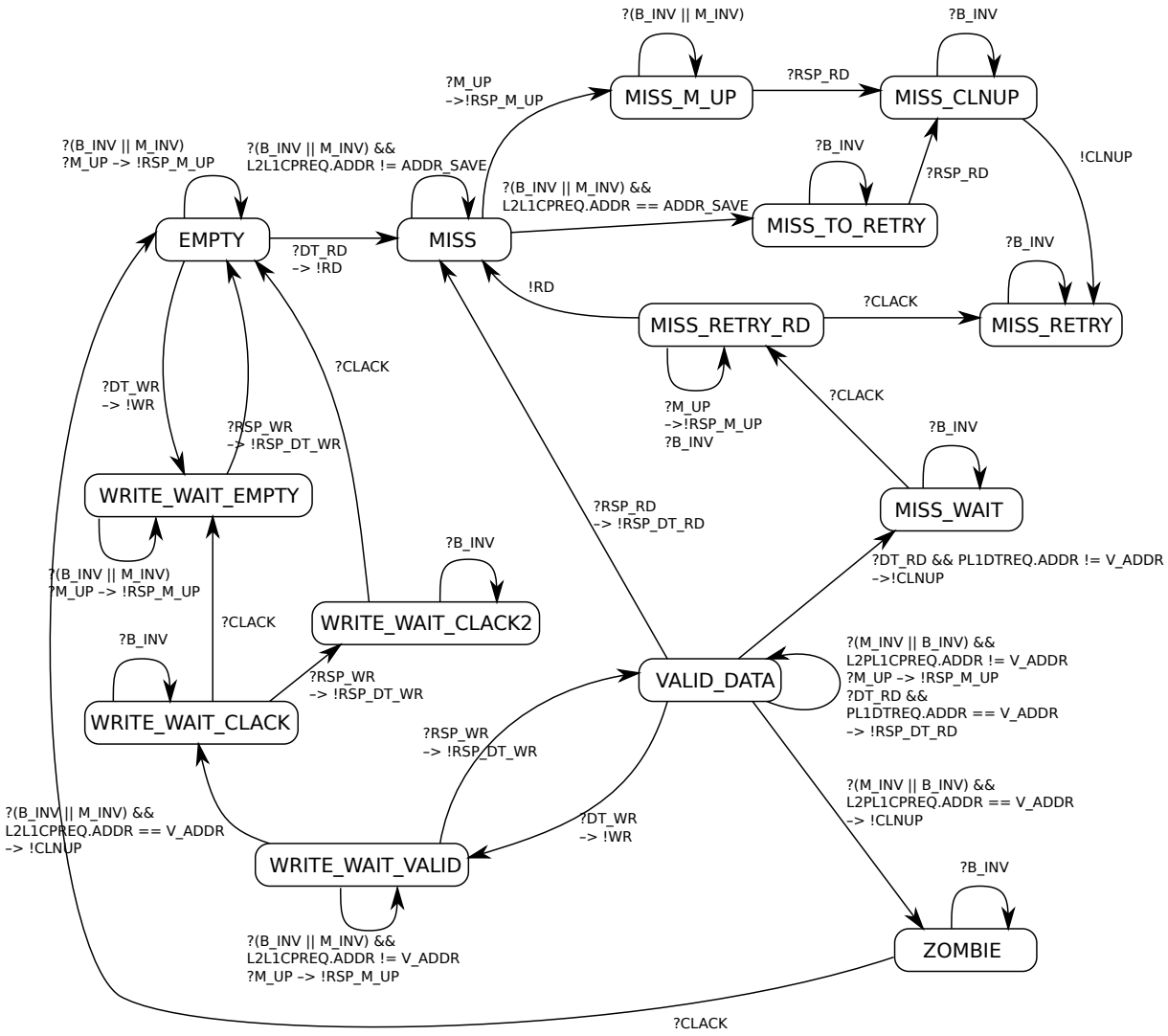

Figure 7: L1 cache finite state machine. '?' denotes the reception of a message; '!' denotes the sending of a message; $\rightarrow$ denotes actions associated to a guard. Some actions are omitted for readability.

B L2 Cache Finite State Machine

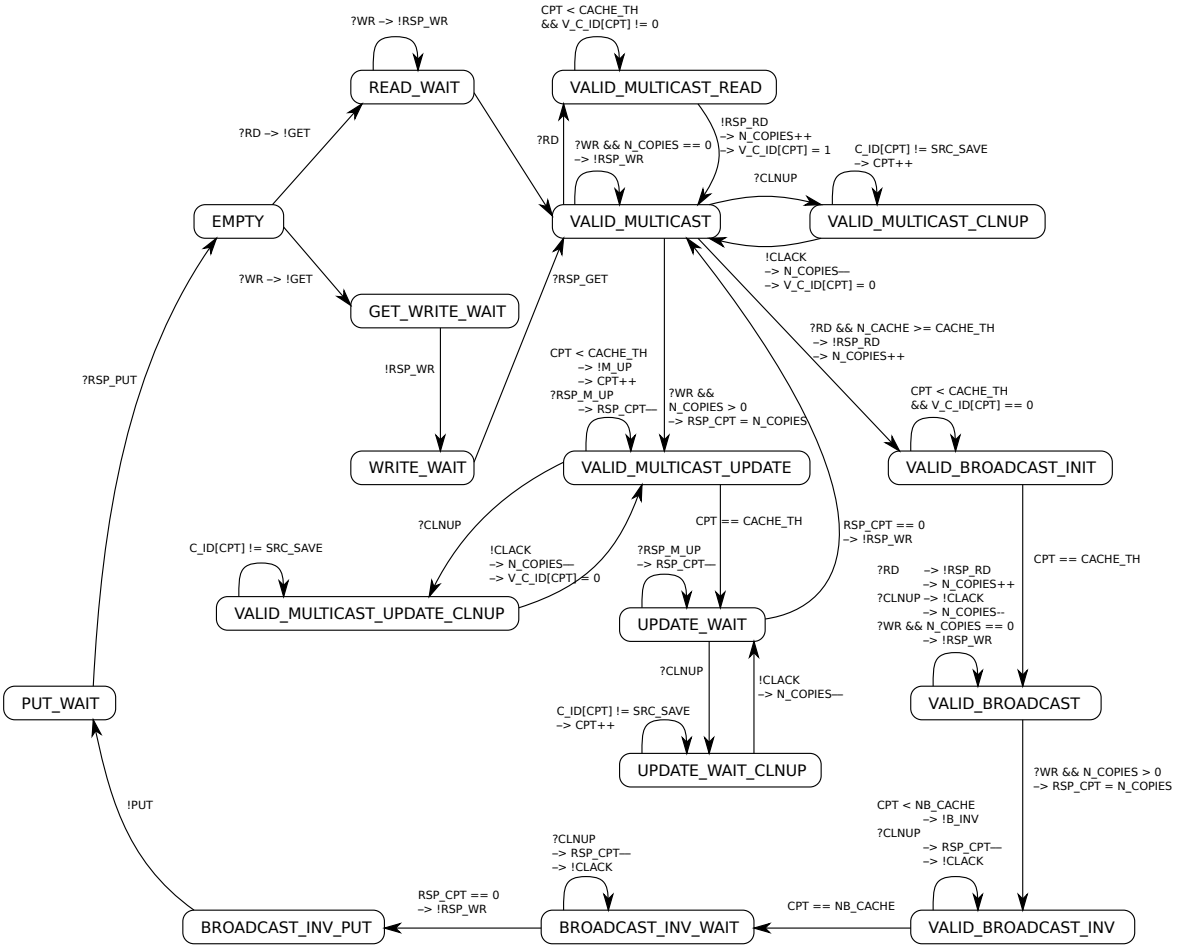

Figure 8: L2 cache finite state machine. '?' denotes the reception of a message; '!' denotes the sending of a message; $\rightarrow$ denotes actions associated to a guard. Some actions are omitted for readability.

C Types of Messages and Associated Channels

Table 3: Types of Messages and Channels on which they are Transported

DT_RD	Read request from processor	PL1DTREQ
DT_WR	Write request from processor	PL1DTREQ
RSP_DT_RD	Read response to processor	L1PDTRSP
RSP_DT_WR	Write response to processor	L1PDTRSP
RD	Read request from L1 to L2	L1L2DTREQ
WR	Write request from L1 to L2	L1L2DTREQ
RSP_RD	Read response from L2 to L1	L2L1DTRSP
RSP_WR	Write response from L2 to L1	L2L1DTRSP
CLNUP	Line eviction from L1 to L2	L1L2CPRSP
CLACK	Line eviction received from L2 to L1	L2L1CLACK
B_INV	Broadcast invalidate	L2L1CPREQ
M_INV	Multicast invalidate	L2L1CPREQ
M_UP	Multicast update	L2L1CPREQ
RSP_M_UP	Multicast update acknowledge	L1L2CPRSP
GET	Read line from memory	L2MEMDTREQ
PUT	Write line to memory	L2MEMDTREQ
RSP_GET	Response to GET request	MEML2DTRSP
RSP_PUT	Response to PUT request	MEML2DTRSP