Integrating Formal Verification with Murφ of Distributed Cache Coherence Protocols in FAME Multiprocessor System Design

Ghassan Chehaibar

BULL, Platforms Hardware R&D
Rue Jean Jaurès, F-78340 Les Clayes Sous Bois, France
ghassan.chehaibar@bull.net

Abstract. Flexible Architecture for Multiple Environments (FAME) is Bull architecture for large symmetrical multiprocessors based on Intel's Itanium® 2 family, which is used in Bull NovaScale® servers series. A key point in the development of this distributed shared memory architecture is the definition of its cache coherence protocol. This paper reports experiences and results of integrating formal verification of FAME cache coherence protocol, on 4 successive versions of this architecture. The goal is to find protocol definition bugs (not implementation) in the early phases of the design, focusing on: cache coherency, data integrity and deadlock-freeness properties. We have performed modeling and verification using Murφ tool and language, because of its easiness of use and its efficient state reduction techniques. The analysis of the results shows that this approach is cost-effective, and in spite of the state explosion problem, it has helped us in finding hard-to-simulate protocol bugs, before the implementation is far ahead.

1 Introduction

Design and verification of complex systems are an outstanding application domain of formal methods. Cache coherence protocol of symmetric multiprocessor (SMP) over a distributed architecture is indeed a very complex system, where concurrency of transactions issued by different agents and the resulting conflicts are very difficult to master and verify without the help of rigorous analysis. Such help is provided by formal methods that allow to describe behaviors in a precise unambiguous language and to automatically prove properties of these descriptions.

Flexible Architecture for Multiple Environments (FAME) is Bull architecture to design large SMPs that can include up to 32 processors [4]. It is based on Intel Itanium®2 family and commercialized in the Bull NovaScale® server series [1]. This non-uniform access memory (NUMA) distributed shared memory multiprocessor is organized in modules managed by a key component, the FAME Scalability Switch (FSS). A FAME machine is obtained by connecting up to 4 modules, through an interconnection network that links the FSSs (Fig. 1 shows the module structure).

From the very beginning of this project we have applied formal protocol verification to the cache coherence protocol of 4 successive versions of this architecture. (Formal verification results of the first version are partially mentioned in [15].)

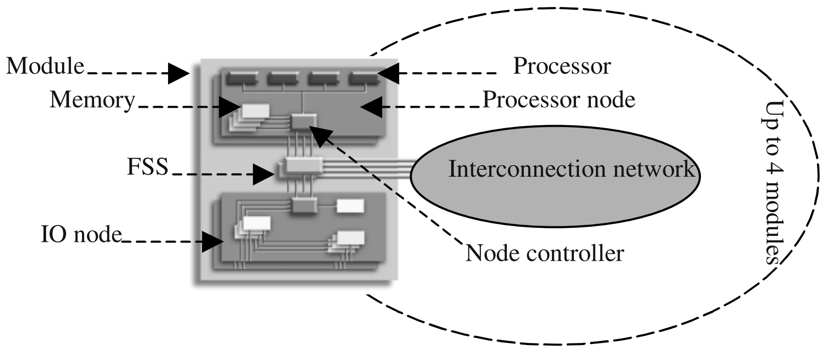

Fig. 1. FAME module architecture. Each *module* contains processor *nodes* and IO nodes that are connected by a switch called FAME Scalability Switch *(FSS).* Here a module contains 2 processor nodes and two IO nodes, a processor node contains four processors and a memory subsystem

Our goal is to apply formal verification as a design aid [3], in order to find protocol definition bugs (not implementation) in the early phases of its specification and to increase confidence in its correctness. Protocol specification verification differs from other formal verification activities that address hardware implementation correctness. like formal verification of properties of the register-transfer level (RTL) descriptions, or equivalence checking between RTL and gate levels. Starting from a reference specification we build an abstracted, simplified and downsized model of the protocol and check that it verifies some properties. As we will see, this approach is cost-effective and allows finding hard-to-simulate protocol bugs before the implementation is far ahead, in spite of state explosion problem.

Among all requirements that must be implemented by FSS, we focus on the essential function of keeping memory coherent, which is ensured by the cache coherence protocol. Thus, formal modeling and verification address this protocol, focusing on coherence handling aspects, abstracting anything else, like routing, networking and resource management.

In order to show the complexity of the problem addressed, Section 2 gives an overview of a distributed cache coherence protocol like FAME's one, highlighting the main issues (conflict handling, race conditions and data integrity) and defining the properties we aim to check. Based on these properties, Section 3 states and informally justifies the protocol abstractions done in the modeling process: event aggregations and resource simplifications. Then in Section 4, we summarize the features of Murφ language and tool, which have made us choose it to model and verify our cache coherence protocol: amenity of the language, shortest explicit error traces, efficient state reduction techniques (symmetry and hash compaction) and asynchronous semantics. In Section 5, we analyze the results obtained in the modeling and verification of the four versions, from two viewpoints: the incremental modeling and verification process, and the cost-benefit Figures. Finally we draw our conclusions from this experience, summing up the benefits of this approach.

2 FAME Cache Coherence Protocols Issues

In order to give insights of the complexity of the addressed problem, we describe the features of cache coherence protocols in distributed shared-memory architecture [7]. We give some information on FAME protocol specifically, *without disclosing the details of this proprietary protocol.*

2.1 Distributed Cache Coherence Protocol

A private cache is associated to a processor in order to reduce the effects of memory access latency and contention. In shared-memory multiprocessor, a memory location can be present in several caches, thus introducing a *consistency* problem. A *cache coherence protocol* ensures that memory is kept coherent, that is, any change made to a memory location is made visible to all other processors. A common solution is to associate to each cache *line* (transfer unit between memory and caches) a state and associated access rights. When a processor initiates an access compatible with the line state, it is performed in the cache (it is a *hit*); otherwise it issues a transaction on the bus (it is a *miss*).

In *writeback* caching policy, all processor loads and stores are performed in the cache: thus even when a processor needs to write a location, first it fetches in its cache the memory line that contains this location, invalidating all the other caches (*read with invalidation* request). *Replacement* occurs when a processor needs to put a new line in its cache, and all the entries that it can fit in (depending on the organization of the cache) contain valid lines: then a replacement algorithm selects a line to be evicted from the cache: if it is not modified, this can be done silently; otherwise, a *memory update* request is sent to memory.

FAME protocol is based on the classical 4-state protocol called MESI [12] (acronym formed by the state initials): M (modified line, this cache owns the only valid copy of the system, and any access by its processor is a hit; this cache is responsible of providing data to other caches), E (exclusive, this cache is the only one to hold a copy, but it is the same as in memory; any access is a hit and a store will change it to M), S (shared line, it can be present in other caches, and data value is the same as in memory; a load causes a hit, but a store causes a miss), I (invalid line: not present or present but stale; any access is a miss). (Sometimes, M state is called *dirty* in the literature).

A cache coherence protocol defines the rules of handling the requests issued on a miss: how to get information on all cache states, cache state transition rules, where and how to send requests, where to find data, collision handling (concurrent requests to the same line). There are two basic kinds of protocols: snoopy-based and directory-based protocols.

In a snoopy-based protocol, any request is snooped by all processors and memory, and their responses are also snooped in a synchronous way: thus memory and cache controllers have all needed information in a synchronous way and can take appropriate actions. This protocol is suitable for a bus-based architecture and does not scale to distributed systems.

In a directory-based protocol, the original idea is a directory that indicates for any line contained in a processor cache, its state and the list of caches that contain it. In distributed shared-memory architecture, where there is a virtual unique global memory address space but memory is physically distributed, each memory piece has its associated directory. Then on a miss, a request to a line mapped in a memory slice (called the *home* memory of the line) is sent to its attached directory, which forwards request to the concerned caches, instead of the bus-broadcast scheme in snoopy-based case. Actually these directories can be distributed in various ways including grouping some of them in one directory or defining directory hierarchy.

As in caches, there are also replacements in directories: when a an entry holding the state of a line has to be evicted out of the directory, then the directory sends invalidation requests to all the caches that hold a copy of this line.

Often in actual implementation of distributed shared memory architecture, both kinds of cache coherence protocol are combined. In FAME, within a processor node, there is a bus-based snoopy-protocol that interacts with a directory based protocol at the module level. All directories are grouped in FSS.

2.2 Cache Coherence Correctness Properties

A cache coherence protocol aims to keep *memory coherent* not to implement some *memory consistency model*, like sequential or processor or weak or release consistency. Any memory model assumes basic memory coherency that is: all writes to the *same memory location* are seen in the same order by all processors [6] (otherwise, you cannot even implement a lock; notice the difference with sequential consistency for instance, where the set of accesses to *all memory locations* is seen in the same order by all processors).

Therefore, the properties to verify are:

1. Cache and directory state coherency, following the definition of the MESI states: for instance, when a line is E/M in some cache, it is I elsewhere. Since directories contain information about caches, there are inclusion relations between cache state and directories. When there is directory hierarchy, there are inclusion relations between directories.
2. Data integrity: a processor does not read stale data and no data modification is lost. This requirement is not implied by cache state coherency. For instance, as said above, a memory update is performed when a cache evicts a line in M state. After the eviction all caches are I (so the states are coherent), but there is an ongoing memory update. If a read request issued by a processor can get to memory before the update (race condition between read and write), it will get stale data.
3. Deadlock-freeness: actually, a lot of deadlock and starvation issues are related to resource management and so are implementation-dependent. Still, at the protocol level, we have an abstract view of outstanding resources that are used to handle coherency like directories and buffers that track request progression.

Besides, deadlock issues rise in coherence conflict resolution policies, where a colliding request can be held-off or retried.

2.3 Cache Coherence Issues

The main behavior issues of a cache coherence protocol, which we derive from the properties to verify, can be summarized as follows:

1. **Basic transaction handling**. What are the transactions of the protocol, how is memory updated, where to find up-to-date data? As hinted above, there are several types of transactions: read, read with invalidation, invalidation, memory update, etc., and each type has a particular cache and directory state transition rule. In FAME we have up to 10 transaction types.
2. **Conflict resolution rules**, which should ensure coherence without deadlock. A key issue of distributed directory-based cache coherence protocols is *conflict resolution*. Two concurrent requests issued by two processors are said to be in conflict (or to collide) if they are to the same address. In a snoopy-based protocol, the bus grant serializes accesses in an atomic way thus resolving conflicts: request *emission is serialized*; requests and responses are snooped *synchronously* by all the caches. But in distributed protocols, requests are issued *concurrently*, there are *multiple conflict points* (where conflicting requests meet) and various *race conditions* arise between requests or between requests and responses. For instance in FAME, within a module, a processor node can send requests to FSS and vice-versa, and there are requests between FSSs of different modules: then conflict points are in processor nodes and in FSS (where there are several types of conflicts depending on the request source). Thus, two concurrent conflicting requests that are issued by nodes in different modules can collide in either requesting node or in either FSS of both modules. Conflict resolution is complicated by race conditions: request and response channels are independent, so a request can overtake a response and vice-versa. For instance, if a node controller sends a request Rq1 to FSS, then FSS sends its corresponding response Rs1 followed by a request Rq2: the node controller may receive Rq2 before Rs1, without knowing whether its request has been acknowledged or not. There are similar race conditions in transfers between modules.
3. **Directory replacement handling**, in relation with conflicts and deadlocks. For instance, if a request on address A is received by a directory and it needs to cause a replacement in order to complete, if B is the line that is chosen to be evicted (and so invalidated) it may run into a coherence conflict with a pending request to B. Thus, the replacement triggering creates a connection between two requests on different addresses through resource (directory) and coherence conflict, which may cause deadlocks.

3 Protocol Behavior and Property Modeling

We aim to build a reduced model at the “right” abstraction level, trying to find a compromise between what is tractable and what is needed to verify cache coherence properties. The behavior details which are not related to the cache coherence protocol issues and properties brought out above are dropped.

3.1 Behavior Modeling

There are mainly two kinds of simplifications that are combined in modeling:

1. Aggregating a sequence of events in one atomic event. This means that the intermediate states between the aggregated events are not observable and some orderings are not possible in the model.
2. Reducing the resources of the system. This involves reducing the elements of the system: determining the number of processors, nodes, modules, memory addresses, choosing which tables or queues are to be modeled, and what information they contain is needed to model the behavior we want to verify.

Event Collapsing

As said above, the main issue of a cache coherence protocol is conflict resolution, and conflicts results from the concurrent behavior of the different agents of the protocol and race conditions between request/response transfers. So the abstraction, particularly the event collapsing one, should capture this concurrency, so that all kinds of conflict be possible in the model.

This is the general event aggregation scheme: a transaction goes through different phases incurring treatment in each agent (processor, node controller, FSS) and transfers between agents. We consider that there are three "treatment centers": the processor bus including the caches within the node, the node controller, and FSS. We can collapse several steps as long as there is no more than one transfer involved between these centers. There are 3 kinds of transfers: between the processor caches and the node controller, between the node controller and the FSS buffers and between two FSS buffers. A typical case is collapsing emission or reception of a transaction with its handling. An agent receives a transaction, then handles it (performing some treatment), then sends a result. We can collapse receiving the transaction and handling it in one event, or handling the transaction and sending the result. If we collapse the three events we could miss conflicts between several transactions received, or we miss some orderings like: a transaction T1 is received before T2, but the results of T2 is sent before that of T1.

Thus, considering that transfers between the agents are atomic and point-to-point, discarding the interconnection network and routing functions, does not miss the requests and responses concurrency from the coherence protocol viewpoint. This assumes we use a formalism based on *asynchronous interleaving semantics*.

So, in our models, events will be either internal events to caches or FSSs, or request/responses transfers with the associated treatment at the reception point.

Resource Reduction

The objects that are modeled are: caches (state and data), memory, node controller and FSS directories. Within nodes and modules, we represent the buffers that keep track of requests, sometimes collapsing several buffers in one.

Concerning, the number of memory line addresses, since the aim of a cache coherence protocol is memory coherency and not some consistency model, it is enough to perform verification with only one address [8]. This remains true for directory replacements, if they are modeled as non-deterministic events, as long as only coherence aspects are considered. However, we aim to capture some deadlock issues related to the resources present in the model, and as pointed above, coherence and resource conflict meet in directory replacements. Therefore, we set the number of addresses to 2 when we want to take replacements into account; otherwise we set it to 1. An additional reason, for using 2 addresses in replacements, is to model conflict rules specified by the protocol as they are without introducing modeling bias. (In our models, when there are 2 addresses they are mapped to the same home memory).

Beside varying the number of addresses, in order to perform incremental verification and be able to vary the configuration of the model in facing state explosion, we need facilities to set these parameters (a home node or module, is the one that contains the home memory):

- Number of processor nodes in a module, number of caches per node.
- Number of memory line addresses in the system.
- Sizes of the different buffers.
- Number of active nodes in home/non-home module: so that we can set a model where one node is active in home module and 2 nodes in non-home module, for instance.
- Option to prevent nodes in home or non-home module from issuing requests.
- For each kind of transaction, a switch to enable it or not (as said above, there is up to 10 kinds of transactions in FAME).

Caches, controllers, FSSs, modules, addresses, buffer index are all *symmetrical* types. Even if some node is home and the others not, we define the fact of being home as a boolean attached to a node, then this boolean can be set non-deterministically at the initial state. Then, in order to take advantage of these symmetries that allow reducing the state space, we need a tool that implements symmetry reduction techniques.

3.2 Property Modeling

Cache Coherence Properties

Cache coherence properties are typically *state invariants*. The fact that there may be transient states where a directory is not accessible and coherence is not maintained is included in the property. Such transient state could be, for instance, that a transaction is ongoing in some buffer. Then the cache coherence property is: we are in a transient state OR the coherence relation is true. An example of coherence relation: if one directory state is E, then all other directories states are I.

Valid Data Properties

In order to verify that a processor does not get stale data and that no data modifications are lost, we use a data model (borrowed from [13]) that avoids manipulating data values.

Data are modeled with two values: valid and invalid. When a processor writes a line, this copy takes the value “valid” and all other copies of the same address in the system become “invalid”. These copies are in memory, caches, and buffers that keep track of requests and hold responses. *This implies the ability to manipulate global variables*. Then, to verify data integrity, we add these state invariants:

- If a cache is not I, it contains valid data.
- When there are no modified data in a cache, data in memory are valid.

Deadlock-Freeness
The actual deadlock-freeness property one expects from a real system is: “a transaction will always inevitably complete (within a bounded time)”. But since we deal with abstract models that do not describe arbitration and starvation prevention mechanisms, and we use asynchronous modeling where it is possible to indefinitely delay the firing of a transition, the general property we would like to verify is: “always, whatever the point it has reached, a request can be completed”. The different cases of request non-termination are: it has gone into a livelock, or it is stopped somewhere.

4 Murφ Language and Tool

Choosing a notation and its associated tool depends on the goal and the application domain. A comprehensive survey on verification methods for cache coherence protocols is given in [14]. Since we deal with complex specification of cache coherence protocol in distributed shared-memory architectures, and we focus on mastering the specification and finding bugs rather quickly, methods based on *explicit state enumeration* are more suitable: because, verification is fully automatic, and *error traces* can be minimal and explicit, giving a scenario showing the error origin. Efficient state reduction techniques are indispensable to take into account the minimal concurrency we need to verify conflict issues. These considerations have led us to choose to use the Murφ language and tool developed by the Hardware Verification Group of Stanford University [9].

Amenity of the Language and Specification Style
Murφ provides familiar data structures and programming constructs. For instance, there are types such as record and arrays that can be indexed over an enumerated type, imperative programming constructs such as if-then-else, switch, for, while… Besides, it is possible in Murφ to define constants that are parameters of the system: number of addresses, of processors, etc… So, we can change the configuration of the model by changing these constants and recompiling.

Building a model consists in defining a collection of global variables, which represent the system resources states and a collection of transitions rules. Each rule has an enabling condition, which is a boolean expression on the state variables, and an action, which is a sequence of statements that modify the values of the state variables, generating a new state: **rule** condition ➔ action_statements **endrule**. A rule is symbolically defined with parameters: it represents a set of instantiated rules. In a rule we can access any global variable.

If the global variable concept does not seem suitable to an architecture reference specification, it is an important mean of *abstraction* and *state reduction* in a model intended to verify the main points of a protocol. We use this global variable access feature in the verification of valid data property (Section 3.2): if caches, node controllers and modules were modeled as processes with local variables that are not accessible globally, we would not be able to simply model this property.

State Reduction Techniques

Murφ provides several state reduction techniques:

The `undefine` statement allows to give a nil value to a variable thus identifying irrelevant values at some point. This reduces the number of states since it avoids having two states that differ in non-relevant parts. In one of our verification tasks, forgetting to `undefine` a variable at some point has multiplied the state count by 10.

The *symmetry reduction* [11]: a special type constructor, `scalarset`, can be used to define a set of symmetrical identifiers (so it is user-provided symmetries). For instance, we declare the types of processor identifiers as `scalarset`. Then, in the state enumeration process, if a state can be obtained from another one by permuting the values of `scalarset` types, then both states are considered equal. As the complexity of trying all possible permutations may become exponential, there are options to limit the number of permutation trial or to use fast heuristic normalization algorithms.

Bit-compaction consists in compacting the state descriptor into a bit-string without loss of information. This reduces the state space but increases computing time. Generally, this reduction is not enough for complex configurations and we rather use the hash-compaction option detailed in the next point.

Probabilistic verification or hash compaction: instead of storing the whole state descriptor, a hash compacted descriptor is stored (typically on 40 bits). Thus, different states could be considered equal. In the verification status, the verifier prints the probabilities of having missed one state or one error [16].

Asynchronous Behavior

The Murφ language is asynchronous without a clock and without event duration. Its fundamental semantics is that of a transition system: there are events (transitions) that can occur when some enabling condition is true, and one event occurs at a time (no simultaneous events). The occurring of an event leads the system from a state to another one. This is insufficient if we want to describe and analyze the low-level design of a hardware piece (RTL level). But it is necessary abstraction means to describe system level protocol transactions, where we need an abstract way to describe all possible interleavings of events due to variable delays and different paths without describing the implementation details.

Verification and Error Diagnosis

The semantic of the model is the reachability graph of the transition system. A state is an assignment of the global variables. A rule is an atomic event. The graph is produced by an explicit state enumeration: beginning with an initial state, all the enabled rules in this state are executed yielding the successors states of the initial one. And this process continues with the generated new states, etc.

The properties to verify are expressed as boolean expressions and incorporated in the model:

- State invariants: boolean expression on global variables that should be satisfied by all the reachable states.
- "Assert" instructions: boolean expressions that should be true in some point during the execution of a rule.

Murφ compiler transforms a model into a C++ program, the verifier that explores the state graph. When an error is found, the verifier halts and prints an error trace. There are 4 kinds of errors:

- A reachable state that violates an invariant.
- An assert instruction result is false during the execution of a rule.
- An undefined variable is accessed during the execution of a rule: this could indicate an uncovered case in the protocol definition.
- A deadlock is reached: a state that has no successors (no rule is enabled).

Since we always use *breadth-first search* option, the error trace is a *minimal* one, producing a *scenario* leading from the initial state to the state exhibiting the problem. So, errors can be found quickly without the need to totally explore the state graph: this moderates state explosion problem consequences on finding errors.

Murφ Choice Motivation Discussion

Among all Murφ advantages listed above, the determining choice factors are symmetry and hash-compaction reductions, which have allowed us to verify fairly huge models (see Section 5.1, particularly Table 2 and its comment). Then the drawback is that liveness property verification is not supported with symmetry reduction.

In [5], a similar protocol to ours is model-checked using Cospan, SPIN and Murφ: the results demonstrate also the benefits to exploiting symmetries with Murφ.

However, even if we cannot verify deadlock-freeness properties like "always a transaction can complete" (Section 3.2), we can verify that there is no total system deadlock (a state where no event can occur). This is a sub-case of the liveness property we aim to, but *benefits outweigh this disadvantage* since we mainly focus on coherence properties and at least sink states can be detected (the limited resources of the model often make a blocked request result into a total deadlock).

(In a previous experiment, we had other property constraints and it was suitable to use LOTOS [2].)

5 Verification and Modeling Outcome

We have applied protocol formal verification to four versions of the FAME cache coherence protocol, which we call: FV1 to FV4. In FV1, modeling started at the very beginning of the cache protocol definition when it was still early thoughts, and went along its specification process. FV2 was a major revision, impacting transaction and conflict handling: in this case, formal modeling and verification started when the protocol definition was fairly mature but not finalized yet. FV3 has kept basic transaction handling but introduced a significant modification in conflict handling. FV4 had no significant impact on coherence protocol, but the evolutions were related to routing and system scaling: new protocol cases were added by this change but the transaction and conflict handling is the same.

In order to assess this experience, we examine two aspects: the modeling and verification process and the cost-benefit analysis.

5.1 Incremental Modeling and Verification

FAME Murφ models conform to the principles stated in Section 3. The global variables are: modules, each module contains FSS and processor nodes, FSS contains directories and buffers for ongoing transactions. A processor node contains memory, caches and input/output buffers of node controller. The cache states, data values, request and responses types are defined as enumerated types. The structures are defined as records and arrays. Identifiers of caches, nodes, addresses, and buffer index are defined as `scalarset` (symmetrical types). Replacements in caches are non-deterministic, but directory replacements occur only when needed and in models with 2 memory line addresses.

The model can be parameterized in order to define the configuration to be verified: the parameters are those listed in Section 3.1. There are 13 rules corresponding to: internal processor events, bus events within a processor node, transfers within a module, internal FSS events and transfers between two modules. The properties of interest (data integrity and cache coherence) are modeled as described in Section 3.2: there are 5 state invariants about directory coherence.

The process interleaves modeling and verification. From the protocol definition specification, we build a first incomplete model and run verification. If an error is found, it can be a modeling bug or a protocol bug. So we are concurrently debugging our model and verifying the protocol definition. Then we make corrections or add new features to the model. Even if we know that a configuration is tractable by the verification tool, we should begin verification with the smallest model and increase sizes of the different parameters in stages: because the same error is longer to detect on a larger configuration than on a smaller one.

Table 1 shows for each protocol major revision (FV1 to FV4), the number of Murφ model versions and the corresponding number of lines of code (LOC). For each case, there is a new model version at three points: model bug detection, protocol bug detection or new feature introduction in the incremental modeling process. So it is related to modeling effort and to issue finding. This explains why there are so many versions in FV1, where modeling started on early protocol definition and a lot of issues were detected, and why so few ones in FV4, where there are no protocol significant modifications and no error detected (see next subsection). The model sizes are similar and tantamount to a few thousands lines.

Table 2 shows the largest graphs that could be reached by the verification without state explosion. For each case we give the figures for the largest configuration with 1 memory address (so without directory replacement) in the model and the largest one with 2 addresses (with directory replacement). In FV1 case, we were using a machine with a 256MB memory, so it was not possible to go very far, while in the other experiments, the machines used had 1 GB of memory. Therefore FV1 largest graphs are not comparable to the other graphs and are not reported here.

Table 1. Incremental modeling effort

Case	Model versions count	LOC (smallest → biggest)
FV 1	41	920 → 3820
FV 2	36	1700 → 2750
FV 3	15	2643 → 3266
FV 4	4	3322 → 3459

The features of the verifications shown in Table 2 are: FV2a: 1 module, 4 processor nodes, 1 cache per node, 1 address, 6 transaction types. FV2b: 1 module, 3 processor nodes, 1 cache per node, 2 addresses, 5 transaction types. FV3a: 2 modules, 2 nodes/module but only 3 modules active in the system, 1 cache/node, 1 address, 7 kinds of transactions. FV3b: 2 modules, 1 node/module, 1 cache/node, 2 addresses, 7 kinds of transactions. FV4a: 3 modules, 1 node/module, 1 cache/node, 1 address, 7 kinds of transactions. FV4b: 3 modules, 1 node/module, 1 cache/node, 2 addresses, 4 kinds of transactions. Even when there are 2 addresses, a node can have at most one pending request. FSS buffer sizes are set, so that it can receives all node requests concurrently. Obviously, increasing the number of request sources (caches, nodes) has more impact than increasing the number of transaction types or addresses, since it increases concurrency in the system.

Notice that these are the states explored taken into account symmetries, so they are not all the states of the underlying graph explored. The graph diameter is a hint about the longest transaction path. The CPU time may be different even for similar counts of states for the same model, because with different parameters configurations, the non-compacted state sizes are different. Generally, enough early in the process, we have to use Murφ hash compaction to avoid state explosion and so perform probabilistic verification (actually we combine bit-compaction and hash compaction).

Table 2. Largest graphs. All this information is provided by Murφ. The "States" column gives the number of states explored. "Rules": number of rules fired. Bounds of omission probabilities induced by hash compaction: P1 is probability of "even one omitted state"; P2 of "even one undetected error". P<=0.000000 does not mean P=0, but that the bound of P, when rounded to 6 digits, gives 0. Diameter is the one of the reachability graph. CPU time is expressed in days

Case	States	Rules	Probabilities bounds	Diameter	CPU (d)
FV 2a	54,842,173	316,784,167	P1<=0.000024 P2<=0.000000	114	3.1
FV 2b	59,069,095	367,365,869	P1<=0.000029 P2<=0.000000	91	1.5
FV 3a	12,732,647	55,006,883	P1<=0.000001 P2<=0.000000	76	0.4

FV 3b	53,908,283	319,449,256	P1<=0.000013 P2<=0.000000	91	1.5
FV 4a	23,203,144	100,615,496	P1<=0.000006 P2<=0.000000	82	2.3
FV 4b	48,418,599	301,928,790	P1<=0.000025 P2<=0.000000	89	7.5

These reduction techniques (symmetry and hash compaction) are indispensable to extend the limit where state explosion occurs and has allowed us to obtain the results we analyze in next subsection.

5.2 Cost-Benefit Analysis

Since our goal is to find protocol definition issues, the benefits can be measured by the number of issues raised by modeling/verification activity. The cost is measured by the number of person.week needed to perform this task (actually the work was achieved by one person, the author). *A protocol issue can be found either by the verifier or as a result of the modeling and abstraction activity*. Modeling induces a thorough analysis of the protocol definition that can lead to finding issues, helping in clarifying, completing and mastering its specification.

When we run verification, there are 3 possible outcomes:

- It is complete with no error found: then we go into another modeling/verifying cycle by adding features to the model or rerun the same model by changing the configuration parameters.
- An error is found and a trace error is produced: then we check whether it is a protocol error or a model error. In order to get shortest error traces, we always use breadth-first search.
- The graph exploration cannot be complete due to memory lack (state explosion): then we use probabilistic verification, we try other configurations by tuning the model parameters, or we give up if we have already tried this.

We classify the issues following 2 criteria, its category and finding origin [10]:

1. Category of the issue: there are three kinds of issues:
   - Uncovered or undefined case: the specification does not define the behavior of the protocol in this case.
   - Ambiguous specification: several interpretations of the specification are possible. One of these leads to an error.
   - Behavioral error: the behavior defined by the protocol specification leads to an error like reading stale data, coherency paradox or deadlock.
2. Origin of the issue detection: an issue can be found: by modeling (during the manual analysis of the protocol in order to model it); or by verification (by running the verification).

Table 3 shows cost, issue count (with their classification), along with the total CPU time consumed. This last figure is given as a hint and is not a rigorous comparison factor, because we have not used the same machines with the same processors in all cases. In FV1, we used 1 machine with small memory size (256 MB); in the other cases we had several machines with 1 GB of memory: we were able to launch up to 3 verifications in parallel. The usage distribution of this CPU time is more meaningful and is given by Table 4.

Table 3. Cost-benefits analysis. The categorization of issues read: A=Ambiguity, U=Uncovered, E=Error (coherence paradox or deadlock). Finding origin is M=Modeling or V=Verification. So EM means an error found by modeling

Case	Cost (p.w)	CPU (days)	Benefits (protocol issues raised)
FV 1	33	13	24 issues (1 AM, 9 UM, 2 EM, 1 AV, 4 UV, 7 EV)
FV 2	17	42	15 issues (5 AM, 3 UM, 4 EM, 3 EV)
FV 3	7	13	9 issues (1 AM, 1 UM, 2 UV, 5 EV)
FV 4	6	46	No issue raised

FV1 is the most costly one and also the one that raised the biggest number of issues, half of them by modeling. This is due to the following reasons: it is our first experience with Murφ and the modeling-verification process started at the very beginning of the protocol definition, when it was still early thoughts. This explains the preponderance of ambiguity and uncovered case issues. Half of the CPU time is wasted on verifications that did not complete, because of the small memory size.

In FV2 case, the protocol definition was mature enough (but not finalized) when the formal modeling started. We were already familiar with Murφ and a small part of the first model could be reused, so productivity increases and benefits are still important. Most of the issues are raised by modeling and are either uncovered or ambiguity issues. Since in this case we had up to 3 machines with more memory, we have tried to make use of it, and verify large configurations (with all kinds of transaction, for instance) which ended with state explosion: this explains the important CPU time consumed.

In FV3, the protocol is a fairly important extension of FV2 but with the basic transaction handling remaining the same (conflict handling is modified and directories distribution is modified). The productivity is further increased, there are more errors found by verification than by modeling. In this case, based on previous experience, we have found the configuration sizes that are manageable without state explosion. So, we have not tried to check larger ones, but instead, we have tried several combinations of up to 3 nodes in the system distributed over 2 modules. This explains that in this case the verification that ended with state explosion are not dominant.

The last case FV4 is a non-significant extension of FV3 from protocol viewpoint: the important modifications are at the routing level and increasing the number of supported modules, while we focus on cache coherence protocol. So, to follow this architecture extension, we tried to verify larger model configurations (up to 3 modules) to check new concurrency cases: this naturally increased again the effort spent on launching verifications that ended with state explosion. However, significant configurations were successfully verified and, as expected, no new issue was detected, increasing confidence in the protocol definition correctness.

Table 4. CPU time distribution. % of CPU time consumed in verifications that detected protocol issues, model bugs, were error-free (terminated with "no error" message), ran out of memory. These are rounded figures: 0% = 0, ~0% = a non-null negligible percentage

Case	Protocol Issues	Model bugs	Error-free	State explosion
FV 1	~0%	3%	46%	51%
FV 2	~0%	1%	30%	69%
FV 3	7%	7%	71%	15%
FV 4	0%	1%	38%	61%

Protocol issues found by verification are usually detected very quickly, since generally it happens on the first models and state graph exploration stops as soon as it detects an error. The time range for finding an error is between less than 1 minute and a few hours. The time consumed on debugging is not very significant either.

Finally, this approach was very fruitful and cost-effective, since it helped finding hard-to-simulate bugs, generally involving tricky conflict cases, in the early development stages. Moreover, these are protocol specification errors, not implementation errors, which are much more costly to detect in later development stages.

6 Conclusion

A distributed directory-based cache coherence protocol is a complex system to design, where the conflict resolution rules are a key issue, difficult to apprehend, because of concurrency and race conditions. Therefore, it is an outstanding domain of application of formal techniques, which provide rigorous analysis and verification methods. We have applied formal verification to FAME cache coherence protocol, aiming at finding protocol errors with abstracted simplified models.

Actually, even abstracted and simplified models of such a protocol, which focus on these coherence aspects, produce huge state spaces. Due to the Murφ reduction techniques (symmetry, hash-compaction), specification style and explicit state enumeration technique, this obstacle is overcome: the errors found by verification are detected very quickly at the beginning of the incremental modeling-verification process and are not impacted by state explosion that comes later. Error traces are minimal and exhibit a scenario explaining the error origin.

The experiments show that verifying an abstracted model is sufficient to find important protocol bugs: these are most of the time very hard-to-simulate errors that involve intricate conflict situations.

Besides, even if state-of-the-art tools were able to handle larger models, we think that building a complete protocol model instead of an abstracted protocol model would not be cost-effective in the beginning of the life-cycle of the development.

A simplified abstracted model is easier to build than a complete model and allows mastering more quickly the protocol specification complexity: then in the early phases of the development process, it helps in finding issues and improves the understanding of the system, as it is shown by the issues found by modeling, which are most of the time protocol specification "holes".

The method of protocol verification aimed at finding bugs and increasing confidence in protocol specification correctness proved to be an efficient protocol design aid. The benefits of this approach are to detect protocol specification errors not implementation ones, and to do it in the early development phases. Since the first experiment on FAME, it was considered fruitful and it was carried on to next versions; now it is a well-established practice in our protocol development process.

References

[1] Bull NovaScale® servers. //www.bull.com/novascale/index.html

[2] Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., Zulian, F.: Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Proc. of FORTE/PSTV'96 (1996) 435-450

[3] Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol Verification as a Hardware Design Aid. IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society (1992) 522-525

[4] FAME Architecture, Statement of Direction www.bull.com/download/whitepapers/fame.pdf

[5] Fisler, K., Girault, C.: Modelling and Model Checking a Distributed Shared Memory Consistency Protocol. Proc. 18th International Conference on Applications and Theory of Petri Nets, LNCS 1420, Springer Verlag (1998) pp 84-103

[6] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In Proc. of the 17th Annual International Symposium on Computer Architecture (1990) 15-26

[7] Hennessy, J., Heinrich, M., Gupta, A.: Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges. In Proc. of the IEEE 87(3) (1999) 418-429 (Special issue on Distributed Shared Memory)

[8] McMillan, K.L., Schwalbe, J.: Formal Verification of the Gigamax Cache Consistency Protocol. Proc. ISSM Int'l Conf. Parallel and Distributed Computing (1991)

[9] Murφ description language and verifier: http://sprout.stanford.edu/dill/murphi.html

[10] 10.NASA Formal Methods Guidebook. Formal Methods, Specification and Verification Guidebook for Software and Computer Systems. Volume I: Planning and Technology Insertion. Release 2.0, [NASA/TP-98-208193] (1998) 88 pages

[11] Norris Ip, C., Dill, D.L.: Better Verification through Symmetry. Formal Methods in System Design, Volume 9, Numbers 1/2 (1996) 41-75

[12] Paramarcos, M., Patel, J.: A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. Proc. of 11^{th} Int'l Symp. Computer Architecture, (1984) 348-354

[13] Pong, F., Dubois, M.: A New Approach for the Verification of Cache Coherence Protocols. IEEE Transactions on Parallel and Distributed Systems, 6(8) (1995) 773-787

[14] Pong, F., Dubois, M.: Verification Techniques for Cache Coherence Protocols. ACM Computing Surveys, Vol.29, 1, (1997) 82-126

[15] Roucairol, G.: Using Formal Verification Methods in an Industrial Environment for a Decade, Conclusions and Perspectives. Keynote at FLOC99, Trento (1999)

[16] Stern, U., Dill, D.L: A New Scheme for Memory-Efficient Probabilistic Verification. In Proc. of FORTE/PSTV'96, (1996) 333-348