Application of Esterel for Modelling and Verification of Cachet protocol on CRF Memory Model*

Siddharth R. Phanse† and R.K Shyamasundar‡
School of Technology and Computer Science
Tata Institute of Fundamental Research
Mumbai 400 005, India, email: {phanse,shyam}@tcs.tifr.res.in

Abstract

Synchronous languages have been widely used for the reliable design of reactive systems and synchronous circuits. as it has become possible to move from the realm of simulations to that of proofs. While ESTEREL *has been used for the modelling and verification of reactive systems, it has not been used that widely for the modelling and verification of cache coherency protocols. In this paper, we shall discuss the use of* ESTEREL *for the modelling and verification of cache coherence protocols, in particular on the adaptive cache coherence protocols called* CACHET *built on the Commit-Reconcile & Fences (CRF) distributed shared memory model. In the paper, we shall (i) first, model the CRF memory model along with the various* CACHET *micro-protocols in* ESTEREL, *(ii) verify properties of the* CACHET *micro-protocols in a graded manner reflecting the protocol design using the synchronous observer method, and (iii) highlight the experiences in overcoming limitations including state-explosion.*

1. Introduction

Cache memory plays a vital role in the design of multiprocessor systems to control the latency for accesses to the memory by the processors. A cache is private to a processor and contains a subset of memory presumably containing the most relevant locations needed for processing; the idea is to enhance the probability of *cache hit*. This naturally leads to multiple copies of the memory in the cache and hence, leads to the problem of *cache consistency*: when a processor updates the value at some location, all caches containing a copy of that location get updated. Establishing cache coherence or consistency has been one of the challenging problems of verification methods.

Modelling, validating and verifying cache coherency is an important topic of research from the point of theory and practice. Various approaches based on *Symbolic Model Checking, State Enumeration*, and *Symbolic State Models* have been used for cache coherence protocol verification such as verification of the FutureBus+ by Ed Clarke using SMV, verification of SCI protocol by David Dill using Murphi etc. Since these methods depend on searching state spaces exhaustively, the verification task becomes increasingly difficult as the number of processors and the complexity of protocol increase. In this study, we show that using ESTEREL [6] and its environment, it is possible to formulate and check *safety* (e.g. data consistencies, deadlock etc.) and *liveness* properties using ESTEREL models of these systems. We illustrate how reachability analysis can be done in the ESTEREL environment for such complex verification problems. To our knowledge, this is the first such study involving ESTEREL and cache coherence protocols. In this paper, we shall

1. Model CRF memory model and the various CACHET micro-protocols [1, 2, 3] in ESTEREL, and
2. Verify properties of CACHET protocols in a graded fashion using the verification environment of ESTEREL.

2. Distributed Shared Memory Models

Cache plays a vital role in controlling the latency between processor and memory access for reading and writing. In a shared-memory system, a *cache memory* is interposed between each processor and the shared-memory system. *Cache* is private to a processor and

*The authors thank V. Roy, A. Bouali (FC2 team, INRIA) and Arvind and J.E. Stoy (MIT) for various discussions.
†Supported by Ada-TIFR proj. *Object Code Validation.*
‡Partially supported by the IFCPAR Project 2202-1: *Formal Specification and Verification of Reactive Hybrid Systems.*

contains a subset of the memory, hopefully containing most of the locations that the processor needs to access; i.e., the "cache-hit" should be maximum. Such caches induce the replication of data and hence, there is the problem of *cache consistency*. If one of the processors updates a copy of memory location, say l, then all copies of location l in cache of other processors have to be updated. Cache states in processors are usually maintained at the cache-line level that typically contains 8 to 64 cells.

In any memory model, *Load* and *Store* are the two basic instructions that perform the reading and writing between the processors and the shared main memory. When a processor requires a data from the memory it loads that location from the memory to its local cache through the instruction "**Load (address)**". The processor can now locally access this data without contacting the memory for further reads on the same location. Similarly, when the processor reads the data and computes a new value for it, a *Store* instruction is given to write back the new value into its cache through the instruction "**Store (address, new value)**". Thus, it is not necessary for the processor to write back this new value to the memory at the very first instance. Such a design prevents communication with the memory every time a new value is generated for that location. A shared-memory system is shown in Figure 1.

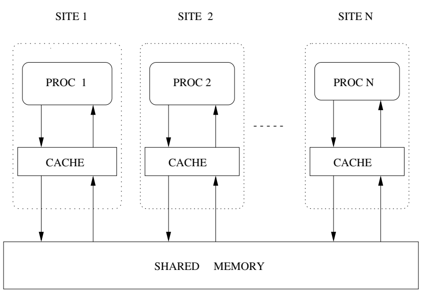

Figure 1. Shared-Memory System

In the case of Distributed Shared Memory (DSM) systems, memory is distributed across processors on different locations. This introduces a considerable latency time during communication among local caches and shared main memory. "**Site** i" refers to both the processor and the cache at location i as in Figure 1.

2.1. The CRF model

We shall confine to the recently proposed mechanism-oriented memory model called Commit-Reconcile & Fences (CRF) which offers the following advantages

1.CRF effectively exposes a semantic notion of caches, and decomposes *Load* and *Store* instructions into finer-grain operations.
2. CRF exposes mechanisms needed to specify protocols precisely and separates how a cache provides correct values from how it maintains coherence.
3. CRF uses a design methodology that separates the correctness and liveness concerns in protocol design thus allowing us to incorporate adaptivity behavior without having to worry about these issues.

We can enforce protocols on the CRF memory model that exposes it to both data replication and instruction reordering at the programming level. It decomposes the *Load* and *Store* instructions into finer-grain operations that operate on a local semantic cache (sache). There are five memory-related instructions in CRF: (i) *Loadl* (load-local), (ii) *Storel* (store-local), (iii) *Commit*, (iv) *Reconcile* and (v) *Fence*. Since, our goal is to study the cache coherence model, we do not consider the "Fence" instruction, which is related to instruction reordering. A formal detailed description of CRF using Term Rewriting Systems (TRS's) can be found in [2]. The above instructions are generated by the processors for their local cache (or sache) cells. A sache cell is a single location of a cache at a *Site* that contains copies of memory location when they are read by the processors. A cell contains the address of the memory location it reads and its value at the instant it is read. Each of these sache cells also has an associated state (its Status), which can be either *Clean* or *Dirty*. The states and instructions are described below.

Clean: The data has not been modified since it was cached or last written back.

Dirty: The data has been modified and has not been written back to the memory since then.

Invalid: This state defines the empty state (also the initial state) of a cell when no memory address is loaded.

Loadl: When a processor needs to access a memory location that is not present in its local cache, it requests the memory manager; the memory loads the address and its value at that instant into a cache cell at the requesting processor's *Site* ; the processor then locally accesses that memory location until it is purged from the cache. In **Loadl(a)**, 'a' denotes the address of the location to be loaded from the memory.

Storel: A *Storel* instruction writes the data into the sache if the address is cached, and sets the sache state to *Dirty* . On a sache miss, *Storel* first loads the location from memory and then completes the store operation locally. In **Storel(a,v)**, 'a' denotes the address into which the new value 'v' is to be written.

Commit: The instruction on a *Dirty* cell writes back

Processor Rules

Rule	Instr.	Cstate	Action	Next Cstate
CRFLoadl	Loadl(a)	Cell(a,v,Clean)	retireLoadl	Cell(a,v,Clean)
		Cell(a,v,Dirty)	retireLoadl	Cell(a,v,Dirty)
CRFStorel	Storel(a,v)	Cell(a,-,Clean)	retireStorel	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	retireStorel	Cell(a,v,Dirty)
CRFCom	Com(a)	Cell(a,v,Clean)	retireCom	Cell(a,v,Clean)
		$a \notin cache$	retireCom	$a \notin cache$
CRFRecon	Recon(a)	Cell(a,v,Dirty)	retire Recon	Cell(a,v,Dirty)
		$a \notin cache$	retireRecon	$a \notin cache$

Background Rules

Rule	Cstate	Mstate	Next Cstate	Next Mstate
CRFCache	$a \notin cache$	Cell(a,v)	Cell (a,v(clean)	Cell(a,v)
CRFWriteback	Cell(a,v,Dirty)	Cell(a,-)	Cell (a,v(clean)	Cell(a,v)
CRFPurge	Cell(a,-,Clean)	Cell(a,v)	$a \notin cache$	Cell(a,v)

Table 1. CRF Memory Model Rules

the dirty value in the cache cell to the memory; the instruction is stalled until it receives a Writeback Acknowledge signal from the main memory after which it changes the cache cell state to *Clean* and completes. In **Commit(a)**, 'a' denotes the address of the location that is to be written back to the memory.

Reconcile: The instruction on a *Clean* cache cell stalls until that address is purged from the cache cell. This instruction is useful to purge any stale copy at a *Site* so that the next request requires a *Load* operation to read an updated address copy from the memory, thus maintaining cache consistency. It has no effect on a *Dirty* sache cell. In **Reconcile (a)**, 'a' denotes the address of the location that is to be purged from the cache.

Retire: This is not an instruction but a message indicating the completion of the previous instruction that was taken for execution.

The CRF Memory Model rules are given in Table 1 in the form of state transition tables. Each row in the table represents a rewriting rule pertaining to an instruction having three components: a precondition, an action and a post-condition. The precondition contains an instruction to be processed (Instruction) and the corresponding cache or memory state (Cstate/Mstate). The action is typically completing that instruction or issuing a protocol message while the post-condition is the cache or memory state after the action is performed (Next Cstate/Mstate). There is one rule for each of the instructions and three background rules that operate between the saches and the memory. The exploitation of the flexibility that the *Commit* and *Reconcile* instructions offer in th memory model is the key to the performance of CACHET .

3. Adaptive Cache Coherence Protocols

Adaptive cache coherence protocols are those capable of changing their actions to address changing program behavior. Shared-memory programs exhibit various access patterns and it is very difficult to have a fixed cache coherence protocol that works well for all access patterns. Adaptive shared-memory systems allow multiple coherence protocols to run at the same time, or allow one coherence protocol to adapt to some identifiable access patterns. The main difference in these systems is regarding what and how access patterns are detected and this may involve using some heuristic mechanisms. The implementation of an adaptive cache coherence protocol involves two issues: *what* adaptivity can be embodied in the protocol, and *how and when* such adaptivity can be invoked.

CACHET was proposed [1] as an Adaptive Cache Coherence Protocol for DSM systems on which programs can exhibit various access patterns. It is a seamless integration of several micro-protocols, each of which has been optimized for a particular memory access pattern. CACHET exploits a wide scope of adaptivity within its micro-protocols to achieve high performance under changing memory access patterns and consists of three *micro-protocols* briefed below.

Cachet Base Protocol: This is the simplest of the three and implements the CRF memory rules directly and uses the memory as a rendezvous. When a *Commit* instruction is executed for an address that is cached in the *Dirty* state, the data is written back to the memory before the instruction can complete. A *Reconcile* instruction for an address cached in the *Clean* state requires that the data be purged from the cache before the instruction can complete. The most attractive characteristic of Cachet Base is its simplicity; no state needs to be maintained at the memory side to store identifiers of the *Site* in which the addresses are currently cached.

Cachet Migratory Protocol: When an address is exclusively accessed by one processor for a reasonable time period, this protocol gives the cache at that *Site* the exclusive ownership so that all instructions on that address become local operations. The protocol ensures that an address can be cached in at most one cache at any time. Therefore, a *Commit* instruction completes even when the address is cached in the *Dirty* state, and a *Reconcile* instruction completes even when the address is cached in the *Clean* state. The exclusive ownership of the address can migrate among different caches whenever necessary.

Cachet WriterPush Protocol: When *Loadl* operations are usually more frequent than the *Storel* operations, WriterPush protocol allows *Reconcile* instruction to complete even when the address is cached in the *Clean* state. Thus in such cases, the following *Loadl* access to the same address causes no cache miss thereby reducing access time. Correspondingly, when a *Commit* instruction is performed on a *Dirty* cell, it does not complete before *Clean* copies of that address are purged from all other caches. Hence, it can be a lengthy process to *Commit* an address that is cached in the *Dirty* state.

4. Overview of Synchronous Observers

One of the interesting techniques of verification of synchronous programs is the method of *synchronous observers* proposed in [4]. The idea has its origins from model checking using temporal logic and testing of programs. The idea is to test whether there is a possibility of violation. If not, we can conclude that the property holds. It is easy to observe that when the program behavior violates a property, it does so at a precise time. Thus, the first task is to identify the first reaction that violates the property on hand. A broad model of the approach is shown in Figure 2. Let S be a

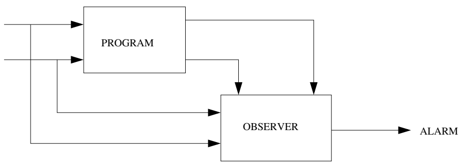

Figure 2. Synchronous Observers

safety property to be verified on program P. Now, design an observer Ω_P for the given property that takes as input both the inputs and outputs of P as shown in Figure 2 and emits an ALARM signal whenever S is violated. The verification thus consists of checking that the parallel composition of the program and its observer never leads to the emission of ALARM. Thus, instead of proving S about P, we establish that $P \,||\, \Omega_{S'}$ does not emit an alarm. Some of the distinct advantages of the approach are: (i) No need to add any explicit synchronization and hence the program remains unaffected, (ii) Modular (due to synchronous composition), (iii) The same language is used for specifying the program and its properties, and Specification becomes executable. Using this approach, safety properties can be easily expressed modularly through *observers*. Tool support for verification using this approach is provided by the XEVE environment[5].

5 Modelling CRF/Cachet in Esterel

Abstract properties of CRF are given below:
1. The CRF model decomposes the basic *Load* and *Store* instructions in a memory model into fine-grain operations.
2. A *Loadl* instruction reads the data from the sache if the address is cached; otherwise it stalls until the value of the address is fetched into the sache.
3. A *Storel* instruction writes new data into the sache if the address is cached, & sets the sache state to *Dirty*.
4. A *Storel* instruction is performed locally without any coordination with other saches.
5. A *Commit* instruction ensures that the modified value in sache is written back to the memory.
6. A *Reconcile* instruction ensures that a stale value is purged from the sache.

We have designed an ESTEREL model for the basic architecture described in section 2. It consists of several ESTEREL modules running in parallel, with modules corresponding to the execution of the processor, the sache and the memory behaviors. The processor awaits the incoming instructions (*Loadl*, *Storel*, *Commit* and *Reconcile*) and takes proper action depending on the type of instruction. These actions constitute coordination with local sache communicating with the memory module as and when the need arises.

Our model replaces the *Load* and *Store* operations with fine-grain instructions, *Loadl* and *Storel* that operate locally on the sache cells. Thus, this model exposes the semantic notion of a cache (as proposed by CRF) on which the operations are done locally. The normal *Load* operation can be thought of as placing a *Reconcile* before each *Loadl* thereby following the *Load* in a Sequential Consistency [3] memory model. Similarly, a *Commit* can be placed before each *Storel* to perform a *Store* operation. For lack of space, we only show ESTEREL fragments to depict the way communication among Processor, Memory and Cache takes place during specific instructions.

```
loop
 await  Commit; (Processor)
 present In_Cache then
  WriteBack value to Memory; (Proc-Memory)
  set Sache to "Clean" (Memory-Cache)
 end present
end loop

loop
  await  Reconcile; (Processor)
  present In_Cache then
   stalls until Sache Purged (Proc-Cache)
  end present
end loop
```

The model was simulated using XES simulation tool and we verified properties 1-6 on this model. It is easy to see that the model satisfies Property 1 trivially.

For the next two properties, we use the synchronous observer method on the Base CACHET protocol which implements the CRF model. These two properties define the basic *Loadl* and *Storel* operations as proposed for the semantic cache. The observers for establishing these properties have to test whether:

1. The Loadl *instruction completes instantly (emits Retire_Loadl) if the requested location is already cached (presence of In_Cache). Otherwise, if cache miss occurs then* Loadl *should not complete until it receives a Cache acknowledgment from the Memory along with the location to be cached.*

2. The Storel *instruction on a Cache hit (presence of In_Cache) completes instantly (emits Retire_Storel) and doesn't emit a Cache_Request signal requesting the Memory for a location which is already cached.*

The structure of observers is given below:

```
module CRF_Loadl:
input Loadl, Cache, Retire_Loadl, In_Cache;
output ALARM_SIGNAL;
loop
      await Loadl;
      present In_Cache then
            present [not Retire_Loadl] then
                  emit ALARM_SIGNAL
           end present
      else
           await Retire_Loadl;
           present [not Cache] then
                 emit ALARM_SIGNAL
           end present
      end present
end loop
end module

module CRF_Storel:
input Storel,Cache_Request,Retire_Storel,
                              In_Cache;
output ALARM_SIGNAL;
loop
  await Storel;
  present In_Cache then
   present [Cache_Request or not
                  Retire_Storel] then
     emit ALARM_SIGNAL
   end present
  end present
end loop
end module
```

The above observers when run in parallel with the main program do not emit the violation signal for all possible cases (note that this is finite). From this, we can conclude that the model satisfies the negation and hence, the required properties. Property 4 directly follows from the fact that the model implements a notion of sache which permits local execution of operations on its cache without being dependent on the current action or the state of the other saches.

Properties 5 and 6 impose constraints for cache consistency in the CRF Memory Model. These were most appropriately verified from the simulation of the model in the XES graphical simulator for ESTEREL programs. In the case of *Commit*, the processor stalls until the memory is updated and it receives a Write-Back_Acknowledge message from the memory. Similarly for *Reconcile*, the instruction is stalled until the sache copy of address is purged. Both these actions ensure that properties 5 and 6 are satisfied in this model.

5.1. Modelling Cachet Protocol in Esterel

The CACHET protocol has been enforced on top of the DSM. Now, we enforce this protocol on the CRF model already developed. The synchronous structuring does not alter the CRF model but only adds further constraints. The "conjunctivity" property makes it easier to build on the existing model. Our modelling of CACHET is based on (i) abstraction & simplifications and (ii) the model for the shared memory. These aspects are further described in the sequel.

5.1.1 Abstraction and Simplification

During this step our goal was to extract details of the CACHET memory model that were important for the cache coherence and then simplify these details to make the model construction in ESTEREL possible. We have omitted the instruction reordering that the CRF model exposes and concentrate entirely on its *Commit* /*Reconcile* features which play an important role in cache consistency protocol design. The communication between the cache and memory is a request/response type of transaction that involves data transfer to and from the main memory. The model constitutes a FIFO (applicable to messages with the same source address) nature of instruction execution where each processor has an arbitrary choice of *Loadl*, *Storel*, *Commit* and *Reconcile* for its next instruction. The structure of the system is detailed below:

a.The process for *Memory* simply uses the global memory variables as a rendezvous for the instructions (*Loadl* or *Storel*) from the processors provided the constraints are followed. These global variables are defined as global output type ESTEREL signals. The memory has a simple request/response behavior, i.e. if there is a request for an address from a processor its memory process reacts by sending a response that contains the requested data.

b. The *Cache* process too has a simple request/response behavior, similar to that of the memory. The local caches (saches) maintain coherence with the main memory by sending appropriate global signals. For example, on a Cache Miss, a cache sends a Cache-Request message to the memory, which responds with the data from the memory. Communication between parallel process modules is done by emission and catching of global signals thus performing the required message passing that takes place in DSM systems.

c. The *Processor* behavior is a bit complicated. After the preceding instruction is completed, it awaits the arrival of the next instruction. Thus, it executes instructions strictly in the order of their arrival. Also, since no instruction ordering is provided for the model it restricts arrival of more than one instruction from the same processor at the same instant. All the processor rules are mandatory, thus, if a memory instruction can be executed, it is executed and *Retire* before the next instruction is taken up for execution.

5.1.2 The Shared Memory Model for Cachet

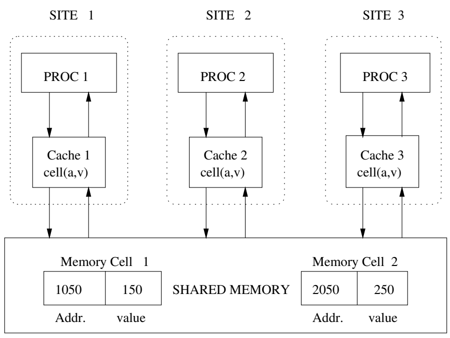

Figure 3. Model of Cachet

We model the full set of given CACHET specification on the CRF as a pure ESTEREL program (without incorporating any C code) which has parallel modules running as individual processes for each of the processors and their caches. For doing this, we propose a "down-scaled" version of the aforementioned DSM system (Figure 1) consisting of three processor sites each having a single sache cell. The shared memory implemented has only two memory cells with static values for addresses (as 1050 and 2050). Figure 3 shows an abstract model of our proposed shared memory system. The different transitions of the Sache cell states are shown in Figure 4.

The *State* of the model consists of current values of the global variables that include those representing the current value of the two shared-memory locations. These global variables are initialized to a pre-assigned value in the Start State and transitions thereon, from one state to the other, take place according to CRF memory rules. There is a simple request and response

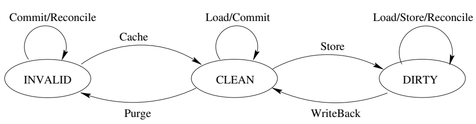

Figure 4. Sache State Transitions in CRF

coordination between the cache and the memory. However, a strong exclusion constraint is put on the processors regarding access of global variables in order to prevent false writing. Individual component modules are synchronized on signals such that any instruction initiated by a processor will definitely be completed within a finite amount of time. This defines a *fairness* property for our model. That is, for example, if a processor starts a *Loadl* instruction, it will eventually get a copy of data and then *Retire* the instruction. We also made certain assumptions which the protocol specification allows without affecting its behavior or causing any inconsistencies. These include restricting the number of instructions per processor at any instant to one and allowing next instruction to be issued only after the previous one completes its execution.

The ESTEREL program has been implemented for running simulations of the CACHET protocols in XES, a graphical simulator for ESTEREL programs. The ESTEREL module corresponding to each *Site* replicates each of the processor instructions thrice and these are issued as input event in the ESTEREL environment using XES. The complete model has been simulated and tested under the XES environment.

6. Verification of Cachet Protocols

Here, we show the verification of these protocols for selected important properties using the *conjunctivity* property of the synchronous formalism. That is, after establishing the properties of the basic CRF, we establish the new properties and show that the basic properties remain invariant. We use the XEVE tool [5] which uses *symbolic bisimulation* and *observers* for verifying the model against the required properties. The tool has been used to build automata in the FC2 format [8, 7] that can be used with the AUTOGRAPH visual tool. XEVE also facilitates checking output signals for possible emissions and this further helps to check observer signals during verification of programs. Also, appropriate hiding of irrelevant signals during verification using XEVE further ensures property by restricting the reachable states to the property being verified. Thus, signal checking in XEVE and analysis of the automata in FC2 format [7] generated by XEVE from abstracted programs enable us to verify the properties.

The most important property is: "*Simultaneous Commit from different processors on the same memory location should not complete*" i.e. they should be *Commit* exclusively. In the sequel, we check as to which protocol satisfies EXCLUSIVE COMMITS and also describe verification of other properties of CACHET protocols.

6.1. Cachet Base Protocol

This is the simplest of the three. The EXCLUSIVE COMMITS property exposes the behaviour in cases when more than one processor sites (each having a *Dirty* copy of the same memory location) simultaneously issue a *Commit* instruction for a location. In such cases, the memory controller would receive a WRITEBACK request from each of the processor simultaneously. Since this protocol doesn't implement any bookkeeping about copies of locations at other *Sites*, the memory controller attempts to write different values on the same location in the memory leading to a state of inconsistency in some processor cache. The Base protocol does not provide any corrective behaviour to such cases. In ESTEREL model, such a situation very rightly shows up with "Multiple Signal Emission" error as the global memory valued signal is emitted with multiple Writeback values simultaneously.

To verify this condition, we have generated a counter example which demonstrates that the property doesn't hold for the Base protocol. We run an observer module for the property in parallel with ESTEREL simulation module and show that the alarm signal is emitted. This is verified from the XEVE tool which generates a counter simulation run showing one instance of simulation where the danger signal is emitted by the observer. The observer module as given below checks if the Memory location is updated when more that one simultaneous *Commit* instructions are issued from different processors for the same location in the Main Memory.

```
module SIMUL_COMMITS:
input COMMIT1_1050, COMMIT2_1050,
        WRITEBACK1,WRITEBACK2,MEMVAL;
output COMMIT_EXCLUSION_VIOLATED;
loop
  await COMMIT1_1050 and COMMIT2_1050;
  present WRITEBACK1 and WRITEBACK2 then
    await tick;
    present MEMVAL then % Updated Mem.
      emit COMMIT_EXCLUSION_VIOLATED
    end present
  end present
end loop
end module
```

In the reduced FSM as realized through AUTOGRAPH– a graphical tool to load and visualize automata– emission of the Alarm signal COMMIT_EXCLUSION_VIOLATED on some transitions could be seen (the automata is not shown for the lack of space). If the memory value is updated under these conditions then the new value is some incorrect garbage value stored in the memory. This leads to a *cache inconsistency* conditions at some processor *Sites*. This is verified by the emission of the alarm signal COMMIT_EXCLUSION_VIOLATED. As the signal is found emitted during verification an input sequence is extracted that leads to a state where the signal is emitted. This sequence is saved in a *.csimul* file, called a tape, and can be simulated on the ESTEREL source using the XES graphical simulator.

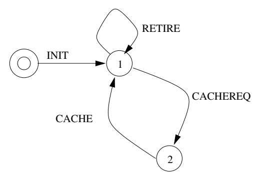

Figure 5. Basic Automata of Loadl

However, this scenario is different for other microprotocols of CACHET where more constraints on the protocol behavior prevents violation of this property. In the Migratory version of CACHET, since exclusivity of memory copies at different *Sites* is maintained, the question of multiple *Commit* instructions issued for the same location does not arise. In the Writer-Push protocol the memory maintains a directory state storing identifiers of *Sites* having a copy of that location. In such cases, this protocol arbitrates (using some kind of decision making) among the multiple Writebacks and allows only one of them to complete. We shall illustrate this later while we verify other protocols. In addition, we illustrate the verification of the following properties of the Base protocol:

1. Simultaneous Loadl *instructions from different sites must be allowed to complete successfully.*
2. A Cache Request should be sent to the memory if and only if the requested location is currently not cached.

For purposes of verification, we abstract from the complete program one that takes only *Loadl* inputs from two processor sites and observe its behavior for simultaneous inputs of *Loadl* from both processors. This does not reduce the generality since for a particular property we can hide away unwanted behavior thereby reducing the scope of verification. The basic behavior of the Base protocol is as shown in Figure 5.

For analyzing the simultaneous *Loadl* behavior, we design observers to check these properties running in parallel synchronously with the above two *Site* program and obtained the minimized automata using the bisimulation reduction method in *Xeve*. Observer 1 as shown in the following program module verifies the behavior of simultaneous *Loadl* from different sites. If simultaneous *Loadl* generate CACHEREQ for the same memory location then both instructions should be allowed to *Retire* by loading that location into respective caches. For this the observer follows those traces in the main program where CACHEREQ is issued for each simultaneously occurring *Loadl* and checks for the presence of RETIRE signal for each of them. If in any case, one or more of these expected RETIRE signals are absent then observer emits an Alarm signal denoting violation of property. Observer 2 verifies the second property by observing that a CACHEREQ is issued only when a cache miss occurs. This observer emits an Alarm if a CACHEREQ is made to the memory during any of the traces (which it follows) involving absence of CACHEMISS on a *Loadl* instruction.

```
module Observer1:
input LOADL, CACHEREQ, RETIRE;
output ALARM_SIGNAL1;
loop
    await LOADL;
    present CACHEREQ then
        await tick;
        present [not RETIRE] then
            emit ALARM_SIGNAL1
        end present
    end present
end loop
end module
```

```
module Observer2:
input LOADL, CACHEREQ, CACHEMISS;
output ALARM_SIGNAL2;
loop
    await LOADL;
    present not CACHEMISS then
        await tick;
        present CACHEREQ then
            emit ALARM_SIGNAL2
        end present
    end present
end loop
end module
```

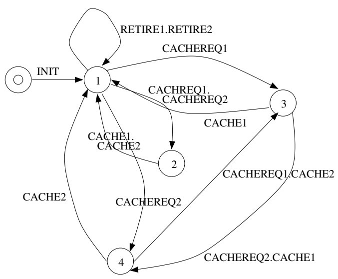

Figure 6. Automata of Retire Simulation

In the above discussions, by saying "presence of RETIRE for an instruction", we mean that it has been correctly executed and the results are generated according to that given in the specification (here loading of locations into cache). We use similar definition for *Retire* when labeling the automaton describing transitions when instructions *Retire* without any state changes. On minimizing, we obtain the automata shown in Figure 6. From this automata, we find that apart from satisfying the necessary conditions like being deadlock-free, the program never emits the observer ALARM signal that signifies the violation of the above mentioned properties. Furthermore, from the transitions of the automata, we find that this behavior follows the CACHET specification wherein a CacheReqeuest signal always follows a CacheMiss while the CacheRequest is always definitely followed by the Cache signal denoting the successful completion of the *Loadl* . This also verifies the property that a CacheRequest is eventually acknowledged by the Memory with a Cache message.

The Observer signal was also verified by the Signal Emission Checking tool of XEVE thereby verifying that the above properties would hold in all conditions. These properties are also enforced for the other micro-protocols and can be assumed to be true for them as well since the basic loading behavior for the Base protocol remains the same for other protocols too.

6.2. Cachet Migratory Protocol

Apart from the two properties of the Base protocol that also hold true for this protocol, this protocol also satisfies the EXCLUSIVE COMMITS property because of the constraints placed on its behavior by the specification. We know from Section 3 that in this protocol locations are exclusively owned by processors. Hence, the condition of addressing simultaneous *Commit* for the same location never arises in this protocol behavior. We verify the most important property: "Ensuring the Exclusive Ownership of memory addresses among different Sites".

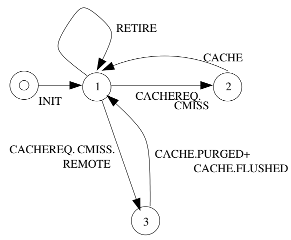

Figure 7. Automata of Migratory Protocol

Since the protocol requires extra states for keeping track of the Memory it becomes more complex making verification of specific properties difficult. Hence, for verifying this property, we abstract from the complete model one that only takes *Loadl* /*Storel* inputs from two processors. This can be done since migration of ownership occurs only when there is *Loadl* or *Storel* with the address being cached at some other *site*. It is required to purge or flush the copy from the other *Site* before loading it. The observer runs in parallel with the program and observes its behavior for purging of copies for proper migration of addresses thereby ensuring the exclusivity of address among different sites. The basic behavior of the Migratory protocol is as shown in Figure 7 where CMISS (or CacheMiss) indicates that a cache miss occurred when the instruction was issued while emission of REMOTE indicates that address was already cached at some other remote *Site*. The actual simulation program along with the observer module for a two *Site* memory system generated a 10 state, 31 transition automata (not shown for brevity). From the following observer modules, we verified that the automata behaves exactly as described in Figure 7.

The first observer verifies the behavior (transitions between vertices 1 & 2) corresponding to loading of a new location from the main memory. To do this, it traces the simulated program up to the point where the memory is sent a CACHEREQ when a CMISS occurs along with absence of REMOTE. Then, it checks if the next action taken by the memory is acknowledging the above request with CACHE signal thereby allowing the Cache to load the location. However, if CACHE is absent, then the observer emits the ALARM_SIGNAL. It may be noted this in effect also forms one of the properties being verified for the Base protocol.

```
module Observer1:
input CACHEREQ, CACHEMISS, CACHE, REMOTE;
output ALARM_SIGNAL;
loop
  await CACHEREQ;
  present [CACHEMISS and not REMOTE] then
      await tick;
      present [not CACHE] then
          emit ALARM_SIGNAL
      end present
  end present
end loop
end module
```

The second observer verifies the other state transitions between vertices 1 & 3 corresponding to loading of a location that is currently cached at some other *Site*. This is achieved by tracing the complete program up to the point where the memory is sent a CACHEREQ when a CMISS and REMOTE occur together. At this point, the only action that the memory controller can take is "wait" until the location is purged from the remote *Site* and then to send the requested memory location's value along with a CACHE acknowledgment. Any other action or behavior if exhibited would violate the property. Now the observer checks if the memory acknowledges the request with CACHE only after it has received either a PURGED or a FLUSHED acknowledge signal from the remote *Site*. However, if the CACHE is absent or occurs before a PURGED or FLUSHED signal is received by the memory, then the observer emits the ALARM_SIGNAL. Since the danger ALARM is always absent when these observers are run in parallel with the complete program, we can conclude that the automata of the program is the same as that shown in Fig. 7.

```
module Observer2:
input CACHEREQ,CACHEMISS,CACHE,REMOTE,
                  PURGED,FLUSHED;
output ALARM_SIGNAL;
loop
  await CACHEREQ;
  present [CACHEMISS and REMOTE] then
    await tick;
    present [not CACHE and not (PURGED
                  or FLUSHED)] then
      emit ALARM_SIGNAL
    end present
  end present
end loop
end module
```

On analyzing the actual automata, we could verify that whenever REMOTE is present the loading instruction completes (emits CACHE) only when the memory gets a PURGED or FLUSHED acknowledge message from the other *Site*. That is, exclusivity is maintained as *Loadl* /*Storel* do not *Retire* until the other *Site* abandons the address by either purging or flushing it.

6.3. Cachet WriterPush Protocol

The most important property for this protocol is: *Ensuring that at any instant a* Clean *cell always contains the same data as the memory.* That is, we have to ensure that a memory location cannot be updated by a WriteBack (leading to *Clean* cell) before all Clean copies have been purged from other caches. However, things become more complicated when there are multiple WRITEBACKs for the same location. This essentially means that the protocol should also satisfy the EXCLUSIVE COMMITS property discussed earlier. The specification provides for some bookkeeping that accommodates this situation by recording all writebacks in a transient state. However, it purges the location from all but one site to which a WriteBack acknowledge is sent. This means that it should satisfy the EXCLUSIVE COMMITS property as it never allows more than one writeback to complete successfully.

In this case, we simulate a kind of arbiter (based on bookkeeping) in the memory controller that writebacks the cache with the largest value while flushing the other competing WRITEBACKs (*Clean* copies are purged as in the other case). However, exclusive WRITEBACKs are allowed to complete simultaneously as already considered for the CRF model. To avoid state explosion, we abstract from the complete program one that only takes *Commit* (for WRITEBACK) and *Storel* (to create *Dirty* copies for WRITEBACK) and obtain the minimized automata. The basic behavior is shown in Figure 8. The following synchronous observer which continuously checks for the violation of these two conditions is designed for establishing the properties.

1. Single WRITEBACK with Clean *copies present at remote sites. Action - Writeback only after purging all remote copies.*
2. Multiple WRITEBACKs. Action - Purging and/or Flushing of some caches at remote sites before writing back cache at only one Site*.*

The observer has two parallel loops one each for property (REMOTE indicates presence of *Clean* copies at remote sites).

```
module Observer:
input WRITEBACK[i],WBACK,WBACKFlush,
                    REMOTE,PURGED;
output ALARM_SIGNAL;
loop  % Case 1
  await  WRITEBACK;
  present REMOTE then
     await tick;
     present  [not PURGED and WBACK] then
       emit ALARM_SIGNAL
     end present
  end present
end loop
||
loop  % Case 2-Multiple WriteBack signals
  await WRITEBACK[n]; % n ≤ i
  present REMOTE then
     await tick;
     present [not(PURGED and
          WBACKFlush[x]) and WBACK] then
        emit ALARM_SIGNAL
     end present
  else
     await tick;
     present  [not WBACKFlush[x] and
                 WBACK ] then % x=n-1
        emit ALARM_SIGNAL
     end present
  end present
end loop
end module
```

The minimized automata generates an automata with 14 state and 50 transitions (not shown for brevity). The observer shown above establishes the behavior corresponding to that shown in the basic automata. The second parallel loop verifies: *for multiple WriteBacks the protocol checks for the presence of WBACKFlush signal for all but one processor (indicated by x = n-1) before writing back the last sache to the memory* – illustrating the satisfaction of EXCLUSIVE COMMIT property that was violated in the Base protocol.

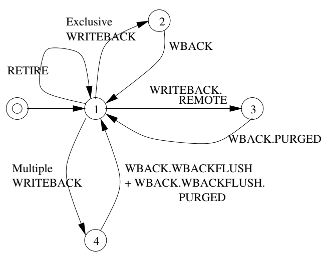

Figure 8. Automata of WriterPush Protocol

7 Discussions

While verifying CACHET protocols we had to handle state explosion, finiteness of testing and restrictions imposed by the tools. Directly writing observer programs for verification over the complete program presented a lot of problems due to the complexity of the automata and the limitations of the tool; for example, large number of input signals in ESTEREL led to a large set of behaviors over single transitions (longer labels) thereby making the analyzing process of automaton very difficult. We overcame most of the issues by: (i) Making appropriate program abstractions on the general automata corresponding to the complete model along with observer modules, (ii) Keeping the number of signals to the minimum relative to the property on hand, working on boolean abstractions as far as possible, and (iii) Hiding and renaming signals to avoid lengthy labels (imposed by the tool as it generates lengthy system names). It would have been nice if we had a tool that would prune the whole branch of the automata that will not enter into checking the correctness of the property reflected through the observer as in LUSTRE (achieved through the *pre* operator).

In short, our experience shows that synchronous languages can be effectively used for the verification of complex protocols with appropriate abstractions due to strong binding between simulation and verification (which exploits the observer property).

References

[1] X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. CSG Memo 414, LCS, MIT, August 1998.

[2] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. *26th Int. Symp. on Computer Architecture, Atlanta* , 1999.

[3] X. Shen, and Arvind. A Methodology for Designing Correct Cache Coherence Protocols for DSM Systems. CSG Memo 398 (A), LCS, MIT, June 1997.

[4] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. Proc. *AMAST'93*, Workshops in Computing, Springer Verlag, 1993.

[5] A.Bouali. XEVE: an ESTEREL Verification Environment. CAV 98, June/July 1998

[6] G. Berry and G. Gonthier. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. SCP, 19, pp. 83-152, 1992

[7] A. Bouali, A. Ressouche, R. de Simone, V. Roy. The FC2TOOLS User Manual (1.0), TR191, INRIA, Apr. 1996.

[8] E. Madeliane, and R. de Simone. The Fc2 Reference Manual. (ftp from cma.cma.fr:pub/verif as *fc2refman.ps.gz*), 1993.