Formal Verification of Safety Properties for a Cache Coherence Protocol
Verificación Formal de Invariantes para un Protocolo de Coherencia de Caché

Sergio Ramírez, Camilo Rocha
Escuela Colombiana de Ingeniería Julio Garavito
Bogotá D.C., Colombia
Email: {sergio.ramirez,camilo.rocha}@escuelaing.edu.co

***Abstract*—This paper presents a case study on the formal specification of a cache coherence protocol and the verification of some of its safety properties. Cache coherence refers to the consistency between the contents of a memory resource shared by many processes, that can have read and write access, and each local copy of the memory contents. The protocol presented in this paper is based on the ESI standard in which a central controller can grant *exclusive* and *shared* access to the memory, and also *invalidate* access to the memory. The formal specification of the protocol is presented as a rewriting logic theory and it is fully executable in the Maude system. The safety properties presented in this paper are linear temporal logic (LTL) formulas expressing invariants about mutual exclusion among the processes accessing the shared memory resource. By using Maude's search functionality and LTL model checker, some of these invariants can be proved automatically for a finite number of processes. This paper also presents an initial exploration on the mechanical verification of an invariant for *any* number of processes by means of deductive techniques based on inductive reasoning.**

***Abstract*—Este artículo presenta un caso de estudio acerca de la especificación formal de un protocolo de coherencia de caché y la verificación formal de algunos de sus invariantes. Coherencia de caché se refiere a la consistencia del contenido de un recurso de memoria compartido por varios procesos, sobre el cual se pueden otorgar accesos de lectura y escritura. El protocolo presentado en este artículo está basado en el estándar ESI en el cual un controlador central puede otorgar acceso *exclusivo* o *compartido* a la memoria, y también puede *invalidar* accesos a la memoria. La especificación formal del protocolo se presenta como una teoría de la lógica de reescritura y es completamente ejecutable en el sistema Maude. Las propiedades presentadas en el artículo son fórmulas de la lógica temporal lineal (LTL) que expresan propiedades de invariancia acerca de la exclusión mutua entre procesos que acceden al recurso de memoria compartido. Usando la funcionalidad de búsqueda en Maude y su verificador de modelos LTL, algunos de estos invariantes pueden ser demostrados automáticamente para una cantidad finita de procesos. Este artículo también presenta una exploración inicial en la verificación mecánica de invariantes para *cualquier* cantidad de procesos usando técnicas basadas en razonamiento inductivo simbólico.**

I. Introduction

When processes in a system maintain caches of a common memory resource, problems may arise with inconsistent data. This is particularly true of CPUs in a multiprocessing system because a process could be left with an invalid cache of memory without any notification of the change. *Cache coherence* is the consistency of shared resource data that ends up stored in multiple local caches. For example, if a process always reads the new value from a memory location after a write made by another process in that same memory location, then the state of the memory is coherent. A coherence protocol is a protocol that maintains the consistency memory location after a write made by another process in that same memory location, then the state of the memory is coherent. A *coherence protocol* is a protocol that maintains the consistency between all the caches in a system of distributed shared memory, and it is intended to manage conflicts and maintain consistency between cache and memory. The ESI model is a cache coherence and memory coherence model [1] in which every cache line is marked with one of three states: *exclusive* (the cache is present only in the current cache and it matches main memory), *shared* (the cache may be stored in other caches of the machine and it matches the main memory), or *invalid* (the cache line is invalid or unused). This paper presents a case study on the formal specification and analysis of critical safety properties of an ESI-based cache coherence protocol found in [1]. This work is a summary of the development and main results reported in [2].

The first contribution of this paper is the formal specification of the cache coherence protocol in rewriting logic [3], a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, and interaction. Rewriting logic can be used for specifying a wide range of systems and languages in various application fields; it also has good properties as a metalogical framework for representing logics (see [4] for more information). The formal specification of the cache coherence protocol is fully executable in Maude [5] and, thus, it can be formally analyzed with the wealth of tools available for rewriting logic such as, for instance, Maude LTL Model Checker [6] and the Maude Invariant Analyzer tool [7].

The second contribution of this paper is to report on the algorithmic verification of safety properties of the cache coherence protocol. These safety properties include invariants about the mutual exclusion for write access to the main memory achieved by the protocol and invariants about the formal representation of the system. Since for a state with a fixed number of processes initially in the system, the set of reachable states is finite, the verification of these invariants can be obtained automatically for specific instances of the system, with the help of Maude's search command and LTL model checker. This paper also explores the limitations of such an algorithmic verification due to the exponential explosion of the system's state search space as the number of initial processes increases.

The third contribution of this paper is to report on an initial exploration of deductive verification of simple safety properties of the cache coherence protocol. One of the advantages of deductive verification over the algorithmic one is that the former is not limited by the number of states in the system and thus can directly be used to prove invariants for infinite state systems, either the set of initial states is infinite or the set of reachable states from some initial state is infinite. In the case of the cache coherence protocol studied in this paper, the set of initial states is, ideally, infinite. That is, the safety properties of the protocol should be proven for *any* reasonable initial configuration with an arbitrary number of processes. Towards this goal, this paper reports on the mechanical verification of an invariant with the help of the Maude Invariant Analyzer tool [7], which is based on inductive reasoning over the rewrite relation induced by the rewriting logic specification of the cache coherence protocol.

Paper outline. Section II includes some preliminary material on rewriting logic and some of its tools. Section III presents an overview of the cache coherence protocol and its formal specification in rewriting logic. Section IV reports on the algorithmic verification of several invariants, while Section V reports on the deductive verification of one invariant. Finally, Section VI includes some related work and concluding remarks.

II. Preliminaries

An *order-sorted signature* is a triple $\Sigma = (S, \leq, F)$ with finite poset of sorts $(S, \leq)$ and finite set of function symbols F. The binary relation $\equiv_{\leq}$ denotes the equivalence relation generated by $\leq$ on S and its point-wise extension to strings in S^*. A *top sort* in Σ is a sort $s \in S$ such that if $s' \in S$ and $s \equiv_{\leq} s'$, then $s' \leq s$. For any sort $s \in S$, the expression $[s]$ denotes the connected component of s, that is, $[s] = [s]_{\equiv_{\leq}}$, called the kind of the connected component. The collection $X = \{X_s\}_{s \in S}$ is an S-sorted family of disjoint sets of variables with each X_s countably infinite. The set of terms of sort s is denoted by $T_\Sigma(X)_s$ and the set of ground terms of sort s is denoted by $T_{\Sigma,s}$, which are assumed nonempty for each s. The expressions $T_\Sigma(X)$ and T_Σ denote, respectively, the set of all terms and the set of all ground terms; $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the respective term algebras. The set of variables of a term t is written *vars*(t) and is extended to sets of terms in the natural way. A *substitution* is an S-indexed mapping $\theta : X \longrightarrow T_\Sigma(X)$ that maps variables of sort s to terms of sort s and is different from the identity for a finite subset of X. The identity substitution is denoted by *id*. The expression $ran(\theta)$ denotes the set of variables introduced by θ. Substitutions extend homomorphically to terms in the natural way. A substitution θ is called *ground* iff $ran(\theta) = \emptyset$. The application of a substitution θ to a term t is denoted by $t\theta$ and the composition of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$.

A Σ-*equation* is an unoriented pair $t = u$ with $t, u \in T_\Sigma(X)_s$ for some sort $s \in S$. A *conditional* Σ-*equation* is a triple $t = u$ **if** γ, with $t = u$ a Σ-equation and the *condition* γ is a finite conjunction of Σ-equalities $\bigwedge_{i \in I} t_i = u_i$; it is called *unconditional* if γ is the empty conjunction. An *equational theory* is a tuple (Σ, E) with order-sorted signature Σ and finite set of (possibly conditional) Σ-equations E. An equational theory (Σ, E) induces the congruence relation $=_E$ on $T_\Sigma(X)$ defined for any $t, u \in T_\Sigma(X)$ by $t =_E u$ iff $(\Sigma, E) \vdash t = u$. The expressions $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$ denote the quotient algebras induced by $=_E$ over the algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$, respectively; $\mathcal{T}_{\Sigma/E}$ is the *initial algebra* of (Σ, E).

A Σ-*sequent* is an oriented pair $t \to u$ with $t, u \in T_\Sigma(X)_s$. A Σ-*rule* is a sentence $t \to u$ **if** γ, where $t \to u$ is a Σ-*sequent* with $t, u \in T_\Sigma(X)_s$ for some sort $s \in S$ and the *condition* γ is a finite conjunction of Σ-equalities. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with equational theory $\mathcal{E}_\mathcal{R} = (\Sigma, E)$ and a finite set of Σ-rules R. A *topmost rewrite theory* is a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ such that for some top sort $\mathfrak{s} = [\mathfrak{s}]$ and for each $t \to u$ **if** $\gamma \in R$, the terms t, u satisfy $t, u \in T_\Sigma(X)_\mathfrak{s}$ and $t \notin X$, and no operator in Σ has $\mathfrak{s}$ as argument sort. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\to_\mathcal{R}$ on $T_{\Sigma/E}(X)$ defined for every $t, u \in T_\Sigma(X)$ by $[t]_E \to_\mathcal{R} [u]_E$ iff there is a *one-step* rewrite proof $\mathcal{R} \vdash t \to u$.

Executability conditions. It is assumed that the set of equations of a rewrite theory $\mathcal{R}$ can be decomposed into a disjoint union $E \uplus A$, with A a collection of axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo* A producing a finite number of A-matching substitutions, or failing otherwise. It is also assumed that the equations E can be oriented into a set of *ground sort-decreasing*, *ground confluent*, and *ground terminating* rules $\overrightarrow{E}$ modulo A. The expression $t{\downarrow_{\Sigma,E/A}} \in T_{\Sigma,s}(X)$ denotes the E/A-*canonical form* of $t \in T_\Sigma(X)$. The rules R in $\mathcal{R}$ are assumed to be *ground coherent* relative to the equations E modulo A [8].

LTL model checking. For a set of atomic predicates Π, $p \in \Pi$, and $[t]_E \in T_{\Sigma/E,\mathfrak{s}}$, $\mathcal{E}_\Pi$ defines the *semantics of* p in $\mathcal{T}_\mathcal{R}$ as follows: it is said that $p([t]_E)$ *holds* in $\mathcal{T}_\mathcal{R}$ iff $\mathcal{E}_\Pi \vdash p(t) = \top$. This defines a Kripke structure $\mathcal{K}_\mathcal{R}^\Pi = (T_{\Sigma/E,\mathfrak{s}}, \to_\mathcal{R}, L_\Pi)$ with labeling function L_Π such that, for each $[t]_E \in T_{\Sigma/E,\mathfrak{s}}$, the semantic equivalence $p \in L_\Pi([t]_E)$ iff $p([t]_E)$ holds in $\mathcal{T}_\mathcal{R}$. Then, all of LTL can be interpreted in $\mathcal{K}_\mathcal{R}^\Pi$ in the standard way [9], including the "always" ($\Box$), "next" ($\bigcirc$), and "strong implication" ($\Rightarrow$) operators.

InvA: *The Maude Invariant Analizer Tool.* For a topmost rewrite theory $\mathcal{R}$ and of a set of state predicates Π specified in Maude, the InvA tool [7] mechanizes: (i) a set of inference rules that can reduce a ground stability or ground invariance property φ to a finite collection of equational proof obligations of the form $t = u$ **if** γ and (ii) rewriting-based reasoning and narrowing procedures, and SMT decision procedures for automatically discharging as many of these equational proof obligations as possible. The main idea is that if all equational proof obligations generated by the InvA are successfully discharged, then the Kripke structure $\mathcal{K}_{\mathcal{R}}^{\Pi}$ associated to the initial reachability model of $\mathcal{R}$ satisfies φ. The reader is referred to [7], [10] for details on (i) and to [11] for details on (ii).

III. Formal Specification

This section presents an overview of the ESI-based cache coherence protocol studied in this paper and the main aspects of its formal specification in the syntax of Maude. The specification under analysis contains one single location, given that if the protocol works for that one location, then it will work for any number of locations as they are independent of each other in the given representation. There is no use of connected cache lines for a number of memory locations, for example. The reader is referred to [2] for more details on the protocol and its full formal specification in rewriting logic.

The *ESI Cache Coherence Protocol* [1] is a protocol that uses a controller for coordinating processes that require access to cache lines (or memory blocks). ESI is a coordination model based on *exclusive* or *shared* access to a resource. The controller can also *invalidate* (or revoke) access to the resource. In the system, every process has an identifier and a label that indicates its state: *critical*, *share*, or *idle*. These states correspond to writing in the memory, reading in the memory, and waiting for an access, respectively.

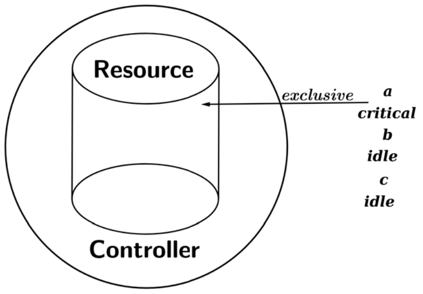
As an example, Figure 1 depicts two states that the protocol can reach. In state (a), process a has exclusive access to the shared resource, while processes b and c are idle (i.e., they have no access or, equivalently, have invalid access to the resource). In state (b), processes a and b have shared access to the resource, while c does not. In general and under fairness constraints, any process will eventually have exclusive or shared access to the resource if it requires it.

Formal Modeling. The system states are representated by the sort `Sys` defined in the functional module `ESI-STATE` [2] as a 4-tuple as follows:

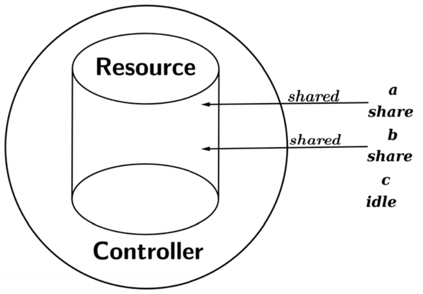
```
sort Sys .
op _:_:_|_ : Nat NatBag NatBag ProcBag
                -> Sys .
```

The arguments of a state are the data in the memory (as `Nat`), a bag (or multiset) of process identifiers with shared or exclusive access to the memory (as `NatBag`), a bag of process identifiers with exclusive access to the memory (as `NatBag`), and a bag of processes that belong to the system (as `ProcBag`). Sort `Nat` is that of natural numbers in Peano notation; natural numbers are used to identify processes and to represent memory contents. Sort `NatBag` represents bags of natural numbers whose contents are process identifiers; these bags are used to identify those processes with exclusive or shared access to the memory. Sort `ProcBag` is a bag of processes. The empty bag of natural numbers is identified by the constant `mtpi` and the empty bag of processes is identified by the constant `mtpr`. For instance, sort `NatBag` defined in module `ESI-STATE` is specified as follows:

```
sort NatBag .
op mtpi : -> NatBag .
op __ : NatBag NatBag
      -> NatBag [assoc comm id: mtpi] .
```



(a) Process a has exclusive access to the resource.



(b) Processes a and b have shared access to the resource.

Fig. 1: Two states in the coherence protocol.

The operation to construct non-empty bags is denoted by juxtaposition and it is commutative, associative, and has identity `mtpi`.

A process is identified by the sort `Proc`:

```
sort Proc .
op <_,_,_> : Nat Mode Nat -> Proc .
```

The arguments of a process are an identifier (as `Nat`), an internal state (as `Mode`), and a local copy of the memory (as `Nat`). Sort `Mode` represents the internal state a process can have, which can take one of the following three values:

```
ops idle share crit : -> Mode .
```

These values indicate, respectively, that a process is an invalid state (maybe waiting for access to the shared resource), with read-only access to the resource, or with exclusive access to the resource.

The protocol transitions are modeled with five rewrite rules. These rewrite rules indicate how to assign, remove, and renew access to the memory, and also how reads and writes are performed on the memory by the process.

```
var  M           : Mode .
vars ID Cac Mem : Nat .
vars IDSE IDSV   : NatBag .
var  PS          : ProcBag .

rl [fill]:
  Mem : IDSV : mtpi
```

```
          | < ID, idle, Cac > PS
  => Mem : ID IDSV : mtpi
          | < ID, share, Cac > PS .

rl [unfill]:
  Mem : ID IDSV : IDSE
          | < ID, share, Cac > PS
  => Mem : IDSV : IDSE
          | < ID, idle, Cac > PS .

rl [fille]:
  Mem : mtpi : IDSE
          | < ID, idle, Cac > PS
  => Mem : ID : ID IDSE
          | < ID, crit, Cac > PS .

rl [flush] :
  Mem : ID IDSV : ID IDSE
          | < ID, M, Cac > PS
  => Cac : IDSV : IDSE
          | < ID, idle, Cac > PS .

rl [store]:
  Mem : ID IDSV : IDSE
          | < ID, M, Cac > PS
  => Mem : ID IDSV : IDSE
          | < ID, M, Mem > PS .
```

The effects of these rules on a state can be summarized as follows:

`fill`:	request shared access to the memory;
`unfill`:	invalidate a shared access to the memory;
`fille`:	request exclusive access to the memory;
`flush`:	invalidate an exclusive access to the memory by first saving into the main memory the contents of the cache; and
`store`:	make a local copy from the memory.

IV. Algorithmic Verification of Invariants

This section presents details on the algorithmic verification in Maude of safety properties for the cache coherence protocol specified in Section III. In particular, this section presents the verification of invariants by means of Maude's `search` command [5] and the Maude LTL Model Checker [6]. This section also includes an overview of some limitations found during the algorithmic verification task.

A safety property asserts that something bad never happens, for instance, that the system at hand can not reach an "error" state. Invariants are among the most important safety properties. One key invariant the formal specification in Section III must satisfy is the achievement of mutual exclusion for accessing the shared memory. Specifically, the cache coherence protocol must satisfy the following mutual exclusion property:

> From any initial state, it should be impossible for the system to reach a state in which two or more processes simultaneously have write access to the shared memory.

As a matter of fact, the ESI protocol has infinitely many (initial) states. However, for any initial state, the set of reachable states is finite and then model checking is feasible (in theory, at least) as a decision procedure for the verification of invariants. In what follows, some invariants and their algorithmic verification (for a fixed number of initial states) are presented, including the above-mentioned one about mutual exclusion.

A. Auxiliary Definitions

The following auxiliary definitions are used in the algorithmic (and deductive) verification of the protocol performed in this section and in Section V:

```
op card   : NatBag -> Nat .
op in     : Nat NatBag -> Bool .
op _===_  : Nat Nat -> Bool [ comm ] .
op subbag : NatBag NatBag -> Bool .
```

These functions specify:

`card`:	the number of elements in a bag of identifiers.
`in`:	if an identifier belongs to a bag of identifiers.
`===`:	if two natural numbers are equal.
`subbag`:	if a bag is contained in another bag.

The recursive definition of these operations is presented below in the syntax of Maude:

```
vars N1 N2   : Nat .
vars NB1 NB2 : NatBag .

eq card(mtpi)    = 0 .
eq card(N1 NB1)  = s(card(NB1)) .
eq 0 === 0      = true .
eq N1 === N1    = true .
eq s(N1) === 0      = false .
eq s(N1) === s(N2) = N1 === N2 .
eq in(N1, mtpi) = false .
eq in(N1, N2 NB1)
 = (N1 === N2) or in(N1,NB1) .
eq subbag(mtpi, NB2)     = true .
eq subbag(N1 NB1, mtpi)  = false .
eq subbag(N1 NB1, N1 NB2)
 = subbag(NB1,NB2) .
ceq subbag(N1 NB1, NB2)
 = false if not(in(N1,NB2)) .
```

These equational definitions can be understood as follows:

`card`:	the empty bag has cardinality 0 and each element of a non-empty bag accumulates one unit to the cardinality of the bag.
`_===_`:	a natural number is equal to itself, including the case in which it is 0; 0 is different to any non-zero number; two non-zero numbers are equals iff their predecessors are equal.
`in`:	the empty bag does not contain elements; an element `N` belongs to a bag `NB` iff `N` is equal to some element in the bag `NB`.
`subbag`:	the empty bag `mtpi` is contained in any bag; a non-empty bag isn't contained in the empty bag; a non-empty bag is contained in another non-empty bag if both bags have a common element and recursively, without this element, the remaining bags satisfy the same property.

The operation `_===_` is called an *equality enrichment* and it defines, operationally and mathematically, the inductive equality for any abstract data type [12].

B. *Verification with Maude's* `search` *Command*

The cache coherence protocol must satisfy some invariants that assert its correct performance. For instance, when a process is modifying the contents of the shared memory, it must be the case that this process has exclusive access to the memory. In this case, any other process must have invalid access since they are not even allowed, at that specific moment, to read from the memory.

The invariants verified in this subsection are listed below with *exclusive* and *valid* denoting, respectively, the bags of process identifiers granted exclusive access and granted shared access to the shared memory maintained by the controller:

1) $|exclusive| \leq 1$.
2) $exclusive \subseteq valid$.
3) $exclusive \neq \mathtt{mtpi} \implies valid = exclusive$.

Property $|exclusive| \leq 1$. This property states that the number of process with write access to the memory is at most one. If this property holds for any reachable state from the set of initial states, then the cache coherence protocol achieves the mutual exclusion property for the processes accessing the shared memory.

The following search command tries to find a state falsifying the given property making an exploration from a initial state with 3 processes, each with internal state *idle*, and the two bags with process identifiers are both empty.

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 >
                         < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag
           | PS:ProcBag
       such that card(NBE:NatBag) > 1 .
```

This search command generates the following output:

```
No solution.
states: 979  rewrites: 6230 in 16ms cpu
        (14ms real) (389375 rewrites/second)
```

The search is performed over 979 reachable states (from the given initial state) and proves that no such state falsifies the property. That is, for the given initial configuration, the property $|exclusive| \leq 1$ holds. Although it is not a formal argument that ensures the correctness of the protocol, it is important to note that the search process could have been done, resulting in the same outcome, from an initial state with different process identifiers or internal cache values. This is a clear limitation of the algorithmic approach but, still, it provides strong evidence about the correctness of the mutual exclusion achieved by the protocol.

Property $exclusive \subseteq valid$. This is a representation invariant for the formal specification in Section III. It states that the controller maintains the sets of bags in such a way that the identifier of a process with exclusive access is also registered in the bag of identifiers with read access.

The following search command tries to find a state that falsifies the property:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 >
                         < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag
           | PS:ProcBag
       such that
       subbag(NBE:NatBag,NBV:NatBag)==false .
```

The search command generates the following output:

```
No solution.
states: 979  rewrites: 6230 in 96ms cpu
        (95ms real) (64895 rewrites/second)
```

This proves that the given property is an invariant (for the given initial state) of the protocol.

Property $exclusive \neq \mathtt{mtpi} \implies valid = exclusive$. This property states that if there is at least a process with exclusive access to the shared memory, then it must be the case that these processes are the only ones with read access to the memory. This is an important property for the correctness of the protocol because it implies that a process cannot have read-only access whenever there is a process with write access to the memory.

The following search command tries to find a state that falsifies the property:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 >
                         < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag
           | PS:ProcBag
       such that
       NBE:NatBag =/= mtpi and
       NBV:NatBag =/= NBE:NatBag .
```

The search command generates the following output and thus the given property is an invariant of the system (from the given initial state):

```
No solution.
states: 979  rewrites: 6942 in 28ms cpu
        (28ms real) (247928 rewrites/second)
```

The reader is referred to [2] for more examples, including the verification of the above properties for initial states having more than 3 processes and the verification of other invariants, with the help of Maude's `search` command.

C. *Verification with Maude LTL Model Checker*

LTL model checking provides a decision procedure for safety properties for the cache coherence protocol because the set of reachable states from a given initial states is finite.

The Maude LTL model checker uses the following satisfaction relation:

```
op _|=_ : State Prop -> Bool .
```

Sort `State` is a generic representation for the system's state sort and can be instantiated by using subsorts. Since `Sys` is the top sort and system state of the cache coherence protocol in Section III, the following subsort declaration is used:

```
subsort Sys < State .
```

Sort `Bool` represents Boolean values and sort `Prop` represents the set Π of atomic predicates (see [5] for more details).

Atomic predicates `idle`, `share`, and `crit` are defined equationally and are used for querying the internal state of a given process; they are defined as follows:

```
eq Mem : IDSV : IDSE | < ID, idle, Cac > PS
    |= idle(ID)
  = true .
eq Mem : IDSV : IDSE | < ID, share, Cac > PS
    |= share(ID)
  = true .
eq Mem : IDSV : IDSE | < ID, crit, Cac > PS
    |= crit(ID)
  = true .
```

Atomic predicates `in-idle`, `in-share`, and `in-crit` are defined equationally and are used to query the internal states of the process as *registered* by the controller in the bags of process identifiers it maintains:

```
 eq Mem : IDSV : IDSE | PS |= in-idle(ID)
  = not (in(ID,IDSV) or in(ID,IDSE)) .
 eq Mem : IDSV : IDSE | PS |= in-share(ID)
  = in(ID,IDSV) and not in(ID,IDSE) .
 eq Mem : IDSV : IDSE | PS |= in-crit(ID)
  = in(ID,IDSV) and in(ID,IDSE) .
```

Given a state s and an LTL formula ϕ (built from the atomic predicates in Π), the Maude LTL Model Checker uses the command $\texttt{modelCheck}(s, \phi)$ for checking the satisfaction of the semantic relation $s \models \phi$.

The following command checks that, for an initial state with three processes, it is never the case that a process has invalid access to the memory but the controller has assigned read or write access to it.

```
red modelCheck( 0 : mtpi : mtpi |
    < 1, idle, 31 >
    < 2, idle, 25 >
    < 3, idle, 44 >,
    [] (idle(3) ->  in-idle(3))) .
```

This model checking command generates the following output:

```
reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi |
        (< 2,idle,25 > < 3,idle,44 >)
                       < 1,idle,31 >,
        [](idle(3) -> in-idle(3))) .
rewrites: 24041 in 32ms cpu (29ms real)
          (751281 rewrites/second)
result Bool: true
```

The reader is referred to [2] for similar verification tasks, also considering different sets of initial states.

Number of Processes	Reachable States	Transitions	Time (ms)
1	9	42	0
2	60	360	4
3	979	7298	24
4	27720	255024	620
5	900469	10075518	28408
6	-	-	Time Limit

TABLE I: Search time for different verification tasks.

D. *Some Limitations of the Algorithmic Verification*

The algorithmic verification task of safety properties for the cache coherence protocol helped in identifying some limitations of the approach. In particular, the state space explosion problem can be experienced with initial states with a fairly small amount of processes.

Table I presents some experimental data on the time required for checking some invariants. For this purpose, the following Maude search command was given:

```
search in ESI-PROP :
      0 : mtpi : mtpi | init
      =>*
      Mem:Nat : NBV:NatBag : NBE:NatBag |
      PS:ProcBag
      such that card(NBV:NatBag) < 0 .
```

Here `init` represents the set of processes in the initial state under consideration. Note that the condition `card(NBV:NatBag) < 0` is unsatisfiable and hence the search command explores *all* reachable states without printing any state. In the case with 6 initial processes, the search task takes longer than 12 minutes, which was the time limit of the verification task for each inviariant.

V. Deductive Verification of Invariants

This section presents an initial exploration on the deductive verification of safety properties for the cache coherence protocol given in Section III. The deductive technique used in this section follows the development for the symbolic verification of invariants of rewrite theories proposed by C. Rocha and J. Meseguer [7], and its implementation in the Maude Invariant Analyzer Tool (InvA). One important feature of this approach is that the verification task does not depend on the finiteness of the set of initial or reachable states.

A. *Predicate Definition*

The invariant verified in this section ensures that no confusion is generated by the controller on the assignment of access to the shared memory resource.

```
ops init good-ids : Sys -> [Bool] .
```

Predicates `init` and `good-ids` specify, respectively, the set of initial states and the set of states in which the collection of processes is such that no two processes share the same identifier. These two predicates are equationally defined as follows:

```
eq init(Mem : mtpi : mtpi | PS)
  = set(exids(PS))
  eq good-ids(Mem : IDSV : IDSE | PS)
     = set(exids(PS))
```

In an initial state the bags of process identifiers maintained by the controller are empty, and the collection of processes in the system is such that no two processes share the same identifier. A "good state" is defined similarly with the difference that no restriction is imposed on the form of the collections of identifiers maintained by the controller. Note that both predicates are satisfied by *infinitely* many states.

Auxiliary functions `set` and `exids` in the definition above specify, respectively, that a bag is a set (i.e., it has no repeated element) and the collection of process identifiers. The reader is referred to [2] for details.

B. Mechanical Verification of the Invariant

The safety property proved in this section for the cache coherence protocol corresponds to the following LTL formula:

$$\texttt{init} \implies \Box\texttt{good-ids}.$$

This formula states that each reachable state from any initial state is such that no two processes have the same identifier.

The verification task is decomposed into two subtasks, namely, in proving the following two properties:

1) $\texttt{init} \implies \texttt{good-ids}.$
2) $\texttt{good-ids} \implies \bigcirc\texttt{good-ids}.$

Formula (1) is a purely equational implication stating that an initial state is a "good state". Formula (2) is a stability property and states, basically, that the set of "good states" is closed under the rewrite relation. Operator '$\bigcirc$' denotes the 'next' LTL operator.

These two formulas can be input to the InvA tool as follows:

```
 (analyze init(S:Sys) implies
     good-ids(S:Sys) in ESI-PREDS .)

 (analyze-stable good-ids(S:Sys)
                 in ESI-PREDS ESI .)
```

It is assumed that module `ESI-PREDS` [2] contains the state predicates and their corresponding auxiliary functions, and module `ESI` [2] contains the specification of the protocol.

When issuing the above-mentioned commands, the InvA tool generates the following output:

```
 rewrites: 6132 in 20ms cpu (20ms real)
           (306600 rewrites/second)
 Checking
   ESI-PREDS ||- init(S:Sys)
   => good-ids(S:Sys) ...
 Proof obligations generated:  1
 Proof obligations discharged: 1
 Success!

 rewrites: 281039 in 172ms cpu (170ms real)
           (1633947 rewrites/second)
 Checking
   ESI-PREDS ||- good-ids(S:Sys)
   => O good-ids(S:Sys) ...
 Proof obligations generated:  20
 Proof obligations discharged: 12
 The following proof obligations
 need to be discharged:
 ...
 5. from store : pending
  good-ids(#5:Nat : #9:Nat mtpi : #6:NatBag |
       #10:ProcBag < #9:Nat,#7:Mode,#5:Nat >)
   = true
  if set(exids(#10:ProcBag
                 < #9:Nat,#7:Mode,#8:Nat >))
   = true .
 ...
```

The tool generates 21 proof obligations and automatically discharges 13 of them. The remaining 8 proofs obligations are output to the user. These proof obligations can all be easily discharged by equational reduction to canonical form in Maude. For instance, proof obligation identified by label 5 can be discharged as follows:

```
 red set(exids(#10:ProcBag
                 < #9:Nat,#7:Mode,#8:Nat >))
  implies
  good-ids(#5:Nat : #9:NatBag mtpi : #6:NatBag
     | #10:ProcBag < #9:Nat,#7:Nat,#5:Nat >) .
```

The output for this command is `true`. For details on the remaining reductions the user is referred to [2]. Since all proof obligations can be discharged, the cache coherence protocol satisfies the invariant $\texttt{init} \implies \Box\texttt{good-ids}$.

VI. Concluding Remarks

A lot of research has been conducted in the formal verification of cache coherence protocols given their critical nature. For example, A. J. Hu et al. [13] describe the experience of applying formal verification to the cache coherence protocol of the HAL S1 System, a shared-memory and/or message-passing multiprocessor. They identify the pitfalls encountered and recommend ways to minimize them. A. Kriouile and W. Serwe [14] use model checking and deductive techniques to develop and validate a generic formal model for a specification proposed by ARM to implement system-level coherence.

This paper summarizes the main results of an undergraduate project that was developed in a period of 3.5 months, approximately. The full report is available online [2]. This paper presents the formal specification of a cache coherence protocol in rewriting logic. This specification is fully executable in the Maude system. Several safety properties have been verified, both algorithmically by model checking and deductively by inductive reasoning. This paper also explored the limitations of such an algorithmic verification due to the exponential explosion of the system's state search space as the number of initial processes increases. However, in the case of the cache coherence protocol studied in this paper, the set of initial states is, ideally, infinite. Towards this goal, this paper reported on the mechanical verification of an invariant with the help of the Maude Invariant Analyzer tool [7], which is based on inductive reasoning over the rewrite relation induced by the rewriting logic specification of the cache coherence protocol. All proofs have been mechanized and are available from [2].

As usual much work remains to be done. In particular, the initial exploration on the verification of invariants reported in Section V witnesses a promising approach for the deductive verification of more challenging invariants for the cache coherence protocol such as some of the ones presented in Section IV. Also, it is worth pursuing the use of these ideas and rewriting logic for the formal verification of larger cache coherence protocols in which other aspects of cache coherence such as, for example, cache miss/hit situations are also considered.

REFERENCES

[1] S. Ray, *Scalable Techniques for Formal Verification*. Springer, 2010.

[2] S. Ramírez, "Especificación formal y verificación de invariantes para un protocolo de coherencia del caché," Escuela Colombiana de Ingeniería, Bogotá, Colombia, Trabajo de grado 2015-1, 2015.

[3] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73 – 155, 1992, selected Papers of the 2nd Workshop on Concurrency and Compositionality. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439759290182F

[4] ——, "Twenty years of rewriting logic," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 7–8, pp. 721 – 781, 2012, rewriting Logic and its Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832612000707

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[6] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker and its implementation," in *Model Checking Software*, ser. Lecture Notes in Computer Science, T. Ball and S. Rajamani, Eds. Springer Berlin Heidelberg, 2003, vol. 2648, pp. 230–234. [Online]. Available: http://dx.doi.org/10.1007/3-540-44829-2_16

[7] C. Rocha and J. Meseguer, "Proving safety properties of rewrite theories," in *Algebra and Coalgebra in Computer Science*, ser. Lecture Notes in Computer Science, A. Corradini, B. Klin, and C. Cîrstea, Eds. Springer Berlin / Heidelberg, 2011, vol. 6859, pp. 314–328. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22944-2_22

[8] P. Viry, "Equational rules for rewriting logic," *Theoretical Computer Science*, vol. 285, pp. 487–517, 2002.

[9] E. M. Clarke, O. Grumberg, and D. Peled, Eds., *Model Checking*. MIT Press, 1999.

[10] C. Rocha and J. Meseguer, "Proving safety properties of rewrite theories," University of Illinois at Urbana-Champaign, Tech. Rep., 2010, available at http://hdl.handle.net/2142/17407.

[11] C. Rocha, "Automatic proof-search heuristics in the Maude Invariant Analyzer tool," *Revista Colombiana de Computación*, vol. 14, no. 2, 2013. [Online]. Available: http://revistas.unab.edu.co/index.php?journal=rcc&page=article&op=view&path%5B%5D=2017

[12] R. Gutiérrez, J. Meseguer, and C. Rocha, "Order-sorted equality enrichments modulo axioms," in *Rewriting Logic and Its Applications*, ser. Lecture Notes in Computer Science, F. Durán, Ed. Springer Berlin Heidelberg, 2012, vol. 7571, pp. 162–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34005-5_9

[13] A. Hu, M. Fujita, and C. Wilson, "Formal verification of the HAL S1 System cache coherence protocol," in *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, Oct 1997, pp. 438–444.

[14] A. Kriouile and W. Serwe, "Formal analysis of the ACE specification for cache coherent systems-on-chip," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science, C. Pecheur and M. Dierkes, Eds. Springer Berlin Heidelberg, 2013, vol. 8187, pp. 108–122. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41010-9_8