FMona: A Tool for Expressing Validation Techniques over Infinite State Systems

J.-P. Bodeveix and M. Filali

IRIT Université Paul Sabatier
118 route de Narbonne, F-31062 Toulouse
{bodeveix,filali}@irit.fr

Abstract. In this paper, we present a generic tool, called FMona, for expressing validation methods. we illustrate its use through the expression of the abstraction technique and its application to infinite or parameterized space problems. After a review of the basic results concerning transition systems, we show how abstraction can be expressed within FMona and used to build a reduced system with decidable properties. The FMona tool is used to express the validation steps leading to synthesis of a finite abstract system;then SMV and/or Mona validate its properties.

Keywords: abstraction, transition systems, model checker, monadic second order logic.

1 Introduction

In recent years, important work has been done in the design and implementation of general specification languages and validation systems. Usually, we distinguish three families of tools: model checkers (SMV [BCMD90], SPIN [Hol91]) to build a finite model and check its temporal properties; automatic proof tools (Mona [HJJ+95]) which offer a complete decision procedure if the underlying logic is decidable and proof assistants (Coq [BBC+97], HOL [GM94] and PVS [ORS92]) which offer an expressive higher order logic (and thus not decidable) and an assistance to the validation of formulas expressed in this logic.

Our experience in using these tools has led us to the following observations:

- model checkers are generally easy to use but their validation algorithm is "hardwired" and the available data structures are generally poor,
- automatic proof tools lack a specification language level to express methods
- and proof assistants lack powerful decision procedures and their integration is a delicate operation. Moreover their use is uneasy and require for instance the knowledge of the underlying type theory, the proof tactics, the tactic language and the underlying decision procedures.

In this paper, we relate an attempt to overcome these problems. We have chosen an intermediate approach combining an automatic proof tool and higher level

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 204–219, 2000.

aspects so that the expression of validation methods becomes easy. Of course, unlike proof assistants, our tool FMona cannot be used to validate the methods themselves. Encoded methods can only be instantiated on given applications. As our automatic proof tool target, we have chosen Mona as the main proof tool. Actually, within its underlying logic (WS1S) we can express transition systems and most of their basic properties without restricting to finite state spaces. Indeed, we can consider parameterized transition systems. We have used FMona to express several validation techniques, e.g. iterative methods with convergence acceleration over parameterized state systems and abstraction methods. In the following, we focus on the use of FMona to express and apply the abstraction method.

The remainder of this paper is organized as follows: section 2 presents the logic underlying the studied validation techniques and its associated tools. Section 3 describes the transition system formalism and states the theorems underlying the abstraction technique. In section 4, we describe the use of the abstraction method to the validation of always true and simulation properties. Sections 5 and 6 illustrate the technique on two concrete examples. Section 7 considers some ongoing work.

2 Monadic Second Order Logic and Related Tools

We recall below the definition of the two variants (WS1S and S1S) of the monadic second order logic of one successor [Tho90]. Then, we present the Mona tool deciding WS1S formulas and its high level interface FMona.

Definition 1 (The S1S and WS1S logics) *Let* $\{x_1, \ldots, x_n\}$ *be a family of first order variables and* $\{X_1, \ldots, X_n\}$ *a family of second order monadic variables. A primitive grammar of this logic can be defined as follows:*

- *A term* t *is recursively defined as:* $t ::= 0 \mid x_i \mid s(t)$
- *A logic formula* f *is recursively defined as:* $f ::= X_i(t) \mid \neg f \mid f \wedge f \mid \exists x_i.\ f \mid \exists X_i.\ f$

Notice that the successor function `s` is the only function.

Validity of a formula A closed formula is valid in S1S or WS1S if it is valid in the interpretation on the set $\mathcal{N}$ of naturals, where s is the successor function, first order quantifiers relate to the naturals and second order quantifiers to the subsets (finite in the case of WS1S) of $\mathcal{N}$. These two logics are decidable [Tho90].

The monadic second order logic naturally supports the method presented since it makes it possible on the one hand to express the concept of sequence of execution on a finite state space and the temporal properties associated, and on the other hand the refinement relation between a parameterized concrete space and a finite abstract space.

The Mona Tool The Mona tool [HJJ+95] implements a decision procedure for WS1S, based on the automata theory. The Mona syntax accepts the constructions of the WS1S logic presented in the preceding section. Mona data types are thus limited to booleans, naturals and finite sets of naturals. Thus, propositional, first order and second order variables are respectively declared, existentially and universally quantified by `var`$_i$, `ex`$_i$ and `all`$_i$ where i is the order of the variable.

The FMona Tool The FMona tool [BF99b] is a high level interface for Mona. For instance, it is possible to declare enumerated and record with update types and quantify over them. Furthermore, FMona allows the definition of higher order macros parameterized by types and predicates. FMona source code is type-checked, macro expanded and translated to pure Mona. The following example defines the transition relation `req1` between the states `st` and `st'`. The complete example is given in section 5.

```
type PC = {out,req,mutex}; type Sys = record{pc1,pc2: PC; y1,y2: nat;};
pred req1(var Sys st,st') =
  st.pc1=out & st'=st with {pc1:=req; y1:=st.y2+1;};
```

Note that the expression `st'=st with { pc1 := req; y1 := st.y2 + 1;}` expresses that `st'` is obtained by updating the fields `pc1` and `y1` of `st`.

We will use the FMona tool to express parameterized transition systems, abstraction relations, the synthesis of finite abstractions and the validation of their safety properties (mainly the so called always true properties).

3 The Formalism and the Basic Results

This section presents the formalism used: transition systems [Arn92]. We recall then the basic results concerning transition systems. The Coq proof assistant has been used to formalize the definitions and to validate the stated theorems.

Notations Given a set S, its complement is denoted $\overline{S}$. In the following, sets and predicates are identified. Given a relation $\varphi \subset A \times B$ and a subset $P \subset A$, we note $\varphi(P) = \{y \in B \mid \exists x : P(x) \wedge \varphi(x, y)\}$.

3.1 Transition Systems, Refinements, Simulations and Implementations

Definition 2 *A labeled transition system is defined by a quadruple $(E, I, L, \rightarrow)$, where E is a set of states, $I \subset E$ is the set of initial states, L is a set of labels and $\rightarrow \subset E \times L \times E$ is the transition relation. We will note $e \xrightarrow{l} e'$ instead of $(e, l, e') \in \rightarrow$.*

Definition 3 (Invariance) *A predicate P is* **invariant** *in the transition system S if: it is true over the initial states and is is preserved by each transition.*

A state $s \in E$ is **reachable** *in S if there exists a sequence $s_0, \ldots, s_n = s$ such that $s_0 \in I$ and $\forall i \in 0..n-1 : s_i \rightarrow s_{i+1}$. The set of reachable states of a*

transition system S will be denoted $\mathcal{A}cc(S)$. A predicate P is said to be **always true** *in S if it holds in all reachable states of S. This is denoted $S \vdash \Box P$.*

Theorem 1 (Sufficient condition for validity) *An invariant property is always true.*

Within FMona, we define the macro `reachable` and `always_true` as follows:

```
pred reachable(type State,
  pred(var State s) init, pred(var State s,s') tr, var State s) =
    ex array nat of State A: ex nat i: A[i]=s & init(A[0]) &
      all nat j where j < i: tr(A[j],A[j+1]);

pred always_true(type State, pred(var State s) p,
                 pred(var State s) init, pred(var State s,s') tr) =
  all State s: reachable(init,tr,s) ⇒ p(s);
```

Definition 4 (Refinement) *Given two transition systems with the same set of labels L,*
$C = (E_c, I_c, L, \rightarrow_c)$ and $A = (E_a, I_a, L, \rightarrow_a)$, and $\varphi \subset E_c \times E_a$. C refines A through φ, denoted $C \sqsubseteq_\varphi A$, if:

- *φ maps each initial state of C to an initial state of A.*
- *Given two states c and c' of E_c such that $c \xrightarrow{l} c'$ and a state $a \in E_a$ in relation with c through φ, there exists a state $a' \in E_a$ in relation with c' such that $a \xrightarrow{l} a'$.*

Definition 5 (Run) *Given a transition system $S = (E, I, L, \rightarrow)$. A run is a relation $\_ \stackrel{*}{\Rightarrow} \_ \in E \times L^* \times E$ inductively defined as:*

- *$e \stackrel{\epsilon}{\Rightarrow} e$*
- *if $e \stackrel{l}{\Rightarrow} e'$ and $e' \xrightarrow{a} e''$ then $e \stackrel{l.a}{\Longrightarrow} e''$*

We define the set of **finite traces** *$\mathcal{T}(S)$ of the system S as follows:*
$\mathcal{T}(S) = \{l \in L^* \mid \exists i \in I, e \in E : i \stackrel{l}{\Rightarrow} e\}$.

Definition 6 (Simulation and implementation) *Given two transition systems C and A. C* **simulates** *A through φ if for every state c of C reachable by a trace t, c has a φ image reachable in A by t. C* **implements** *A if the set of traces of C is a subset of the set of traces of A.*

Theorem 2 (Sufficient conditions) *Refinement is a sufficient condition for simulation. Simulation is a sufficient condition for implementation.*

The validation of always true properties relies on the following theorems which state two equivalent sufficient conditions; the first one is expressed at the abstract level and the second one at the concrete level.

Theorem 3 (Always true properties preservation) *Given two transition systems C and A such that C simulates A through φ. If the abstraction of P by φ ($\overline{\varphi(\overline{P})}$) is always true over A, then P is always true over C. More formally, we have the two equivalent formulas:*

$$\frac{C simulates_{\varphi} A \quad A \vdash \Box\overline{\varphi(\overline{P})}}{C \vdash \Box P} \qquad \frac{C simulates_{\varphi} A \quad \varphi^{-1}(\mathcal{A}cc(A)) \Rightarrow P}{C \vdash \Box P} \tag{1}$$

3.2 Abstraction

The abstraction technique aims at verifying the properties of a transition system through the reduction of its state space: the original system is said concrete and the reduced one is said abstract. In fact, it can be considered as the reverse of the refinement technique where we derive a concrete system from an abstract one. The abstraction technique synthesizes an abstract system from the concrete one. Relevant properties are studied over the abstract system and inherited by the concrete one. This method has been used by [CGL94] to reduce the state explosion resulting from an exhaustive search over a finite state space. It is also used by [BLO98] for analyzing infinite state space systems. Given an abstraction function, they propose heuristics for the construction of the abstract system whereas in our approach the abstract system is built in an automatic way by the FMona tool.

In the following, we recall the basic definitions and results.

Definition 7 (Abstraction) *Let $C = (E_c, I_c, L, \rightarrow_c)$ be a transition system, E_a a set of so called abstract states and φ a relation over $E_c \times E_a$. The abstraction of C through φ is the transition system $(E_a, I_a, L, \rightarrow_a)$ where I_a is the image by φ of I_c and for each label l, $\stackrel{l}{\rightarrow}_a$ is the set of images through φ of the pairs connected by $\stackrel{l}{\rightarrow}_c$.*

It follows that the abstraction of a transition can be expressed by the generic and higher order FMona macro:

```
pred tr_a(type State_c, type State_a, pred(var State_c c, c') tr_c,
          pred(var State_c c, var State_a a) φ, var State_a a,a') =
  ex State_c c,c': φ(c,a) & φ(c',a') & tr_c(c,c');
```

Theorem 4 (Refinements and abstractions) *Let φ a total relation between two state spaces E_c and E_a. A transition system over E_c refines its abstraction through φ.*

4 Automatic Validation through Abstraction

Let us recall that the abstraction introduced in the paragraph 3.2 consists in defining a finite reduced system starting from a concrete system, a finite state

space and a total relation known as the abstraction. It is the reverse of a refinement as the starting system is the concrete one. Moreover, for a refinement, the two transition systems are provided.

One generally considers two approaches for the expression of properties to be validated:

- the first one is based on states: a property is expressed as a temporal logic predicate over the state space (more exactly over the state variables defining the interface of the system).
- the second one is based on transitions: we are interested in a simulation property between a concrete system and an abstract system (also called the reference system).

In the following, we illustrate the abstraction method by considering the two approaches: to validate the mutual exclusion implemented by the bakery algorithm, we adopt a state based approach; to validate some classes of cache coherency protocols, we adopt the transition based approach.

4.1 Specification and Synthesis of the Abstract Transition System

The formulas defining the transitions of the abstract transition system, as introduced in definition 7, are quantified over the domain of the concrete space and thus are not propositional. It follows that the existence of a transition between two abstract states is not necessarily decidable. In order to be in a decidable context, we consider the framework of the WS1S logic for expressing:

- the infinite or parameterized concrete state space (with first and second order variables),
- the transitions of the concrete system,
- the abstraction relation between the concrete and abstract systems.

Thus, according to definition 7, an abstract transition is a WS1S predicate over two abstract variables. Consequently, the properties of an abstract system can

- either be studied within the WS1S logic, which assumes the encoding of temporal logic operators in this formalism [Tho90]. Then, safety properties (which are inherited) can be expressed and automatically decided [ABP97].
- or be studied using a model checker as SMV [BCMD90]. Such a use requires the synthesis of a propositional expression of abstract transitions. Since this alternative resorts to a dedicated tool, it seems to be more efficient on the considered examples. The synthesis of the propositional expression can be performed through two methods:
  - by an exhaustive exploration of the abstract state space: the existence of a transition between two abstract states is determined by the validity of a closed WS1S formula (definition 7).

- by a symbolic reduction to propositional logic of an abstract transition expressed in WS1S. This reduction is possible within the Mona tool where the set of solutions of a predicate with free variables is encoded by an transition system whose transitions are labeled by propositional formulas. In this context the free variables are booleans and the automaton generated by Mona contains a unique transition.

Let us illustrate the synthesis of the finite transition system over a trivial example. The state space of the concrete transition system is a finite set of naturals of unknown size, the concrete transition is defined by the predicate $\mathrm{Init}_c(S) = (S = \{0\})$ and the transition $\mathrm{Tr}_c(S, S') = \exists_1 x, y : y \in S \wedge S' = S \backslash \{y\} \cup \{x\}$. As an abstraction, we consider a state reduced to a unique boolean b, and the abstraction relation $\varphi(S, b) = (b \Leftrightarrow \forall_1 x, y : x \in S \wedge y \in S \Rightarrow x = y)$. The abstract transition system is defined by the predicate $\mathrm{Init}_a(b) = \exists_2 S : \mathrm{Init}_c(S) \wedge \varphi(S, b)$ and the transition $\mathrm{Tr}_a(b, b') = \exists_2 S, S' : \varphi(S, b) \wedge \mathrm{Tr}_c(S, S') \wedge \varphi(S', b')$. The Mona tool can automatically simplify the previous monadic second order logic formulas to propositional formulas. Actually, we have synthesized the finite transition system defined by: $\mathrm{Init}_a(b) = b$ and $\mathrm{Tr}_a(b, b') = b \Rightarrow b'$.

4.2 Verification of Always True Properties

Given a total abstraction relation φ, the validation of a formula on the concrete transition system relies on the deduction rules of theorem 3. In this theorem, the abstract system is the abstraction through φ of the concrete system. By theorem 4, φ being total, the concrete system simulates its abstraction. Then, the rules (1) can be simplified as follows:

$$\frac{\mathrm{Abs}_\varphi(C) \vdash \Box\overline{\varphi(\overline{P})}}{C \vdash \Box P} \qquad \frac{\varphi^{-1}(\mathcal{A}cc(\mathrm{Abs}_\varphi(C))) \Rightarrow P}{C \vdash \Box P}$$

Given these two rules, the validity of P is derived form the properties of the abstract transition system. Let us recall that the synthesis can be expressed in WS1S. Consequently, the two preceding rules give two decidable sufficient conditions. To summarize, given a user defined abstraction relation, we have two automatic verification methods:

Method 1

1. Construction of the abstract transition system.
2. Construction of the reachable states $\mathcal{A}cc$ of the abstract transition system.
3. Definition of a superset of the reachable states of the concrete transition system as the inverse image by φ of $\mathcal{A}cc$.
4. At the concrete level, we show that $\varphi^{-1}(\mathcal{A}cc) \Rightarrow P$.

Note that the term $\varphi^{-1}(\mathcal{A}cc)$ can be interpreted as a lemma automatically proven through an exhaustive exploration of the abstract state space. This lemma helps in the validation of P.

Method 2

1. Construction of the abstract transition system.
2. Computation of the abstraction of P: $\overline{\varphi(\overline{P})}$.
3. Verification of this abstraction over the reachable states of the abstract transition system.

Since the abstract state space is finite, this verification can be performed by a model checker like SMV. This second method uses a dedicated tool for the verification of always true properties.

4.3 Validation of Simulation Relations

The goal is to determine the existence of a simulation relation between a concrete system and a reference system. For that, we provide a projection of the concrete space to the state space of reference and a total abstraction relation. Then, we show that the restriction of the concrete system to some superset of its reachable states refines the reference system.

Thus, given a user defined projection π and a total abstraction relation φ_{abs}, the construction of the simulation relation automatically proceeds according to the following steps:

1. Construction of the abstract transition system (WS1S): the concrete system refines the abstract system.
2. Construction of the accessible states $\mathcal{A}cc$ of the abstract transition system (WS1S).
3. Restriction of the concrete transition system to the reverse image of the accessible states of its abstraction.
4. Validation of the refinement between the reduced concrete transition system and the reference system (WS1S formula).

4.4 Abstraction Heuristics

To reduce parameterized or infinite data structures, we apply the heuristics presented in the following paragraph. The abstraction function associates with a parameterized or infinite type a finite approximation. We consider three classes of types: integer types, arrays with opaque[1] index and finite values, arrays with natural index and finite value, and arrays with opaque index and values. We have considered the following abstractions which can be expressed in WS1S:

1. We associate with natural state space variables a family of boolean variables coding the comparisons between these variables or the variables and the constants of the problem. Notice that this heuristics is in fact the one used in an automatic way by [Les97].

[1] An opaque type supports only the assignment and comparison operations.

2. We associate with an array with opaque index and finite values within the set $\{x_1, \ldots, x_n\}$ a family of bounded counters $\{c_1, \ldots, c_n\}$ with value in the set $\{0, \ldots, k, +\}$. A counter C_i indicates the number of indexes with value x_i. This enumeration being bounded by k, a higher number of occurrences is denoted $+$.
3. We associate with an array t with natural index and finite values the number of alternations $t(i) \neq t(i-1)$, counted in the finite set $\{0, \ldots, k, +\}$.
4. We associate with an array with opaque index and values the number of different values in the array, counted in the finite set $\{0, \ldots, k, +\}$.

Notice that the heuristics we present relate to the construction of an abstraction function, the abstract transition system being automatically built and validated by the FMona tool for the considered class of problems. On the other hand, [BLO98] supposes the existence of an abstraction function and proposes heuristics for the construction of the abstract system.

5 Application to the Bakery Mutual Exclusion Protocol

The transition system of the Bakery mutual exclusion algorithm over two processes is described in FMona by the following code. Its state space contains two finite-typed variables `pc1` and `pc2` and two naturals `y1` and `y2`. The state space is thus infinite.

```
type PC = {out,req,mutex}; type Sys = record{pc1,pc2: PC; y1,y2: nat;};

pred Init(var Sys st)= st.pc1=out & st.pc2=out & st.y1=0 & st.y2=0;
pred req1(var Sys st,st')= st.pc1=out &
      st'=st with {pc1:=req; y1:=st.y2+1;};
pred req2(var Sys st,st')= st.pc2=out &
      st'=st with {pc2:=req; y2:=st.y1+1;};
pred enter1(var Sys st,st')=
  st.pc1=req & (st.y2=0 | st.y1≤st.y2) & st'=st with{pc1:=mutex;};
pred enter2(var Sys st,st')=
  st.pc2=req & (st.y1=0 | st.y2<st.y1) & st'=st with{pc2:=mutex;};
pred leave1(var Sys st,st')= st.pc1=mutex &
      st'=st with {pc1:=out; y1:=0;};
pred leave2(var Sys st,st')= st.pc2=mutex &
      st'=st with {pc2:=out; y2:=0;};
```

We notice that it is not possible to bound the natural variables `y1` and `y2`. Actually, when considering the execution sequence $r_1(e_1r_2l_1e_2r_1l_2)^*$, the sequence of `(y1,y2)` values is $(0,0)\xrightarrow{r_1(e_1r_2l_1e_2r_1l_2)^k}(2k+1,0)$ where r, e, l respectively abbreviate `req,enter,leave`.

We seek to establish that the mutual exclusion property is always true:

```
pred excl(var Sys st) = ∼(st.pc1 = mutex & st.pc2 = mutex);
```

The predicate `excl` is *not* an invariant. Actually, it is not preserved by `enter1`. However, as shown in the following, it is always true, i.e., true in any reachable state. In order to get a finite state space, the fields `pc1` and `pc2` being finite, it is enough to abstract the fields $y1$ and $y2$. We consider the abstraction which consists in coding the comparisons between these two fields. Then, we get the abstract system `Sys_a` and the abstraction relation φ defined as follows:

```
type Cmp = {z1z2,z1,z2,inf12,inf21,eq};
type Sys_a = record{pc1,pc2: PC; y1y2: Cmp;};
pred φ(var Sys st, var Sys_a a) =
  a.pc1 = st.pc1 & a.pc2 = st.pc2 &
    if st.y1 = 0 & st.y2 = 0 then a.y1y2 = z1z2
    elsif st.y1 = 0 then a.y1y2 = z1
    elsif st.y2 = 0 then a.y1y2 = z2
    elsif st.y1 < st.y2 then a.y1y2 = inf12
    elsif st.y2 < st.y1 then a.y1y2 = inf21
    else a.y1y2 = eq endif;
```

The relation φ being total, according to theorem 4, the `Bakery` transition system refines its abstraction through φ. The abstraction through φ of the `Bakery` system, computed by FMona, is a finite state system of which transitions can be expressed using WS1S and reduced to propositional logic using Mona. Consequently, the computation of the reachable states is possible. Mona validates that any state reachable by this transition system cannot be the image of a state where two processes would not be in mutual exclusion. The following logical formula (extracted from the transition system produced by Mona) encodes the set of reachable states of the abstract system:

```
  st1 = idle   ∧ st2 = idle   ∧ cmp = z1z2  (* 0 = y1 = y2 *)
∨ st1 = idle   ∧ st2 = req    ∧ cmp = z2    (* 0 = y1 < y2 *)
∨ st1 = idle   ∧ st2 = mutex  ∧ cmp = z2    (* 0 = y1 < y2 *)
∨ st1 = req    ∧ st2 = idle   ∧ cmp = z1    (* 0 = y2 < y1 *)
∨ st1 = req    ∧ st2 = req    ∧ cmp = inf12 (* 0 < y1 < y2 *)
∨ st1 = req    ∧ st2 = req    ∧ cmp = inf21 (* 0 < y2 < y1 *)
∨ st1 = req    ∧ st2 = mutex  ∧ cmp = inf21 (* 0 < y2 < y1 *)
∨ st1 = mutex  ∧ st2 = idle   ∧ cmp = z1    (* 0 = y2 < y1 *)
∨ st1 = mutex  ∧ st2 = req    ∧ cmp = inf12 (* 0 < y1 < y2 *)
```

According to theorems 2 and 3, it follows that the mutual exclusion predicate `excl` is always true in the `Bakery` system.

Note that the validation of the abstract system can also be achieved by SMV after reducing abstract transitions to propositional logic. Such a reduction can be performed by Mona. The following FMona predicate defines the transitions of the abstract system according to the abstraction relation φ and the concrete transition relation `Trans`[2](`tr_a` has been defined in section (3.1)).

```
pred bakery_tr_a(var Sys_a a,a') = tr_a(Trans,φ,a,a');
```

[2] FMona automatically synthesizes type parameters.

The Mona tool produces a transition system representing the reduction of the relation `Trans_a`. Since the transitions of this system are labeled by *propositional formulas* relating the arguments of `Trans_a`, a SMV expression of the abstract transition can be extracted and analyzed.

It is interesting to note that since the abstraction did not modify the interface fields `pc1` and `pc2`, it also defines a mutual exclusion algorithm over a finite state space similar to Peterson's algorithm [PS85] comprising apart from the fields `pc1` and `pc2` a finite field `y1y2` that can be encoded by three booleans.

6 Application to Cache Coherency Protocols

In this section we consider cache coherency protocols for shared memory multiprocessor machines [Ste90]. In such an architecture, each processor has a private cache where it stores copies of the shared memory data. The protocol ensures coherence between the multiple copies of a same data. Informally, atomic coherency means that the processors behave as if data were not duplicated. For this purpose, a *given* finite state is associated with each copy. For instance, in the Illinois protocol [AB86], the `State` type can be encoded as the enumeration {`Inv, Excl, Shared, Dirty`} where `Inv` marks an invalid copy, `Excl` an exclusively held copy, `Shared` a shared copy, and `Dirty` a modified copy. The data structures handled by such protocols are described by the following declarations, where `Proc` and `Word` are two opaque types. In the following, we denote by *control* the array `Ctr` and by *data* the array `C` and the memory `M`.

```
var nat NProc;
type Proc  = ... NProc; type State = {Inv, Shared, Excl, Dirty};
type Word  = record{b1:bool; b2:bool;}; #   see section 6.2
type Sys   =
record{Ctr:array Proc of State; C:array Proc of Word; M:Word;};
```

This protocol is parameterized by the number of processors, defined by the type `Proc`, and by the size of the words, defined by the type `Word`. The validity of such a protocol is expressed either by the satisfaction of an *always true* property, or by the simulation of a canonic atomic memory [BF99b]. The canonic atomic transition system (6.1) includes a register and the operations for reading and writing in this register.

6.1 Specification of the Transition System

The transition system defining atomic protocols establishes the interface of a memory access protocol comprising the `Init` predicate and three transitions: the reading of a value `v` by the processor `p`, the writing of a value `v` by the processor `p`. and the Skip transition which does nothing.

```
type Word = record{b1:bool; b2:bool;}; #   see section 6.2
pred Init_atomic(var Word r) = true;
```

```
pred Read_atomic(var Word v, var Word r,r') = (v = r) & (r' = r);
pred Write_atomic(var Word v, var Word r,r') = (r' = v);
pred Skip(var Word r,r') = (r' = r);
```

The transition system of an atomic coherency protocol must implement the preceding reference system.

6.2 Elimination of the Opaque Type Word

The expression of the abstract system in the WS1S logic requires to fix the size of `Word`. For this purpose, we use the result established in [BF99a]. It concerns the reduction of formulas where the comparisons between elements of some opaque type `D` only occurs in terms of the form $f(x) = g(x)$ or $f(x) = k$ where f, g and k are identically quantified at the top level of the formula. Let n be the number of such functions or variables. The validity of the formula over a domain D of size $\geq n$ is equivalent to its validity over a domain of size n.

Here, the opaque type `Word` must be eliminated from formulas expressing data abstraction and transition refinement. In both cases, variables of type `Word` are either global arrays or global variables (the array of words `C`, the memory `M` and the value to be read or written `d`). According to our result, we can restrict the type `Word` to a domain of size three. Consequently, we have chosen a record with two booleans.

6.3 Abstraction and Reduction

In this paragraph, we show how the abstract transition system is actually synthesized. The abstraction relation φ^{ca} can be structured as the conjunction of a control abstraction and a data abstraction.

Control abstraction associates to the (parameterized size) array `Ctr` a fixed family of counters, through the application of the second heuristic of §4.4. Thus, we introduce the abstraction data type as a record with three bounded counters, one for shared caches, one for exclusive caches and one for dirty caches. The abstraction relation `φ_ctr` maps the Illinois control array to these counters.

```
#   array abstraction through bounded counters
type B_Cpt = {Zero, One, More};
pred abstr_p(type State, pred(var State s) p, var B_Cpt f) =
 if all State s: ~p(s) then f=Zero
 elsif ex State s1: p(s1) & all State s: s≠s1⇒~p(s) then f=One
 else f=More endif;
type Sys_a_ctr = #   abstraction of Illinois control
       record{c_Shared: B_Cpt; c_Excl: B_Cpt; c_Dirty: B_Cpt;};
pred φ_ctr(var Ctr s, var Sys_a_ctr a) =
 abstr_p(pred(var Proc p):s[p]=Shared,a.c_Shared) &
 abstr_p(pred(var Proc p):s[p]=Excl,a.c_Excl) &
 abstr_p(pred(var Proc p):s[p]=Dirty,a.c_Dirty);
```

Data abstraction associates to the array C and the memory M the cardinal, counted over $\{1, 2, +\}$, of the set of valid data in the system.

```
type Sys_a_data = {One_d,Two_d,More_d};
pred φ_data(var Sys s, var Sys_a_data c) =
  if (all Proc p: s.Ctr[p]≠Inv ⇒ s.C[p]=s.M) then c = One_d
  elsif (ex Proc p,q: s.Ctr[p]≠Inv & s.Ctr[q]≠Inv & s.C[p]≠s.C[q]
            & s.C[p] ≠ s.M & s.C[q] ≠ s.M) then c = More_d
  else c = Two_d endif;
```

Consequently, the abstract domain consists in a fixed family of bounded counters $(B_i)_{i \leq K}$, defined here by the record Sys_a. The abstraction relation φ is then defined as the conjunction of the abstractions of the control and data parts. Note that for efficiency purposes, this conjunction has been restricted to the reachable states of the control part. Otherwise, the complexity of the computation is too high and the resulting formula cannot be decided by Mona.

```
type Sys_a = record{ ctr_a: Sys_a_ctr; data_a: Sys_a_data;};
pred φ(var Sys s, var Sys_a a) =
  is_abs_ctr_Acc(a.ctr_a) & φ_ctr(s.Ctr, a.ctr_a) & φ_data(s,a.data_a);
```

6.4 Validation of the Simulation Relation

In this section, we show the simulation of the canonic transition system, reduced to one register R (see §6.1) by the abstract one. Here, we are interested in the refinement approach. With this intention, one introduces a projection between the state space of the Illinois protocol and the space reduced to one register. The relation ill_atm expresses that the contents of the register R is either equal to the memory if the caches are all invalid, or equal to the common value of the non-invalid caches.

```
pred ill_atm(var Sys s, var Word r) =
 ((all Proc p:s.Ctr[p]=Inv)⇒s.M=r) & all Proc p:s.Ctr[p]≠Inv⇒s.C[p]=r;
```

It should be noted that this relation does not make it possible to define a refinement with the atomic model. In fact, the Illinois transition system must be restricted to some superset, called is_Acc of its reachable states. For this purpose, we consider the inverse image through φ of the reachable states of its abstraction.

```
pred is_Acc(var Sys c) = ex Sys_a a: φ(c,a) & is_abs_Acc(a);
```

Then, the validity of the Illinois protocol can be stated: we show that the relation ill_atm establishes a refinement between the reduced concrete transition system and the atomic one. More precisely, such a refinement is specified by the following conjunction which is transformed by FMona to pure Mona code and validated by Mona.

```
# Illinois validation
refinement_init(restrict_init(Init,is_Acc),Init_atomic,ill_atm) &
(all Word v: refinement_tr(restrict_tr(Read_Miss(v),is_Acc)
                           ,Read_atomic(v),ill_atm)) &
(all Word v: refinement_tr(restrict_tr(Write_Miss(v),is_Acc)
                           ,Write_atomic(v),ill_atm)) &
refinement_tr(restrict_tr(Flush,is_Acc),Skip,ill_atm);
```

7 Other Validation Techniques

Apart from abstraction techniques, we have also expressed in FMona iteration based techniques. It should be stressed that we have applied them on parameterized systems. However, since the state spaces of the considered problems are not fixed, we have no decidability results. It follows that the user must provide an iteration bound to apply the proposed macros. Backward iteration techniques have been successfully applied to mutual exclusion algorithms on ring networks [Mar85]. However, forward and backward analysis fail for some well known problems (e.g. termination detection [DFvG83], dining philosophers, Szymanski mutual exclusion protocol [Szy90]). To overcome such problems, convergence acceleration techniques have been proposed [ABJN99]. The basic idea consists in approximating the transitive closure of the transition relation. We can express such techniques in FMona. In a forthcoming paper [BF00], we present some new acceleration techniques and their expression in FMona. These techniques allowed us to validate the above-mentioned problems.

8 Conclusion

In this paper, we have illustrated the use of FMona to express the abstraction technique and its applications. Thanks to the higher order features of FMona, the validation steps of the method could be expressed in a generic way and instantiated on specific problems.

FMona has also been used to express other well known methods such as iterative methods applied to parameterized systems. It should be stressed that these methods have been generically defined and have been applied to well know problems (mutual exclusion on parameterized rings, parameterized multiprocessor memory protocols, infinite space bakery algorithm).

For efficiency reasons, we have also connected FMona to propositional solvers. We also plan to consider the validation of the methods themselves and how to integrate them smoothly into FMona.

References

AB86. J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986. 214

ABJN99. Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling Global Conditions in Parameterized System Verification. In *Proc. 11th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145. Springer Verlag, 1999. 217

ABP97. A. Ayari, D. Basin, and A. Podelski. Lisa: A specification language based on ws2s. In *11th International Conference of the European Association for Computer Science Logic (CSL '97)*, volume 1414 of *LNCS*, pages 18 – 34. Springer-Verlag, 1997. 209

Arn92. A. Arnold. *Systèmes de transitions finis et sémantiques des processus communicants*. Etudes et recherches en informatique. MASSON, 1992. 206

BBC+97. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. 204

BCMD90. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10E20 states and beyond. In *5th Symposium on Logic in Computer Science*, June 1990. 204, 209

BF99a. J.-P. Bodeveix and M. Filali. Reduction and quantifier elimination techniques for program validation. *Formal Methods in System Design*, to appear, 1999. 215

BF99b. J.-P. Bodeveix and M. Filali. The FMONA tool. Technical Report http://www.irit.fr/ACTIVITES/EQ_COS/MF/FMONA, IRIT, may 1999. 206, 214

BF00. J.-P. Bodeveix and M. Filali. Experimenting acceleration methods for the validation of infinite state systems. In Dr. Pao-Ann Hsiung, editor, *International Workshop on Distributed System Validation and Verification*, Institute of Information Science, Academia Sinica, Taiwan, R.O.C., april 2000. to appear. 217

BLO98. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionnaly and automatically. In *Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, BC, Canada, june 1998. Springer-Verlag. http://www.csl.sri.com/∼owre/cav98.html. 208, 212

CGL94. E.M. Clarke, O. Grumber, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, september 1994. 208

DFvG83. E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, june 1983. 217

GM94. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1994. 204

HJJ+95. J.G. Henriksen, J.L. Jensen, M.S. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A.B. Sandholm. Mona: Monadic second-order logic in practice. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–73, Aarhus, May 1995. 204, 206

Hol91. G.J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, 1991. 204

Les97. Lessens, D. and Saïdi, H. Abstraction of parameterized networks. *Electronic notes in theoretical computer science*, 9:12, 1997. http://www.elsevier.nl/locate/entcs/volume9.html. 211

Mar85. A.J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5(3):265–276, October 1985. 217

ORS92. S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607, 1992. 204

PS85. J.L. Peterson and A. Silberschatz. *Operating system concepts*. Addison-Wesley, 1985. 214

Ste90. P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):11–25, June 1990. 214

Szy90. B.K. Szymanski. Mutual exclusion revisited. In *fifth Jerusalem conference on information technology*, pages 110–117. IEEE Computer Society Press, 1990. 217

Tho90. W. Thomas. Automata on infinite objects. In J.v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–192. MIT Press, 1990. 205, 209