MCMT: A Tutorial

Silvio Ghilardi[1] and Silvio Ranise[2]

[1] Dipartimento di Informatica, Università degli Studi di Milano (Italia)
[2] Dipartimento di Informatica, Università di Verona (Italia)

Abstract. The main purpose of this paper is to explain how to use MCMT, an infinite state model checker for checking the safety of systems whose state variables are arrays, called array-based systems. MCMT is founded on a declarative framework based on many-sorted first-order logic extended with theories for the specification of sets of states and transitions of array-based systems. Imperative programs, parametrised, timed, and distributed systems can all be formalized as array-based systems. We use a parametrised version of a real-time mutual exclusion protocol to illustrate the main features of MCMT.

Remark. This tutorial *was written for version 1.0*, hence it does not cover acceleration and abstraction features (these features are essential to handle certain classes of problems, e.g. software model checking problems). However, the old tutorial has been modified by the first author to cover some novelties (at specification level) offered by version 2.5.

1 Introduction

MCMT [GR10] is an infinite state model checker for checking the safety of systems manipulating array variables, called array-based systems. One major goal of MCMT is to provide a declarative and flexible verification tool for array-based systems such as parametrised, timed, and distributed systems or imperative algorithms (e.g., sorting programs).

The actual version of MCMT only supports the verification of safety properties although the theoretical framework underlying the system permits also the verification of a sub-set of liveness properties. A safety problem is specified by a formula I of first-order logic characterizing the set of initial states of an array-based system S, a finite set Tr of formulae specifying the transitions of S, a formula U for the set of unsafe states, obtained by negating the safety property we would like the system S to satisfy. The verification of a safety property is thus reduced to repeatedly compute the pre-images of U until a fixed-point is reached or the intersection with the set of initial states is non-empty. If a fixed-point has been reached and the intersection with I is empty, then we conclude that S is safe with respect to the safety property under consideration; otherwise, S is unsafe. In the second case, MCMT also returns a sequence of transitions that leads the system from an initial to an unsafe state. The repeated computation of pre-images and the checks for fixed-point and empty intersection with the initial states is known as backward reachability procedure. To mechanize this

procedure, MCMT puts some constraints on the format of I, Tr, and U so that pre-image computation is closed with respect to a certain class of first-order formulae and both fixed-point and empty intersection checks can be reduced to decidable Satisfiability Modulo Theories (SMT) problems for formulae containing universal quantifiers. At a very high-level of abstraction, the architecture of MCMT is client-server where the client implements the backward reachability procedure while the server is an SMT solver (in our case, Yices[3]) which is invoked to solve the SMT problems encoding fixed-point and empty intersection checks.

1.1 Pointers to the literature

The formal framework underlying MCMT is described in [GNRZ08]. In particular, it is described the backward reachability procedure for checking the safety of infinite state systems whose states are declaratively specified by first-order formulae. In this context, checks for fixed-point and safety are reduced to Satisfiability Modulo Theories (SMT) problems of formulae containing (universal) quantifiers. In [GRV08,GR09b], some heuristics are described to reduce the number of quantified variables and—most importantly—of instances while preserving the completeness of SMT solving. The invariant synthesis techniques used in the tool are described in [GR09a]. Finally, for a complete description of the functionalities offered by MCMT and its usage, the reader can consult the on-line User Manual available in the form of a Technical Report at the address `http://homes.dsi.unimi.it/~ghilardi/mcmt/GhRa-RI_MCMT.pdf`.

1.2 Plan of the paper

Section 2 introduces the abstract input language of the tool. Section 3 briefly describes the abstract calculus underlying the main loop of MCMT. Section 4 describes a parametrised version of Fischer's protocol for mutual exclusion (Section 4.1), illustrates its formalization in the abstract syntax (Section 4.2), and then in the concrete syntax (Section 4.3). Finally (Section 4.4), various ways of invoking MCMT on the previously described safety problem and a description of its outputs is offered.

2 MCMT Abstract Input Language

The input language of MCMT can be seen as a parametrised extension of the one used by UCLID (`http://www.cs.cmu.edu/~uclid`). Formally, it is a sub-set of multi-sorted first-order logic, extended with the ternary expression constructor "if-then-else" (which is standard in the SMT-LIB format). The concrete syntax is fully described in the on-line User Manual and is illustrated on an example in the following Section.

[3] `http://yices.csl.sri.com`

Sorts. We use the following distinguished sorts: Ind for indexes, $Elem_1, ..., Elem_m$ for elements of arrays, and $Arr_1, ..., Arr_m$ for array variables (where Arr_k corresponds to arrays of elements of sort $Elem_k$, for $k = 1, ..., m$).

Theories. We assume that the mono-sorted theories T_I and T_{E_k} are given over the sorts Ind and $Elem_k$, respectively, for $k = 1, ..., m$. The three-sorted theories $A_I^{E_k}$ are obtained as the combination of the theories T_I and T_{E_k} for each $k = 1, ..., m$ by adding the sort Arr_k to Ind and $Elem_k$, by taking the union of the symbols of T_I and T_{E_k}, and by adding the binary symbol $\_[\_]_k : Arr_k \times Ind \rightarrow Elem_k$ for reading the content of an array at a given index (the subscript k is omitted if clear from the context). Finally, we let $A_I^E := \bigcup_{k=1}^m A_I^{E_k}$.

Formats of formulae. We use two classes of formulae to describe sets of states: $\forall \underline{i}.\phi(\underline{i}, \underline{a})$ and $\exists \underline{i}.\phi(\underline{i}, \underline{a})$, where $\underline{i}$ is a tuple of variables of sort Ind, $\underline{a}$ is a tuple of length m of array variables of sorts $Arr_1, ..., Arr_m$, and ϕ is quantifier-free formula containing at most the variables in $\underline{i} \cup \underline{a}$ as free variables. The former are called $\forall^I$-formulae and the latter $\exists^I$-formulae. An $\exists^I$-formula $\exists \underline{i}.\phi$ is *primitive* when ϕ is a conjunction of literals; it is *differentiated* when it is primitive and ϕ contains as a conjunct the variable distinction $i_k \neq i_l$ for each $1 \leq k < l \leq n$ (where n is the length of the tuple $\underline{i}$). By applying simple logical manipulations, it is always possible to transform any $\exists^I$-formula into a disjunction of primitive differentiated ones. To specify transitions, we use a particular class of formulae (called *transition formulae*) corresponding to a generalization of the usual notion of guarded assignment system:

$$\exists i_1, i_2, e. \left(G(i_1, i_2, e, \underline{a}) \ \wedge \ \bigwedge_{k=1}^{m} \forall j.\, a'_k[j] = \mathit{Upd}(j, i_1, i_2, e, \underline{a}) \right),$$

where i_1, i_2 are variables of sort Ind (having at most two existentially quantified variables is not too restrictive since many disparate systems can be formalized in this format as shown by the experiments available on-line), e is a variable of sort $Elem_k$ (for some $k = 1..., m$), $\underline{a}$ is a tuple of array state variables, a_k (in $\underline{a}$) is the actual value of a state variable and a'_k is its value after the execution of the transition (for $k = 1, ..., m$), G is a conjunction of literals (called the *guard*), and Upd is a function defined by cases, i.e. by suitably nested if-then-else expressions whose conditionals are again conjunctions of literals. The format for transition formulae above—because of the presence of the existentially quantified variable e over data values—is the first significant amelioration of the actual version of MCMT as it allows one to specify classes of systems which were not previously accepted by the tool such as real time systems or those with non-deterministic updates. Notice that the theory T_{E_k} over the sort $Elem_k$ of the variable e must be Linear Arithmetic (over the integers or the reals). This limitation allows us to maintain the closure of the class of $\exists^I$-formulae under pre-image computation by exploiting quantifier elimination (implemented only in the latest version of MCMT).

Safety problem. Let I be a $\forall^I$-formula describing the set of initial states, Tr a finite set of transition formulae, and U an $\exists^I$-formula for the set of unsafe states. The *safety problem* solved by MCMT consists in establishing whether there

$$\frac{K \quad [K \text{ is primitive differentiated}]}{\mathsf{Pre}(\tau_1, K) \mid \cdots \mid \mathsf{Pre}(\tau_m, K)}\ \mathsf{PreImg} \qquad \frac{K}{K_1 \mid \cdots \mid K_n}\ \mathsf{Beta}$$

$$\frac{K \quad [K \text{ is } A_I^E\text{-unsatisfiable}]}{\times}\ \mathsf{NotAppl} \qquad \frac{K \quad [I \wedge K \text{ is } A_I^E\text{-satisfiable}]}{\mathsf{UnSafe}}\ \mathsf{Safety}$$

$$\frac{K \quad [K \wedge \bigwedge\{\neg K' | K' \preceq K\} \text{ is } A_I^E\text{-unsatisfiable}]}{\times}\ \mathsf{FixPoint}$$

Fig. 1. The calculus underlying MCMT

exists an $n \geq 0$ such that the formula

$$I(\underline{a}^0) \wedge \tau(\underline{a}^0, \underline{a}^1) \wedge \cdots \wedge \tau(\underline{a}^{n-1}, \underline{a}^n) \wedge U(\underline{a}^n) \tag{1}$$

is A_I^E-satisfiable, where $\underline{a}^h = a_1^h, ..., a_m^h$ for $h = 0, ..., n$, and $\tau := \bigvee_{\tau_i \in T_r} \tau_i$. If there is no such n, then the system is *safe* (w.r.t. U); otherwise, it is said to be *unsafe* since the A_I^E-satisfiability of (1) implies the existence of a run (of length n) leading the system from a state in I to a state in U.

3 MCMT Backward reachability

MCMT implements backward reachability to solve safety problems. For $n \geq 0$, the *n-pre-image* of an $\exists^I$-formula $K(\underline{a})$ is $Pre^0(\tau, K) := K$ and $Pre^{n+1}(\tau, K) := Pre(\tau, Pre^n(\tau, K))$, where $Pre(\tau, K) := \exists \underline{a}'.(\tau(\underline{a}, \underline{a}') \wedge K(\underline{a}'))$. It is easy to show [GNRZ08] that the class of $\exists^I$-formulae is closed under pre-image computation under the assumption that T_{E_k} admits elimination of quantifiers (if an existentially quantified variable of sort $Elem_k$ occurs in a transition formula). The formula $BR^n(\tau, U) := \bigvee_{i=0}^n Pre^i(\tau, U)$ represents the set of states which are backward reachable from the states in U in at most $n \geq 0$ steps. So, backward reachability consists of computing $BR^n(\tau, U)$ for increasing values of n and checking whether $BR^n(\tau, U) \wedge I$ is A_I^E-satisfiable or $\neg(BR^n(\tau, U) \rightarrow BR^{n-1}(\tau, U))$ is A_I^E-unsatisfiable. In the first case (*safety* test), one concludes the unsafety of the system while in the second (*fixed-point* test), it is possible to stop computing pre-images as no new states can be reached and, if the safety test has been passed, one can infer the safety of the system.

Figure 1 introduces the Tableaux-like calculus used by MCMT to implement backward reachability [GR09a]. We initialize the tableau with the $\exists^I$-formula $U(a)$ representing the set of unsafe states. The computation of the pre-image is realized by applying rule PreImg (we use square brackets to indicate the applicability condition of a rule), where $\mathsf{Pre}(\tau_h, K)$ computes the $\exists^I$-formula which is logically equivalent to $Pre(\tau_h, K)$. Since the $\exists^I$-formulae labeling the consequents of the rule PreImg may not be primitive and differentiated (because of nested if-then-else expressions and incompleteness of variable distinction), we need to apply the Beta rule to an $\exists^I$-formula so as to eliminate the conditionals by case-splitting and derive $K_1, \ldots, K_n$ primitive differentiated $\exists^I$-formulae whose disjunction is A_I^E-equivalent to K. By repeatedly applying PreImg and Beta, it is

possible to build a tree whose nodes are labelled by $\exists^I$-formulae whose disjunction is equivalent to $BR^n(\tau, U)$ for some $n \geq 0$. Indeed, there is no need to fully expand the tree; it is useless to apply the rule PreImg to a node ν labelled by an A_I^E-unsatisfiable $\exists^I$-formula (rule NotAppl). One can terminate the whole search because of the safety test (rule Safety), in which case one can extract from the branch a *bad trace*, i.e. a sequence of transitions leading the array-based system from a state satisfying I to one satisfying U. A branch can be terminated by the fixed-point test described by rule FixPoint, where $K' \preceq K$ means that K' is a primitive differentiated $\exists^I$-formula labeling a node preceding the node labeling K (nodes can be ordered according to the strategy for expanding the tree).

4 Fisher's Mutual Exclusion Algorithm

To illustrate the concrete input syntax of MCMT and some of its options, we present the verification of the safety of a parametric and real-time mutual exclusion protocol. We consider the variant of Fischer's real-time based synchronization protocol for mutual exclusion described in Chapter 13 ("A Case Study: Fischer's Protocol") of the book *Operational Semantics for Timed Systems: A Non-standard Approach to Uniform Modeling of Timed and Hybrid Systems* by H. Rust (LNCS 3456, Springer, 2005). Among the many available formalizations of the protocol, we choose this one as it was almost precisely in the formal framework underlying MCMT and it was thus easy to use.

Below, after a brief high-level description of the algorithm, we give its formalization in the high-level specification language of the previous Section, present the same formalization in the concrete input language of MCMT, and finally describe how the tool solves the safety problem in the default setting and with some options.

4.1 Informal description

Fischer's real-time based synchronization protocol is meant to ensure mutual exclusion of access to a shared resource via real-time properties of a shared variable v by a set of n processes (for $n \geq 1$). The idea is to use just one shared variable v for coordinating the access to the critical section. The variable can contain a process id or some neutral value, called *noProc*. When a process wants to enter its critical section, it first has to wait until $v = noProc$; then it sets v to its process id, waits some time, and then reads v again. If v has kept the old value, i.e., the id of the process considered, the process may enter its critical section; on leaving the critical section, v is set to *noProc* again. If v has not kept its old value, the attempt has failed and the process must go back and wait again till $v = noProc$. Two time distances are relevant for the correctness of the protocol: the time from ensuring that $v = noProc$ holds up to the moment at which v is set to the process id; this span is required to be smaller than or equal to a value called a. The other time span is that from setting v to the process id to checking it again. This is assumed to be greater than or equal to a value called b. Mutual exclusion is ensured for $b > a$.

State variables: n, v, s, r, w	
I	$n = 0 \wedge v = noProc \wedge \forall i. s[i] = nc$
τ_0	$\exists c \left(c > 0 \wedge n' = n + c \right)$
τ_1	$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = nc \quad \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ rv\ \mathsf{else}\ s[j] \end{array} \right) \end{array} \right)$
τ_2	$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = rv \wedge v = noProc \quad \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ w1\ \mathsf{else}\ s[j] \wedge \\ r'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ n\ \mathsf{else}\ r[j] \end{array} \right) \end{array} \right)$
τ_3	$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = w1 \wedge n + c \leq r[i] + a \quad \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ w2\ \mathsf{else}\ s[j] \wedge \\ v' = i \wedge \\ w'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ n + c\ \mathsf{else}\ w[j] \end{array} \right) \end{array} \right)$
τ_4	$\exists i, c. \left(\begin{array}{l} c > 0 \wedge v \neq i \wedge s[i] = w2 \wedge n \geq w[i] + b \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ rv\ \mathsf{else}\ s[j] \end{array} \right) \end{array} \right)$
τ_5	$\exists i, c. \left(\begin{array}{l} c > 0 \wedge v = i \wedge s[i] = w2 \wedge n \geq w[i] + b \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ cr\ \mathsf{else}\ s[j] \end{array} \right) \end{array} \right)$
τ_6	$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = cr \quad \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ nc\ \mathsf{else}\ s[j] \wedge \\ v' = noProc \end{array} \right) \end{array} \right)$
U	$\exists i_1, i_2. (s[i_1] = cr \wedge s[i_2] = cr \wedge i_1 \neq i_2)$

Fig. 2. The formalization of Fischer's protocol

4.2 Formalization in mcmt: abstract syntax

We assume T_I to be the empty theory, T_{E_1} is the theory of an enumerated datatype with constants $nc, rv, w1, w2, cr$ over the sort $Elem_1$; the other four theories we need are all identical copies of Linear Arithmetic (over reals/integers). We have two shared variables, v and n (for the former see the informal description above, the latter represents time). In addition, we need three (local) variables: $s : Arr_1$, $v : Arr_2$, and $n : Arr_3$.[4] While s stores the states of each process in the system, r and w associate each process with the time point when the variable v has been read and written, respectively. The formulae characterizing the set of initial states, the seven transitions, and the set of unsafe states are depicted in Figure 2. If a state variable α is not mentioned in a transition τ_i, then it is

[4] Local state variables range over standard arrays while global or shared state variables denote single values. These single values are internally tretaed by MCMT as arrays containing the same value in all their cells. Since version 2.5, the user needs not to care about this internal representation and can specify global variables just giving them a name consisting of a string of characters.

assumed to be unchanged and the conjunct $\forall j.\alpha'[j] = \alpha[j]$ must be added to τ_i (if α is global the conjunct $\alpha' = \alpha$ must be added). The set of initial states is given by a $\forall^I$-formula while that of unsafe states is described by an $\exists^I$-formula which is also primitive differentiated.

For Fischer's protocol, the existentially quantified variable c can always be eliminated as T_{E_2} is Linear Arithmetics over the reals, that is well-known to admit elimination of quantifiers. To illustrate, consider the computation of $Pre(\tau_5, U)$, i.e.

$$\exists i, c \left(\begin{array}{l} c > 0 \wedge v = i \wedge s[i] = w2 \wedge n \geq w[i] + b \wedge \\ \forall j. \left(\begin{array}{c} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ cr\ \mathsf{else}\ s[j] \end{array} \right) \end{array} \right) \wedge \exists i_1, i_2. \left(\begin{array}{l} s'[i_1] = cr \wedge \\ s'[i_2] = cr \wedge \\ i_1 \neq i_2 \end{array} \right), \quad (2)$$

where s', n' are also existentially quantified. Without loss of generality, it is possible to instantiate i with i_1 so that after some simple manipulations we derive:[5]

$$\exists i_1, i_2.(s[i_1] = w2 \wedge s[i_2] = cr \wedge v = i_1 \wedge n \geq w[i_1] + b \wedge i_1 \neq i_2),$$

which is an $\exists^I$-formula. To quickly obtain this formula, call it N_1, rewrite the updates as

$$n' = n + c \qquad s' = \lambda j.(\mathsf{if}\ j = i\ \mathsf{then}\ cr\ \mathsf{else}\ s[j])$$

eliminate the existentially quantified variables n', s' by substitution and finally perform β-conversion.

To illustrate the need of quantifier-elimination in Linear Arithmetics, consider to compute $Pre(\tau_3, N_1)$:

$$\exists i, c \left(\begin{array}{l} c > 0 \wedge s[i] = w1 \wedge n + c \leq r[i] + a \qquad \wedge \\ \forall j. \left(\begin{array}{c} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ w2\ \mathsf{else}\ s[j] \wedge \\ v' = i \wedge \\ w'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ n + c\ \mathsf{else}\ w[j] \end{array} \right) \end{array} \right) \wedge$$
$$\exists i_1, i_2.(s'[i_1] = w2 \wedge s'[i_2] = cr \wedge v' = i_1 \wedge n' \geq w'[i_1] + b \wedge i_1 \neq i_2),$$

where all the primed array variables are (implicitly) existentially quantified. As before, it is possible to identify i with i_1 without loss of generality so as to

[5] In MCMT, state formulae are kept in primitive and differentiated form. As a consequence, we must produce three disjuncts corresponding to the cases when the existentially quantified variable i in τ_5 is equal (or distinct) to one (all) of the existentially quantified variables i_1, i_2 in U: one disjunct is obtained by instantiating i to i_1, one by instantiating i to i_2, and the the last by requiring that i be different from both i_1 and i_2 (i.e. $i \neq i_1 \wedge i \neq i_2$). After the fixed-point check, just one of the three disjunct is kept while the the other two are subsumed.

obtain:[6]

$$\exists i_1, i_2, c. \begin{pmatrix} s[i_1] = w1 \wedge s[i_2] = cr \wedge v = i_1 \wedge \\ b \leq 0 \wedge c > 0 \wedge n + c \leq r[i_1] + a \end{pmatrix},$$

which still contains the existential quantifier over the real valued variable c and it is thus not an $\exists^I$-formula. However, by using standard quantifier-elimination in Linear Arithmetics, we can the derive the equivalent $\exists^I$-formula:

$$\exists i_1, i_2.(s[i_1] = w1 \wedge s[i_2] = cr \wedge v = i_1 \wedge b \leq 0 \wedge n < r[i_1] + a).$$

The sharp-eyed reader may have already realized that b, being an increment of time, should be positive so that the node just computed should be discarded without loss of information. While this observation is true, to prove safety this assumption is formally not needed (because if b is negative and $a < b$, the critical section is not reachable at all). In addition, MCMT will be able of delete this node because it is subsumed by others (see node labelled with 3 in Figures 3 and 4 below).

Let us apply rule Safety on formula N_1 above. Discharging the applicability condition of the rule amounts to checking the A_I^E-unsatisfiability of the following formula:

$$\exists i_1, i_2.(s[i_1] = w2 \wedge s[i_2] = cr \wedge v = i_1 \wedge n \geq w[i_1] + b \wedge i_1 \neq i_2) \wedge \\ n = 0 \wedge v = noProc \wedge \forall i.s[i] = nc.$$

It is not difficult to see that instantiating i with i_1 and i_2 considering these as free-constants yields an A_I^E-unsatisfiable quantifier-free formula; thereby implying that the intersection between the current set of backward reachable states and the initial set of states is empty.

Similarly, the applicability condition of FixPoint applied on N_1 reduces to checking the A_I^E-unsatisfiability of the following formula, obtained by negating $N_1 \rightarrow U$:

$$\exists i_1, i_2.(s[i_1] = w2 \wedge s[i_2] = cr \wedge v = i_1 \wedge n \geq w[i_1] + b \wedge i_1 \neq i_2) \wedge \\ \forall i'_1, i'_2.(s[i'_1] \neq cr \vee s[i'_2] \neq cr \vee i'_1 = i'_2).$$

It is tedious but straightforward to check that by instantiating (i'_1, i'_2) to (i_1, i_1), (i_1, i_2), (i_2, i_2), and (i_2, i_1) and considering i_1 and i_2 as free constants, the resulting quantifier-free formula is A_I^E-satisfiable;[7] thereby, implying that more pre-images of N_1 must be computed.

---

[6] Notice that some simplification steps have been applied, e.g., we derive $b \leq 0$ from $n + c \geq n + c + b$. One of the novelty of MCMT v. 1.0 is an enhanced simplification routine for Linear Arithmetic which allows us to control the size of $\exists^I$-formulae representing sets of backward reachable states.

[7] That such instantiations suffice follows from general facts, see [GNRZ08].

4.3 Formalization in mcmt: concrete syntax

We encode the enumerated data-type of sort $Elem_1$ with a finite sub-set of the integers as follows: 1 for *nc*, 2 for *rv*, 3 for *w1*, 4 for *w2*, and 5 for *cr*. To do this, we use the following command

```
:smt (define-type locations (subrange 1 5))
```

which tells MCMT (and more precisely, to its backhand SMT solver Yices) to include the declaration of the type `locations` whose values are the integers in the range [1...5].

We introduce the array variables together with their types for the elements:

```
:local  s locations
:global n real
:global v int
:local  r real
:local  w real
```

which tells MCMT that the system contains 5 array variables all of whose indices range over an implicitly declared subset of the naturals while their elements take values over the types `locations`, `real`s, `int`egers, `real`s, and `real`s, respectively. The special value *noProc* will be encoded by the integer value `-1`.

Then, we declare the two (time) parameters a and b of the protocol as follows:

```
:smt (define a::real)
:smt (define b::real)
```

which tells MCMT (and more precisely, to its backhand SMT solver Yices) to include the declaration of two (time invariant) constants of type `real`. The assumption that $a < b$ can be encoded as follows:[8]

```
:system_axiom
:var x
:cnj  (<  a  b)
```

The keyword `:system_axiom` tells MCMT that the formula that follows is an invariant of the system being specified. Since the formula after `:system_axiom` must always be universally quantified, the keyword `:var` introduces an implicitly universally quantified index variable (in this case `x`) that may occur in the formula after the keyword `:cnj`. Finally, the keyword `:cnj` must be followed in this case by a single quantifier free formula. The set of initial states is specified as follows:

```
:initial
:var x
:cnj (=  n  0) (=  v  -1) (=  s[x]  1)
```

The keyword `:initial` tells `mcmt` that the variable introduced by the keyword `:var` (in this case `x`) is implicitly universally quantified and may occur in the formula after the keyword `:cnj`. The meaning of the keyword `:cnj` is similar to that of block introduced by `:system_axiom`, but here you can insert after it a list of quantifier free formulae intended conjunctively.

[8] Without this assumption, the protocol is not correct and in fact dropping it causes MCMT to find an unsafety trace of length 10.

The set of unsafe states is specified as follows:

```
:unsafe
:var z1
:var z2
:cnj (=  s[z1]  5) (=  s[z2]  5)
```

The keyword `:unsafe` tells `mcmt` that the variables introduced by the keyword `:var` (in this case `z1` and `z2`) are implicitly existentially quantified and may occur in the formula after the keyword `:cnj`. Notice that the keyword `:cnj` can only be followed by a list of literals (not formulae as in the previous case) in Yices input language which are intended to be in conjunction. You need not add the conjunct `(not (= z1 z2))` to make the formula differentiated: this is done automatically by the tool. It is worth noticing that the present release of MCMT accepts multiple unsafety conditions intended disjunctively (see the User Manual for the appropriate syntax to be used).

In all formulae of the input specification files for MCMT, the array reading operation *can be applied only to variables*: you cannot for instance write `(> s[(+ z2 1)] 1)`, you must use an extra existentially quantified variable, say `z3`, and write the literals conjunction `(> s[z3] 1) (= z3 (+ z2 1))` instead. This is a special case of flattening for terms that is imposed by MCMT input syntax (more detailed instructions and motivations can be found in the User Manual).

Finally, we specify the transitions. To help readability, we also propose (on the right) the abstract version of the transition in Figure 2:

```
:comment ---------------------------------------------------
:eevar c real

:transition
:var j
:guard (>  c  0)
:numcases 1                    ∃c. ( c > 0 ∧ n′ = n + c )
:case
 :val s[j]
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:comment ---------------------------------------------------
```

The existentially quantified variable over data is introduced by the keyword `:eevar` and it can be used in any transition of the system. The keyword `:transition` introduces the block of the transition: a transition is composed by the variable declarations, the guard, the cases of a case distinction and the updates (within each case) of the (local and global) system variables.

– The index variable `j` is universally quantified and can only be used in the case distinctions (the list of literals following the keyword `:case`) and in the updates, i.e. in the terms after the keyword `:val`, see below,

– The index variables `x`, `y` can also be introduced by the keyword `:var`; these variables are existentially quantified (for instance, the former corresponds to the variable i in the abstract versions of transitions 2, 3, etc., see below). The existentially quantified variables are absent in the first transition, because the first transition only modifies global variables. The use of the existentially quantified variables is subject to precise constraints, see the User Manual.
– `:guard` introduces a list of literals in Yices format which may contain `c` and the declared existentially quantified variables.
– The number after `:numcases` specifies the number of cases of the update function (this number is 1 in the first transition, because there is no case distinction there).
– The list of literals after the keyword `:case` specifies the condition under which a certain update should be performed and the terms after the keyword `:val` specify how each array variable (in the order in which they are declared) must be updated. In the first transition, the list of literals after `:case` is empty (one could put `true` instead), because there is no case distinction there.

This transition will be identified by the name `t1` in the system. The remaining transitions are specified as follows and will be named `t2`, ..., `t7`:[9]

```
:comment ------------------------------------------------------
:transition
:var x
:var j
:guard (>  c  0) (=  s[x]  1)
:numcases 2
:case (= x j)
 :val  2
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:case (not (= x j))
 :val s[j]
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:comment ------------------------------------------------------
```

$$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = nc \\ \forall j. \left(\begin{array}{l} n' = n + c) \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ rv\ \mathsf{else}\ s[j] \end{array} \right) \end{array} \right) \wedge$$

[9] Notice that in Figure 2 the naming of the transition is the following: $\tau_0, \ldots, \tau_6$. The correspondence should be obvious: `t1` corresponds to τ_0, `t2` corresponds to τ_1, and so on.

```
:comment -------------------------------------------------------
:transition
:var x
:var j
:guard (>  c  0) (=  s[x]  2) (=  v  -1)
:numcases 2
:case (= x j)
 :val  3
 :val (+  n  c)
 :val v
 :val n
 :val w[j]
:case (not (= x j))
 :val s[j]
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:comment -------------------------------------------------------
```

$$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = rv \wedge v = noProc \quad \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ w1\ \mathsf{else}\ s[j] \wedge \\ r'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ n\ \mathsf{else}\ r[j] \end{array} \right) \end{array} \right)$$

```
:comment -------------------------------------------------------
:transition
:var x
:var j
:guard (>  c  0) (=  s[x]  3) (<= (+  n  c) (+  r[x]  a))
:numcases 2
:case (= x j)
 :val  4
 :val (+  n  c)
 :val x
 :val r[j]
 :val (+  n  c)
:case (not (= x j))
 :val s[j]
 :val (+  n  c)
 :val x
 :val r[j]
 :val w[j]
:comment -------------------------------------------------------
```

$$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = w1 \wedge n + c \leq r[i] + a \quad \wedge \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ w2\ \mathsf{else}\ s[j] \wedge \\ v' = i \wedge \\ w'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ n + c\ \mathsf{else}\ w[j] \end{array} \right) \end{array} \right)$$

```
:comment ------------------------------------------------------------
:transition
:var x
:var j
:guard (>  c  0) (=  s[x]  4) (>=  n (+  w[x]  b)) (not (= v x) )
:numcases 2
:case (= x j)
 :val  2
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:case (not (= x j))
 :val s[j]
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:comment ------------------------------------------------------------
```

$$\exists i, c. \begin{pmatrix} c > 0 \wedge v \neq i \wedge s[i] = w2 \wedge n \geq w[i] + b \wedge \\ \forall j. \begin{pmatrix} n' = n + c \wedge \\ s'[j] = \text{if } j = i \text{ then } rv \text{ else } s[j] \end{pmatrix} \end{pmatrix}$$

```
:comment ------------------------------------------------------------
:transition
:var x
:var j
:guard (>  c  0) (=  s[x]  4) (>=  n (+  w[x]  b)) (=  v x)
:numcases 2
:case (= x j)
 :val  5
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:case (not (= x j))
 :val s[j]
 :val (+  n  c)
 :val v
 :val r[j]
 :val w[j]
:comment ------------------------------------------------------------
```

$$\exists i, c. \begin{pmatrix} c > 0 \wedge v = i \wedge s[i] = w2 \wedge n[i] \geq w[i] + b \wedge \\ \forall j. \begin{pmatrix} n' = n + c \wedge \\ s'[j] = \text{if } j = i \text{ then } cr \text{ else } s[j] \end{pmatrix} \end{pmatrix}$$

```
:comment ------------------------------------------------------------
:transition
:var x
:var j
:guard (>  c  0) (=  s[x]  5)
:numcases 2
:case (= x j)
 :val  1
 :val (+  n  c)
 :val  -1
 :val r[j]
 :val w[j]
:case (not (= x j))
 :val s[j]
 :val (+  n  c)
 :val  -1
 :val r[j]
 :val w[j]
:comment ------------------------------------------------------------
```

$$\exists i, c. \left(\begin{array}{l} c > 0 \wedge s[i] = cr \\ \forall j. \left(\begin{array}{l} n' = n + c \wedge \\ s'[j] = \mathsf{if}\ j = i\ \mathsf{then}\ nc\ \mathsf{else}\ s[j] \wedge \\ v' = noProc \end{array} \right) \end{array} \right) \wedge$$

As mentioned above, there are syntactic constraints to be respected (most of them are checked by the MCMT parser). The reader is referred to the User Manual for complete information; we just mention some important syntax constraints here. First, the update of the global variables must be repeated in all cases of the case distintions in the literally identical way. Second, when the variable `x` is declared, there must be at least two cases in the case distinctions and the first case must be `(= x j)`. Third, the cases must be mutually exclusive and exhaustive. Moreover, the variable `y` can be declared only when `x` is declared too, etc.

4.4 Running mcmt

In order to solve the problem, it is sufficient to type at the command prompt the following command:

```
mcmt < Fischer.in
```

provided that the specification described in the previous section has been saved in the file named `Fischer.in` and that the MCMT executable is in the path of the shell. As a result, the system will output the following message:

```
MCMT - version 2.5
MCMT is linked to the SMT solver Yices version 1.0.39 (@ SRI, Stanford)
------------------------------------------------------------
 node 1= [t6_1][0]
 node 2= [t1][t6_1][0]
 node 3= [t4_1][t6_1][0]
 node 4= [t4_1][t1][t6_1][0]
 node 3 is deleted!
 node 5= [t3_1][t4_1][t1][t6_1][0]
 node 6= [t6_2][t4_1][t1][t6_1][0]
 node 7= [t2_1][t3_1][t4_1][t1][t6_1][0]
 node 8= [t5_1][t3_1][t4_1][t1][t6_1][0]
 node 9= [t1][t6_2][t4_1][t1][t6_1][0]
 node 10= [t4_2][t6_2][t4_1][t1][t6_1][0]
 node 11= [t1][t5_1][t3_1][t4_1][t1][t6_1][0]
 node 12= [t4_2][t1][t6_2][t4_1][t1][t6_1][0]
 node 10 is deleted!
 node 13= [t3_2][t4_2][t1][t6_2][t4_1][t1][t6_1][0]
 node 14= [t2_2][t3_2][t4_2][t1][t6_2][t4_1][t1][t6_1][0]
 node 15= [t5_2][t3_2][t4_2][t1][t6_2][t4_1][t1][t6_1][0]
 node 16= [t1][t5_2][t3_2][t4_2][t1][t6_2][t4_1][t1][t6_1][0]

============================================================================
Global fixpoint reached!

System is SAFE!

Max depth:10   #nodes:16   #deleted nodes:2   #SMT-solver calls:363 ...
============================================================================
```

MCMT executes backward reachability by trying to symbolically execute (backward) one of the available transitions among `t1`, ..., `t7` starting with the set of unsafe states which is identified by `[0]` in the output above. Doing this, it produces several nodes of the search space (which is organized as a tree along the lines of what has been explained in Section 3). For example, the first node (named `node 1`) has been generated by applying transition `t6` to the set of unsafe states (this amounts to apply rule PreImg in Figure 1). The number after the label of transition `t6`, namely `1`, tells us that the transition has been applied by identifying the existentially quantified variable `x` occurring in it with the first existentially quantified variable occurring in the formula characterizing the set

of unsafe states (i.e. `z1`). This is done by `mcmt` whenever it is possible in order to keep the number of existentially quantified variables occurring in the set of backward reachable states as low as possible. This is important to reduce the cost of quantifier instantiation when performing fixed-point checking. However, this is not always possible; the interested reader is pointed to [GRV08] for details about this point. Similarly, the second node (named `node 2`) has been generated by applying transition `t1` to `node 1` (since this has been generated by applying `t6` to the set of unsafe states). Once a node has been generated, it may become redundant because it is subsumed by other nodes: this is the case with `node 3` which is deleted because the disjunction of the formulae labelling nodes `1`, `2`, and `4` is implied by it.

When no more nodes can be expanded (because of rules NotAppl or FixPoint in Figure 1), a global fix-point has been reached and the system is said to be safe (since rule Safety in Figure 1 is not applicable). Finally, some statistics are reported such as the depth of the tree (`10`), the number of nodes obtained by applying rule PreImg (`16`), the number of redundant nodes which have been deleted (`2`), the number of calls to the backhand SMT-solver (`336`), etc.

If one is only interested in the final result, it is possible to run the system in silent mode as follows:

```
mcmt -s < Fischer.in
```

In this case, the output is much more compact:

```
MCMT - version 2.5
MCMT is linked to ....
------------------------
STATS=SAFE:10:16:2:363:0
```

However, it is possible to extract much more information about the set of backward reachable states by generating a report listing the formulae labelling the tree as well as a graphical representation of the tree itself. This can be done by invoking the system as follows:

```
mcmt -r "Fischer" < Fischer.in
```

which generates a LaTeX file (named `Fischer.report.tex`)[10] containing a list of the formulae labelling the nodes of the tree representing the search space as well as a file (named `Fischer.report.dot`) containing a picture of the search space in Dot format, which can be read by Graphwiz.[11] Figures 3 and 4 contain an excerpt of the LaTeX file and the tree representing the search space for the Fischer example, respectively. In Figure 4, deleted nodes are depicted in gray.

Finally, it is sometimes possible to reduce the search space explored by MCMT by using invariants (sometimes, it is even possible to avoid non termination in this way). There are various available strategies to synthesize invariants which can be tried to reduce the response time of the system. The interested reader is pointed to the on-line User Manual for a complete overview.

[10] The option "`-r STRING`" tells MCMT to create a LaTeX file named `STRING.report.tex` and a dot file named `STRING.report.dot`.

[11] Available at `http://www.graphviz.org`.

List of reachable nodes

- **Kept** node 0 at depth 0 labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (= s[z1] 5) (= s[z2] 5))}$$

- **Kept** node 1 at depth 1 generated by applying the transition $\tau_6(\texttt{z1})$ on node 0 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (<= (+ b w[z1]) n) (= s[z1] 4) (= s[z2] 5)}\ \texttt{(= z1 v))}$$

- **Kept** node 2 at depth 2 generated by applying the transition τ_1) on node 1 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (= s[z1] 4) (= s[z2] 5) (= z1 v))}$$

- **Deleted** node 3 at depth 2 generated by applying the transition $\tau_4(\texttt{z1})$ on node 1 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (< n (+ a r[z1])) (<= b 0) (= s[z1] 3) (= s[z2] 5)}$$

- **Kept** node 4 at depth 3 generated by applying the transition $\tau_4(\texttt{z1})$ on node 2 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (< n (+ a r[z1])) (= s[z1] 3) (= s[z2] 5)}$$

- **Kept** node 5 at depth 4 generated by applying the transition $\tau_3(\texttt{z1})$ on node 4 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (< 0 a) (= s[z1] 2) (= s[z2] 5) (= 0 (+ 1 v))}$$

- **Kept** node 6 at depth 4 generated by applying the transition $\tau_6(\texttt{z2})$ on node 4 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (< n (+ a r[z1])) (<= (+ b w[z2]) n) (= s[z1] 3)}\ \texttt{(= s[z2] 4) (= z2 v))}$$

- **Kept** node 7 at depth 5 generated by applying the transition $\tau_2(\texttt{z1})$ on node 5 and labelled by

$$\exists \texttt{z1}, \texttt{z2}.\ \texttt{(and (< 0 a) (= s[z1] 1) (= s[z2] 5) (= 0 (+ 1 v))}$$

Fig. 3. Excerpt of the file `Fischer.report.tex`

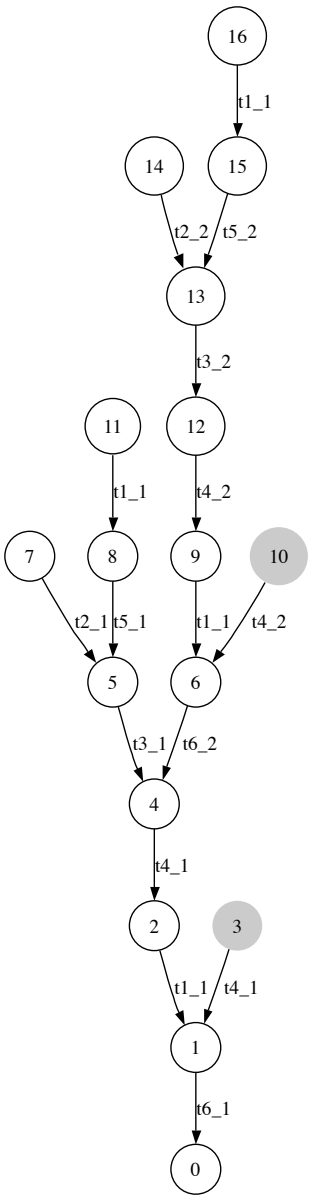

Fig. 4. Picture of search space for Fischer (file `Fischer.report.dot`)

References

GNRZ08. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model-Checking of Array-based Systems. In *Proc. of IJCAR*, LNCS, 2008.

GR09a. S. Ghilardi and S. Ranise. Goal Directed Invariant Synthesis for Model Checking Modulo Theories. In *Tableaux 09*, LNAI, pages 173–188. Springer, 2009.

GR09b. S. Ghilardi and S. Ranise. Model Checking Modulo Theory at work: the intergration of Yices in MCMT. In *AFM 09 (co-located with CAV09)*, 2009.

GR10. S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *Proc. of IJCAR 2010*, LNCS, 2010.

GRV08. S. Ghilardi, S. Ranise, and T. Valsecchi. Light-Weight SMT-based Model-Checking. In *Proc. of AVOCS 07-08*, ENTCS, 2008.