Model-Checking Modulo Theories at Work: the integration of Yices in MCMT

Silvio Ghilardi
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
ghilardi@dsi.unimi.it

Silvio Ranise
Dipartimento di Informatica
Università degli Studi di Verona
silvio.ranise@univr.it

ABSTRACT
Recently, the notion of an array-based system has been introduced as an abstraction of infinite state systems (such as parametrised systems) which allows for model checking safety properties by SMT solving. Unfortunately, the use of quantified first-order formulae to describe sets of states makes checking for fix-point and unsafety extremely expensive. In this paper, we describe (static and dynamic) techniques to overcome this problem which have been implemented in the (declarative) model checker MCMT. We describe how such techniques have been combined with Yices (the back-end SMT solver) and discuss some interesting experimental results.

Keywords
Declarative model checking, Backward reachability, SMT, Yices

1. INTRODUCTION
Safety properties for classes of infinite state systems can be checked by a (backward) reachability analysis, i.e. repeatedly computing the pre-image of the set of unsafe states (obtained by complementing the property to be verified) and checking for fix-point and emptiness of the intersection with the set of initial states. This analysis is a decision procedure for some classes of parametrised systems [1], i.e. systems consisting of a finite (but unknown) number n of identical processes which can be modelled as extended finite state automata, manipulating variables whose domains can be unbounded, like integers. The challenge is to verify properties for any number n of processes.

Backward reachability analysis is parametrised with respect to the topology of the parametrised system and the domains of the variables manipulated by each process in the system. Sometimes, the topology of the system can be safely abstracted away (e.g., by counting abstraction [7]) so as to obtain a guarded transition systems where only numeric variables are used. Systems based on constraint solving techniques capable of handling this kind of transition systems have been successful for some classes of systems (see, e.g., again [7]). However, for many parametrised, counting abstractions are not precise and spurious counter-examples are introduced. To overcome this problem, there have been attempts at designing 'topology-aware' backward reachability analyses (see, e.g., [2, 3, 4]). Usually, these approaches encodes the topology in a suitable data-structure which allows for the finite representation of an infinite set of states of the parametrised systems while the data manipulated by each process in the system are handled declaratively by (numerical) constraints. These approaches are hybrid as they exploit both dedicated data structures and (declarative) constraints to represent the infinite sets of states of the system. This implies that the computation of the pre-image also requires manipulations of constraint expressions for the data variables and of the structure encoding the topology of the system. So, while manipulations on constraints depends only on very general properties of the constraint structure itself and can thus be re-used without much effort, this is not the case for the topology which is represented by *ad hoc* data structures. In fact, every time the topology changes, the computation of the pre-image must be designed from scratch and the requirements ensuring the termination of the backward analysis must be checked; e.g., the existence of a suitable pre-order (i.e. it is a reflexive and transitive binary relation) on configurations which finitely represent infinite set of states (see [1] for details).

In [9], to overcome all these problems and obtain a fully declarative approach to backward reachability of infinite state systems, we proposed the notion of *array-based systems* as a suitable abstraction for parametrised systems as well as sequential programs manipulating arrays, or lossy channel systems. The idea is to use classes of first-order formulae to represent an infinite set of states of the system so as to simplify the computation of pre-images. As it is standard in deductive verification of software, arrays are modelled by functions. In this framework, both the topology and the data manipulated by the systems are specified by using a suitable class of first-order structures, which is modularly and uniformly obtained by combining a class of structures for the topology and one for the data via the functions representing arrays. More precisely, both the topology and the data of the system are specified by first-order theories, which are pairs formed by a (first-order) language and a class of (first-order) structures. In this way, the union of the lan-

guages augmented with the function symbols representing arrays yields the language of the formulae representing the states of the system.

In order to mechanize the backward reachability analysis using formulae to represent states, three requirements must be satisfied. First, the class of formulae used to represent states must be closed under pre-image computation. Second, the tests for fix-point and safety should be reduced to decidable logical problems. Third, sufficient conditions on the theories specifying the topology and the data must be identified so as to guarantee termination of the backward reachability analysis. All these requirements have been investigated in our previous works [9, 12] and the interested is pointed to those papers for details. In particular, we recall that the computation of the pre-image for the class of guarded assignment systems consists of simple manipulations of formulae and that it is straightforward to show closure under pre-image computation [12]. We also recall how it is possible to reduce the checks for fix-point and safety to Satisfiability problems Modulo the Theories (SMT) for the topology and the data of the system of first-order formulae containing (universal) quantifiers. Under suitable hypotheses on the theories, this SMT problem turns out to be decidable [9, 12]. However, this is not yet enough to ensure the termination of the backward reachability analysis: in [9], we showed how to introduce a pre-order on the set of configurations of the system and observed that termination is achieved when such a pre-order is a well-quasi-order[1] (this is the case, among others, of broadcast protocols and lossy channels systems).

In this paper, we focus on the pragmatics of our approach and discuss techniques that allowed us to implement a prototype tool, called MCMT,[2] which proved to be competitive with the state-of-the-art model checker (for parametrised systems) PFS [2]. The crucial problem is solving the SMT problem for fix-point and safety checking, because it is required to be able to handle quantified formulae which makes the off-the-shelf use of SMT solvers problematic. In fact, even when using classes of formulae with decidable satisfiability problem, currently available SMT solvers are not yet mature enough to efficiently discharge formulae containing (universal) quantifiers, despite the fact that this problem has recently attracted a lot of efforts (see, e.g., [6, 8, 5]). To alleviate this problem, we have designed a general decision procedure for the class of formulae satisfying the requirements above, based on quantifier instantiation and SMT solving [9]. Unfortunately, the number of instances required by the instantiation algorithm is still very large and preliminary experiments have shown poor performances. This fact together with the observation that the size of the formulae generated by the backward search algorithm grows very quickly demand a principled approach to the pragmatics of efficiently integrating SMT solvers in backward analysis.

More precisely, we describe *instance reduction* heuristics which allows us to reduce the number of instances to be considered for the tests of fix-point and non-empty intersection with the initial set of states. We consider two types of techniques: *static* and *dynamic*. We illustrate the former by showing how a careful classification of the transitions based on their shape allows us to reduce the number of (universally) quantified variables and consequently also the number of their instances. For dynamic instance reduction techniques, we will see how to *filter* instances of formulae which cannot contribute to show unsatisfiability by (computationally inexpensive) reasoning *modulo* certain theories of *enumerated data-types*. We also describe that during the exploration of the symbolic state space by backward reachability, we have observed a sort of "locality principle" for fix-point testing whereby using the more recently visited formulae representing the set of backward reachable states has proved to detect unsatisfiability earlier; we call this *chronological fix-point checking*.

Finally, we describe the general architecture of our model-checker and its main loop while emphasizing the central role played by the SMT solver Yices;[3] and discuss some experimental results about a heuristics that we have implemented in the latest release of the tool.

2. ARRAY-BASED SYSTEMS AND MCMT

To make the paper self-contained, we briefly recall the notion of array-based system and related safety problem. The basic ingredients of an array based system are a theory T_I (with related signature Σ_I) for describing the topology of a system (i.e. T_I is a theory specifying the structure of the identifiers, also called indexes in the following, of processes in a parametrized system) and another theory T_E (with related possibly multi-sorted signature Σ_E) for describing data (i.e. numerical values, program counter locations, etc.); the array-based system has *array variables* $a, b, \ldots$ for local data: these are function variables having the index sort as source sort and some data sort as target sort. For example, to specify a system which requires only to identify processes by their identifiers, T_I will be the theory of equality (without functions); to state that processes are arranged in a linear array, T_I is the theory of linear orders whose signature consists just of a binary relation $<$ which allows one to locate processes 'on the left' or 'on the right' of a given process. Indeed, most available SMT solvers provide support for these theories: the former can be handled by using (part of) a congruence closure algorithm while the latter by using (part of) a decision procedure for Linear Arithmetics. Since it is more common to use constraints to handle the content of data variables, we only mention that SMT solvers (and Yices in particular) offer support for several instances of T_E. In particular, the theory of enumerated datatype (i.e. the theory whose signature consists of only a finite set of constants which are constrained to be pairwise distinct and to name all the elements in the domain of any structure in the class of models) is particularly useful to model control locations, i.e. the states of the extended automaton representing the processes in the system.

Once T_I and T_E are fixed, we consider the tuple of function symbols representing all array variables manipulated by the system. For simplicity, here, we consider array-based system with only one array variable a. Then, the specification of

[1]A well-quasi-ordering $\leq$ on a set X is a quasi-ordering (i.e., a reflexive, and transitive binary relation) such that any infinite sequence of elements $x_0, x_1, x_2, \ldots$ from X contains an increasing pair $x_i \leq x_j$ with $i < j$.

[2]`http://homes.dsi.unimi.it/~ghilardi/mcmt`

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

an array-based system consists of a formula $I(a)$ describing the *initial* sets of states and a *transition* formula $\tau(a,a')$ relating actual a and updated a' array variables. A *safety* or reachability problem for the array based system $\mathcal{S}=(a,I,\tau)$ is a formula $U(a)$ specifying a set of states the system should not be able to reach starting from a state in I and firing τ finitely many times.

Formulae I,τ,U are subject to some syntactic restrictions [9, 12, 11] that guarantee that all safety and fix-point tests needed for a standard *backward reachability analysis* [1] can be reduced to quantifier-free SMT problems (modulo T_I and T_E) by instantiation, i.e. to the kind of tests that can be efficiently handled by state-of-the-art SMT solvers like Yices. In the following, we use two classes of formulae to describe states: *primitive differentiated* and $\forall^I$-formulae. The former is used to describe sets of unsafe state and consists of the existential closure of conjunctions of literals containing all the literals needed to say that quantified variables range over distinct indexes (see [11] for details). The following is an example of primitive differentiated formula:

$$\exists \mathtt{x},\mathtt{y}.(\mathtt{x}\neq\mathtt{y}\wedge \mathtt{a}[\mathtt{x}]=q_2\wedge \mathtt{a}[\mathtt{y}]=q_2)$$

saying that there exists two distinct processes in the system, identified by the identifiers `x` and `y` (which are constrained to be distinct by the literal $\mathtt{x}\neq\mathtt{y}$) such that the control location of the processes `x` and `y` is q_2 (so, if q_2 identifies the critical section, then the formula above specifies the violation of mutual exclusion). In other words, the formula describes all the configurations of the system where there exists two distinct processes which are both in control location q_2.

The class of $\forall^I$-formulae is used to describe the set of initial states or invariants of the system. For example, to say that all processes initially are in control location q_0, it is possible to use the formula: $\forall \mathtt{x}.\mathtt{a}[\mathtt{x}]=q_0$. Another example is the negation of the primitive differentiated formula above, i.e.

$$\forall \mathtt{x},\mathtt{y}.((\mathtt{a}[\mathtt{x}]=q_2\wedge \mathtt{a}[\mathtt{y}]=q_2)\Rightarrow \mathtt{x}=\mathtt{y})$$

describing the set of states of the system satisfying mutual exclusion.

For transition formulae $\tau(a,a')$, we use formulae that correspond to guarded assignment systems. An example of such formulae is the following:

$$\exists x.(\mathtt{a}[\mathtt{x}]=q_2\wedge\forall \mathtt{j}.a'[j]=\textit{if } \mathtt{j}=\mathtt{x} \textit{ then } q_3 \textit{ else } q_0),$$

where the first conjunct is the *guard* and the second is the update. Notice that the update can be seen to model a broadcast action in a parametrised system: a process `x` which is in the control location q_2 moves to q_3 while all the other processes react by going to control location q_0.

2.1 An example specification in MCMT

To illustrate how to work with MCMT, we consider the verification of the key safety property of a protocol used to maintain coherence in a system with multiple (local) caches as used, for example, in commercial multiprocessor systems. In the following, we specify the Illinois cache coherence protocol (see, e.g., [7] for details) as an array-based system in MCMT.[4] Each cache in the protocol may have four possible control locations: `invalid`, `dirty`, `shared`, or `exclusive`. This is specified to MCMT by the following declarations:[5]

```
:smt (define-type locations (subrange 1 4))
```

The keyword `:smt` tells the model checker that what follows is a Yices expression and should be added to the current context of Yices. In this case, we introduce the new type symbol `locations` in the current Yices context whose values may range over the set of integers $\{1,2,3,4\}$; where `1` encodes `invalid`, `2` encodes `dirty`, etc. If we use integers to identify the different copies of a cache, the state of the Illinois protocol can be naturally represented by an array mapping a finite sub-set of the integers to locations. This is declared as follows:

```
:local a locations
```

where the keyword `:local` tells MCMT that `a` is an array variable whose indexes are a finite sub-set of the integers and whose elements must be of type `locations`. This declaration also implicitly defines the theory T_E of the elements stored in the array as the theory of an enumerated data-type containing four distinct elements, identified by the numerals from `1` to `4`. At the beginning, all the caches in the Illinois cache coherence protocol are in the state `invalid`. The corresponding declaration in MCMT is the following:

```
:initial
:var x
:cnj (= a[x] 1)
```

The keyword `:initial` indicates that we are going to specify the set of initial states, `:var` specifies which (implicitly universally quantified) variable is allowed to occur in the formula, and `:cnj` says that the space separated list of formulae (in Yices syntax) is intended to be read conjunctively. The elements of the array variable `a` are accessed by using square brackets as in C: this is a peculiarity of the MCMT syntax. The logical reading of the piece of specification above is the following formula: $\forall \mathtt{x}.\mathtt{a}[\mathtt{x}]=\mathtt{1}$, where `x` is a variable of type `int`.

The negation of cache coherence (describing the set of unsafe states) is that there should not be any two local caches in `dirty` or one cache in `dirty` and another one in `shared`. This is specified in MCMT as follows:

```
:unsafe
:var x
:var y
:cnj (= a[x] 2) (> a[y] 1) (< a[y] 4)
```

[4]The file containing the complete specification of this example (with many others) is available from `http://homes.dsi.unimi.it/~ghilardi/mcmt`.

[5]The syntax of MCMT includes a sub-set of Yices syntax. Roughly, the rule is that all the keywords starting with a semicolon are specific to the model checker while the remaining should be valid expressions of the input language of Yices.

Indeed, the keyword `:unsafe` tells the tool that we are going to specify the set of unsafe states, the two `:var`'s introduce the (implicitly existentially quantified) variables which are allowed to occur in the formula, and furthermore the two variables are assumed to be mapped to two distinct integers (this is because, when describing sets of states, MCMT uses primitive differentiated formulae.The logical reading of this piece of specification is thus the following:

$$\exists \mathtt{x}, \mathtt{y}.(\mathtt{x} \neq \mathtt{y} \wedge \mathtt{a}[\mathtt{x}] = 2 \wedge \mathtt{a}[\mathtt{y}] > 1 \wedge \mathtt{a}[\mathtt{y}] < 4).$$

Notice that by using integers to encode locations, we are able to avoid the explicit use of disjunction as $\mathtt{a}[\mathtt{y}] > 1 \wedge \mathtt{a}[\mathtt{y}] < 4$ is logically equivalent to $\mathtt{a}[\mathtt{y}] = 2 \vee \mathtt{a}[\mathtt{y}] = 3$.

The full specification of the Illinois protocol requires eleven transitions to be specified [7]. For lack of space, we just give two of them:

```
:comment t1                    :comment t2
:transition                    :transition
:var x                         :var x
:var j                         :var j
:guard (= a[x] 1)              :guard (= a[x] 2)
:numcases 2                    :numcases 2
:case (= x j)                  :case (= x j)
 :val 2                         :val 1
:case (not (= x j))            :case (not (= x j))
 :val 1                         :val a[j]
```

The transition on the left is a (so-called) 'write miss,' i.e. one cache (identified by `x`) gets an exclusive copy of the data and it moves to the control location `dirty` if it was `invalid`, whereas all others (identified by `j`) go to `invalid`. The logical reading of this piece of specification is the following:

$$\exists \mathtt{x}. \left(\begin{array}{l} \mathtt{a}[\mathtt{x}] = 1 \;\; \wedge \\ \forall \mathtt{j}.\mathtt{a}[\mathtt{j}] = (\textit{ if } \mathtt{x} = \mathtt{j} \textit{ then } 2 \textit{ else } 1) \end{array} \right),$$

where $\mathtt{a}[\mathtt{x}] = 1$ is called a *guard* (cf. the keyword `:guard`) and $\forall \mathtt{j}.\mathtt{a}[\mathtt{j}] = (\textit{ if } \mathtt{x} = \mathtt{j} \textit{ then } 2 \textit{ else } 1)$ is called a *case-defined function* update (cf. the keyword `:case`). The values stored in the elements of the array `a` after the execution of the transition are specified after the keyword `:val`. (If there is more than one array variable, for each case of the function, the user is supposed to specify the values of each array variable, according to the order in which they were declared at the beginning of the input file.) In its current release, MCMT supports case-defined functions with at most 20 cases, specified by the `:numcases` keyword above. According to the logical reading, the variable `x` is implicitly existentially quantified while the variable `j` is implicitly universally quantified. And this is so, despite the fact that the two variables are introduced by the same keyword `:var`. To clarify this point, we must say that, in MCMT, transitions may have at most two existentially quantified variables and one universally quantified variable: the former are assumed to have `x` and `y` as identifiers while the latter to have `j` as identifier. This rigid convention is to simplify parsing as much as possible and it will be relaxed in future releases of the tool.

The transition on the right is called a 'replacement,' according to which the content of a `dirty` cache is written in the main memory: the cache gets `invalid`ated, whereas the remaining ones maintain their status. The logical reading of the transition on the right is given by following formula:

$$\exists \mathtt{x}. \left(\begin{array}{l} \mathtt{a}[\mathtt{x}] = 2 \;\; \wedge \\ \forall \mathtt{j}.\mathtt{a}[\mathtt{j}] = (\textit{ if } \mathtt{x} = \mathtt{j} \textit{ then } 1 \textit{ else } \mathtt{a}[\mathtt{j}]) \end{array} \right).$$

Since to specify the initial and final set of states, and all the transitions (also those not shown above), we have used only the equality symbol, the theory T_I over the indexes can be assumed to be the theory of equality. This is a declarative specification of the topology assumed by the cache coherence protocols to ensure the coherence property. In general, it is possible to characterize several different topologies for a distributed system by choosing a suitable theory T_I, such as the theory of linear orders for linearly ordered collections of processes, the theory of trees or graphs for the corresponding topologies. MCMT supports the declarative specifications of several different background theories both by using the `:smt` directive—as explained above—to enrich the Yices context with new type declarations and the keyword `:system_axiom` that allows one to extend the context with axioms concerning certain predicate symbols playing the role, for example, of recognizers for trees or graphs.

Two remarks are in order. First, because of the declarative approach underlying MCMT, alternative specifications of the same cache coherence protocol can be fed to the system. For example, those obtained by using counting abstraction [7] are easy to formalize in MCMT and Yices gives support to handle the resulting SMT problems which are modulo the theory of Linear Arithmetics. In the case of the Illinois protocol considered above, its counting abstraction is easily handled by MCMT. For details on the performances on this and other specifications, the reader is referred to Section 3. The second remark is about the input specification language of MCMT which is quite difficult to understand by humans. However, this should not be regarded as a drawback since the language has been designed for ease of parsing or as a target language of translators from richer and human-friendly specification languages.

2.2 More specification constructs

We consider here a couple of interesting constructs of MCMT specification language not exemplified by the (partial) specifications of the Illinois protocol given above.

Shared variables. An important mechanism to propagate information in distributed systems is to have shared variables that processes can read or update. In MCMT, it is possible to declare every such variable, say `g`, as follows:

```
:global g int
```

which means that `g` can be seen as a (single) integer variable. However, the system requires to dereference its value as a 'standard' array variable (declared by `:local`) such that

$$\forall \mathtt{x}, \mathtt{y}.\mathtt{g}[\mathtt{x}] = \mathtt{g}[\mathtt{y}], \qquad (1)$$

i.e. `g` stores the same value in every one of its cells. In other words, every process in the system has a local copy of the shared variable whose value is identical to the value of the local copies of all the other processes. Indeed, transitions

must be designed in such a way to ensure that processes always have an identical copy of the same value. Internally, the tool exploits the information contained in the `:global` declaration by asserting the invariant (1) in the current Yices context.

Universal guards. In the two transitions of the (original) specification of the Illinois protocol, we have encountered only *existential* conditions, specified by the keyword `:guard`. However, there are other transitions of the protocol for which *universal* (also called, global) conditions are required. In general, this kind of transitions turns out to be useful for specifying some class of parametrised systems, i.e. distributed systems consisting of a finite collection of processes (see [13] for examples of this). To illustrate the notion of universal condition, consider a guard saying that a process i can execute a transition if a certain condition is satisfied by *all* processes $j \neq i$. MCMT supports also this kind of condition by the keyword `:uguard`. Internally, the tool automatically translates such universal guards into existential ones by recasting the parametrised system under consideration in the so-called *stopping failures model* [13], which is quite close to the *approximate* model of [3, 2]. The key property of a parametrized system modelled according to the stopping failures model is that processes may fail without warning at any time. To formalize this, we add q_{crash} to the set of control locations of each process in the system and a set of transitions saying that it is always possible to go from any control location to q_{crash}, for any process. Using q_{crash} it is possible to transform a universal into an existential guard at the expense of introducing more runs in the systems, thereby preserving safety and even some liveness properties such as recurrence (in practice, the system provides "real" safety certifications, but it can—quite rarely—give spurious traces of unsafety). To illustrate, consider the universal guard saying that a process i can execute a transition if a certain predicate C is satisfied by all processes $j \neq i$. In the stopping failures model, this can be expressed without the universal quantification as follows: the process i takes the transition without checking the predicate C and, concurrently, all processes $j \neq i$ not satisfying C move to the location q_{crash}; moreover, all processes $j \neq i$ satisfying C behave as originally prescribed. For more details on this issue, the interested reader is pointed to [12, 10].

3. ANATOMY OF MCMT

To solve safety problems, MCMT uses a refinement of 'backward reachability analysis' (see, e.g., [1]). This procedure repeatedly computes the pre-images of the set of unsafe states. The key peculiarity of MCMT—as anticipated by the specification of the Illinois protocol in the previous section—is the use of a *completely declarative approach* where (a) both the topology of and the elements manipulated by distributed systems are formalized by (first-order) theories, (b) sets of states are described by (a certain class of) logical formulae, namely disjunctions of primitive differentiated formulae, and (c) transitions are also described by (a certain class of) logical formulae, corresponding to guarded assignment systems. There are two main crucial consequences of this approach. First, let a be the array state variables of the system, $K(a)$ be a primitive differentiated formula describing the set of reachable states, and $\tau(a, a')$ be the formula describing one of the transitions in the system (representing a guarded assignment); the computation of the pre-image $Pre(\tau, K)$ of $K(a)$ with respect to $\tau(a, a')$ is greatly simplified. In fact, it is possible to show [12] that $Pre(\tau, K)$, i.e.

$$\exists a'.(\tau(a, a') \wedge K(a'))$$

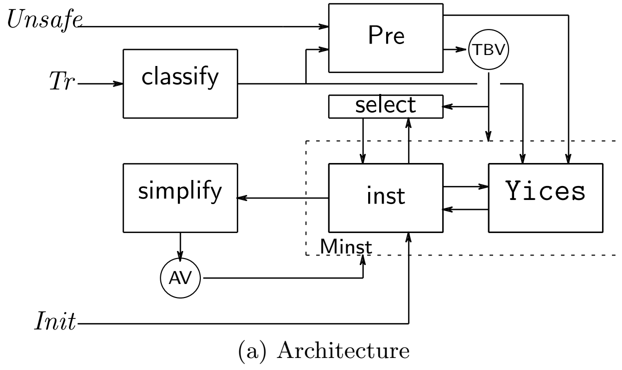
is logically equivalent to a disjunction of primitive differentiated formulae. The second important consequence of the declarative approach underlying MCMT is that checking for fix-point and intersection with the set of initial states can be reduced to a satisfiability problem modulo the combined theories T_I and T_E which are "connected" by the array a. The formulae to be checked for satisfiability are of the form $\exists \underline{i} \forall \underline{j} \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}])$, where $\underline{i}, \underline{j}$ are tuples of index variables and ψ is a quantifier free formula (notice that such formulae contain universally quantified variables in $\underline{j}$ and the existentially quantified variables in $\underline{i}$ may be regarded as Skolem constants). Under suitable hypotheses on T_I and T_E, it can be shown that the satisfiability of these formulae is decidable [9]. The proof of the decidability result suggests a two-phase procedure: (1) instantiate the universally quantified $\underline{j}$ to the Skolem constants $\underline{i}$ in all possible ways and (2) check the satisfiability of the resulting quantifier-free formulae modulo the theory obtained by the combination of T_I and T_E. Indeed, for efficiency, the two phases should be interleaved, the check for satisfiability can be done incrementally, and the satisfiability in the combination of T_I and T_E can be solved by using standard combination methods readily available in state-of-the-art SMT solvers. The instantiation phase is one of the main bottle-necks of the system and many of the heuristics implemented in MCMT have been developed to reduce the number of instantiations to be considered.

The main modules of MCMT and their connections are depicted in Figure 1(a) while its main loop can found in Figure 1(b). Notice the key role played by Yices (cf. Figure 1(a)), as (almost) all modules in the system invoke the available SMT solver. The integration of Yices in our model checker has been greatly simplified by the availability of the lightweight API. Particularly crucial to efficiently handle the SMT problems of quantified formulae for fix-point and safety checks is the incrementality of satisfiability checks so that the quantifier-free instances are incrementally added and unsatisfiability is reported as soon as possible. A nice set of interface functionalities is still missing from the SMT-Lib standardization effort.[6] Once available, this would enable us to perform interesting experimental investigations in using several SMT solvers by adopting the standard SMT-Lib language.

3.1 Main loop

The system takes as input the description of a safety problem consisting of the formulae describing the initial set of states (*Init*), its transitions (*Tr*), and the set of unsafe states (*Unsafe*). First of all, transitions are processed to extract relevant information for the successive computation of pre-images (module classify at lines 1–2). This is done in order to reduce the number of instantiated formulae in the satisfiability checks at lines 3, 9, and 10. Then, the formula $\exists \underline{i}.\forall \underline{j}.\, Unsafe(\underline{i}) \wedge Init(\underline{j})$ is checked for satisfiability

[6] `http://www.smt-lib.org/`



(a) Architecture

fun MCMT($Init$, Tr, $Unsafe$)
1 $Tr^\ell \leftarrow \emptyset$;
2 **foreach** $\tau \in Tr$ **do** $Tr^\ell \leftarrow$ classify$(\tau) \cup Tr^\ell$;
3 **if** inst($Init$, getinds($Unsafe$), $Unsafe$) = sat
then return unsafe;
4 $P \leftarrow Unsafe$; AV $\leftarrow \emptyset$; TBV $\leftarrow \emptyset$;
5 **do**
6 AV $\leftarrow$ AV $\cup P$;
7 **foreach** $\tau^\ell \in Tr^\ell$ **do**
8 $N \leftarrow$ simplify(Pre(τ^ℓ, P));
9 **if** Minst(AV $\cup$ TBV, getinds(N), N) = sat
then begin TBV $\leftarrow$ TBV $\cup N$;
10 **if** inst($Init$, getinds(N), N) = sat
then return unsafe;
end
11 **end**
12 **if** TBV $\neq \emptyset$ **then** $P \leftarrow$ select(TBV);
13 **while** TBV $\neq \emptyset$;
14 **return** safe;

(b) Main loop

Figure 1: High-level description of mcmt

(line 3). The module inst implements the two-phase decision procedure for formulae of the form $\exists \underline{i} \forall \underline{j} \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}])$ sketched above; it takes the formulae $Init$ and $Unsafe$ as input together with the set of (existentially quantified) index variables occurring in $Unsafe$ (computed by the function getinds); inst uses Yices to check the satisfiability of the resulting quantifier-free formulae in the combination of the theories over indexes and elements (see Figure 1(a)). If the satisfiability check is positive, then the intersection between the set of initial states and that of unsafe states is non-empty and the array-based system is trivially unsafe (line 3). Otherwise (lines 4–13), the main part of the algorithm for backward reachability analysis is entered. The loop maintains two sets AV and TBV of primitive differentiated formulae; the former stores the sets of states which have been already visited while the latter those which are still to be visited (see the two circles in Figure 1(a)). The variable P contains the set of states which is currently under consideration; when entering the loop for the first time, this is the set of unsafe states (line 4). In the loop, the pre-images of (formulae in) P with respect to each transition in Tr are computed and simplified (line 8). Each pre-image is tested for fix-point as follows. The formula

$$\exists \underline{i}.\forall \underline{j}.P(\underline{i}) \quad \wedge \bigwedge_{V \in \mathsf{AV} \cup \mathsf{TBV}} \neg V(\underline{j})$$

is checked for satisfiability (line 9); this is done by the module Minst which goes over the set AV∪TBV and incrementally invokes the module inst to handle the instantiation of each formula V (see the dashed box in Figure 1(a)). Each (instance of a) formula V in AV ∪ TBV is incrementally added and checked in the current Yices context. (This is similar to the *forward redundancy elimination* of resolution-style automated theorem provers.) If Minst concludes the unsatisfiability of the formula, then a fix-point has been reached (as $P \rightarrow \bigvee \mathsf{AV} \cup \mathsf{TBV}$ is valid) and the formula is discarded; otherwise, it is added to the set TBV of the already visited states (line 9). Then, it is tested whether the computed pre-image has a non-empty intersection with the set of initial states and, if the case, unsafety of the array-based system is reported (line 10). Once all the pre-images of a certain set of states have been computed, a new set of states is selected (and deleted from) the set TBV of states yet to be visited, if TBV is non-empty and the main loop is executed again with the new choice of P.

Since the set AV is handled as a queue, MCMT explores *on-the-fly* (i.e. while generating it) and *breadth-first* (according to the order in which the user listed the transitions) the 'state space graph' of the safety problem under consideration. The nodes of this graph are labelled by the primitive differentiated formulae representing sets of backward reachable states. MCMT (using the option `-r`) is capable of producing a list of the visited nodes, the formulae labelling them, and a graphical representation of the state space graph. The first two in LaTeX while the last one in the format supported by the graphical tool GraphViz.[7]

3.2 Main modules

To complete the high-level description of MCMT, we now give more details about the heuristics embodied in each module of the architecture in Figure 1(a).

Pre. Let a be the array state variables of an array-based system, $P(a) := \exists \underline{k}\, P(\underline{k}, a[\underline{k}])$, where P is a primitive differentiated conjunction of literals, and $\tau(a, a') := \exists \underline{i}\ (G(\underline{i}, a[\underline{i}]) \wedge a' = \lambda j.F(\underline{i}, a[\underline{i}], j, a[j]))$ be a transition corresponding to a guarded assignment, where G is a conjunction of literals and F is a case-defined function update (the use of the λ-abstraction can be avoided by using universal quantification as seen for the Illinois protocol). It is straightforward to show [12] that $Pre(\tau, P)$ is equivalent to the following formula:

$$\exists \underline{i}\, \exists \underline{k}.G(\underline{i}, a[\underline{i}]) \wedge P(\underline{k}, F(\underline{i}, a[\underline{i}], \underline{k}, a[\underline{k}])). \qquad (2)$$

So, in principle, the implementation of Pre simply amounts to (repeatedly) build up formula (2). However, there are two crucial problems in using formula (2) without any further manipulation. First, the number of existentially quantified variables in the formula has grown since $\exists\ \underline{k}$ is augmented with $\exists\ \underline{i}$. Indeed, this put an additional burden on the quan-

[7] `http://www.graphviz.org/`

tifier instantiation routine of the module inst. It would be desirable to find ways to limit the growing number of existentially quantified variables in the prefix of the pre-image or, even better, to ensure it remains constant. To this end, the techniques described in [12] have been implemented in the module classify which is described below. The second problem in directly using (2) is that there is no guarantee it to be in primitive differentiated form. To avoid this problem, the module Pre performs a case-analysis according to the case-defined function update F so as to compute several primitive differentiated formulae whose disjunction is logically equivalent to (2). In order to reduce the number of inconsistent updates (i.e. unsatisfiable formulae) so as to reduce the number of formulae to be visited (i.e. added to TBV), a satisfiability check is required. While Yices can be invoked to do this, for efficiency reasons, it is better to minimize the number of calls to the available SMT solver. To this end, the module classify besides deciding how it is possible to control the grow of the existential prefix for each transition, it also derives useful information which allows for computationally inexpensive satisfiability checks (see below for more on this). If all the cheap tests are passed, Yices must indeed be invoked.

classify. As explained above, the goal of this module is to reduce the number of existentially quantified variables $\exists \underline{i}$ that should be added to $\exists \underline{k}$ in the prefix of (2). The idea to do this consists of trying to identify as many as possible variables in $\underline{i}$ with those already present in $\underline{k}$. This is justified by the observation that it is often the case that naively adding the extra variables in $\underline{i}$ yields primitive differentiated formulae which are immediately discharged by the successive fix-point test (line 9 of Figure 1(b)); thereby wasting computational resources. Under suitable hypotheses on the theories over indexes and elements, it is possible to predict whether this will happen before entering the main loop implementing backward reachability analysis as it depends on the form of the guards and the case-defined function updates used to define transitions (see [12] for details). Concretely, this is done by generating certain proof obligations for Yices, whose unsatisfiability imply that some or all of the additional variables in $\underline{i}$ can be identified with those in $\underline{k}$. For example, in the case of the Illinois protocol, classify establishes that transition `t2` may avoid to introduce a new (existentially quantified) variable while for `t1`, this is not the case and a new variables should be considered. The module classify is capable of establishing the behavior of transition with one and two existentially quantified variables and indicates whether 0, 1, or 2 new variables must be introduced by the module Pre. We call this technique *static instance reduction*. In the pseudo-code of Figure 1(b), we indicate that a transition τ have been analyzed by classify and the resulting information is attached to the transition using the label ℓ. Recall that τ^ℓ also contains additional information to allow for the cheap satisfiability tests mentioned above and used by Pre but also by inst (see below). To illustrate this point, consider an array variable a whose elements are an enumerated data-type. Then, τ^ℓ stores the value of a after the execution of the transition (since we perform a backward analysis, this is the only relevant value). In this way, to check for unsatisfiability, it is sufficient to search in a primitive differentiated formula, two literals of the form $a[i] = c$ and $a[i] = d$, for some index i, where c and d are two distinct 'values' of the enumerated data-type. We call this form of dynamic instance reduction as *filtering modulo enumerated data-types* and we will see another use of it for formulae describing sets of states below.

inst. As already said, one of the main bottle-necks in MCMT is the generation of instances for the checks of fix-point and non-empty intersection with the initial set of states. While the module classify attempts to *statically* control the number of instances of a universally quantified formula by reducing its number of universally quantified variables, our experience showed that this is not enough and we need also some technique to *dynamically* reduce the number of such instances. To this end, the key observation is that there is a large amount of instances which are useless in checking for unsatisfiability. To understand the problem, recall that the tool is required to check the satisfiability of the following kind of formulae:

$$\varphi(\underline{i}) \quad \wedge \quad \bigwedge\{\forall \underline{j}.\neg\psi(\underline{j}) \mid \psi \in \mathsf{AV} \cup \mathsf{TBV}\}$$

where φ and the ψ's are primitive differentiated formulae. Roughly, the algorithm implemented in inst consists of asserting φ in the actual Yices context and then incrementally enumerating all the instances of the ψ's. Indeed, before considering a new instance, a satisfiability check is performed. Now, consider again the case where there is an array variable a whose elements belong to an enumerated data-type. Assume that φ contains a literal of the form $a[i_k] = c$ (for $i_k \in \underline{i}$) and a ψ contains a literal of the form $a[j_l] = d$ (for $j_l \in \underline{j}$) such that c and d are two distinct values of the enumerated data-type. Then, it may happen—during the enumeration of the instances of a ψ—that j_l is instantiated to i_k and this implies that the instances under consideration are trivially satisfiable. To see this, rewrite φ as $a[i_k] = c \wedge \varphi'$ and the instance of ψ as $a[i_k] = d \wedge \psi'$ so as to obtain (after simple logical manipulations) the formula

$$(a[i_k] = c \wedge \varphi') \quad \wedge \quad (a[i_k] = d \rightarrow \neg\psi'),$$

which is trivially satisfiable, if φ is so. The module inst filters out useless instances by using this observation. This is another facet of the filtering modulo enumerated data-types technique applied to state formulae instead, as before, to transitions: also the primitive differentiated formulae produced by Pre are decorated with additional information such as the values in the arrays which range over enumerated data-types. Despite its simplicity, this heuristics dramatically reduces the numbers of instances which are passed to the SMT solver to check the satisfiability of a single universally quantified formula. To appreciate the importance of this heuristics, it is sufficient to observe that, to realize a fix-point check for a newly visited set of states (line 9 of Figure 1(b)), the module Minst is required to iterate the application of inst for each formula in the set AV of already visited states. Observe also that AV grows at each iteration of the backward analysis loop. Fortunately, our experience shows that few iterations in Minst are often sufficient and that we have a higher probability for an earlier detection of unsatisfiability when considering first those formulae which have been added to AV more recently. We call this form of dynamic instance reduction as *chronological fix-point checking*.

To conclude the description of the module inst, we mention two further heuristics. First, in the generate-and-check algorithm implemented in inst, the (incremental) satisfiability check is done after considering each new instance. It is possible to change the frequency of this test by using the option `-p`. Second, another heuristic belonging to the class of the dynamic instance reduction can be activated by the option `-f`. It consists of fixing the instantiation of a certain number of the universally quantified formulae after visiting a given number of sets of states. This makes the system incomplete but it may give significant speed-ups.

select. When describing the main loop of the tool in Figure 1(b), we claimed that the symbolic state space of the problem is explored breadth-first according to the order in which the user listed the transitions. This is not precise because of the function select used to extract the next set of states to be visited (line 12 of Figure 1(b)). In fact, along the lines of the dynamic instance reduction technique, we try to visit as a large part of the state space as possible by using first those formulae with fewer variables; it is possible to disable this heuristics by using the option `-b`.

simplify. This module performs some simple manipulations of the primitive differentiated formulae computed by Pre such as eliminating duplicated literals, orienting (according to some total ground ordering) equalities, or perform some simple normalization on arithmetic terms. It would be interesting to have some simplification routines used inside Yices available through the API so as to reuse the wealth of well-engineered techniques to simplify constraints for our tool and avoid to waste time duplicating functionalities.

4. EXPERIMENTS

MCMT has been implemented using the API light of Yices and release 0.2.1 can be downloaded at the following address `http://homes.dsi.unimi.it/~ghilardi/mcmt`.

As benchmarks, we have derived three sets of benchmarks for MCMT from the safety problems in [3, 2]: the first is of mutual exclusion protocols (with 7 problems, Table 1), the second is of cache coherence protocols (with 4 problems, Table 2), and the last is obtained as the counting abstraction of the previous problems (Table 3). We used the theory of finite linear orders as T_I for the first set of problems and the theory of equality as T_I for the other two. The theory T_E for the first two sets is the combination of an enumerated datatype theory for the control locations with theories for the data manipulated by the processes while for the third sets is Linear Arithmetics (over the Integers). The termination of the backward reachability analysis for some of these benchmarks specifying broadcast protocols and some versions of the mutual exclusion protocols (e.g., bakery) is guaranteed by applying results from [9]. In particular, it is possible to show termination when T_E is the theory of an enumerated datatype (i.e. each process in the parametrised system is modelled by a finite state automaton updating variables whose domains consist only of finitely many values). However, according to the experience in [7], even when termination is not guaranteed, we have obtained it in practice for all our benchmarks.

Columns 2-5 of the Tables report the statistics (timings are in seconds and obtained on a Pentium Dual-Core 3.4 GHz with 2 Gb Sdram) of the implementation of the main loop in Figure 1(b) while columns 6-9 show the results for the following variant (available in the latest release of the tool only). We have replaced the instruction at line 12 with the following block of code:

12_1 **do**
12_2 $\quad P \leftarrow \mathsf{select}(\mathsf{TBV});$
12_3 **while** $\mathsf{TBV} \neq \emptyset \wedge \mathsf{Minst}(\mathsf{AV} \cup \mathsf{TBV}, \mathsf{getinds}(P), P) = \mathsf{unsat};$

This is equivalent to check for fix-point a newly computed set of backward reachable states before computing its pre-images: if the test is positive (when the corresponding proof obligation is unsatisfiable), we can not only avoid to compute its pre-images but also eliminate it (recall that select has the side-effect of deleting the selected formula from its input set) so that its instances are no more considered when checking for fix-point or non-empty intersection with the set of initial states. (This technique is similar to the *backward redundancy elimination* in resolution-style theorem provers.) When the test for fix-point fails, the loop is exited and the primitive differentiated formula P starts the main loop in Figure 1(b).

Table 1 shows the usefulness of forward redundancy elimination as the seize of the problem grows. For the other two tables, the situation is less clear. However, we remark that all the cache coherence protocols that we have considered so far (except the German) are quite small and a brute force search of the tiny search space (see the column '#nodes') is likely to be more successful. Interestingly, there is some gain in using forward redundancy elimination on the last problem in this set (a difficult version of the German protocol, which is well-known to be a significant benchmark for verification tools).

Although a comparative analysis is somewhat difficult in lack of a standard for the specifications of safety problems, we report that MCMT performs comparably with the model checkers PFSand UNDIP on small to medium sized problems and outperforms them on larger instances.[8] These tools use a partially declarative approach by combining *ad hoc* algorithms to explore the symbolic state space of a system and constraint solvers for numerical data in order to reason on the elements manipulated by the processes. By comparing the statics of such systems with those of MCMT, it is apparent that a significantly smaller number of sets of states is visited during backward analysis. This is so because a single primitive differentiated formula is capable of representing a large amount of (minimal) configurations that a (partially) symbolic model checker, like PFS or UNDIP, is forced to enumerate. However, we must point that the completely symbolic representation of sets of states contains a lot of redundancy so that powerful heuristics are required to make our approach to scale up to large and interesting problems. The techniques for instance reductions described above can be seen as an important class of heuristics capable of handling this redundancy.

[8] `http://www.it.uu.se/research/docs/fm/apv/tools/{pfs,undip}`

	depth	#nodes	#calls	time	depth	#nodes	#calls	time
Bakery	9	29	221	**0.104**	9	28	252	0.124
Burns	14	57	497	0.216	14	57	429	**0.188**
Java M-lock	9	23	353	0.156	9	23	343	**0.152**
Dijkstra	13	40	392	0.148	13	38	256	**0.096**
Dijkstra (rv)	14	138	6905	5.756	14	127	4451	**3.324**
Szymanski	17	143	3266	2.208	17	136	3164	**2.036**
Szymanski (a)	23	2358	902017	24m19s	23	1745	475505	**11m19s**

Table 1: Mutual exclusion algorithms

	depth	#nodes	#calls	time	depth	#nodes	#calls	time
Mesi	3	2	175	0.032	3	2	177	**0.028**
Moesi	3	2	304	0.048	3	2	306	0.048
Illinois	4	8	998	**0.196**	4	8	1006	0.216
German	26	2985	322335	8m39s	26	2442	125060	**3m51s**

Table 2: Cache coherence protocols

	depth	#nodes	#calls	time	depth	#nodes	#calls	time
Mesi	3	2	18	**0.008**	3	2	22	0.012
Moesi	3	2	21	**0.016**	3	2	25	0.024
Illinois	3	4	82	0.036	3	3	56	**0.020**
German	9	13	46	0.028	9	13	63	0.028

Table 3: Cache coherence protocols: counting abstraction

Experimental results for a powerful heuristics for invariant generation on the same set of benchmarks are also reported in [11]. The command line option to active such a heuristics is `-i`.

5. DISCUSSION

We have described the architecture and the main loop of our model-checker MCMT based on a careful combination of SMT solving (Yices), a backward reachability algorithm, and heuristics to solve the proof obligations encoding the fix-point tests and the checks for non-empty intersection with the initial set of states.

We plan to extend MCMT in two directions. First, we intend to implement or to import a quantifier elimination module for the the variables ranging over the elements of the array (for some theories T_E admitting quantifier elimination, like for instance Linear Arithmetic and its fragments). Once this module is available, existentially quantified variables for elements can be used in the guards of the transitions: this will enable MCMT to deal, for example, with some of the benchmarks in [3]. Second, we want to support two-dimensional arrays to model communication channels between processes: this extension is crucial for MCMT to be able to treat—in a natural way—concurrent processes communicating over channels (see, e.g., [3] for examples of the verification of this kind of distributed systems).

6. REFERENCES

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.

[2] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*, volume 4424 of *LNCS*, pages 721–736, 2007.

[3] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, volume 4590 of *LNCS*, pages 145–157, 2007.

[4] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. of VMCAI*, volume 4905 of *LNCS*, pages 22–36, 2008.

[5] L. de Moura and N. Bjørner. Efficient e-matching for smt solvers. In *Proc. of CADE*, LNCS, 2007.

[6] D. Déharbe and S. Ranise. Satisfiability solving for software verification. *Int. Journal on STTT*, 2009. To appear.

[7] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. of CAV*, number 1855 in LNCS, 2000.

[8] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *Proc. of CADE-21*, LNCS, 2007.

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

[10] S. Ghilardi and S. Ranise. A Note on the Stopping Failures Models. 2009. Draft, available from MCMT distribution.

[11] S. Ghilardi and S. Ranise. Goal-Directed Invariant Synthesis in Model Checking Modulo Thoeries. In *Proc. of TABLEAUX 09*, LNCS, 2009. Full version available at `http://homes.dsi.unimi.it/~ghilardi/allegati/GhRa-RI325-09.pdf`.

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

[13] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.