Symbolic Bounded Conformance Checking of Model Programs*

Margus Veanes and Nikolaj Bjørner

Microsoft Research, Redmond, WA, USA
{margus,nbjorner}@microsoft.com

Abstract. Model programs are high-level behavioral specifications typically representing Abstract State Machines or ASMs. Conformance checking of model programs is the problem of deciding if the set of traces allowed by one model program forms a subset of the set of traces allowed by another model program. This is a foundational problem in the context of model-based testing, where one model program corresponds to an implementation and the other one to its specification. Here model programs are described using the ASM language AsmL. We assume a background $\mathcal{T}$ containing linear arithmetic, sets, and tuples. We introduce the Bounded Conformance Checking problem or BCC as a special case of the conformance checking problem when the length of traces is bounded and provide a mapping of BCC to a theorem proving problem in $\mathcal{T}$. BCC is shown to be highly undecidable in the general case but decidable for a class of model programs that are common in practice.

1 Introduction

We consider behavioral specifications given in the form of model programs. Model programs are mainly used to describe protocol-like behavior of software systems, and the underlying update semantics is based on ASMs [17]. However, model programs usually depend on additional parameters that are needed for executability. At Microsoft, model programs are used in the Spec Explorer tool in the Windows organization as an integral part of the *protocol quality assurance process* [16] for model-based testing of public application-level network protocols. A central problem in the context of model-based testing is to determine if an implementation *conforms* to a given specification, meaning that the traces that are observed from the implementation under test do not contradict the model. Traditionally, model-based testing is used at system-level, as a black-box testing technique where the implementation code is not visible to the tester. White-box testing on the other hand, is used at the unit-level by the developers of the code and is based on different techniques. Here we

* Microsoft Research Technical Report: MSR-TR-2009-28

assume that the implementation is also given or abstracted as a model program and consider the conformance checking problem as a theorem proving problem between the implementation and the model. The general conformance checking problem is very hard but can be approximated in various ways. One way is to bound the length of the traces, which leads to the Bounded Conformance Checking problem, or BCC, and is the topic of this paper.

Model programs typically assume a rich background universe including tuples (records) and sets, as well as user defined data structures. Moreover, unlike traditional sequential programs, model programs often operate on a more abstract level, for example, they use set comprehensions and parallel updates to compute a collection of elements in a single atomic step, rather than one element at a time, in a loop. The definition of model programs here extends the prior definitions to *nondeterministic* model programs, by allowing internal choices. Two model programs, written in AsmL [4, 18], are illustrated in Figure 1.

```
type Vertex = Integer
type Edge = (Vertex, Vertex)
IsSource(v as Vertex, E as Set of Edge) as Boolean
  return not exists e in E where Second(e) = v
Sources(E as Set of Edge) as Set of Vertex
  return {First(e) | e in E where IsSource(First(e),E)}
```

Model program P	Model program Q

`var E as Set of Edge`
`var V as Set of Vertex =`
`  {x,y|(x,y) in E}`
`[Action]`
`Step(v as Vertex)`
`  require v in V and IsSource(v,E)`
`  forall w in V`
`  remove (v,w) from E`
`  remove v from V`
	`var D as Set of Edge`
	`var S as Set of Vertex = Sources(D)`
	
	`[Action]`
	`Step(v as Vertex)`
	`  require S<>{} and v=Min(S)`
	`  D' = {e|e in D where First(e)<>v}`
	`  S := (S\{v}) union Sources(D')`
	`  D := D'`

Fig. 1. P specifies a topological sorting of a directed graph $G = (V, E)$ as follows. The *Step*-action of P requires that the vertex v has no incoming edges and removes all outgoing edges from v. Thus, starting from a given initial graph G with n vertices, a trace $Step(v_1), Step(v_2), \ldots, Step(v_n)$ is allowed in P if and only if $(v_1, v_2, \ldots, v_n)$ is a topological sorting of G. Similarly, the model program Q describes a particular implementation where during each step the vertex with minimum integer id is selected. As in ASMs, the top-level loop of a model program is implicit: while there exists an enabled action, one enabled action is chosen and executed.

$$
\begin{array}{ll}
T^{\sigma} & ::= x^{\sigma} \mid \mathit{Default}^{\sigma} \mid \mathit{Ite}(T^{\mathbb{B}}, T^{\sigma}, T^{\sigma}) \mid \mathit{TheElementOf}(T^{\mathbb{S}(\sigma)}) \mid \\
 & \quad \pi_i(T^{\sigma_0 \times \cdots \times \sigma_{i-1} \times \sigma \times \cdots \times \sigma_k}) \\
T^{\sigma_0 \times \sigma_1 \times \cdots \times \sigma_k} & ::= \langle T^{\sigma_0}, T^{\sigma_1}, \ldots, T^{\sigma_k} \rangle \\
T^{\mathbb{Z}} & ::= k \mid T^{\mathbb{Z}} + T^{\mathbb{Z}} \mid k * T^{\mathbb{Z}} \\
T^{\mathbb{B}} & ::= \mathit{true} \mid \mathit{false} \mid \neg T^{\mathbb{B}} \mid T^{\mathbb{B}} \wedge T^{\mathbb{B}} \mid T^{\mathbb{B}} \vee T^{\mathbb{B}} \mid T^{\mathbb{B}} \Rightarrow T^{\mathbb{B}} \mid \forall x\, T^{\mathbb{B}} \mid \exists x\, T^{\mathbb{B}} \mid \\
 & \quad T^{\sigma} = T^{\sigma} \mid T^{\mathbb{S}(\sigma)} \subseteq T^{\mathbb{S}(\sigma)} \mid T^{\sigma} \in T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{Z}} \leq T^{\mathbb{Z}} \\
T^{\mathbb{S}(\sigma)} & ::= \{T^{\sigma} \mid_{\bar{x}} T^{\mathbb{B}}\} \mid \emptyset^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cup T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cap T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \setminus T^{\mathbb{S}(\sigma)} \\
T^{\mathbb{A}} & ::= f^{(\sigma_0, \ldots, \sigma_{n-1})}(T^{\sigma_0}, \ldots, T^{\sigma_{n-1}})
\end{array}
$$

Fig. 2. Well-formed expressions in $\mathcal{T}$. Sorts are shown explicitly here. An expression of sort σ is written T^{σ}. The sorts $\mathbb{Z}$ and $\mathbb{B}$ are for integers and Booleans, respectively, k stands for any integer constant, x^{σ} is a variable of sort σ. The sorts $\mathbb{Z}$ and $\mathbb{B}$ are *basic*, so is the *tuple sort* $\sigma_0 \times \cdots \times \sigma_k$, provided that each σ_i is basic. The *set sort* $\mathbb{S}(\sigma)$ is not basic and requires σ to be basic. All quantified variables are required to have basic sorts. The sort $\mathbb{A}$ is called the *action sort*, $f^{(\sigma_0, \ldots, \sigma_{n-1})}$ stands for an *action symbol* with fixed arity n and argument sorts $\sigma_0, \ldots, \sigma_{n-1}$, where each argument sort is a set sort or a basic sort. The sort $\mathbb{A}$ is *not* basic. The only atomic relation that can be used for $T^{\mathbb{A}}$ is equality. $\mathit{Default}^{\mathbb{A}}$ is a nullary action symbol. Boolean expressions are also called *formulas* in the context of $\mathcal{T}$. In the paper, sort annotations are mostly omitted but are always assumed.

In Section 2 we define model programs. In Section 3 we define the problem of *bounded conformance checking* or *BCC* and show its reduction to a theorem proving problem in $\mathcal{T}$. Section 4 discusses the complexity of *BCC*. Section 5 is about related work.

2 Model programs

We consider a background $\mathcal{T}$ that includes linear arithmetic, Booleans, tuples, and sets. All values in $\mathcal{T}$ have a given *sort*. Well-formed expressions of $\mathcal{T}$ are shown in Figure 2. Each sort corresponds to a disjoint part of the universe. We do not add explicit sort annotations to symbols or expressions but always assume that all expression are well-sorted. A value is *basic* if it is either a Boolean, an integer, or a tuple of basic values.

The expression $\mathit{Ite}(\varphi, t_1, t_2)$ equals t_1 if φ is true, and it equals t_2, otherwise. For each sort, there is a specific *Default* value in the background. In particular, for Booleans the value is *false*, for set sorts the value is $\emptyset$, for integers the value is 0 and for tuples the value is the tuple of defaults of the respective tuple elements.

The function *TheElementOf* maps every singleton set to the element in that set and maps every other set to *Default*. Note that *extensionality* of sets: $\forall v\, w\, (\forall y(y \in v \leftrightarrow y \in w) \rightarrow v = w)$, allows us to use set comprehensions as terms: the *comprehension term* $\{t(\bar{x}) \mid_{\bar{x}} \varphi(\bar{x})\}$ represents the set such that $\forall y(y \in \{t(\bar{x}) \mid_{\bar{x}} \varphi(\bar{x})\} \leftrightarrow \exists \bar{x}(t(\bar{x}) = y \wedge \varphi(\bar{x})))$. We make use of explicit definitions in terms of $\mathcal{T}$ such as *Min* (used in Figure 1), that returns the minimum element from a set of integers, or 0 when the set is empty,

$$Min(X) \stackrel{\text{def}}{=} TheElementOf(\{y \mid y \in X \wedge \forall\, z(z \in X \Rightarrow y \leq z)\}).$$

In the general case, model programs also use *maps*. We assume a standard representation of maps as function graphs, maps are needed to represent dynamic ASM functions, see [7], maps are not used in the current paper.

Actions. There is a specific *action sort* $\mathbb{A}$, values of this sort are called *actions* and have the form $f(v_0, \ldots, v_{\text{arity}(f)-1})$. $Default^{\mathbb{A}}$ has arity 0. Two actions are equal if and only if they have the same action symbol and their corresponding arguments are equal. An action $f(\bar{v})$ is called an *f-action*. Every action symbol f with arity $n > 0$, is associated with a unique *parameter variable* f_i for all i, $0 \leq i < n$.[1]

Choice variables. A *choice variable* is a variable[2] χ that is associated with a formula $\exists x \varphi[x]$, called the *range condition* of χ, denoted by $\chi^{\exists x \varphi[x]}$. The following axiom is assumed to hold for each choice variable:

$$IsChoice(\chi^{\exists x \varphi}) \stackrel{\text{def}}{=} (\exists\, x\, \varphi[x]) \Rightarrow \varphi[\chi^{\exists x \varphi}]). \tag{1}$$

In the general case, the sort of χ may be non-basic and χ is a map (a Skolem function), in which case the range condition must hold for the elements in the range of the map, see [7].

Model programs. The following definition extends the former definition of model programs by allowing nondeterminism through *choice variables*. An *assignment* is a pair $x := t$ where x is a variable and t is a term (both having the same sort). An *update rule* is a finite set of assignments where the assigned variables are distinct.

Definition 1 (Model Program). A *model program* is a tuple $P = (\Sigma, \Gamma, \varphi^0, R)$, where

[1] In AsmL one can of course use any formal parameter name, such as v in Figure 1, following standard conventions for method signatures.

[2] Pronounced "chi".

- Σ is a finite set of variables called *state variables*;
- Γ is a finite set of *action symbols*;
- φ^0 is a formula called the *initial state condition*;
- R is a collection $\{R_f\}_{f\in\Gamma}$ of *action rules* $R_f = (\gamma, U, X)$, where
  - γ is a formula called the *guard of* f;
  - U is an update rule $\{x := t_x\}_{x\in\Sigma_f}$ for some $\Sigma_f \subseteq \Sigma$, U is called the *update rule of* f,
  - X is a set of *choice variables of* f

  All unbound variables that occur in an action rule, including the range conditions of choice variables, must either be state variables, parameter variables, or choice variables of the action. The sets of parameter variables, state variables and choice variables must be disjoint.

Intuitively, choice variables are "hidden" parameter variables, the range condition of a choice variable determines the valid range for its values. For parameter variables, the range conditions are typically part of the guard. We often say *action* to also mean an action rule or an action symbol, if the intent is clear from the context. The case when all parameter variables and choice variables of a model program are basic is an important special case when symbolic analysis becomes feasible, which motivates the following definition.[3]

Definition 2 (Basic Model Programs). An update rule is *basic* if all parameter variables and choice variables that occur in it are basic. An action rule is *basic* if its update rule is basic. A model program is *basic* if its action rules are basic and the initial state condition implies that all nonbasic state variables are empty sets.

Representing standard ASMs as model programs. Standard *ASM update rules* can be translated into update rules of model programs. A detailed translation from standard ASMs to model programs is given in [7]. Intuitively, a forall-statement (such as the one used in Figure 1) translates into a comprehension expression, and each choose-statement introduces a new choice variable. An important property of the translation is that, if choose statements are not allowed to occur inside forall statements in the ASM update rules, then the translation yields a basic model program. When a choose-statement is nested inside a forall-statement, the resulting model program will depend on a non-basic choice variable or a *choice function* (Skolem function). In the general case, the translation also adds

[3] The standard notion of basic ASMs is more restrictive, in particular model programs allow unbounded exploration, quantifiers may be unbounded.

an additional state variable that indicates collisions of updates and in this way captures "error" states. We assume here that update rules of actions in a model program correspond to ASM update rules where some choice variables occur as parameters of the action, in which case their range conditions are typically part of the guard.

States. A *state* is a mapping of variables to values. Given a state S and an expression E, where S maps all the free variables in E to values, E^S is the *evaluation of E in S*. Given a state S and a formula φ, $S \models \varphi$ means that φ is true in S. A formula φ is *valid* (in $\mathcal{T}$) if φ is true in all states. *Since $\mathcal{T}$ is assumed to be the background theory we usually omit it, and assume that each state also has an implicit part that satisfies $\mathcal{T}$, e.g. that $+$ means addition and $\cup$ means set union.* In the following let P be a fixed model program.

Definition 3. Let a be an action $f(v_0, \ldots, v_{n-1})$ and S a state. A *choice expansion of S for a* is an expansion S' of $S \cup \{f_i \mapsto v_i\}_{i<n}$ with choice variables of f.

Definition 4. An f-action a is *enabled* in a state S if there exists a choice expansion of S for a that satisfies the guard of f.

Definition 5. An f-action a *causes a transition* from a state S_1 to a state S_2, if a is enabled in S_1, S_1' is a choice expansion of S_1 that satisfies the guard of a, for each assignment $x := t$ of f, $x^{S_2} = t^{S_1'}$, and for any other state variable x, $x^{S_2} = x^{S_1}$.

Example 1. Let P be the model program in Figure 1. The set of initial states of $[\![P]\!]$ includes for example the state $S_0 = \{V \mapsto \{1,2,3\}, E \mapsto \{\langle 1,2\rangle, \langle 2,3\rangle\}\}$. The action *Step*(1) is enabled in S_0 because $S_0 \cup \{v \mapsto 1\} \models v \in V \wedge \neg\exists w(w \in V \wedge \langle w, v\rangle \in E)$. The action *Step*(1) causes a transition from S_0 to $S_1 = \{V \mapsto \{2,3\}, E \mapsto \{\langle 2,3\rangle\}\}$. ⊠

A *labeled transition system* or *LTS* is a tuple $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}$ is a set of *states*, $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of *initial states*, L is a set of labels and $T \subseteq \mathcal{S} \times L \times \mathcal{S}$ is a *transition relation*.

Definition 6. Let $P = (\Sigma, \Gamma, \varphi^0, R)$ be a model program. The *LTS of P*, denoted by $[\![P]\!]$ is the LTS $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}_0 = \{S \mid S \models \varphi_0\}$; L is the set of all actions over Γ; T and $\mathcal{S}$ are the least sets such that, $\mathcal{S}_0 \subseteq \mathcal{S}$, and if $S \in \mathcal{S}$ and there is an action a that causes a transition from S to S' then $S' \in \mathcal{S}$ and $(S, a, S') \in T$.

Definition 7. A model program P is *deterministic* if forall transitions (S, a, S_1) and (S, a, S_2) in $[\![P]\!]$, $S_1 = S_2$.

Clearly, any model program without choice variables is deterministic.

Definition 8. A *run* of P is a sequence of transitions $(S_i, a_i, S_{i+1})_{i<\kappa}$ in $[\![P]\!]$, for some $\kappa \leq \omega$, where S_0 is an initial state of $[\![P]\!]$. The sequence $(a_i)_{i<\kappa}$ is called an (*action*) (κ-)*trace* of P.

3 Symbolic Bounded Conformance Checking

We are now ready to define the central problem of the paper in Definition 10. Let P and Q be fixed model programs with the same set of action symbols. Let $k \geq 0$ be a fixed bound. We assume here that P and Q have initial state conditions that require that all the state variables are initially equal to *Default*. Under this assumption, we drop the initial state condition from the definition. This assumption is needed in order to avoid tedious special cases, when for example the initial conditions are false, etc. Note that, by adding an additional initialization action, any values can be assigned to the state variables.

Definition 9. *Q k-conforms to P*, $Q \sqsubseteq_k P$, if for all $l \leq k$, all l-traces of Q are l-traces of P. *Q conforms to P*, $Q \sqsubseteq P$, if $Q \sqsubseteq_k P$ for all k.

If $Q \sqsubseteq_k P$, then P is more liberal by allowing more traces up to length k. Intuitively, when P is a specification model program and Q is an implementation model program and $Q \sqsubseteq_k P$, then Q behaves as expected by P within k steps. Conformance testing is an approximation of k-conformance up to some k, where k depends on the maximum length of the test cases. In the more general case, when one distinguishes between *observable* and *controllable* actions in the context of asynchronous systems, one needs to consider a more general form of conformance notion, such as alternating refinement [2] or ioco [26], that is outside the scope of this paper. Note that a most general model program is one where all actions have an empty update rule and all guards are *true*, such a model program is trivially conformed to by any other model program

Example 2. Let P and Q be the model programs in Figure 1. Assume that there is an additional *Init*-action in both P and Q that first initializes the state variables to a concrete graph G and then enables the *Step*-action. One can show that $Q \sqsubseteq_k P$ for all k and thus $Q \sqsubseteq P$. In this particular case, if one shows that, for all input graphs with k vertices $Q \sqsubseteq_{k+1} P$, then $Q \sqsubseteq P$ follows. Note also that $P \not\sqsubseteq_2 Q$. ⊠

Definition 10 (BCC). *Bounded Conformance Checking* or *BCC* is the problem of deciding if $Q \sqsubseteq_k P$.

In order to reduce BCC into a theorem proving problem, we construct a special formula from given P, Q and k, as defined in Definition 11. Given an expression E and a step number $i > 0$, we write $E[i]$ below for a copy of E where each (unbound) variable x in E has been uniquely renamed to a variable $x[i]$. We assume also that $E[0]$ is E.

Definition 11 (Bounded Conformance Formula). Let P and Q be model programs $(\overline{x_\star}, \Gamma, (\gamma_{f,\star}, U_{f,\star}, X_{f,\star})_{f\in\Gamma})$, for $\star = P, Q$. Assume that $\overline{x_Q} \cap \overline{x_P} = \emptyset$ and that the choice variables in P and Q are disjoint.[4] Assume also that each action rule includes an assignment for all the state variables.[5] The *bounded conformance formula* for P, Q, and k is:

$$BCF(Q,P,k) \stackrel{\text{def}}{=} (\overline{x_Q} = \overline{\mathit{Default}} \wedge \overline{x_P} = \overline{\mathit{Default}}) \Rightarrow \mathit{Conforms}(0,k)$$

$$\mathit{Conforms}(k,k) \stackrel{\text{def}}{=} \mathit{true}$$

$$(i<k)\ \mathit{Conforms}(i,k) \stackrel{\text{def}}{=} \bigwedge_{f\in\Gamma} (\forall\, \overline{f_j[i]}\, \overline{\chi_{f,Q}[i]} (\gamma_{f,Q}[i] \wedge \overline{\mathit{IsChoice}(\chi_{f,Q}[i])} \Rightarrow$$
$$\exists\, \overline{\chi_{f,P}[i]} (\gamma_{f,P}[i] \wedge \overline{\mathit{IsChoice}(\chi_{f,P}[i])} \wedge$$
$$\Big(\bigwedge_{x:=t_x \in U_{f,Q}\cup U_{f,P}} x[i+1] = t_x[i]$$
$$\Rightarrow \mathit{Conforms}(i+1,k)))))$$

where $\overline{f_j[i]} = f_0[i] \ldots f_{\mathrm{arity}(f)-1}[i]$ are the parameter variables of action f for step i (the parameter variables of f are shared between P and Q), and $\overline{\chi_{f,P}[i]}$ and $\overline{\chi_{f,Q}[i]}$ are the choice variables of f in P and Q, respectively, for step i.

Notice that all parameter variables, and choice variables have distinct names in each step. This implies that all oracles and parameters are local to a single step, and do not carry over from one step to the next. The only connection between the steps happens via the state variables. Note also that if both P and Q are deterministic, then the resulting formula is essentially a universal formula. If P has choice variables then the bounded conformance formula has a k-depth quantifier alternation.

The following theorem allows us to check k-conformance by proving that the corresponding bounded conformance formula is valid in $\mathcal{T}$.

[4] Alternatively rename those variables in Q for example.
[5] Add an assignment $x := x$ for each state variable x that is not assigned.

Theorem 1. *$BCF(Q, P, k)$ is valid in $\mathcal{T}$ if and only if $Q \sqsubseteq_k P$.*

Proof (Sketch). For $k = 0$ the statement holds trivially. Assume $k > 0$. Both directions are proved separately. For the direction ($\Longrightarrow$) we assume that $Q \not\sqsubseteq_k P$ and get a shortest run of length $l \leq k$ where the last action is enabled in Q but not in P. From the run we can construct a state where $\neg BCF(Q, P, l)$ is true. Note that if $\neg BCF(Q, P, l)$ is satisfiable then so is $\neg BCF(Q, P, l')$, for $l' > l$. The proof of the direction ($\Longleftarrow$) is similar. $\boxtimes$

4 Complexity of BCC

Here we look at the complexity of BCC. First we note that the problem is effectively equivalent to the validity problem of formulas in second-order Peano arithmetic with sets (Π^1_1-complete). This implies that there exists no refutationally complete procedure for checking k-conformance in general (even for $k = 1$). Second, we note that, even if we restrict the background universe to *finite* sets, the problem is still undecidable, by being co-re-complete. Third, we show that BCC is decidable over *basic* model programs. The reason for this is that for basic model programs, the set variables can be eliminated, and the problem reduces to Presburger arithmetic.

Undecidability of BCC. We use the result that the validity problem of formulas in Presburger arithmetic with unary relations is Π^1_1-complete [1, 19]. The Π^1_1-hardness part is an immediate consequence of the results in [1, 19], by considering model programs that have one action with a set-valued parameter and a linear arithmetic formula as the guard. The inclusion in Π^1_1 can be shown similarly to the proof of the Σ^1_1-completeness of the BMPC problem in [7].

Corollary 1. *BCC is Π^1_1-complete.*

Now suppose that the sets in the background are finite and consider the satisfiability problem in $\mathcal{T}$ over finite sets that is re-complete [7].

Corollary 2. *BCC over finite sets is co-re-complete.*

Decidability of BCC over basic model programs. Basic model programs are common in practical applications. The two main reasons for this are: 1) actions typically only use parameters that have basic sorts, see for example the Credits model in [31]. 2) the initial state is usually required to have fixed initial values or default values for all the state variables.

Let $\mathcal{T}^0$ stand for the fragment of $\mathcal{T}$ where all variables are basic. We use decidability of $\mathcal{T}^0$, that follows as a special case from the decision procedure for $\mathcal{T}^{\prec}$ in [7], that is by reduction to linear arithmetic.

Theorem 2. *BCC of basic model programs is decidable.*

Proof (Sketch). Let P and Q be basic model programs and k a step bound. Let $\psi = BCF(Q, P, k)$. The subformula $\bigwedge_{x\in\Sigma} x[i+1] = t_x[i] \Rightarrow Conforms(i{+}1, k)$ of ψ is equivalent to the formula $Conforms(i{+}1, k)\{x[i{+}1] \mapsto t_x[i] \mid x \in \Sigma\}$ where $x[i+1]$ has been replaced by $t_x[i]$. Apply this transformation successively to eliminate each occurrence of $x[i+1]$ for $i < k$. Finally, eliminate each (initial) state variable by replacing it with the default value. The resulting formula, say φ, is equivalent to ψ and does not use any state variables. Moreover, since P and Q are basic, φ is in $\mathcal{T}^0$. The statement follows from Theorem 1 and decidability of $\mathcal{T}^0$. ⊠

It is possible to carry out the reduction in Theorem 2 in polynomial time in the size ψ. First, the formula ψ is translated into logic without sets but with unary relations, by replacing set variables with unary relations and by eliminating set comprehensions and set operations in the usual way, e.g., $t \in S$, where S is a set variable, becomes the atom $R_S(t)$, where R_S is a unary relation symbol. It is easy to show by induction over expressions that such a translation can be done in polynomial time in the size of ψ and preserves the structure of ψ.

We iterate the following transformation on the resulting formula, say ψ_i, starting with $i = k$, repeating the transformation for $i := i - 1$, until $i = 0$. For ease of exposition assume also that there is a single set valued state variable S.

The formula ψ_i has a subformula of the form (2) where $\mathbf{Q}\bar{y}\rho$ is assumed to be on Prenex form so that ρ is quantifier free,

$$\forall x(R_{i+1}(x) \Leftrightarrow \varphi[x]) \Rightarrow \mathbf{Q}\bar{y}\rho[R_{i+1}(t_1), \ldots, R_{i+1}(t_n)] \tag{2}$$

where R_{i+1} corresponds to the value of S at step $i+1$ and φ as well as each t_j may only contain values of S from step i. The formula (2) is equivalent to (3) where we may assume that $\bar{y}$ do not occur free in φ.

$$\mathbf{Q}\bar{y}(\forall x(R_{i+1}(x) \Leftrightarrow \varphi[x]) \Rightarrow \rho[R_{i+1}(t_1), \ldots, R_{i+1}(t_n)]) \tag{3}$$

The formula (3) is equivalent to (4) (where $\bar{z}$ are Boolean).

$$\mathbf{Q}\bar{y} \forall \bar{z}(\underbrace{(\bigwedge_{j=1}^{n} z_j \Leftrightarrow \varphi[t_j])}_{\delta} \Rightarrow \rho[z_1, \ldots, z_n]) \tag{4}$$

Formula δ is equivalent to (5) by using the encoding in [15, p 129],

$$\forall\, x\, \forall\, w\, (\underbrace{(\bigvee_{j=1}^{n} (x = t_j \wedge w = z_j))}_{\Phi[w \Leftrightarrow \varphi]} \Rightarrow (w \Leftrightarrow \varphi[x])). \tag{5}$$

Now consider the formula $\Phi[w \Leftrightarrow \varphi]$, where φ is $\mathbf{Q}\bar{u}\gamma[\bar{u}]$ in Prenex form. The formula $\Phi[w \Leftrightarrow \varphi]$ is equivalent to

$$\mathbf{Q}\bar{u}\mathbf{Q}^c\bar{u}'\Phi[(w \wedge \gamma[\bar{u}]) \vee (\neg w \wedge \neg\gamma[\bar{u}'])] \tag{6}$$

where $\mathbf{Q}^c$ is the complement of quantifier prefix $\mathbf{Q}$; (6) is equivalent to

$$\mathbf{Q}\bar{u}\mathbf{Q}^c\bar{u}'\forall b\forall b'(\underbrace{(\gamma[\bar{u}] \Leftrightarrow b \wedge \gamma[\bar{u}'] \Leftrightarrow b')}_{\gamma'} \Rightarrow \Phi[(w \wedge b) \vee (\neg w \wedge \neg b')]) \tag{7}$$

Using the same encoding from [15] as above, γ' is equivalent to

$$\forall\bar{v}\forall d(((\bar{v} = \bar{u} \wedge d = b) \vee (\bar{v} = \bar{u}' \wedge d = b')) \Rightarrow (d \Leftrightarrow \gamma[\bar{v}])) \tag{8}$$

Combining the above equivalences, it follows that (2) is equivalent to

$$\mathbf{Q}\ldots((8) \Rightarrow \Phi[(w \wedge b) \vee (\neg w \wedge \neg b')]) \Rightarrow \rho[\bar{z}]) \tag{9}$$

The reduction from (2) to (9) shows that no t_j or φ needs to be duplicated and clearly the Prenex form of (9) has the same size as (9). The formula (2) is replaced in ψ_i with (9) to get ψ_{i-1}.

Finally, recall that the initial values of set variables are empty sets, which means that $\forall x\,(R_0(x) \Leftrightarrow \mathit{false})$, so each occurrence of an atom $R_0(t)$ is replaced in ψ_0 with *false*.

The above reduction can also be carried out in a more general setting, independent of the background theory, by first introducing auxiliary predicates that define all the subformulas of ψ, by applying a transformation similar to [27] or [24], and then eliminating the predicates (as a form of deskolemization) by equivalence preserving transformations similar to the transformations shown above.

The overall reduction shows that the computational complexity of BCC of basic model programs, regarding both the lower and the upper bound, is the same as that of Presburger arithmetic, stated here as a corollary of the above reduction and [15].

Corollary 3. *The upper bound of the computational complexity of BCC of basic model programs is* $2^{2^{2^{cn}}}$ *and the lower bound is* $2^{2^{cn}}$*, where* c *is a constant and* n *is the size of the input* (P, Q, k) *for BCC.*

5 Related work

The bounded model program checking problem or BMPC [7, 28, 30] is a bounded path exploration problem of a given model program. BMPC is a generalization of bounded model checking to model programs. The technique of bounded model checking by using SAT solving was introduced in [5] and the extension to SMT was introduced in [14], a related approach is described in [3]. BMPC reduces to satisfiability modulo $\mathcal{T}$. Unlike BCC, the resulting formula for a BMPC problem is typically existential with no quantifier alternation, even for nondeterministic model programs, since choice variables and parameter variables are treated equally. BMPC is therefore better suited for analysis using the SMT approach. General reachability problems for transition systems as theorem proving problems are also discussed in [25].

Formulating a state refinement relation between two symbolic transition systems as a theorem proving problem, where one system describes an implementation and the other one its specification, has a long standing in automatic verification of hardware, with seminal work done in [11] for verifying control properties of pipelined microprocessors. In particular the work generated interest in the use of uninterpreted functions for hardware verification problems [10]. Refinement techniques related to ASMs are discussed in [8]. Traditionally, such techniques are based on state transitions, rather than action traces and use untyped ASMs; the main motivation is incremental system design. Various refinement problems between specifications are also the topic of many analysis tools, where sets and maps are used as foundational data structures, such as RAISE, Z, TLA+, B, see [6]. The ASM method is also described in [6]. In some cases, like in RAISE, the underlying logic is three-valued in order to deal with undefined values in specifications. In many of those formalisms, frame conditions need to be specified explicitly, and are not implicit as in the case of model programs or ASMs. In Alloy [20], the analysis is reduced to SAT, by finitizing the data types. A file system case study of a refinement problem using Alloy is discussed in [22]. In our case the analysis is reduced to a theorem proving problem in $\mathcal{T}$, by restricting the search depth rather than the size of the data types.

As future and ongoing work, we use the state of the art SMT solver Z3 [13] for our experiments on satisfiability problems in $\mathcal{T}$. Our current experiments use a lazy quantifier instantiation scheme that is on one hand not limited to basic model programs, but is on the other hand also not complete for basic model programs, some of the implementation

aspects are discussed in [31] in the context of BMPC. In particular, the scheme discussed in [31] is inspired by [9], and extends it by using model checking to implement an efficient incremental saturation procedure on top of Z3. The saturation procedure is similar to CEGAR [12], the main difference is that we do not refine the level of abstraction, but instead lazily instantiate axioms in case their use has not been triggered during proof search. Implementation of the reduction of BCC of basic model programs to linear arithmetic is future work. In that context the reduction to Z3 does not need to complete all the reductions to linear arithmetic, but can take advantage of built-in support for *Ite* terms, sets, and tuples.

Model programs are used as high-level specifications in model-based testing tools such as Spec Explorer [29] and NModel [23]. In Spec Explorer, one of the supported input languages is the abstract state machine language AsmL [17, 18]. In that context, sanity checking or validation of model programs is usually achieved through simulation and explicit state model checking and search techniques [21, 29].

References

1. R. Alur and T. A. Henzinger. A really temporal logic. In *Proc. 30th Symp. on Foundations of Computer Science*, pages 164–169, 1989.
2. R. Alur, T. A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.
3. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In A. Valmari, editor, *SPIN*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.
4. AsmL. http://research.microsoft.com/fse/AsmL/.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
6. D. Bjørner and M. Henson, editors. *Logics of Specification Languages*. Springer, 2008.
7. N. Bjørner, Y. Gurevich, W. Schulte, and M. Veanes. Symbolic bounded model checking of abstract state machines. Technical Report MSR-TR-2009-14, Microsoft Research, February 2009. Submitted to IJSI.
8. E. Börger and R. F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
9. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation: 7^{th} International Conference, (VMCAI'06)*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
10. R. E. Bryant, S. M. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 470–482. Springer, 1999.

11. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Conference on Computer-Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
12. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
13. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, LNCS. Springer, 2008.
14. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction (CADE'02)*, volume 2392 of *LNCS*, pages 438–455. Springer, 2002.
15. M. J. Fisher and M. O. Rabin. Super-exponential complexity of presburger arithmetic. In B. F. Caviness and J. R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 122–135. Springer, 1998. Reprint from *SIAM-AMS Proceedings*, Vol VII, 1974, pp. 27-41.
16. W. Grieskamp, D. MacDonald, N. Kicillof, A. Nandan, K. Stobie, and F. Wurden. Model-based quality assurance of Windows protocol documentation. In *First International Conference on Software Testing, Verification and Validation, ICST*, Lillehammer, Norway, April 2008.
17. Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, 1995.
18. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
19. J. Y. Halpern. Presburger arithmetic with unary predicates is Π^1_1 complete. *Journal of Symbolic Logic*, 56:637–642, 1991.
20. D. Jackson. *Software Abstractions*. MIT Press, 2006.
21. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
22. E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *ABZ*, volume 5238 of *LNCS*, pages 294–308. Springer, 2008.
23. NModel. http://www.codeplex.com/NModel, public version released May 2008.
24. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.
25. T. Rybina and A. Voronkov. A logical reconstruction of reachability. In M. Broy and A. Zamulin, editors, *PSI 2003*, volume 2890 of *LNCS*, pages 222–237. Springer, 2003.
26. J. Tretmans. Model based testing with labelled transition systems. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.
27. G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.
28. M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *FORTE'08*, volume 5048 of *LNCS*, pages 53–68. Springer, 2008.
29. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.

30. M. Veanes and A. Saabas. On bounded reachability of programs with set comprehensions. In *LPAR'08*, volume 5330 of *LNAI*, pages 305–317. Springer, 2008.
31. M. Veanes, A. Saabas, and N. Bjørner. Bounded reachability of model programs. Technical Report MSR-TR-2008-81, Microsoft Research, May 2008.