Int J Software Informatics, Vol.3, No.2-3, June/September 2009, pp. 149–170 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Symbolic Bounded Model Checking of Abstract State Machines

Margus Veanes, Nikolaj Bjørner, Yuri Gurevich, and Wolfram Schulte

(Microsoft Research, Redmond, WA, USA)

Abstract Abstract State Machines (ASMs) allow modeling system behaviors at any desired level of abstraction, including a level with rich data types, such as sets, sequences, maps, and user-defined data types. The availability of high-level data types allow state elements to be represented both abstractly and faithfully at the same time.

In this paper we look at symbolic analysis of ASMs. We consider ASMs over a fixed state background $\mathcal{T}$ that includes linear arithmetic, sets, tuples, and maps. For symbolic analysis, ASMs are translated into guarded update systems called model programs. We formulate the problem of bounded path exploration of model programs, or the problem of Bounded Model Program Checking (BMPC) as a satisfiability problem modulo $\mathcal{T}$. Then we investigate the boundaries of decidable and undecidable cases for BMPC. In a general setting, BMPC is shown to be highly undecidable (Σ_1^1-complete); and even when restricting to finite sets the problem remains re-hard (Σ_1^0-hard). On the other hand, BMPC is shown to be decidable for a class of *basic* model programs that are common in practice.

We use Satisfiability Modulo Theories (SMT) for solving BMPC; an instance of the BMPC problem is mapped to a formula, the formula is satisfiable modulo $\mathcal{T}$ if and only if the corresponding BMPC problem instance has a solution. The recent SMT advances allow us to directly analyze specifications using sets and maps with specialized decision procedures for expressive fragments of these theories. Our approach is extensible; background theories need in fact only be partially solved by the SMT solver; we use simulation of ASMs to support additional theories that are beyond the scope of available decision procedures.

Key words: model programs; abstract state machines; satisability modulo theories; model checking

Veanes M, Bjørner N, Gurevich Y, Schulte W. Symbolic bounded model checking of abstract state machines. *Int J Software Informatics*, **2009, 3(2-3): 149–170. http://www.ijsi.org/1673-7288/3/149.htm**

1 Introduction

We look at behavioral specifications given in the form of abstract state machines[19] (ASMs). For symbolic analysis, we transform an ASM specification into a canonical form and represent it as a model program. Intuitively, a model program is a collection of guarded parallel assignments, the formal definition is given below. Model programs have mainly been used in the context of model based testing. A key feature provided

* Corresponding author: Margus Veanes, Email: margus@microsoft.com
Manuscript received 2009-02-06; revised 2009-07-10; accepted 2009-07-15.

by model based testing tools such as Spec Explorer[36] and NModel[26] is that model programs can be represented in different languages, including AsmL[4] and C#, and a model program can use a library of data types including sets, sequences, and maps, as well as user defined data types to describe a state. At Microsoft, model programs are used as an integral part of the *protocol quality assurance process*[18] for model-based testing of public application-level network protocols. Correctness assumptions about a model can be expressed through (state) invariants. A state where an invariant is violated is *unsafe*. A part of the model validation process is *safety analysis*, which aims at identifying unsafe states that are reachable from the initial state. Safety analysis of model programs is undecidable in general but can be approximated in various ways. One way is to bound the number of steps from the initial state, which leads to bounded model checking of model programs or *bounded model program checking* and is the topic of this paper.

Model programs typically rely on a rich background universe including sequences, tuples (records), sets and bags (multisets), as well as user defined data structures. Moreover, unlike traditional sequential programs, model programs often compute on a more abstract level, for example, they use set comprehensions to compute a collection of elements in a single atomic step, rather than one element at a time, in a loop. Set comprehensions can be used to specify infinite sets and may encode unbounded or even an infinite number of updates.

The satisfiability modulo theories (SMT) based symbolic bounded model checking of model programs is introduced in Ref. [35]. The use of SMT solvers for automatic software analysis was introduced in Ref. [14] as an extension of SAT-based bounded model checking[5]. The SMT based approach is better suited for dealing with more complex background theories, such as linear arithmetic; instead of encoding the verification task as a *propositional* formula the task is encoded as a *quantifier free* formula. The decision procedure for checking the satisfiability of the formula may use combinations of background theories[29]. Unlike for sequential programs, bounded model checking of model programs is undecidable[37]. Here we sharpen the undecidability result, by showing that it is Σ^1_1-complete, and we also provide an algorithm for a decidable fragment. The definition of model programs here extends the prior definitions by introducing the concept of choice variables that allow specifications of nondeterministic ASMs. A typical model program is illustrated in Example 1.

Example 1 *The below model program, named* Topsort*, is written in AsmL. It has two state variables V and E and it has one unary action called Step with an integer parameter v. The guard of the action requires that v has not been visited (v is still in V) and there are no edges in E that enter v. The update rule of the action removes all edges from E that exit v and removes v from V.*

```
var V as Set of Integer
var E as Set of (Integer,Integer)
IsSource(v as Integer) as Boolean
  return not exists w in V where (w,v) in E

[Action] Step(v as Integer)
  require v in V and IsSource(v)
```

```
E := E - {(v,w) | w in V}
V := V - {v}
```

This model program specifies all traces of Step-actions that describe a topological sorting of a directed acyclic graph $G = (V, E)$. Starting from a given initial state where G is a specific graph with n vertices, an action trace $Step(v_1), Step(v_2), \ldots, Step(v_n)$ exists if and only if $(v_1, v_2, \ldots, v_n)$ is a topological sorting of G. Note that if G is cyclic, then no such action trace exists; for example if $G = (\{1, 2\}, \{(1, 2), (2, 1)\})$ in the initial state, then neither $Step(1)$, nor $Step(2)$ is enabled. □

The rest of the paper is structured as follows. In Section 2 we define model programs formally, we provide a translation of standard ASMs into model programs, and we define the problem of *bounded model program checking* or *BMPC*. In Section 3 we prove two undecidability results: in the general case BMPC is Σ^1_1-complete, and in the case when sets are required to be finite, BMPC is *re*-complete. In Section 4 we show decidability and describe an algorithm of BMPC for a restricted class of model programs that are common in practice. Section 5 is about related work.

2 Model Programs and Bounded Model Program Checking

The notions that are automatically available in the context of a computation are often referred to as the *background* [8]. This includes not only the elements or values themselves, like integers, but also operations on them, like addition. In this paper we assume a background that is *multi-sorted*, where each element has a fixed sort, thus any two elements that have distinct sorts are distinct. There is a sort $\mathbb{Z}$ for integers, there is a sort $\mathbb{B}$ for Booleans, given sorts $\sigma_0, \sigma_1, \ldots, \sigma_{k-1}$, there is a ($k$-)*tuple sort* $\sigma_0 \times \sigma_1 \times \cdots \times \sigma_{k-1}$ for $k > 0$. A sort is *basic* if it is either $\mathbb{Z}$, $\mathbb{B}$, or a tuple sort of basic sorts. For every basic sort σ, there is also a (σ-)*set sort* $\mathbb{S}(\sigma)$ that is not a basic sort, we do not consider nested sets (sets including other sets as elements) in this paper. An element is *basic* if it has a basic sort. The background is named $\mathcal{T}$. Well-formed expressions of $\mathcal{T}$ are shown in Figure 1. We do not add explicit sort annotations to symbols or expressions but always assume that all expression are well-sorted.

The interpretation of the arithmetical operations, Boolean operations, tuple operations, and set operations shown in Figure 1 is standard. The expression $Ite(\varphi, t_1, t_2)$ equals t_1 if φ is true, and it equals t_2, otherwise. For each sort, there is a specific *Default* value. 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 the *extensionality axiom* for 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* $\{t(\vec{x}) \mid_{\vec{x}} \varphi(\vec{x})\}$ represents a set such that

$$\forall y(y \in \{t(\vec{x}) \mid_{\vec{x}} \varphi(\vec{x})\} \leftrightarrow \exists \vec{x}(t(\vec{x}) = y \wedge \varphi(\vec{x})))$$

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

Figure 1. 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.

2.1 Maps

The definitions provided here show how the expressions in $\mathcal{T}$ are used to define map operations. These definitions are needed in later sections to provide a translation from ASMs to model programs, and to provide the axioms that support the symbolic analysis.

We assume a standard representation of maps as function graphs. A *map* $m = \{k_i \mapsto v_i\}_{i<\kappa}$ is represented as a set of *key-value* pairs $\{\langle k_i, v_i \rangle\}_{i<\kappa}$. Updating a map m with a key-value pair $\langle k, v \rangle$ produces a new map that is the same as m except that k maps to v.

$$\mathit{Update}(m, k, v) \stackrel{\text{def}}{=} \{e \mid e \in m \wedge \pi_0(e) \neq k\} \cup \mathit{Ite}(v = \mathit{Default}, \emptyset, \{\langle k, v \rangle\})$$

Given a set u of tuples, we write $\pi_i(u)$ for $\{\pi_i(x) \mid x \in u\}$. The definition of $\mathit{Update}(m, u)$, where u is a set of key-value pairs (where no key occurs twice, but where some value may be *Default*) is analogous:

$$\mathit{Update}(m, u) \stackrel{\text{def}}{=} \{e \mid e \in m \wedge \pi_0(e) \notin \pi_0(u)\} \cup \{e \mid e \in u \wedge \pi_1(u) \neq \mathit{Default}\}$$

Lookup of a value based on a key is defined as follows.

$$\mathit{Lookup}(m, k) \stackrel{\text{def}}{=} \mathit{TheElementOf}(\{v \mid \langle k, v \rangle \in m\})$$

We also write $m(k)$ as a shorthand for $\mathit{Lookup}(m, k)$. Note that maps are extensional, since keys that are mapped to *Default* are removed from the map, i.e. given two maps

m_1 and m_2:

$$m_1 = m_2 \Leftrightarrow \forall k(m_1(k) = m_2(k)).$$

A map m is *finite* if $m(k) = Default$ for all but finitely many k. In most discussions below maps are assumed to be finite.

2.2 Actions

There is a specific *action sort* $\mathbb{A}$, values of this sort are called *actions* and have the form $f(v_0, \ldots, v_{n-1})$, where f has arity n and each v_i has some expected argument sort σ_i. $Default^{\mathbb{A}}$ is a nullary function symbol. Two actions are equal if and only if they have the same action symbol and their corresponding arguments are equal. An action with action symbol f is called an *f-action*.

If $n > 0$ then we assume that f is associated with a unique variable f_i of sort σ_i, for each i, $0 \leqslant i < n$, called the *i'th parameter variable* of f. In AsmL one can of course use any formal parameter name (such as v in Example 1, following standard conventions for method signatures).

2.3 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 *A model program is a tuple $P = (\Sigma, \Gamma, \varphi^0, R)$, where*

- 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 variables, disjoint from Σ, called* choice variables of f*, each $\chi \in X$ is associated with a formula $\exists x\varphi[x]$, called the* range condition *of χ, denoted by $\chi^{\exists x\varphi[x]}$.*

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

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. If the choice variable is nonbasic, it is assumed to be a map (possibly infinite), and the range condition must hold for the elements in the range of that map.

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 variables of a model program are basic is an important special case when symbolic analysis becomes feasible, which motivates the following definition.

Definition 2 *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 all state variables have the initial value Default.*

2.4 Representing standard ASMs as model programs

We show here how *ASM update rules* can be represented as update rules of model programs. We consider an ASM with dynamic functions Σ with positive arities (nullary dynamic functions are handled similarly) and construct a model program where the dynamic functions are map valued state variables. The model program has an additional Boolean state variable *Inconsistent*, the purpose of this variable is to represent states resulting from an inconsistent state update. The kinds of ASM update rules we are covering here are *skip*, *dynamic function update*, *if-then-else*, *do-in-parallel*, *choose* and *forall*. The ASM update rules are shown below in the definition of $\mathbf{u}$; $\mathbf{u}$ is defined by induction over the structure of ASM update rules. Given $g \in \Sigma$ and an ASM update rule U, we define a translation $\mathbf{u}(U, g)$ to an expression that represents the state updates produced by U to update g. The different kinds of update rules are shown below in the definition of $\mathbf{u}$.

We assume that g is unary, generalization to arbitrary arities is trivial by using tuples. Each occurrence of a choose statement in U has a unique position identifier p, we write $choose_p$ below to indicate that id. For each $choose_p$ in U there is a distinct choice variable named χ_p, also called an *oracle*. If a choice is made in the context of a forall statement, the corresponding oracle is a set, and is queried for a Boolean answer that determines which of the two branches of the choose statement is to be selected. If a choice is not context dependent, i.e., the choose statement is not in the scope of any forall statements, then the corresponding oracle is a Boolean valued choice variable. The list of context variables $\vec{y}$ is propagated down in the translation, as the third argument of $\mathbf{u}$. When $\vec{y}$ is empty then the expression $\vec{y} \in \chi_p$ below stands for the expression $\chi_p = true$.

$$
\begin{aligned}
\mathbf{u}(U, g) &\stackrel{\text{def}}{=} \mathbf{u}(U, g, ()) \\
\mathbf{u}(U, g, \vec{y}) &\stackrel{\text{def}}{=} \emptyset, \quad \text{if } U \text{ does not contain } g \\
\mathbf{u}(g(t) := u, g, \vec{y}) &\stackrel{\text{def}}{=} \{\langle t, u\rangle\} \\
\mathbf{u}(\textit{if } \varphi \textit{ then } U_1 \textit{ else } U_2, g, \vec{y}) &\stackrel{\text{def}}{=} \mathit{Ite}(\varphi, \mathbf{u}(U_1, g, \vec{y}), \mathbf{u}(U_2, g, \vec{y})) \\
\mathbf{u}(\textit{do-in-parallel } U_1\ U_2, g, \vec{y}) &\stackrel{\text{def}}{=} \mathbf{u}(U_1, g, \vec{y}) \cup \mathbf{u}(U_2, g, \vec{y}) \\
\mathbf{u}(\textit{forall } x \textit{ where } \varphi[x] \textit{ do } U[x], g, \vec{y}) &\stackrel{\text{def}}{=} \{u \mid_u \exists\, x(\varphi[x] \wedge u \in \mathbf{u}(U[x], g, (x, \vec{y})))\} \\
\mathbf{u}(\textit{choose}_p\ U_1\ U_2, g, \vec{y}) &\stackrel{\text{def}}{=} \mathit{Ite}(\vec{y} \in \chi_p, \mathbf{u}(U_1, g, \vec{y}), \mathbf{u}(U_2, g, \vec{y}))
\end{aligned}
$$

A *conditional choose* statement needs a more powerful oracle. In the following translation we assume that $\chi^{\exists x \varphi}$ is a choice function that satisfies the condition

$IsOracle(\chi^{\exists x\varphi})$:

$$IsOracle(\chi^{\exists x\varphi}) \stackrel{\text{def}}{=} \forall\, \vec{y}((\exists\, x\, \varphi[x]) \Rightarrow \varphi[\chi^{\exists x\varphi}(\vec{y})]), \tag{1.1}$$

where $\vec{y}$ is the list of context variables of the choose statement. If $\vec{y}$ is empty then $\chi^{\exists x\varphi}$ is a witness for x in $\exists x\varphi$. The translation is:

$$\mathbf{u}(\mathit{choose}_p\ x\ \mathit{where}\ \varphi[x]\ \mathit{do}\ U[x], g, \vec{y}) \stackrel{\text{def}}{=} \mathit{Ite}(\exists x\varphi[x], \mathbf{u}(U[\chi_p^{\exists x\varphi}(\vec{y})], g, \vec{y}), \emptyset).$$

An update set U for a dynamic function g is *inconsistent* if it includes two pairs with the same key but a different value:

$$\begin{aligned} \mathit{IsInconsistent}(U, g) &\stackrel{\text{def}}{=} \exists\, x\, y\, z(\langle x, y\rangle \in \mathbf{u}(U, g) \wedge \langle x, z\rangle \in \mathbf{u}(U, g) \wedge y \neq z) \\ \mathit{IsInconsistent}(U) &\stackrel{\text{def}}{=} \bigvee_{g\in\Sigma} \mathit{IsInconsistent}(U, g) \end{aligned}$$

Definition 3 *The canonical model program with action Step for an ASM has the following components. Let U and Σ be as above. The set of state variables of P is $\Sigma \cup \{\mathit{Inconsistent}\}$. The initial state condition of P is $\bigwedge_{g\in\Sigma} g = \emptyset \wedge \mathit{Inconsistent} = \mathit{false}$. The guard of Step is true. The update rule of Step has an assignment to detect inconsistent updates, as well as an assignment for each dynamic function $g \in \Sigma$:*

$$\begin{aligned} \mathit{Inconsistent} &:= \mathit{IsInconsistent}(U) \\ g &:= \mathit{Ite}(\mathit{IsInconsistent}(U), g, \mathit{Update}(g, \mathbf{u}(U, g))) \end{aligned}$$

Each choice variable χ_p^φ is a choice variable of the Step action.

The following examples illustrate the translation from standard ASMs to model programs. Example 2 illustrates a case when the resulting update rule is not basic. This case arises when choose statements are nested inside forall statements. Example 3 illustrates that forall statements can be nested within choose statements, but the resulting update rule is still basic. Example 4 illustrates an ASM that has a potential update inconsistency within one step.

Example 2 *This is an example of a an ASM update rule U that removes some elements from the domain of a dynamic function g. We use AsmL. The AsmL expression x `in` g stands for the condition $g(x) \neq \mathit{Default}$. The syntax for an update $g(x) := \mathit{Default}$ is in AsmL `remove` x `from` g.*[*]

```
forall x in g
  choose y in {true,false}
    if y then remove x from g else skip
```

The expression $\mathbf{u}(U, g)$ is as follows, where the oracle χ (we omit the corresponding choose statement identifier) is consulted regarding which elements are to be removed.

$$\mathbf{u}(U, g) = \{u \mid_u \exists\, x\, (g(x) \neq \mathit{Default} \wedge u \in \mathit{Ite}(x \in \chi, \{\langle x, \mathit{Default}\rangle\}, \emptyset))\}$$

Note that the condition $\exists y(y \in \{\mathit{true}, \mathit{false}\})$ is trivially true, moreover, the oracle $\chi = \chi^{\exists y(y=\mathit{true}\vee y=\mathit{false})}$ is in this case a set (as if we used the unconditional choose statement of a standard ASM). □

[*] AsmL does not allow direct access to *Default*.

Example 3 *Consider the update rule of the Step action in Example 1. Let us hide the parameter of the action by turning it into an internal choice and let us write the update rule using standard ASM style. Note that the body of the conditional choose statement below is a parallel block, i.e., choose refers to x rather than the statements in the body. Let U be:*

```
choose x in V where IsSource(x)
  remove x from V
  forall y in V
    remove (x,y) from E
```

Let φ be the formula

$$\exists x(V(x) = \mathit{true} \wedge \mathit{IsSource}(x)).$$

We get the following expressions for $\mathbf{u}(U, E)$ and $\mathbf{u}(U, V)$:

$$\begin{array}{rcl} \mathbf{u}(U,E) & = & \mathit{Ite}(\varphi, \emptyset \cup \{u \mid_u \exists\, y\, (V(y) = \mathit{true} \wedge u \in \{\langle\langle \chi^{\varphi}, y\rangle, \mathit{Default}\rangle\})\}, \emptyset) \\ & = & \mathit{Ite}(\varphi, \{\langle\langle \chi^{\varphi}, y\rangle, \mathit{Default}\rangle \mid_y V(y) = \mathit{true}\}, \emptyset) \\ \mathbf{u}(U,V) & = & \mathit{Ite}(\varphi, \{\langle \chi^{\varphi}, \mathit{Default}\rangle\}, \emptyset) \end{array}$$

Note that the choice variable χ^{φ} is the same variable in both $\mathbf{u}(U, E)$ and $\mathbf{u}(U, V)$. □

Example 4 *Consider the following AsmL update rule U that is a parallel block of two choose statements, where g is a dynamic function from integers to integers.*

```
choose x in {1,2} g(x) := 1
choose x in {1,2} g(x) := 2
```

The choose statements give rise to two distinct choice variables $\chi_1^{\varphi_1}$ and $\chi_2^{\varphi_2}$ (of sort $\mathbb{Z}$), where both range conditions are $\exists x(x \in \{1, 2\})$. The translation $\mathbf{u}(U, g)$ is

$$\mathit{Ite}(\varphi_1, \{\langle \chi_1^{\varphi_1}, 1\rangle\}, \emptyset) \cup \mathit{Ite}(\varphi_2, \{\langle \chi_2^{\varphi_2}, 2\rangle\}, \emptyset)$$

which can be simplified to $\{\langle \chi_1^{\varphi_1}, 1\rangle, \langle \chi_2^{\varphi_2}, 2\rangle\}$, because φ_1 and φ_2 are trivially true. If we wish to analyze U to see if there are potential inconsistent updates, we can see that the condition $\mathit{IsInconsistent}(\mathbf{u}(U, g))$ is indeed satisfiable for example for $\chi_1 = \chi_2 = 1$. Note however, that this does not mean that U is invalid as an update rule, because the canonical model program for U includes the inconsistency checking.

Let us also consider the following modification U' of U where the second choose statement is in the body of the first choose statement.

```
choose x in {1,2}
  g(x) := 1
  choose y in {1,2} difference {x}
    g(y) := 2
```

In this case φ_1 is the same as before and φ_2 is $\exists x(x \in \{1, 2\} \setminus \{\chi_1\})$ The translation $\mathbf{u}(U', g)$ is (in simplified form) $\{\langle \chi_1^{\varphi_1}, 1\rangle, \langle \chi_2^{\varphi_2}, 2\rangle\}$ but $\mathit{IsInconsistent}(\mathbf{u}(U', g))$ is unsatisfiable in this case. □

In most cases $\mathit{IsInconsistent}(U)$ can never true, in which case the inconsistency checking can be eliminated. One way to show this is to over approximate by showing that $\mathit{IsInconsistent}(U)$ is unsatisfiable, for example, this is the case for the update rule in Example 3. Note however, that $\mathit{IsInconsistent}(U)$ may be satisfiable but the corresponding state is never-the-less not reachable from the initial state, we revisit this point later.

In the general case, we assume 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. For the most part of the paper we are concerned with *basic* model programs.[†]

With AsmL one can either use the style of standard ASMs, or write more in the style of Definition 1, as in Example 1, where state variables are assigned with *total updates* and update inconsistency checking is not needed. We often use the latter style in modeling. This allows us to omit the update inconsistency checking and we can think of such AsmL models as direct representations of model programs.

AsmL allows also a mixture of both modeling styles. This is supported by the theory of partial updates [21], which is outside the scope of this paper.

Typically, in a model program written in AsmL the initial state is given by the initializers of the state variables. In the model program in Example 1, however, the initializers have been omitted and the initial state condition is therefore unconstrained or *true*. Notice also that Definition 1 allows unbounded set comprehensions, that are not allowed in AsmL.

2.5 States

A *state* is a mapping of variables to values. Given a state S and an expression E, 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.

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 definitions we assume a fixed model program P.

Definition 4 *Let a be an action $f(v_0, \ldots, v_{n-1})$. A choice expansion of a state S for a is an expansion S' of $S \cup \{f_i \mapsto v_i\}_{i<n}$ with choice variables of f, such that S' satisfies (1) for each choice variable $\chi^{\exists x \varphi[x]}$ of f.*

Note that if an action is parameterless and has no choice variables then $S = S'$ above. Note also that in the case of a nonbasic update rule, there is at least one choice function that is infinite, unless the corresponding range condition is false.

Definition 5 *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 6 *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, $x^{S_2} = x^{S_1}$.*

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

Example 5 *Let P be the model program in Example 1. The set of initial states of $[\![P]\!]$ is the set of all states of $[\![P]\!]$, in particular the state*

$$S_0 = \{V \mapsto \{1,2,3\}, E \mapsto \{\langle 1,2\rangle, \langle 2,3\rangle\}\}$$

is a possible initial state. 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

$$S_1 = \{V \mapsto \{2,3\}, E \mapsto \{\langle 2,3\rangle\}\}.$$

□

2.6 Runs and traces

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 7 *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 8 *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.

Example 6 *The model program in Example 1 is deterministic. The following action on the other hand, has an internal choice. The update rule is essentially the same as illustrated in Example 3. The action Step is parameterless and v is a choice variable. The range condition associated with v is also the guard of the action. Note that if the guard is omitted then the action is enabled in all states but produces no updates when the range condition of v is false.*

```
[Action] Step()
  require exists v in V where IsSource(v)
  choose v in V where IsSource(v)
    E := E - {(v,w) | w in V}
    V := V - {v}
```

□

Definition 9 *A run of P is a sequence of transitions $(S_i, a_i, S_{i+1})_{i<\kappa}$ in $[\![P]\!]$, for some $\kappa \leqslant \omega$, where S_0 is an initial state of $[\![P]\!]$. The sequence $(a_i)_{i<\kappa}$ is called an (*action*) trace of P. The run or the trace is* finite *if $\kappa < \omega$.*

Example 7 *Let P and S_1 be as in Example 5. $(Step(1), Step(2))$ is an action trace of P from S_0, whereas $(Step(1), Step(3))$ is not, because $Step(2)$ is enabled in S_1 but $Step(3)$ is not enabled in S_1.* □

2.7 Bounded model program checking

We are now ready to formulate the central problem of this paper in Definition 11. We first need the following additional definition.

Definition 10 *Given a condition φ, a bound $k \geqslant 0$ and a state S_0, φ is reachable in P from S_0 within k steps if the following holds:*

- *S_0 is an initial state of P,*
- *there is an $l \leqslant k$ and a run $(S_i, a_i, S_{i+1})_{i<l}$ of P, such that $S_l \models \varphi$.*

The trace $\alpha = (a_i)_{i<l}$, when it exists, is called a trace for φ from S_0.

Example 8 *Let P and S_0 be as in Example 5. Then $V = \emptyset$ is reachable in P from S_0 within 3 steps, $(Step(1), Step(2), Step(3))$ is a trace for $V = \emptyset$ from S_0.* □

Definition 11 (BMPC) *Bounded Model Program Checking or BMPC is the problem of deciding, given inputs P, k, and φ as above, if φ is reachable in P from some initial state of P within k steps.*

In order to reduce BMPC into a theorem proving problem, we construct a special formula from given P, k, and φ, as defined in Definition 12. Given an expression E and a step number $i > 0$, we write $E[i]$ below for a copy of E where each variable x in E has been uniquely renamed to a variable $x[i]$.[‡] We assume also that $E[0]$ is E. For each step number i, there is an additional variable $action[i]$ of sort $\mathbb{A}$ that records the selected action for step i. Recall the definition of $IsOracle(\chi)$ (see Equation (1)).

Definition 12 (Bounded reachability formula) *Let P be a model program $(\Sigma, \Gamma, \varphi^0, (\gamma_f, U_f, X_f)_{f\in\Gamma})$. The* bounded reachability formula *for P, step bound k and reachability condition φ is:*

$$
\begin{aligned}
BRF(P,\varphi,k) &\stackrel{def}{=} \varphi^0 \wedge \Big(\bigwedge_{0\leqslant i<k} STEP(P,i)\Big) \wedge \Big(\bigvee_{0\leqslant i\leqslant k} \varphi[i]\Big)\\
STEP(P,i) &\stackrel{def}{=} \bigvee_{f\in\Gamma} \Big(action[i] = f(f_0[i],\ldots,f_{arity(f)-1}[i])\\
&\qquad \wedge \gamma_f[i] \bigwedge_{\chi\in X_f} IsOracle(\chi)[i]\\
&\qquad \bigwedge_{x:=t\,\in U_f} x[i+1] = t[i] \bigwedge_{x\in\Sigma,\ x:=\_\notin U_f} x[i+1] = x[i]\Big)
\end{aligned}
$$

Notice that all parameter variables, and choice variables have distinct names in each formula $STEP(P,i)$. This implies that all oracles and parameters are local to a single step, and do not carry over from one step to the next.

[‡] Bound variables need not be renamed.

Example 9 *Let P be the Topsort model program, let φ be $V = \emptyset$ and let k be 2. Then*

$$\begin{aligned} BRF(P,\varphi,k) &= true \wedge P_0 \wedge P_1 \wedge (V[0] = \emptyset \vee V[1] = \emptyset \vee V[2] = \emptyset), \\ where\ P_i &= action[i] = Step(v[i]) \wedge \\ &\quad v[i] \in V[i] \wedge \neg(\exists w\,(w \in V \wedge \langle w, v[i]\rangle \in E[i])) \wedge \\ &\quad V[i+1] = V[i] \setminus \{v[i]\} \wedge \\ &\quad E[i+1] = E[i] \setminus \{\langle v[i], w\rangle \mid w \in V[i]\}. \end{aligned}$$

□

Some model programs may reach states where no action is enabled. For example the *Topsort* model program in Example 1 is such, because the *Step* action is not enabled in a state where $V = \emptyset$. The construction in Definition 12 will not directly work for such model programs, we want the model program to be *total* in the following sense.

Definition 13 *A model program P is* total *if for all states S of $[\![P]\!]$, there is a transition (S, a, S') in $[\![P]\!]$ for some S'.*

There is a trivial transformation that one can use to transform every model program into a total model program: add a new action to it whose enabling condition is true and whose update rule is empty. The following theorem enables us to turn the BMPC problem into a satisfiability problem. We write $S \restriction \Sigma$ below for $\{x \mapsto x^S\}_{x\in\Sigma}$.

Theorem 1 *Given P, k and φ as above, where P is total,*

1. *$BRF(P,\varphi,k)$ is satisfiable if and only if φ is reachable in P within k steps.*
2. *If $S \models BRF(P,\varphi,k)$ then there is $l \leqslant k$ such that $(action[i]^S)_{i<l}$ is a trace for φ from $S \restriction \Sigma$.*

Proof. (2): Assume $S \models BRF(P,\varphi,k)$. From the definition of $BRF(P,\varphi,k)$ follows that $S \models \varphi^0$, $S \models STEP(P,i)$ for $0 \leqslant i \leqslant k$, and $S \models \varphi[l]$ for some $l \leqslant k$, fix the value of l. Let $S_0 = S \restriction \Sigma$. So S_0 is an initial state of $[\![P]\!]$. If $l = 0$ the statement follows trivially. If $l > 0$, let S'_i, for $i \leqslant l$, be the reduction of S to all the variables with step number i, and construct S_i from S'_i by erasing the step numbers on variables. In particular $S_l \models \varphi$. By induction on i, $i < l$, and by using the definition of $STEP(P,i)$, it follows that (S_i, a_i, S_{i+1}) is a transition in $[\![P]\!]$ where $a_i = action[i]^S$. So there is an action trace $(a_i)_{i<l}$ for φ from S_0.
(1($\Rightarrow$)): Follows from (2) and Definition 10.
(1($\Leftarrow$)): Assume φ is reachable in P within k steps. So there is a run $(S_i, a_i, S_{i+1})_{i<l}$ of P, for some $l \leqslant k$, where $S_0 \models \varphi^0$ and $S_l \models \varphi$. Since P is total, we can extend the run with additional transitions (S_i, a_i, S_{i+1}) for $l < i < k$. Moreover, for each transition $\tau_i = (S_i, a_i, S_{i+1})$, for $i < k$, there is a choice expansion S'_i of S_i for a_i that provides values for the choice variables and parameter variables of a_i that are needed to show that τ_i is a transition in $[\![P]\!]$. Let S'_k be S_k. Construct a state S''_i from S'_i, for $i \leqslant k$, by annotating all variables with the step number i. Let

$$S = \bigcup_{i<k} (S''_i \cup \{action[i] \mapsto a_i\}) \cup S''_k.$$

It is straightforward to show that $S \models BRF(P, \varphi, k)$. □

2.8 Inconsistency checking of ASMs

We illustrate how BMPC can be used for inconsistency checking of ASMs. Consider the canonical model program P for an ASM (see Definition 3). Let k be a step bound. If the formula $BRF(P, \mathit{Inconsistent}, k)$ is satisfiable then there exists a run of the ASM of length at most k from the initial state such that the final step produces an inconsistent update set.

If there are some choice variables in P, those can be made into parameters of the *Step* action. The resulting action trace is then a witness for the inconsistency. In particular, this problem is decidable for *basic* model programs as shown in Section 4.

3 Undecidability of BMPC

The BMPC problem is highly undecidable in general. In this section we show undecidability of some minimal cases of the problem. First we show that the problem is recursively equivalent to the halting problem of Turing machines (Σ^0_1-complete, or *re*-complete) for a small class of *nonbasic* model programs when sets in the background are restricted to be *finite*.

Next we show that the problem is recursively equivalent to the satisfiability problem of formulas in second-order Peano arithmetic with sets (Σ^1_1-complete) and remains Σ^1_1-hard for a very restricted fragment, even when the background sets are restricted to be *recursive*. This result has important implications for program analysis techniques that use the SMT approach, where it is very often the case that the background is assumed to include $\mathcal{T}$ or an equivalent variation of it, and a model is sought that requires infinite sets or maps. It shows that there exists no refutationally complete procedure for $\mathcal{T}$, i.e., there exists no semi-decision procedure that allows one to check for unsatisfiability of formulas in $\mathcal{T}$. Moreover, this holds for a very restricted fragment of $\mathcal{T}$.

3.1 Σ^0_1-completeness of BMPC over finite sets

We assume here that the sets in the background are finite. Under this assumption, it is easy to see that BMPC is in Σ^0_1, because models are finite (including all the oracles) and can therefore be systematically enumerated and checked for satisfiability of the given formula. Next we prove Σ^0_1-hardness for a restricted fragment.

In the proof we use a reduction from the halting problem of 2-register machines. A *2-register* machine has two registers that contain positive integers or 0. It has a finite set of instructions, and can check if a register contains 0. In one step it can increment by one or decrement by one, any of the register values. A 2-register machines *halts* if it reaches a certain final instruction. The *halting problem of 2-register machines* is, given a 2-register machine M and initial values (m, n) for its two register, to decide if M halts. This problem is Σ^0_1-complete, i.e., it is (recursively) equivalent the the halting problem of Turing machines. It is very easy to define a quantifier free formula in $\mathcal{T}$ that describes one step $x \vdash_M y$ where x and y are *configurations* of M, of the form

$$\langle \mathit{instruction}, \mathit{value\ in\ register\ 1}, \mathit{value\ in\ register\ 2} \rangle.$$

For example we can use the definition from [9, Theorem 2.1.15]. Let $STEP_M(x, y)$ be

a formula such that $STEP_M(x, y)$ holds if and only if M makes the step $x \vdash_M y$.

We can assume, without loss of generality, that M is such that the initial instruction is 1 and the final instruction is k and when the final instruction is reached then both registers are zero. Let $halts_M$ be the following formula (where X is a variable with the sort $\mathbb{S}(\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}))$ and l is a variable of sort $\mathbb{Z}$). The construction of $halts_M$ is based on the idea of *shifted pairing*[22], see Fig. 2.

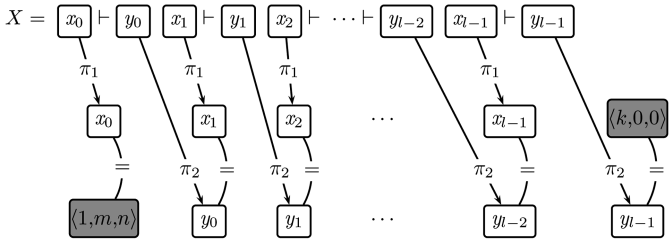

Figure 2. *Shifted pairing*; used in the definition of $halts_M$

$$
\begin{aligned}
&halts_M(m,n) \stackrel{\text{def}}{=} \\
&\qquad X = \{\langle j,x,y\rangle \mid \langle j,x,y\rangle \in X \wedge STEP_M(x,y) \wedge 0 \leqslant j < l\} \wedge \\
&\qquad \{\langle \pi_0(z), \pi_1(z)\rangle \mid z \in X\} \cup \{\langle l, \langle k,0,0\rangle\rangle\} = \\
&\qquad \{\langle 0, \langle 1,m,n\rangle\rangle\} \cup \{\langle \pi_0(z)+1, \pi_2(z)\rangle \mid z \in X\}.
\end{aligned}
$$

Theorem 2 *Given M, m and n as above, the formula $halts_M(m,n)$ is satisfiable if and only if M halts with initial register values (m,n).*

Proof. ($\Leftarrow$): Assume that M halts on (m,n). So there is a finite sequence of configurations $(x_j)_{j \leqslant l}$ where $x_0 = \langle 1,m,n\rangle$, $x_l = \langle k,0,0\rangle$ and $x_j \vdash_M x_{j+1}$ for $j < l$. Let $X = \{\langle j, x_j, x_{j+1}\rangle | 0 \leqslant j < l\}$. It is easy to check that $halts_M(m,n)$ is true for the given X and l.
($\Rightarrow$): Assume that $halts_M(m,n)$ is true for some X and l. From the first equality it follows that all elements of X are of the form $\langle j,x,y\rangle$ where $j < l$ and $x \vdash_M y$. From the second equality it follows that X must be a sequence in j, i.e., for all $j < l$, there is a unique element $\langle j, x_j, y_j\rangle$ in X. Moreover, it follows that $x_0 = \langle 1,m,n\rangle$, and for all j, $0 < j < l$, $x_j = y_{j-1}$, and $y_l = \langle k,0,0\rangle$. Hence, there exists a computation $\langle 1,m,n\rangle \vdash_M^* \langle k,0,0\rangle$ and therefore M halts on (m,n). □

Theorem 2 implies the Σ_1^0-hardness result because the halting problem of 2-register machines is Σ_1^0-hard, and shows also that three comprehensions of tuples and one nonbasic oracle are enough. One can further sharpen the result.

Let M be a 2-register machine that is *universal* in the following sense: given a Turing machine T and an input v, let $\ulcorner T, v \urcorner$ be an effective encoding of T and v as an input for M, so that M accepts $\ulcorner T, v \urcorner$ if and only if T accepts v. Let P be a model program with a state variable x of sort $\mathbb{Z} \times \mathbb{Z}$ with initial value $\langle 0,0\rangle$, one action $Halts()$ whose guard is $halts_M$ where l, X, m and n are all choice variables, and whose update rule is $x := \langle m,n\rangle$. In this case P is also *universal* in the following sense: $BRF(P, x = \ulcorner T, v \urcorner, 1)$ is satisfiable iff T accepts v, showing that both the model program as well as the bound can be fixed and the formula must be a single equality, but BMPC remains Σ_1^0-hard under these restrictions.

3.2 Σ^1_1-completeness of BMPC

Here we consider the general case of BMPC. Intuitively, Σ^1_1 corresponds to second-order Peano arithmetic with unary relations or sets. See Ref. [32] for a precise definition of the analytical hierarchy, including Σ^1_1. For Σ^1_1-hardness we can use the following theorem.*

> **Halpern**[23]**.** *The satisfiability problem of formulas in Presburger arithmetic with one unary relation is Σ^1_1-complete.*

This result for Presburger arithmetic with more than one set variable is also shown in Ref. [1]. The following is an immediate consequence of Halpern's Theorem, for example by using a Presburger formula with one set valued choice variable as an action guard.

Corollary 1 *BMPC is Σ^1_1-hard for a single step and with a single set-valued choice variable.*

We give a more direct proof of Corollary 1 by using the recurrence problem of Turing machines. A Turing machine T *recurs* if, starting from the empty tape, T visits its initial state infinitely often. The *recurrence problem* of Turing machines is the problem of deciding if a Turing machine recurs. The following result is also used in Ref. [23].

> **Harel-Pnueli-Stavi**[24]**.** *The recurrence problem of Turing machines is Σ^1_1-complete.*

Harel-Pnueli-Stavi Theorem holds also for 2-register machines, since one can effectively transform a Turing machine T into a 2-register machine M such that the latter "mimics" the computations of T.

Theorem 3 *BMPC is Σ^1_1-complete. Moreover the problem remains Σ^1_1-hard with a single choice variable and when sets are recursive.*

Proof: We reduce the recurrence problem of 2-register machines to BMPC using an infinite version of the shifted pairing idea that is illustrated in Figure 3, to describe infinite computations. Let M be a 2-register machine and define (note that the final configuration and the length restrictions are omitted in this case):

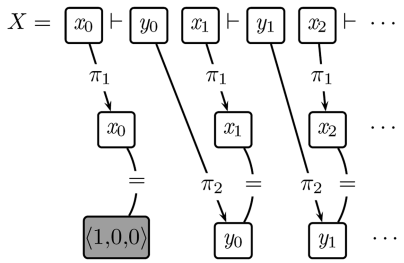

Figure 3. *Infinite shifted pairing*; used in the definition of $recurs_M$

*The problem in Ref. [23] is stated in terms of *validity* which is Π^1_1-complete.

$$
\begin{aligned}
\mathit{recurs}_M \quad \stackrel{\text{def}}{=} \quad & X = \{\langle j, x, y\rangle \mid \langle j, x, y\rangle \in X \wedge \mathit{STEP}_M(x, y)\} \wedge \\
& \{\langle \pi_0(z), \pi_1(z)\rangle \mid z \in X\} = \\
& \{\langle 0, \langle 1, 0, 0\rangle\rangle\} \cup \{\langle \pi_0(z) + 1, \pi_2(z)\rangle \mid z \in X\} \wedge \\
& \forall x(x \in X \Rightarrow \exists y(y \in X \wedge \pi_0(x) < \pi_0(y) \wedge \\
& \qquad \pi_0(\pi_1(x)) = \pi_0(\pi_1(y)) = 1)).
\end{aligned}
$$

The last conjunct of recurs_M states that the instruction 1 occurs infinitely often in X. It is easy to show that M recurs if and only if recurs_M is satisfiable. (Note that in the case of finite sets, recurs_M is not satisfiable.) The construction also shows that the values for X can be restricted to be *recursive sets*, because for any 2-register (or Turing) machine M the set $\{\langle i, M^{(i)}, M^{(i+1)}\rangle\}_{i \geqslant 0}$ is recursive, where $M^{(0)}$ is the initial configuration with registers being 0 (tape being empty) and $M^{(i)} \vdash_M M^{(i+1)}$.

Finally, to show that BMPC is in Σ^1_1, we reduce the satisfiability of a formula $\psi = \mathit{BRF}(P, \varphi, k)$ effectively to satisfiability in second-order arithmetic with sets. First, ψ is rewritten into an equivalent standard form in $\mathcal{T}$ where *Ite*-terms and comprehension terms do not occur.

Let the translation function be τ. We assume that each sort σ is encoded as a unique number $\tau(\sigma)$, where nested sorts are encoded using a standard pairing function in Peano arithmetic. Each $\mathit{Default}^\sigma$ is encoded as a pair of $\tau(\sigma)$ and 0. The translation is lifted to arbitrary terms and formulas in the usual way. The translation preserves the structure of the formula and is such that ψ is satisfiable in $\mathcal{T}$ if and only if $\tau(\psi)$ is satisfiable in second-order arithmetic with sets. □

4 A Decision Procedure for BMPC of Basic Model Programs

Here we look at a fragment of model programs that are common in practice. Recall the definition of a *basic* model program (Definition 2). In most common situations, actions only use parameters that have basic sorts, see for example the Credits model sample in Ref. [39]. Moreover, the initial state is usually required to have fixed initial values for all state variables. We may assume, without loss of generality, that there is a special additional "initialization" action that can be used to initialize the state variables as desired.

Note also that in $\mathcal{T}$ set sorts are not allowed to be nested, i.e., one can only construct sets of basic values. For example, it is not possible to construct a powerset in $\mathcal{T}$. This limitation is not present in model programs in general but is an artifact of the background $\mathcal{T}$. Here this limitation is important however for the decidability result.

Example 10 *The model program* Topsort *in Example 1 is not basic because the state variables E and V are uninitialized. If we were to initialize E and V to be empty, and add the following action, that updates E to be a set of edges satisfying a linear arithmetic condition $\varphi[x, y]$, then the resulting model program would be basic.*

```
var initialized = false
[Action] Init(n as Integer)
```

```
require not initialized
initialized := true
V := {1..n}
E := {(x,y) | x in {1..n}, y in {1..n}, φ[x,y]}
```

□

We first provide a decision procedure for a stratified fragment $\mathcal{T}^{\prec}$ of formulas in $\mathcal{T}$. The decision procedure is by reduction to linear or Presburger arithmetic. Let $X(\varphi)$ denote the collection of all set variables that occur in a formula φ.

Definition 14 ($\mathcal{T}^{\prec}$) *A formula φ is* stratified *or is in $\mathcal{T}^{\prec}$ if*

- *φ has the form $\psi \wedge \bigwedge_{x \in X(\varphi)} x = t_x$, and*
- *the relation $\prec \stackrel{def}{=} \{(y, x) \mid x \in X(\varphi), y \in X(t_x)\}$ is well-founded.*

The equation $x = t_x$ is called the definition of x *in φ.*

Decision procedure for $\mathcal{T}^{\prec}$

Let $\varphi \in \mathcal{T}^{\prec}$. We provide a series of transformations of φ, each of which preserves equivalence to φ, such that the final formula is a linear arithmetic formula. Apply the following transformations to φ in the given order.

Eliminate set variables. Let $(v_i)_{i<k}$ be a fixed sequence of $X(\varphi)$ such that $v_j \not\prec v_i$ if $j > i$, i.e. the definition of v_i does not mention v_j for any $j > i$. This sequence exists because $\prec$ is well-founded.

Let φ_0 be φ. Given φ_j, and the definition $v_j = S_j$ in φ_j, construct φ_{j+1} from φ_j by replacing each occurrence of v_j (other than in its definition) by S_j. Clearly φ_{j+1} is logically equivalent to φ_j. So φ_k is logically equivalent to φ and has the form $\psi \wedge \bigwedge_{i<k} v_i = S_i$ where ψ and all the S_i are set-variable free. Since all set variables are existentially quantified, the formula $\psi \wedge \bigwedge_{i<k} v_i = S_i$ is true if an only if ψ is true.

Eliminate *TheElementOf***.** Apply the following transformation repeatedly to atomic formulas α that contain *TheElementOf*:

$$\alpha[\mathit{TheElementOf}(s)] \rightsquigarrow \exists x\, \alpha[\mathit{Ite}(\{x\} = s, x, \mathit{Default})].$$

Eliminate *Ite***.** Apply the following transformation repeatedly to atomic formulas α that contain *Ite*:

$$\alpha[\mathit{Ite}(\varphi, s_1, s_2)] \rightsquigarrow (\varphi \wedge \alpha[s_1]) \vee (\neg\varphi \wedge \alpha[s_2]).$$

Normalize set comprehensions. The variables y_i below are assumed to be fresh. After this step all element terms of comprehensions are tuples of variables.

$$\{\langle t_1(\vec{x}), \ldots, t_n(\vec{x})\rangle \mid \varphi(\vec{x})\} \rightsquigarrow \{\langle y_1, \ldots, y_n\rangle \mid \exists \vec{x}(y_1 = t_1(\vec{x}) \wedge \cdots \wedge y_n = t_n(\vec{x})) \wedge \varphi(\vec{x})\}.$$

Translate set operations. Since all set variables v have been eliminated the only atomic set terms are comprehensions. Because of the previous step, all element terms only include tuples of variables. It suffices to show the translation of the set operations on comprehensions. Recall also that terms are well-sorted so both sides have the same sort.

$$\begin{aligned}
\{\langle\vec{y}\rangle \mid \psi_1(\vec{y})\} \cup \{\langle\vec{y}\rangle \mid \psi_2(\vec{y})\} &\rightsquigarrow \{\langle\vec{y}\rangle \mid \psi_1(\vec{y}) \vee \psi_2(\vec{y})\},\\
\{\langle\vec{y}\rangle \mid \psi_1(\vec{y})\} \cap \{\langle\vec{y}\rangle \mid \psi_2(\vec{y})\} &\rightsquigarrow \{\langle\vec{y}\rangle \mid \psi_1(\vec{y}) \wedge \psi_2(\vec{y})\},\\
\{\langle\vec{y}\rangle \mid \psi_1(\vec{y})\} \setminus \{\langle\vec{y}\rangle \mid \psi_2(\vec{y})\} &\rightsquigarrow \{\langle\vec{y}\rangle \mid \psi_1(\vec{y}) \wedge \neg\psi_2(\vec{y})\}.
\end{aligned}$$

Translate $\in$ and $\subseteq$. When set operations have been eliminated, all sets are in the form of comprehensions. So element-of and subset atoms can thus be eliminated in the following way.

$$\begin{aligned}
t \in \{u(\vec{x}) \mid \psi(\vec{x})\} &\rightsquigarrow \exists\vec{x}\,(\psi(\vec{x}) \wedge t = u(\vec{x})),\\
\{\langle\vec{x}\rangle \mid \psi_1(\vec{x})\} \subseteq \{\langle\vec{x}\rangle \mid \psi_2(\vec{x})\} &\rightsquigarrow \forall\vec{x}\,(\psi_1(\vec{x}) \rightarrow \psi_2(\vec{x})).
\end{aligned}$$

Expand tuple variables. For each variable x of the sort $\sigma_1 \times \cdots \times \sigma_k$ for $k > 1$. Apply the following transformation repeatedly until all variables have the base sort β. This process clearly terminates.

$$Qx\,\varphi(x) \rightsquigarrow Qx_1 \ldots Qx_n\,\varphi(\langle x_1, \ldots, x_n\rangle).$$

Unwind tuples. Apply the following transformations until there are no more tuple operations. Note that at this point a tuple term can only appear in an equality or as an argument of a projection. This process clearly terminates.

$$\begin{aligned}
\langle t_1, \ldots, t_k\rangle = \langle u_1, \ldots, u_k\rangle &\rightsquigarrow t_1 = u_1 \wedge \cdots \wedge t_k = u_k\\
\pi_i(\langle t_0, \ldots, t_i, \ldots, t_k\rangle) &\rightsquigarrow t_i.
\end{aligned}$$

After the above transformations we get a linear arithmetic formula that is logically equivalent to the original formula. □

We now show that BMPC over basic model programs reduces to $\mathcal{T}^{\prec}$.

Theorem 4 *BMPC of basic model programs is decidable.*

Proof: Let P be a basic model program $(\Sigma, \Gamma, \varphi^0, (\gamma_f, U_f, X_f)_{f\in\Gamma})$. Let φ be a reachability condition, and let k be a step bound. We can assume, without loss of generality, that P has one action *Step*. Let $\psi = BRF(P, \varphi, k)$. Let X be the set of nonbasic state variables of P.

Since P is basic, the initial value of each set variable is $\emptyset$, i.e., $\varphi^0 = \bigwedge_{x\in X} x = \emptyset$. Each step formula $STEP(P, i)$ provides a definition $x[i+1] = t_x[i]$ for $x[i+1]$ as a conjunct of ψ. It follows by induction on i that $x[i] \prec y[i+1]$, for $i < k$ and all $x, y \in X$, is a valid well-founded order of all the set variables. Thus ψ is in $\mathcal{T}^{\prec}$. Decidability follows from the reduction of $\mathcal{T}^{\prec}$ to linear or Presburger arithmetic and decidability of Presburger arithmetic[16]. □

The following result follows from the proof of Theorem 4 and the construction of the canonical representation of an ASM as a model program (Definition 3). *Bounded inconsistency checking of ASMs* is the problem of deciding if an ASM produces an inconsistent update set within a given number of steps (see Section 2.8).

Corollary 2 *Bounded inconsistency checking of standard ASMs that do not allow nesting of choose statements within forall statements reduces to satisfiability in* $\mathcal{T}^{\prec}$.

5 Related Work

Preliminary versions of some of the results in this paper have appeared in Refs. [35, 37, 39]. We use the state of the art SMT solver Z3[13] for our experiments. 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 Ref. [39]. In particular, the scheme discussed in Ref. [39] is inspired by Ref. [11], 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 BMPC of basic model programs to linear arithmetic is ongoing work. In that context the reduction to Z3 does not need to complete all the reductions in the decision procedure of $\mathcal{T}^{\prec}$, 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[36] and NModel[30]. In Spec Explorer, one of the supported input languages is the abstract state machine language AsmL[4,20]. In that context, sanity checking or validation of model programs is usually achieved through simulation and explicit state exploration and search techniques[26,36]. Model checking of ASMs is studied in Refs. [40, 41] using explicit state model checking with the SMV model checker; different temporal logics are considered for expressing properties. Besides safety properties, we have not considered general temporal properties, also, their semantics is unclear in the case of bounded-depth exploration. The *unbounded* reachability problem for model programs without comprehensions and with parameterless actions is shown to be undecidable in Ref. [17], where it is called the hyperstate reachability problem. General reachability problems for transition systems are discussed in Ref. [33] where the main results are related to guarded assignment systems.

The decidable fragment BAPA[28] is an extension of Boolean algebra with Presburger arithmetic. The sets in BAPA are finite and bounded by a maximum size and the cardinality operator is allowed. Comprehensions are not possible and the element-of relation is not allowed, i.e. integers and sets can only be related through the cardinality operator. A decidable fragment of bag (multiset) constraints combined with summation constraints are considered in Ref. [31] where summation constraints can be used to express set cardinality.

Sets and maps are used as foundational data structures in many modeling and analysis methods such as RAISE, Z, TLA+, B, see Ref. [6]. The ASM method is also described in Ref. [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[25], the analysis is reduced to SAT, by finitizing the data types. In our case the analysis is reduced to SMT, and rather than bounding the size of the data, the search depth is bounded. Traditional *untyped* ASMs often assume a rich background[7] that includes hereditarily finite sets and maps, and there is a specific *Undef* element in the universe to deal with partial functions (for Booleans the *Undef* value is *false*).

The data structures that are allowed in the Jahob verification system[10] also allow sets and set operations. The algorithm described in Section 4 is similar, in some parts, to the translation scheme described in Ref. [10, Appendix B]. Their translation leads to a first-order formula targeted to a resolution theorem prover. As shown in Section 3, such a translation cannot, in general, provide a semi-decision procedure for $\mathcal{T}$, because of Σ^1_1-hardness of the satisfiability problem for $\mathcal{T}$.

The reduction of the theories of arrays, sets and multisets to the theory of equality with uninterpreted function symbols and linear arithmetic is used in Ref. [27] for constructing interpolants for these theories.

The technique of bounded model checking by using SAT solving was pioniered in Ref. [5] and the extension to SMT was introduced in Ref. [14], a related approach was proposed in Ref. [2]. Besides Z3[13], other SMT solvers that support arrays and sets are described in Refs. [3, 15, 34].

Acknowledgements

We thank Wolfgang Reisig for many valuable comments and suggestions.

References

[1] Alur R, Henzinger TA. A really temporal logic. In: Proc. 30th Symp. on Foundations of Computer Science, 1989. 164–169.

[2] Armando A, Mantovani J, Platania L. Bounded model checking of software using SMT solvers instead of SAT solvers. Valmari A, ed, SPIN, volume 3925 of LNCS, Springer, 2006. 146–162.

[3] Armando A, Ranise S, Rusinowitch M. A rewriting approach to satisfiability procedures. Inf. Comput., 2003, 183(2): 140–164.

[4] AsmL. http://research.microsoft.com/AsmL, http://www.codeplex.com/AsmL

[5] Biere A, Cimatti A, Clarke E, Zhu Y. Symbolic model checking without BDDs. Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'99), volume 1579 of LNCS, Springer, 1999. 193–207.

[6] Bjørner D, Henson M. eds. Logics of Specification Languages. Springer, 2008.

[7] Blass A, Gurevich Y. Background, reserve, and Gandy machines. In: Proc. of the 14th Annual Conference of the EACSL on Computer Science Logic, Springer, 2000. 1–17.

[8] Blass A, Gurevich Y. Background of computation. Bulletin of the European Association for Theoretical Computer Science EATCS, 2007, 92: 82-114.

[9] Börger E, Grädel E, Gurevich Y. The Classical Decision Problem. Springer, 1997.

[10] Bouillaguet C, Kuncak V, Wies T, Zee K, Rinard M. On using firstorder theorem provers in the Jahob data structure verification system. Computer Science and Artificial Intelligence Laboratory Technical Report MIT-CSAIL-TR-2006-072, Massachusetts Institute of Technology, November 2006.

[11] Bradley AR, Manna Z, Sipma HB. What's decidable about arrays? In: Verification, Model Checking, and Abstract Interpretation: 7th International Conference, (VMCAI'06), volume 3855 of LNCS, Springer, 2006. 427–442.

[12] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement. Emerson EA, Sistla AP, eds, CAV, volume 1855 of Lecture Notes in Computer Science, Springer, 2000. 154–169.

[13] de Moura L, Bjørner N. Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08), LNCS, Springer, 2008.

[14] de Moura L, Rueß H, Sorea M. Lazy theorem proving for bounded model checking over infinite domains. Proc. of the 18th International Conference on Automated Deduction (CADE'02), vol. 2392 of LNCS, Springer, 2002. 438–455.

[15] Dutertre B, de Moura L. A fast linear arithmetic solver for DPLL(T). CAV'06, vol. 4144 of LNCS, 2006. 81–94.

[16] Fischer MJ, Rabin MO. Super-exponential complexity of Presburger arithmetic. SIAMAMS: Complexity of Computation: Proc. of a Sym-posium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics, 1974. 27–41.

[17] Grieskamp W, Gurevich Y, Schulte W, Veanes M. Generating finite state machines from abstract state machines. SIGSOFT Softw. Eng. Notes, 2002, 27(4): 112–122.

[18] Grieskamp W, MacDonald D, Kicillof N, Nandan A, Stobie K, Wurden F. Model-based quality assurance of Windows protocol documentation. First International Conference on Software Testing, Verification and Validation, ICST, Lillehammer, Norway, April 2008.

[19] Gurevich Y. Specification and Validation Methods, chapter Evolving Algebras 1993: Lipari Guide, Oxford University Press, 1995. 9–36.

[20] Gurevich Y, Rossman B, Schulte W. Semantic essence of AsmL. Theor. Comput. Sci., 2005, 343(3): 370–412.

[21] Gurevich Y, Tillmann N. Partial updates. Theoretical Computer Science, 2005, 336(2–3): 311–342.

[22] Gurevich Y, Veanes M. Logic with equality: partisan corroboration and shifted pairing. Inf. Comput.,1999, 152(2): 205–235.

[23] Halpern JY. Presburger arithmetic with unary predicates is Π^1_1 complete. Journal of Symbolic Logic, 1991, 56: 637–642.

[24] Harel D, Pnueli A, Stavi J. Propositional dynamic logic of nonregular programs. Journal of Computer and System Sciences, 1983, 26(2): 222–243.

[25] Jackson D. Software Abstractions. MIT Press, 2006.

[26] Jacky J, Veanes M, Campbell C, Schulte W. Model-based Software Testing and Analysis with C#. Cambridge University Press, 2008.

[27] Kapur D, Majumdar R, Zarba CG. Interpolation for data structures. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE 2006), ACM, 2006. 105–116.

[28] Kuncak V, Nguyen HH, Rinard M. An algorithm for deciding BAPA: Boolean algebra with Presburger arithmetic. Nieuwenhuis R, edit, CADE 2005, volume 3632 of LNAI, Springer, 2005. 260–277.

[29] Nelson G, Oppen DC. Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst.,1979, 1(2): 245–257.

[30] NModel. http://www.codeplex.com/NModel, public version released May 2008.

[31] Piskac R, Kuncak V. Decision procedures for multisets with cardinality constraints. Logozzo F, Peled D, Zuck LD, eds, VMCAI, volume 4905 of LNCS, Springer, 2008. 218–232.

[32] Rogers H. Theory of Recursive Functions and Effective Computability. MIT Press, 2nd edition, 1987.

[33] Rybina T, Voronkov A. A logical reconstruction of reachability. Broy M, Zamulin A, eds, PSI 2003, volume 2890 of LNCS, Springer, 2003. 222–237.

[34] Stump A, Barrett CW, Dill DL, Levitt JR. A decision procedure for an extensional theory of arrays. LICS'01, IEEE, 2001. 29–37.

[35] Veanes M, Bjørner N, Raschke A. An SMT approach to bounded reachability analysis of model programs. FORTE'08, volume 5048 of LNCS, Springer, 2008. 53–68.

[36] Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann N, Nachmanson L. Model-based testing of object-oriented reactive systems with Spec Explorer. Hierons R, Bowen J, Harman

M, eds, Formal Methods and Testing, volume 4949 of LNCS. 39–76.

[37] Veanes M, Saabas A. On bounded reachability of programs with set comprehensions. LPAR'08, volume 5330 of LNAI, Springer, 2008. 305–317.

[38] Veanes M, Saabas A. Using satisfiability modulo theories to analyze abstract state machines (abstract). ABZ 2008, volume 5238 of LNCS, Springer, 2008. 355.

[39] Veanes M, Saabas A, Bjørner N. Bounded reachability of model programs. Technical Report MSR-TR-2008-81, Microsoft Research, May 2008.

[40] Winter K. Model checking for abstract state machines. Journal of Universal Computer Science, 1997, 3(5): 689–701.

[41] Winter K. Model Checking Abstract State Machines. VDM Verlag Dr. Müller, 2008.