Bounded reachability of model programs[*]

Microsoft Research Technical Report MSR-TR-2008-81

Margus Veanes
Microsoft Research, Redmond
margus@microsoft.com

Ando Saabas[†]
Institute of Cybernetics, TUT, Tallinn, Estonia
ando@cs.ioc.ee

Nikolaj Bjørner
Microsoft Research, Redmond
nbjorner@microsoft.com

May 2008

Abstract

Model programs represent labeled transition systems and are used to specify expected behavior of systems at a high level of abstraction. Such programs are common as high-level executable specifications of complex protocols. Model programs typically use abstract data types such as sets and maps, and comprehensions to express complex state updates. Such models are mainly used in model-based testing as inputs for test case generation and as oracles during conformance testing. Correctness assumptions about the model itself are usually expressed through state invariants. An important problem is to validate the model prior to its use in the above-mentioned contexts. We introduce a technique of using Satisfiability Modulo Theories or SMT to perform bounded reachability of a fragment of model programs. We analyze the bounded reachability problem and prove decidability and undecidability results of restricted cases of this problem. We use the Z3 solver for our implementation and benchmarks, and we use AsmL as the modeling language. The translation from a model program into a verification condition of Z3 is incremental and involves selective quantifier instantiation of quantifiers that result from set comprehensions and bag axioms.

[*]This report subsumes part of the material in [41].
[†]Part of this work was done during an internship at Microsoft Research, Redmond.

Contents

1 Introduction

Programs that use high-level data types are commonly used to describe executable specifications [26] in form of so called *model programs.* An important and growing application area in the software industry is the use of model programs for specifying and documenting expected behavior of application-level network protocols [17]. Model programs typically use abstract data types such as sets and maps, and comprehensions to express complex state updates. Correctness assumptions about the model are usually expressed through state invariants. An important problem is to validate a model prior to its use as an oracle or final specification. One approach is to use Satisfiability Modulo Theories or SMT to perform bounded reachability analysis or bounded model-checking of model programs. The use of SMT solvers for automatic software analysis has recently been introduced [1] as an extension of SAT-based bounded model checking [6]. The SMT based approach makes it possible to deal with more complex background theories. Instead of encoding the verification task of a sequential program 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 [33].

Unlike traditional sequential programs, model programs typically operate on a more abstract level and in particular make use of (set and bag) comprehensions as expressions that are computed in a single step, rather than computed, one element at a time, in a loop. In this report we consider an extension of the SMT approach to reachability analysis of model programs where set comprehensions are supported at the given level of abstraction and not unwound as loops. We identify a fragment of model programs using the array property fragment [10] that remains decidable for bounded reachability analysis.

The construction of the formula for bounded reachability of sequential programs is based on the semantics of the behavior of the program as a transition system. The resulting formula encodes reachability of some condition within a given bound in that transition system. If the formula is satisfiable, a model of the formula typically is a witness of some bad behavior. The semantics of a model program on the other hand, is given by a *labeled* transition system, where the labels record the actions that caused the transitions. Using the action label is conceptually important for separating the (external) trace semantics of the model program from its (internal) state variables. The trace semantics of model programs is used for example for conformance testing. When composing model programs, shared actions are used to synchronize steps. We illustrate how composition of model programs [43] can be used for scenario oriented or user directed analysis.

This report also provides a complete characterization of the decidable and undecidable cases of the bounded reachability problem of model programs in terms of the complexity of action parameter types. We show in Section 4 that already the single step reachability problem is undecidable if a single set-valued parameter is allowed. In Section 5 we show that the bounded reachability problem remains decidable provided that all parameters have basic (non-set valued) types. This result is orthogonal to the decidable fragment of bounded reachability of model programs that use the array property fragment [10]. We use the SMT solver Z3 [12] for our experiments and we use AsmL [19] as the modeling language. Related work is discussed in Section 8.

2 Model programs

The semantics of model programs in their full generality builds on the abstract state machine (ASM) theory [18]. Model programs are primarily used in model-based testing tools like Spec Explorer [39, 42] where one of the supported input languages is the abstract state machine language AsmL [3, 19]. The NModel tool [34, 26] and Spec Explorer 2007 [17] use plain C# for describing model programs. Spec Explorer 2007 uses, in addition, a coordination language Cord for scenario control [16] and model composition. Typically, a model program makes use of a rich background theory [7] $\mathcal{T}$, that contains integer arithmetic, finite collections (sets, maps, sequences, bags), and tuples, as well as user defined data types.

2.1 Background theory

Let the signature of $\mathcal{T}$ be Σ. For each *sort* S (representing a type) the theory for S and its signature are denoted by $\mathcal{T}_S$ and Σ_S, respectively. All function symbols and constants in Σ, and all variables are typed, and when referring to terms over Σ we assume that the terms are well-typed. For a term t, the set of symbols that occur in it is called the *signature of* t and is denoted by $\Sigma(t)$. Boolean sort $\mathbb{B}$ is explicit, and formulas are represented by Boolean terms. We use the notation $t[x]$ to indicate that the free logical variable x may occur in t. Given term s we also use the notation $t[s]$ to indicate the substitution of s for x in t. The integer sort is $\mathbb{Z}$. Given sorts D and R, $\{D \mapsto R\}$ is the *map sort* with *domain sort* D and *range sort* R. The map sort $\{D \mapsto \mathbb{B}\}$ is also denoted by $\{D\}$ and called a *set sort* with domain sort D. For each sort S there is a designated constant $\mathit{default}_S$ denoting a special value in (the type represented by) S. For Booleans, that value is *false*. The use of $\mathit{default}_S$ is to represent partial maps, with range sort S, as total maps that map all but finitely many elements to $\mathit{default}_S$. In particular, sets are represented by their characteristic functions as maps.

2.1.1 Maps

For each map sort $S = \{D \mapsto R\}$, the signature Σ_S contains the binary function symbol read_S, the ternary function symbol write_S and the constant empty_S. The function $\mathit{read}_S : S \times D \longrightarrow R$ retrieves the element for the given key of the map. The function $\mathit{write}_S : S \times D \times R \longrightarrow S$ creates a new map where the key has been updated to the new value. The constant empty_S denotes the empty map. The theory $\mathcal{T}_S$ contains the classical map axioms (see e.g. [10]), which we repeat here for clarity and to introduce some notation:

$$\forall m\,x\,v\,y(\mathit{read}(\mathit{write}(m,x,v),y) = \mathit{Ite}(x = y, v, \mathit{read}(m,y))), \tag{1}$$

$$\forall m_1\,m_2(\forall x(\mathit{read}(m_1,x) = \mathit{read}(m_2,x)) \rightarrow m_1 = m_2). \tag{2}$$

All symbols are typed, i.e. have the expected sort, but we often omit the sort annotations as they are clear from the context. The value of an if-then-else term $\mathit{Ite}(\varphi, t_1, t_2)$ (in a given structure) is: the value of t_1, if φ holds; the value of t_2, otherwise. The second axiom above is extensionality. $\mathcal{T}_S$ also contains the axiom for the empty map:

$$\forall x(\mathit{read}(\mathit{empty}, x) = \mathit{default}_R). \tag{3}$$

2.1.2 Sets

For each set sort $S = \{D\}$, the signature Σ_S contains additionally the binary set operations for union $\cup_S$, intersection $\cap_S$, set difference $\setminus_S$, and subset $\subseteq_S$. The theory $\mathcal{T}_S$ contains the appropriate axiomatization for the set operations. We write $x \in s$ and $x \notin s$ as abbreviations for $\mathit{read}(s,x)$ and $\neg\mathit{read}(s,x)$, respectively. A *set comprehension term* s of sort S has the form $\mathit{Compr}(t[x], x, r, \varphi[x])$ or

$$\{t[x] : x \in r, \varphi[x]\}, \tag{4}$$

where $t[x]$ is a term of sort D called the *element term of* s, x is a logical variable of some sort E called the *variable of* s, r is a term of sort $\{E\}$ called the *range of* x, and $\varphi[x]$ is a formula called the *restriction condition of* s. When the restriction condition is *true*, we write the set comprehension as $\{t[x] : x \in r\}$. Given a closed set comprehension term s as (4), the constant s' *defines* s by (5).

$$\forall y(y \in s' \leftrightarrow \exists x(y = t[x] \wedge x \in r \wedge \varphi[x])). \tag{5}$$

The element term $t[x]$ of s is *invertible for* x, if 1) the function $\mathbf{f} = \lambda x.t[x]$ is injective, 2) there exists a formula $\psi_t[y]$ that is true iff y is in the range of $\mathbf{f}$, and 3) there exists a term $t^{-1}[y]$ such that $t^{-1}[y] = \mathbf{f}^{-1}(y)$ for all y such that $\psi_t[y]$ holds. If $t[x]$ is invertible, then the existential quantifier in (5) can be eliminated and (5) can be simplified to (6). (Just extend the body of the existential formula with the conjunct $t^{-1}[y] = x \wedge \psi_t[y]$ and substitute $t^{-1}[y]$ for x.)

$$\forall y(y \in s' \leftrightarrow t^{-1}[y] \in r \wedge \varphi[t^{-1}[y]] \wedge \psi_t[y]) \tag{6}$$

We say that a set comprehension term s is *normalizable* if the element term of s is invertible for the variable of s. The form (6) is called the *normal form definition for* s.

2.1.3 Range expressions

For the sort $S = \{\mathbb{Z}\}$ of integer sets, Σ_S contains the binary function symbol $\mathit{Range} : \mathbb{Z} \times \mathbb{Z} \longrightarrow S$. A term $\mathit{Range}(l,u)$ is called a *range expression* with l

as its *lower bound* and u as its *upper bound*. We also use the notation $\{l..u\}$ for $\mathit{Range}(l,u)$. The interpretation of a range expression is the set of integers from its lower bound to its upper bound. $\mathcal{T}_S$ contains the axiom (7) for range expressions, where it is assumed that $\mathcal{T}_{\mathbb{Z}}$ includes Pressburger arithmetic.

$$\forall x\, l\, u(x \in \{l..u\} \leftrightarrow l \leq x \wedge x \leq u) \tag{7}$$

Note that a formula $t \in \{l..u\}$ simplifies to $l \leq t \wedge t \leq u$, and a formula $t \notin \{l..u\}$ simplifies to $l > t \vee t > u$. More generally, any formula that is a Boolean combination of range expressions and set operations can be simplified to linear equations. Similarly, range expressions that are used as sets and that do not depend on bound variables (inside nested comprehesion terms) can also be eliminated by introducing fresh constants and adding constraints corresponding to (7).

The theories for sets are assumed to contain definitions for all closed set comprehension terms. When considering particular model programs below, the signature Σ is expanded with new application specific constants. However, for technical reasons it is convenient to assume that all those constants are available in Σ a priori, so that the extension with set comprehension definitions is already built into the theories.

Example 1 Let s be $\{m+x : x \in \{1..c\}\}$ where m and c are application specific integer constants. The term $m+x$ is invertible for x; let ψ_{m+x} be *true* and let $(m+x)^{-1}$ be $y-m$. The normal form definition for s is $\forall y(y \in s' \leftrightarrow y-m \in \{1..c\})$, which reduces to $\forall y(y \in s' \leftrightarrow 1 \leq y-m \wedge y-m \leq c)$.

Example 2 Let s be $\{x+x : x \in \{1..c\}\}$ where c is an application specific constant. The term $x+x$ is invertible provided that $\mathcal{T}_{\mathbb{Z}}$ supports divisibility by a constant; let $(x+x)^{-1}$ be $y/2$ and let ψ_{x+x} be $\mathit{Divisible}(y,2)$. The normal form definition for s is $\forall y(y \in s' \leftrightarrow y/2 \in \{1..c\} \wedge \mathit{Divisible}(y,2))$, or equivalently $\forall y(y \in s' \leftrightarrow 2 \leq y \wedge y \leq 2 \cdot c \wedge \mathit{Divisible}(y,2))$.

2.1.4 Arrays

A class of model programs, e.g. those used typically in protocol specifications, do not depend on the full background theory but only on a fragment of it. The particular fragment of interest is when all map sorts have domain sort $\mathbb{Z}$ and $\mathcal{T}_{\mathbb{Z}}$ is Pressburger arithmetic, with $\Sigma_{\mathbb{Z}}$ including $\{+,-,<,=\}$ and integer numerals. In particular, multiplication is omitted. Multiplication by a numeral is used as a convenient shorthand for repeated addition. In this case, the set comprehension term in Example 1 is normalizable. This fragment is called *array theory* [10] and has useful properties that are exploited below.

Note that it *is* possible to express divisibility constraints by for example introducing auxiliary variables and eliminating positive occurrences of $\mathit{Divisible}(t,k)$ by $k \cdot z = t$, and negative occurrences by $k \cdot z + u = t \wedge 1 \leq u < k$ for fresh z and u. One can even consider extending the array fragment to Büchi arithmetic [22].

2.2 Variables and values

We refer to the part of the global signature Σ that only includes symbols whose interpretation is fixed by the background theory $\mathcal{T}$ as Σ^{static}; including for example arithmetic operations and numerals and set operations. We let $\Sigma^{\text{var}} = \Sigma \setminus \Sigma^{\text{static}}$ denote the uninterpreted symbols. We let Σ^{var}_S and Σ^{static}_S indicate the corresponding signatures restricted to the sort S. Note that Σ^{var} includes an unlimited supply of variables for all sorts, treated as uninterpreted constants. A ground term over Σ^{static} is called a *value term*.

2.3 Actions

There is an *action sort* $\mathbb{A}$. The theory $\mathcal{T}_{\mathbb{A}}$ axiomatizes a collection $\Sigma_{\mathbb{A}}$ of *action symbols* as free constructors. For each action symbol f of arity n, the sort of f is $\mathbb{A}$ if $n = 0$ and the sort of f is $S_1 \times \cdots \times S_n \longrightarrow \mathbb{A}$ otherwise, where each S_i is a sort distinct from $\mathbb{A}$. In other words, actions cannot take actions as parameters. An *action* is value term $f(t_1, \ldots, t_n)$ where f is an action symbol.

2.4 Model program definition

For all action symbols f with arity $n \geq 0$, and all i, $1 \leq i \leq n$, there is a unique *parameter variable* (not in Σ^{var}) denoted by f_i. We write Σ_f for $\{f_i\}_{1 \leq i \leq n}$. Note that if $n = 0$ then $\Sigma_f = \emptyset$.

Definition 1 A *model program* P is a tuple (V_P, A_P, I_P, R_P), where

- V_P is a finite set of *state variables*, let Σ_P denote $\Sigma^{\text{static}} \cup V_P$;
- A_P is a finite set of *action symbols*;
- I_P is a formula over Σ_P, called the *initial state condition*;
- R_P is a family $\{R^f_P\}_{f \in A_P}$ of *action rules* $R^f_P = (G^f_P, U^f_P)$, where
  - G^f_P is a formula over $\Sigma_P \cup \Sigma_f$ called the *guard of* R^f_P;
  - U^f_P, called the *update rule of* R^f_P, is a block $\{v := t^f_v\}_{v \in V^f_P}$ of *assignments* where t^f_v is a term over $\Sigma_P \cup \Sigma_f$ and $V^f_P \subseteq V_P$.

This definition is a variation of model programs that syntactically restricts the update rules to be block assignments. Note that this does not restrict expressivity because we can always replace conditional update rules with corresponding *if-then-else terms*. One can therefore treat this definition as a *guarded assignment normal form* for model programs.

We often say *action* to also mean an action rule or an action symbol, if the intent is clear from the context.

Example 3 (*Credits*) The following model program is written in AsmL. It specifies how a client and a server need to use message ids, based on a sliding window protocol. It models part of the credits-algorithm in the SMB2 [38] protocol.

```
var $window$ as Set of Integer = $\{0\}$
var $maxId$ as Integer = $0$
var $requests$ as Map of Integer to Integer = {->}

[Action]
Req(m as Integer, c as Integer)
    require $m \in window$ and $c > 0$
    $requests := requests.Add(m, c)$
    $window := window - \{m\}$

[Action]
Res(m as Integer, c as Integer)
    require $m \in requests$ and $requests(m) \geq c$ and $c \geq 0$
    //require $requests.Size > 1$ or $window$ <> {} or $c > 0$   <-- bug
    $window := window + \{maxId + i \mid i \in \{1..c\}\}$
    $requests := requests.RemoveAt(m)$
    $maxId := maxId + c$

[Invariant]
ClientHasEnoughCredits()
    require $requests$ = {->} implies $window$ <> {}
```

The *Credits* model program illustrates a typical use of model programs as protocol-specifications. Actions use parameters, maps and sets are used as state variables and a comprehension expression is used to compute a set. Each action has a guard and an update rule given by a basic ASM. For example, the guard of the `Req` action requires that the id of the message is in the current window of available ids and that the number of credits that the client requests from the server is positive. The state invariant associated with the model program is that the client must not starve, i.e. there should always be a message id available at some point, so that the client can issue new requests.

Let P be a fixed model program. A *P-state* is a mapping of V_P to values.[1] Given an action $a = f(a_1, \ldots, a_n)$, let θ_a denote the parameter assignment $\{f_i \mapsto a_i\}_{1 \leq i \leq n}$. Given a P-state S, an extension of S with the parameter assignment θ is denoted by $(S; \theta)$.

Let S be a P-state, an f-action a is *enabled* in S if $(S; \theta_a) \models G_P^f$. The action *$a$ causes a transition from S to S'*, where

$$S' = \{v \mapsto t_v^{f\,(S;\theta_a)}\}_{v \in V_P^f} \cup \{v \mapsto v^S\}_{v \in V_P \setminus V_P^f}.$$

[1]More precisely, this is the foreground part of the state, the background part is the canonical model of the background theory $\mathcal{T}$.

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 2 Let P be a model program. The *LTS of* P, denoted by $[\![P]\!]$ is the LTS $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}_0$, is the set of all P-states s such that $s \models I_P$; L is the set of all actions over A_P; 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 $(s, a, s') \in T$.

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*) *trace* of P. The run or the trace is *finite* if $\kappa < \omega$.

2.5 Composition of model programs

Under composition, model programs synchronize their steps for the same action symbols. The guards of the actions in the composition are the conjunctions of the guards of the component model programs. The formal definition is a simplification of the parallel composition of model programs from [43].

Definition 3 Let P and Q be model programs such that $A = A_P = A_Q$ and $V_P^f \cap V_Q^f = \emptyset$ for all $f \in A$. Let

$$P \oplus Q \stackrel{\text{def}}{=} (V_P \cup V_Q, A, I_P \wedge I_Q, (G_P^f \wedge G_Q^f, U_P^f \cup U_Q^f)_{f \in A}).$$

The disjointness of the variables updated by the same action may be relaxed by using a more general form of update rules and a mechanism for combining updates. Composition can be used to do scenario oriented modeling [43]. In Section 7 we illustrate how composition can also be used to do scenario oriented analysis, or assist the theorem prover with lemmas.

3 Bounded reachability of model programs

Let P be a model program and let φ be a Σ_P-formula. The main problem we are addressing is whether φ is reachable in P within a given bound.

Definition 4 Given φ and $k \geq 0$, φ *is reachable in* P *within* k *steps*, if there exists an initial state s_0 and a (possibly empty) run $(s_i, a_i, s_{i+1})_{i<l}$ in P, for some $l \leq k$, such that $s_l \models \varphi$. If so, the action sequence $\alpha = (a_i)_{i<l}$ is called a *reachability trace for* φ and s_0 is called an *initial state for* α.

Note that, given a trace α and an initial state s_0 for it, the state where the condition is reached is reproducible by simply executing α starting from s_0. This provides a cheap mechanism to check if a trace produced by a solver is indeed a witness. In a typical model program, the initial state is uniquely determined by an initial assignment to state variables, so the initial state witness is not relevant.

Note also that an important use of action parameters is to make all nondeterminism explicit, by providing a parameter and making a choice based on that parameter using a conditional update rule. Therefore update rules considered here do not have the nondeterministic *choose* construct of nondeterministic ASMs [18].

3.1 Reachability formula

The basic idea of generating a reachability formula for bounded model checking and to use SAT to check this formula was introduced in [6]. Here we use a similar translation scheme and apply it to model programs. Given a state variable or action parameter x we use $x[i]$ to denote a new variable or parameter for step number i. For step 0, we assume that $x[0]$ is x, i.e. the original variable is used. For a term t, $t[i]$ produces a term by induction over the structure of terms where all state variables and action parameters are given step number i.

The *bounded reachability formula* for a given model program P, step bound k and reachability condition φ is:

$$\begin{aligned} \mathit{Reach}(P,\varphi,k) &\overset{\text{def}}{=} I_P \wedge (\bigwedge_{0\le i<k} P[i]) \wedge (\bigvee_{0\le i\le k} \varphi[i]) && (8)\\ P[i] &\overset{\text{def}}{=} \bigvee_{f\in A_P} \big(\mathit{action}[i] = f(f_1[i],\ldots,f_n[i]) \wedge G_P^f[i] && (9)\\ &\qquad \bigwedge_{v\in V_P^f} v[i+1] = t_v^f[i] \quad \bigwedge_{v\in V_P\setminus V_P^f} v[i+1] = v[i]\big) \end{aligned}$$

A *skip* action has the action rule $(\mathit{true},\emptyset)$. We have the following theorem.

Theorem 1 *Let P be a model program that includes a skip action, let $k \ge 0$ be a step bound and let φ be a reachability condition. Then $\mathit{Reach}(P,\varphi,k)$ is satisfiable if and only if φ is reachable in P within k steps. Moreover, if M satisfies $\mathit{Reach}(P,\varphi,k)$, let $M_0 = \{v \mapsto v^M\}_{v\in V_P}$, let $a_i = \mathit{action}[i]^M$ for $0 \le i < k$, and let α be the sequence $(a_i)_{i<k}$. Then α is a reachability trace for φ and M_0 is an initial state for α.*

Proof Follows easily from the definition of the bounded reachability formula and the definition of bounded reachability. ⊠

3.2 Array model programs

We consider here the fragment of $\mathcal{T}$ when $\mathcal{T}_\mathbb{Z}$ is Pressburger arithmetic and all map sorts have domain sort $\mathbb{Z}$. We call model programs that only depend on this fragment of $\mathcal{T}$, *array model programs*. In the following lemma we refer to the *array property* fragment introduced in [10]. An example of a model program in this fragment is the Credits model program in Example 3 that models the credit negotiation algorithm in the SMB2 protocol [38]. The purpose of this model program is explained in detail in [44].

Lemma 1 *Let P be an array model program and assume that all set comprehension definitions of P are normalizable, have a variable range that is a range expression, and that $P[i]$ is quantifier free. Assume also that I_P and φ are in the array property fragment. Let $k \geq 0$. Then $Reach(P, \varphi, k)$ is in the array property fragment.*

The following is a corollary of Lemma 1 and [10, Theorem 1], using the fact that the only range sort theory besides $\mathcal{T}_{\mathbb{Z}}$ is $\mathcal{T}_{\mathbb{B}}$ and thus this fragment of $\mathcal{T}$ is decidable. We also refer to SAT$_A$ in [10, Definition 9].

Corollary 1 *Let P and φ be as in Lemma 1. Then SAT$_A$ is a decision procedure for $Reach(P, \varphi, k)$.*

The decision procedure SAT$_A$ eliminates universal quantifiers by restricting the universal quantification to a finite index set generated from the formula. In our case the formula under consideration is $\psi = Reach(P, \varphi, k)$. We assume here that the set (comprehension) definitions are conjuncts of the respective step formula.

Typically, a set comprehension uses a range expression, see e.g. the Credits example in Example 3, and the index set for this formula yields at least four indices (the boundary cases for the range and its negation). The size of the index set grows at least proportionally to k, because each step formula introduces new indices, and thus the elimination process increases the size of the final quantifier free formula at least quadratically.

In our elimination scheme, the index set used to eliminate quantifiers of a given step formula, *only* originates from that step formula. For the set of model programs we have encountered so far, this restricted elimination preserves completeness of SAT$_A$ for satisfiability of ψ. While we do not yet have identified a general class of model programs where this restriction remains complete, we can use Z3 to *lazily* augment the constraints we generate by model-checking the model returned by Z3. Section 6 explains the way we use Z3 lazily.

4 One step reachability of model programs is undecidable

The bounded reachability problem of model programs is undecidable in the general case. In this section we pin down various minimal cases of the undecidability with respect to various restrictions on the background. In all cases it is enough to restrict the reachability bound and the number of action symbols to 1, i.e. the undecidability arises already using a single step and a single action symbol. We call it the *one step reachability problem.* In Section 5 we argue that these undecidable cases are minimal in some sense.

4.1 Theory $\boldsymbol{TS}(\mathcal{B})$

Here we assume that we have a small *base theory* $\mathcal{B}$ (for example Presburger arithmetic) and that we define a theory $TS(\mathcal{B})$ that extends $\mathcal{B}$ with *tuples* and *sets*.

It is assumed that the language of $\mathcal{B}$ does not include the new symbols. It is convenient to restrict the set of all possible expressions of $TS(\mathcal{B})$ to a set of well-formed expressions that are shown in Figure 1. When considering a formula of $TS(\mathcal{B})$ as defined in Figure 1, it is assumed that by default all set variables are *existentially quantified*, i.e. have an outermost existential quantifier. We write $TS(\mathcal{B})$ both for the class of expressions as defined in Figure 1, as well as the axioms of $TS(\mathcal{B})$.

The axioms of $TS(\mathcal{B})$ include the axioms of $\mathcal{B}$, the axioms for tuples stating that for each arity k the k-tuple constructor is a free constructor, axioms for set union, set intersection, element-of relation, subset relation, and the extensionality axiom for sets. Given a model $\mathfrak{A}$ of $TS(\mathcal{B})$, i.e., a structure $\mathfrak{A}$ in the language of $TS(\mathcal{B})$ that is a model of the axioms of $TS(\mathcal{B})$, the *comprehension term* $s = \{t(\overline{x}) \mid_{\overline{x}} \varphi(\overline{x})\}$, where t and φ may include parameters, has the interpretation $s^{\mathfrak{A}}$ in $\mathfrak{A}$ such that

$$\mathfrak{A} \models \forall y(y \in s^{\mathfrak{A}} \leftrightarrow \exists \overline{x}(t(\overline{x}) = y \wedge \varphi(\overline{x})))$$

which is well-defined due to the extensionality axiom:

$$\forall v\, w(\forall y(y \in v \leftrightarrow y \in w) \rightarrow v = w).$$

Example 4 Let $\mathcal{P}$ be Presburger arithmetic. The following is a *range expression*, in $TS(\mathcal{P})$:

$$\{z \mid x \leq z \wedge z \leq y\}$$

where we omit the z from $|_z$. We often use the abbreviation $\{x..y\}$ for a range from x to y. The following is a *direct product* $v \times w$ between two sets v and w:

$$\{\langle x, y\rangle \mid x \in v \wedge y \in w\}.$$

Note that, not all well-formed $TS(\mathcal{P})$ expressions can be used in a model program, in a model program all expressions are quantifier free and each set comprehension variable has a finite range.

4.2 Undecidable cases

Theorem 2 *One can effectively associate a deterministic 2-register machine M with a formula $halts_M(m,n)$ in $TS(\mathcal{P})$ with integer parameters m and n, such that M halts on (m,n) if and only if $halts_M(m,n)$ holds.*

Proof Let $STEP_M(\langle i,m,n\rangle, \langle i',m',n'\rangle)$ be the program formula for M as defined in [8, Theorem 2.1.15]. In our context, the formula is a quantifier free Presburger formula with the given parameters that uses only increment (by one) and

$$\begin{array}{llll} \textit{Basic elements:} & E & ::= & T_{\mathcal{B}} \mid \langle E,\ldots,E\rangle \mid \pi_i(E) \mid x \mid \mathit{ite}(F,E,E) \\ \textit{Sets:} & S & ::= & \{E \mid_{\overline{x}} F\} \mid \emptyset \mid S \cup S \mid S \cap S \mid S \setminus S \mid v \mid \mathit{ite}(F,S,S) \\ \textit{Formulas:} & F & ::= & F_{\mathcal{B}} \mid \neg F \mid F \wedge F \mid F \vee F \mid \forall x\, F \mid \exists x\, F \mid \\ & & & E = E \mid S \subseteq S \mid S = S \mid E \in S \end{array}$$

Figure 1: Well-formed expressions in $TS(\mathcal{B})$. The theory $\mathcal{B}$ has terms $T_{\mathcal{B}}$ and Formulas $F_{\mathcal{B}}$. It is assumed that all terms in $T_{\mathcal{B}}$ have sort β. Set variables are denoted by v and basic variables (tuple variables or variables of sort β) are denoted by x. The grammar omits sorts (type annotations) for ease of readability. For example in a set operation term $s_1 \diamond s_2$, it is assumed that both s_1 and s_2 have the same sort, in an element-of atom $t \in s$ it is assumed that if the sort of t is σ then the sort of s is $\{\sigma\}$, a tuple (t_1, t_2) has the sort $\sigma_1 \times \sigma_2$ provided that t_i has sort σ_i, etc.

decrement (by one) operations and equality. The formula holds exactly when $x = (i,m,n) \vdash_M (i',m',n') = x'$, i.e. when x' is the successor configuration of x in M. There is a finite number k of instructions, i.e. $1 \le i, i' \le k$. We can assume, without loss of generality, that M is such that the initial instruction is 1 and the final instruction is $k > 1$ and when the final instruction is reached then both registers are zero. Let halts_M be the following formula where s has the sort $\{\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})\}$:

$$\begin{array}{l} \mathit{halts}_M(m,n) \stackrel{\text{def}}{=} \exists s \exists l(\mathit{valid}_M(m,n,s,l)) \\ \mathit{valid}_M(m,n,s,l) \stackrel{\text{def}}{=} \\ \qquad s = \{\langle j,x,y\rangle \mid \langle j,x,y\rangle \in s \wedge \mathit{STEP}_M(x,y) \wedge 1 \le j \wedge j \le l\} \wedge \\ \qquad \{\langle \pi_0(z), \pi_1(z)\rangle \mid z \in s\} \cup \{\langle l, \langle k,0,0\rangle\rangle\} = \\ \qquad \{\langle 1, \langle 1,m,n\rangle\rangle\} \cup \{\langle \pi_0(z)+1, \pi_2(z)\rangle \mid z \in s\} \end{array}$$

First we prove the direction "$\Rightarrow$". Assume that M halts on (m,n). So there is a finite sequence of configurations $(x_j)_{1 \le j \le l}$ where $x_1 = \langle 1,m,n\rangle$, $x_l = \langle k,0,0\rangle$ and $x_j \vdash_M x_{j+1}$ for $1 \le j < l$. Let $s = \{\langle j, x_j, x_{j+1}\rangle | 1 \le j < l\}$. It is easy to check that $\mathit{valid}_M(m,n,s,l)$ holds.
Next we prove the direction "$\Leftarrow$". Assume that $\mathit{valid}_M(m,n,s,l)$ holds for some s and l. From the first equality it follows that all elements of s are of the form $\langle j,x,y\rangle$ where $1 \le j < l$ and $x \vdash_M y$. From the second equality it follows that s must be a sequence in j, i.e., for all j, $1 \le j < l$, there is a unique element $\langle j, x_j, y_j\rangle$ in s. Moreover, it follows that $x_1 = \langle 1,m,n\rangle$, for all j, $1 < j < l$, $x_j = y_{j-1}$, 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). $\boxtimes$

The following is an immediate consequence of the proof of Theorem 2.

Corollary 2 *$TS(P)$ is undecidable. Undecidability arises already for formulas of the form $\exists v \exists x \varphi$, where φ is quantifier free and uses at most three unnested comprehensions.*

```
type Config = (Integer, Integer, Integer)
steps as Set of (Integer,Config,Config)
length as Integer
[Action] halts_M(m as Integer, n as Integer)
  require valid_M(m, n, steps, length)
```

Figure 2: Model program P_M.

The construction of $halts_M$ in Theorem 2 shows that comprehensions together with pairing (or tuples) leads to undecidability of the one step reachability problem, because $valid_M$ can be used as an enabling condition of an action as illustrated in Figure 2, and the halting problem of 2-register machines is undecidable.

Only a small fragment of Presburger arithmetic is needed. In particular, divisibility by a constant is not needed. The proof of the theorem does not change if M is assumed to be a Turing machine (assume M has two input symbols and the configuration (i, m, n) represents a snapshot of M where i is the finite state of M, m represents the tape content to the left of the tape head and n represents the tape content to the right of the tape head), only the construction of `STEP` is different. However, in that case one needs to express divisibility by 2 to determine the input symbol represented by the lowest bit of the binary representation of m or n, which can be encoded using an additional existential quantifier.

Another consequence of the construction in Theorem 2 is that decidability of the bounded reachability problem cannot in general be obtained by fixing the model program or by limiting the the number of set variables (without disallowing them).

Corollary 3 *There is a fixed model program P_u over $TS(\mathcal{P})$ with one set-valued state variable, one integer-valued state variable, and an action symbol with two integer-valued parameters, such that the following problem is undecidable: given an action a, decide if a is enabled in P_u.*

Proof Let M_u be a 2-register machine that is *universal* in the following sense, given a Turing machine M and an input v (over a fixed alphabet), let $\ulcorner M, v \urcorner$ be an effective encoding of M and v as an input for M_u, so that M_u accepts $\ulcorner M, v \urcorner$ if and only if M accepts v. Such a 2-register machine exists and can be constructed effectively [25, Theorem 7.9]. Let P_u be like P_{M_u} in Figure 2. Let M be a Turing machine and v an input for M. Then $\mathtt{halts}_{M_u}(\ulcorner M, v \urcorner)$ is enabled in P_u iff (by Theorem 2) M_u halts on $\ulcorner M, v \urcorner$ iff M accepts v. ⊠

4.3 Basic model programs

A *basic* value or sort is a non-set value or sort. A parameter or state variable is *basic* if its sort is basic. We say that a model program is *basic* if all action parameters are basic, each state variable is either basic or a set of basic elements, and the initial values of set valued state variables are either fixed or defined by basic state variables.

Example 5 The model program P_u in Corollary 3 is not basic because the initial value of `steps` is undefined. The following model program on the other hand is basic, where `STEP` and `k` are the same as above.

```
[Action] halts(maxCounter as Integer, l as Integer)
  let steps = {(j,(i,m,n),(i',m',n')) | i,i' in {1..k}, j in {1..l},
              m,n,m',n' in {1..maxCounter}, STEP((i,m,n),(i',m',n'))}
  require {(j,x) | (j,x,y) in steps} union {(l,(k,0,0))} =
          {(1,(1,m,n))} union {(j+1,y) | (j,x,y) in steps}
```

It seems as if it is possible to express the halting problem just using bounded reachability of basic model programs. This is not the case as is shown in Section 5. Intuitively, a comprehension adds "too many" elements.

4.4 Undecidable extensions of basic model programs

If we add a `chooseSubset` construct to basic model programs that chooses nondeterministically a subset from a given set, we are again able to encode the halting problem as follows.

```
[Action]
halts(maxCounter as Integer, l as Integer)
  let steps = {(j,(i,m,n),(i',m',n')) | i,i' in {1..k}, j in {1..l},
              m,n,m',n' in {1..maxCounter}, STEP((i,m,n),(i',m',n'))}
  require {(j,x) | (j,x,y) in chooseSubset(steps)} union {(l,(k,0,0))} =
          {(1,(1,m,n))} union {(j+1,y) | (j,x,y) in chooseSubset(steps)}
```

This is easy to see, since a formula $A[\texttt{chooseSubset}(s)]$ is equivalent to the formula $v \subseteq s \wedge A[v]$ where v is a fresh existentially quantified variable, for example a parameter (assuming s does not include universally quantified variables) This allows us to construct the undecidable formula shown in the proof of Theorem 2.

An extension of basic model programs that leads to undecidability of the one step reachability problem is if we allow *set cardinality.* We can then express integer multiplication as follows, given two (non-negative) integers m and n: $m \cdot n \stackrel{\text{def}}{=} |\{1..m\} \times \{1..n\}|$. Also, if we allow *bag comprehensions* we can define the cardinality of a set s as $|s| \stackrel{\text{def}}{=} \{\!\{0 | x \in s\}\!\}[0]$. Either of these extensions allows us to effectively encode diophantine equations (e.g. $5x^2y + 6z^3 - 7 = 0$ is a diophantine equation). Let $p(\overline{x})$ be a diophantine equation and let $\texttt{P}(\overline{x})$ be an action whose enabling condition is the encoding of $p(\overline{x})$. Then $\texttt{P}(\overline{n})$ is enabled iff $\overline{n}$ is an integer solution for $p(\overline{x})$. The problem of deciding whether a diophantine equation has an integer solution is known as *Hilbert's 10th problem* and is undecidable [32].

5 Bounded reachability of basic model programs is decidable

We show that the bounded reachability problem of *basic* model programs over a background $\mathit{TS}(\mathcal{B})$ is decidable provided that $\mathit{Th}(\mathcal{B})$ is decidable, where $\mathit{Th}(\mathcal{B})$ is the closure of $\mathcal{B}$ under entailment, i.e., for an arbitrary closed first-order formula φ in the language of $\mathcal{B}$ it is decidable if $\varphi \in \mathit{Th}(\mathcal{B})$.

5.1 Stratified fragment of $\boldsymbol{TS}(\mathcal{B})$

The proof has two steps. First, we show that there is a fragment of $\mathit{TS}(\mathcal{B})$, denoted by $\mathit{TS}(\mathcal{B})_{\prec}$ that reduces effectively to $\mathcal{B}$. Second, we show that the bounded reachability problem of basic model programs over $\mathit{TS}(\mathcal{P})$ reduces to $\mathit{TS}(\mathcal{P})_{\prec}$. Let $\mathcal{B}$ be fixed. Let $V(\varphi)$ denote the collection of all set variables that occur in a formula φ over $\mathit{TS}(\mathcal{B})$.

Definition 5 ($\boldsymbol{TS}(\mathcal{B})_{\prec}$) A $\mathit{TS}(\mathcal{B})$ formula φ is *stratified* if

- φ has the form $\psi \wedge \bigwedge_{v\in V(\varphi)} v = S_v$, and
- the relation $\prec \stackrel{\text{def}}{=} \{(w, v) | v \in V(\varphi), w \in V(S_v)\}$ is well-founded.

The equation $v = S_v$ is called the *definition of* v in φ.

5.2 Reduction from basic model programs over $\boldsymbol{TS}(\mathcal{B})$

Theorem 3 *$\mathit{TS}(\mathcal{B})_{\prec}$ reduces effectively to $\mathcal{B}$.*

Proof Let φ be a stratified $\mathit{TS}(\mathcal{B})$ formula. We provide a series of transformations of φ, each of which preserves equivalence to the original formula, such that the final formula is a formula over the language of $\mathcal{B}$. Apply the following transformations to φ in the given order.

1. *Eliminate set variables.* Let $(v_i)_{i<k}$ be a fixed sequence of $V(\varphi)$ 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. Remove the set variables.

2. *Eliminate if-then-else terms.* Apply the following transformation repeatedly to atomic formulas α that contain *ite* subterms:

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

3. *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.

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

4. *Translate set operations.* Since all set variables v have been eliminated the only atomic set terms are comprehensions. Because of step 2, 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 typed so both sides have the same sort.

$$\begin{array}{rcl} \{\langle\overline{y}\rangle \mid \psi_1(\overline{y})\} \cup \{\langle\overline{y}\rangle \mid \psi_2(\overline{y})\} & \rightsquigarrow & \{\langle\overline{y}\rangle \mid \psi_1(\overline{y}) \vee \psi_2(\overline{y})\} \\ \{\langle\overline{y}\rangle \mid \psi_1(\overline{y})\} \cap \{\langle\overline{y}\rangle \mid \psi_2(\overline{y})\} & \rightsquigarrow & \{\langle\overline{y}\rangle \mid \psi_1(\overline{y}) \wedge \psi_2(\overline{y})\} \\ \{\langle\overline{y}\rangle \mid \psi_1(\overline{y})\} \setminus \{\langle\overline{y}\rangle \mid \psi_2(\overline{y})\} & \rightsquigarrow & \{\langle\overline{y}\rangle \mid \psi_1(\overline{y}) \wedge \neg\psi_2(\overline{y})\} \end{array}$$

5. *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.

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

6. *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.

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

7. *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.

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

After the above transformations we get a formula over the language of $\mathcal{B}$ that is logically equivalent to the original formula. ⊠

The following corollary is immediate using the decidability of Presburger arithmetic $\mathcal{P}$.

Corollary 4 *$TS(\mathcal{P})_{\prec}$ is decidable.*

We also get the following corollary that is the main result of this section.

Corollary 5 *Bounded reachability of basic model program over $\mathit{TS}(\mathcal{P})$ is decidable.*

Proof Let P be a basic model program over $\mathit{TS}(\mathcal{P})$ let φ be a reachability condition, and let k be a step bound. It is easy to see that $\psi = \mathit{Reach}(P, \varphi, k)$ can be written as a stratified $\mathit{TS}(\mathcal{P})$ formula: First, we can assume that there is only one action symbol (with a specific parameter that identifies a particular action). Since P is basic, the initial value of each state variable $v_{(0)}$ must be defined. In each step formula for step i, the value $v_{(i+1)}$ is given a definition that uses only variables or parameters from state i and parameters are basic. The definition can be written on a form that uses *ite* and is a top level equation of the generated formula. The only variables that are not given definitions are parameters, but all parameters are basic. Satisfiability of ψ in the language that includes the state variables reduces to entailment of the existential closure of ψ from $\mathit{TS}(\mathcal{P})$, which by Theorem 3, reduces to $\mathcal{P}$ and is thus decidable. $\boxtimes$

5.3 Encoding general array operations in $\boldsymbol{TS}(\mathcal{P})$

General integer arrays and array read and write operations are, strictly speaking, not in the $\mathit{TS}(\mathcal{P})$ fragment but can easily be encoded using tuples and comprehensions. For example, given an array variable v from integers to integers with the default value 0, encode it as the graph $\tilde{v}$ of v. The relation $\mathit{Read}(\tilde{v}, l, x)$ that holds when $v[l] = x$, can be defined through

$$\mathit{Read}(\tilde{v}, l, x) \stackrel{\text{def}}{=} \mathit{ite}(\{x\} = \{\pi_1(y) \mid y \in \tilde{v} \wedge \pi_0(y) = l\}, \mathit{true}, x = 0)$$

and the corresponding write operation $\mathit{Write}(\tilde{v}, l, x)$ can be defined through

$$\mathit{Write}(\tilde{v}, l, x) \stackrel{\text{def}}{=} \{y \mid y \in \tilde{v} \wedge \pi_0(y) \neq l\} \cup \{(l, x)\}.$$

Using this encoding one can for example transform the *Credits* model program in Example 3 into an equivalent model program over $\mathit{TS}(\mathcal{P})$.

6 Implementation using Z3

Z3 [12, 45] is a state of the art SMT solver. SMT generalizes Boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. Of particular relevance to model-programs, Z3 exposes a theory of extensional arrays, which has a built-in decision procedure. Thus, terms built up using the array constructs *read* and *write* are automatically subjected to the axioms (1) and (2). Constant arrays are also supported natively, such that axiom (3) can be obtained as a side-effect of declaring a constant array *const*(*default*).

Boolean algebras, also known as sets, are implemented natively in Z3 as a layer on top of the extensional array theory. Thus, adding and removing elements from a set is obtained by using *write*, set membership uses *read*, and the empty sets are the constant sets:

$$\begin{aligned} s' = s \cup \{x\} &\quad\leftrightarrow\quad s' = \mathit{write}(s, x, \mathit{true}) \\ s' = s \setminus \{x\} &\quad\leftrightarrow\quad s' = \mathit{write}(s, x, \mathit{false}) \\ x \in s &\quad\leftrightarrow\quad \mathit{read}(s, x) \\ \emptyset &\quad\leftrightarrow\quad \mathit{const}(\mathit{false}) \end{aligned}$$

The set operations $\cup, \cap, \setminus$ are encoded using a generalized *write*, which we will call *write-set*. It has the semantics:

$$\begin{gathered} \forall m\, m'\, m''\, x \quad (\mathit{read}(\mathit{write\text{-}set}(m, m', m''), x) = \\ \mathit{Ite}(\mathit{read}(m, x) = \mathit{read}(m', x), \mathit{read}(m'', x), \mathit{read}(m', x)), \end{gathered}$$

such that the set operations can be encoded using:

$$\begin{aligned} s \cup s' &\quad\leftrightarrow\quad \mathit{write\text{-}set}(\mathit{const}(\mathit{false}), s, s') \\ s \cap s' &\quad\leftrightarrow\quad \mathit{write\text{-}set}(\mathit{const}(\mathit{true}), s, s') \\ s \setminus s' &\quad\leftrightarrow\quad \mathit{write\text{-}set}(s', \mathit{const}(\mathit{false}), s) \end{aligned}$$

Z3 hides these encodings, such that expressions involving sets can be formulated directly using the usual set operations.

Support for constructs which are not included in Z3's API natively can be added via external axiomatization. Such axiomatizations typically require the use of quantifiers in Z3, which potentially makes the solver incomplete and can cause spurious models to be returned.

Fortunately, since model programs are executable, the feasibility of traces provided by the solver can easily be checked by simply executing them. While, in principle, the traces could be executed directly on the model program via the AsmL compiler, we use the approach to translate them to C# and execute the traces on C# code. This provides several benefits: we can conveniently use .Net API's for reflection, we can add auxiliary methods for evaluating and saving intermediate results for pinpointing error locations (in case an erroneous trace is provided by the solver) etc. Additionally, this eases the adoption of other languages for describing model programs. For example, NModel [34] uses C# as the modelling language. In this case, we would only need to provide a parser from C# to the internal abstract syntax to be able to use the framework.

In the following sections, we explain how the axiomatization of bags (multisets) and set comprehensions makes use of trace checking and model refinement.

Bags in Z3 AsmL makes use of abstract datatypes suitable for high-level reasoning, such as sets, bags etc. Since the bag type is not supported directly by Z3, they need to be axiomatized for Z3, which requires the use of quantifiers. As such, the axiomatization of bags makes an interesting case study about iterative refinement of Z3 models.

Since Z3 supports maps, bags can be axiomatized using maps: a bag is a map from some type T to a natural number. The axioms for bag operations are the following (we will use the notation $s[x]$ for reading the value of the map s at position x):

$$\begin{aligned}
\forall x, s_1, s_2.((s_1 \uplus s_2)[x] &\equiv s_1[x] + s_2[x]) \\
\forall x, s_1, s_2.((s_1 \cap s_2)[x] &\equiv \mathit{ite}(s_1[x] < s_2[x], s_1[x], s_2[x])) \\
\forall x, s_1, s_2.((s_1 \setminus s_2)[x] &\equiv \mathit{ite}(s_1[x] < s_2[x], 0, s_1[x] - s_2[x]))
\end{aligned}$$

The empty bag is defined in Z3 as $\emptyset = \mathit{const}(0)$ i.e. the default value of the map is 0. Following the bag axioms, adding an element to a bag is $s \cup \{x\} = \mathit{write}(s, x, s[x] + 1)$ and removing an element is $s \setminus \{x\} = \mathit{ite}(s[x] = 0, s, \mathit{write}(s, x, s[x] - 1))$.

Since Z3 supports quantifiers, adding axioms such as $\forall x, s_1, s_2.((s_1 \uplus s_2)[x] \equiv s_1[x] + s_2[x])$ is straightforward. Quantifiers are implemented via pattern matching, for example $(s_1 \uplus s_2)[x]$ would be a pattern in the given forall formula. However, relying on such axioms is incomplete as they are only expanded if search explicitly builds a subterm that matches the pattern. If the subterm is not encountered, the axiom is ignored.

There are essentially two ways how a returned model can be incorrect in the context of bag operations. The first one is the case where the interpretation of a bag operation does not follow the particular axiom. Let us look at the following simple example:

```
var bag as Bag of Integer = Bag {i}

Action AddToBag(param as Bag of Integer)
  bag := bag union param

...//actions not adding to bag

Invariant()
  bag <> Bag {}
```

The invariant states that the bag *bag* should never empty. Since the bag is initially non-empty, and it is only possible to add elements to the bag, the invariant clearly. However, by not instantiating the bag union axiom, the wrong interpretation to the union operator can be given (so the resulting interpretation is $\mathit{bag} \uplus \mathit{param} = \emptyset$) and an incorrect model returned.

The second way the model can fail is when Z3 applies the axioms correctly when building the bag terms, but then adds extra skolem constants to the map representing the bag for checking array inequalities. That is, the model would be correct if the extra skolem constants would be ignored, but is incorrect together with them. The following example explains this.

```
var bagA as Bag of Integer = Bag{i}
var bagB as Bag of Integer = Bag{k}

Action AddToBagA(param as Bag of Integer)
  bagA := bagA union param

Action AddToBagB(param as Bag of Integer)
  bagB := bagB union param

Invariant()
  bagA <> bagB
```

The invariant here is that bag_A is never equal to bag_B. Assuming $i \neq k$, a state breaking the invariant can be reached in 2 steps, by adding i to bag_B ($param_0 = Bag\{i\}$) and k to bag_A ($param_1 = Bag\{k\}$). Z3 might give a model where the parameters include i and k respectively (as they should), but also include some irrelevant values. This means that the axiom interpretations are correct on the values that actually matter, but are not correct on the auxiliary skolem constants. The trace checker would of course reject such a trace.

Array property fragment We now describe the general approach we use in dealing with these kinds of errors. In [10], Bradley et al describe a decision procedure for a fragment of first other logic with the theory of arrays. The so called array property is a formula of the form $(\forall\bar{i})(\phi_I(\bar{i}) \rightarrow \phi_V(\bar{i}))$, where $\bar{i}$ is a vector of variables, and $\phi_I(\bar{i})$ and $\phi_V(\bar{i})$ are the index guard and the value constraint, following their respective restrictions. The restriction on the value constraint is that the quantified variables $i \in \bar{i}$ only appear as array reads (which in turn must not be nested). A formula with an array property formula $\psi[(\forall\bar{i})(\phi_I(\bar{i}) \rightarrow \phi_V(\bar{i}))]$ is equisatisfiable with the formula $\psi[\bigwedge_{i \in I_\psi}(\phi_I(i) \rightarrow \phi_V(i))]$, where I_ψ is the *index set* of the formula ψ. The index set of a formula consists of the terms used to read from an array, and certain subterms of the index guards.

We do not follow the reduction of Bradley literally, for example we do not explicitly remove array writes. Instead we lazily repair the model using the index terms when needed. Thus the index set generation differs slightly from that of Bradley. The index set I_{MP} of a model program MP consists of two sets, the read set I_R and the write set I_W. The read set is the set of terms used in accessing the arrays in the model program directly, i.e. $I_R = \{i \mid a[i] \in MP\}$, where $a[i]$ is an array access. The write set consists of terms used to refer to array positions which are written to, together with its 2 closest neighbors, so each array write generates 3 terms: $I_W = \{j \mid i-1 \leq j \leq i+1, a[i \leftarrow x] \in MP\}$. The reason why $i-1$ and $i+i$ are included in the set is that an array write in some context $C[a[i \leftarrow x]]$ should be rewritten as $C[b] \wedge b[i] = e \wedge \forall j.(j \neq i \rightarrow a[j] = b[j])$. To meet the syntactic requirement of the array property fragment, the third conjunct has to be rewritten as $\forall j.(j \leq i-1 \vee i+1 \leq j \rightarrow a[j] = b[j])$.

The terms $i-1$ and $i+1$ meet the syntactic requirement to be included in the index set.

Looking back at the first example, if trace evaluation shows that the interpretation of the union operator is wrong, we can fix the interpretation on the index set, which includes i. The instantiation $(bag \uplus param)[i] = bag[i] + param[i]$ is added to the original formula, which makes the original interpretation explicitly invalid. Feeding the new formula to the solver, it correctly finds that there can be no model for this formula.

The same approach does not solve the problem explained in the second example, since even if we fix the interpretation on the extra skolem constants, the solver will pick new ones. This is where we can again make use of the generated index set. Any value in the bag that is not equal to some value of an index term is ignored, i.e. it is removed from the bag. This approach gives us the correct trace.

This approach is complete in case bags of integer bags. The bag axioms can be expressed in the array property fragment, considering the bag axioms to be axiom schemes, i.e., they can be instantiated to specific cases where bag operation are used in a formula. Doing that, quantification over bag variables can be eliminated. For example in a formula ψ where we have a bag union, the following rewrite steps can be taken:

$$\begin{array}{rl} \psi[s_1 \uplus s_2] & \rightsquigarrow \\ \psi[s] \wedge s = s_1 \uplus s_2 & \rightsquigarrow \\ \psi[s] \wedge \forall x.(s[x] = s_1[x] + s_2[x]) & \end{array}$$

The formula $\forall x.(s[x] = s_1[x] + s_2[x])$ falls in the array property fragment fragment (assuming the index guard to be *true*). The same is true for intersection and difference axioms.

Encoding type restrictions While integer bag axioms fall into the decidable fragment of array theory, the implementation in Z3 is not completely straightforward. One reasons is that there is no built in type for natural numbers, but bags are maps from some type T to $\mathbb{N}$. A naive encoding of bags in Z3 would use integers instead of natural numbers, but this can quickly lead to spurious counterexamples, since then bags could contain a *negative* number of elements.

Similar problems arise when encoding other datatypes, for example enumerated types (enums). With the naive translation of enums as integers, Z3 can give a model that is valid for the formula, but not for the original model program, since the enum is outside of the expected range.

Thus we need to add axioms which restrict the general integer and map types to more specific ones. For bags the predicate specifying that a map s is a bag would be $IsBag(s) \equiv \forall x.(s[x] >= 0)$. For every implicitly existentially quantified bag variable b in the formula, we could add the application of the predicate on b to the formula, i.e. $IsBag(b)$. This guarantees that when Z3 assigns a value to b, its range can only be $\mathbb{N}$. For an enum variable x where the enum range is from $0..n$, the restriction is simply $0 \leq x \wedge x \leq n$.

This approach however does not work in the general case, because datatypes can be nested, for example there can be a set of bags. In this case there would be no explicit bag variables to put the restriction on. Thus we need to recursively generate axioms for nested datatypes. In the case of a set of bags, one axiom would define a predicate stating that the set can only contain elements which are bags and the second axiom for the predicate stating the range of bags must be non-negative.

This checking can be extended to other map based datatypes. The general form of a map axiom is

$$\begin{aligned} &\forall s.(IsSomeMap(s) \equiv \\ &\quad \forall x.(\mathit{RangeRestriction}(s[x]) \wedge \\ &\quad (s[x] \neq DefaultValue \Rightarrow \mathit{DomainRestriction}(x)))). \end{aligned}$$

For example the axioms for the type set of bag of 4-element enum are:

$$\begin{aligned} &\forall x, s.(IsSetofBagOfEnum(s) \equiv \\ &\quad \forall x.(s[x] \neq false \Rightarrow IsBagOfEnum(x))) \end{aligned}$$

and

$$\begin{aligned} &\forall x, s.(\mathit{IsBagOfEnum}(s) \equiv \\ &\quad \forall x.(s[x] \geq 0 \wedge (s[x] \neq 0 \Rightarrow 0 \leq x \wedge x \leq 3))). \end{aligned}$$

If we now have a parameter x of type set of bag of enum, the restriction on it would be $\mathit{IsSetOfBagOfEnum}(x)$.

Set comprehensions One of the biggest advantages of using a language like AsmL together with bounded model checking is the availability of comprehensions in the language. When checking programs in a C or a Java-like language, loops need to be unrolled, making the approach unfeasible for programs which have longer loop runs. With comprehensions support, many loops (especially for-next and foreach loops) can be expressed via comprehensions. These can then be described as a single step in a bounded reachability formula, instead of unrolling them. Unfortunately, this also means that the bounded reachability formula becomes undecidable in the general case (as opposed to the case where all loops are unrolled). However, it is possible to isolate fragments in the set comprehension formula which are decidable (as shown in the previous sections) or have good heuristics for solving them.

A comprehension definition $\forall y(y \in s' \leftrightarrow \exists x(y = t[x] \wedge x \in r \wedge \varphi[x]))$ does not immediately admit existential quantifier elimination, since the quantifier appears in a negative context. One way to eliminate the quantifier was explained in Section 2. This method applies when the element term is invertible and there is only one comprehension variable. If r is a range term in the form $\{m..n\}$, the formula is in the array property fragment.

Another way to remove the existential quantifiers even if the element term is not invertible, or there are more than one variable in the comprehension is by rewriting the comprehension definition into two definitions

$$\forall \overline{y}(\overline{y} \in \overline{r} \wedge \varphi[\overline{x}] \rightarrow t[\overline{x}] \in s')$$

and

$$\forall y(y \in s' \rightarrow \exists \overline{x}(y = t[\overline{x}] \wedge \overline{x} \in \overline{r} \wedge \varphi[\overline{x}])).$$

The latter can then be skolemized and admits the form

$$\forall y(y \in s' \leftrightarrow y = t[\overline{sk(y)}] \wedge \overline{sk(y)} \in \overline{r} \wedge \varphi[\overline{sk(y)}]).$$

While these equations are not in array property fragment, using the index set for formula refinement is a very good heuristic.

Incremental refinement and axiom repair We use the iterative refinement and index set based method for axiom repair for both bag and set comprehension axiomatizations. The general refinement loop works as follows. A trace provided by the Z3 solver is executed step by step on the generated program via reflection, and after each step it is checked whether the state given in the model matches the actual state. If it does not match, we know at which action the mismatching state was reached. By examining the statements in the action we can check which of the axioms was not instantiated correctly and on which variables, consequently pinpointing the exact error source. The interpretation on this operation can then be fixed, by adding new formulas to the original model formula, giving explicit instantiations of the "misinterpreted" axiom on each index term. The new formula can be sent back to Z3 and a new trace obtained, which might again be erroneous (on some other axiom application), in which case it is again fixed and the refinement loop continues. This approach is similar to CEGAR [11] (counter example guided abstraction refinement), the main difference being 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.

7 Experiments

As the concrete input language of model programs we use a subset of AsmL [3]. Model programs have the same meaning as in the Spec Explorer tool [42] or in NModel [34]. The difference is that here the analysis is done symbolically using a theorem prover, rather than using explicit state exploration through execution. An action rule is given by a method definition annotated with the [Action] attribute, with the method name being the action symbol and the method signature providing the signature term for the action. The conjunction of all the require-statements defines the precondition. The main body of the method defines the update rule, where parallel update is the default in AsmL.

```
var counter as Map of Integer to Integer = {0->$n$, 1->$n$}
[Action]
Execute(bar as Integer)
   require $bar \in counter$
   if $counter(bar) = 1$
      $counter := RemoveAt(counter, bar)$
   else
      $counter(bar) := counter(bar) - 1$
```

Figure 3: *Count*(n) model program.

```
var current as Integer
[Action]
Execute(bar as Integer)
   require $current \leq bar$
   $current := bar$
```

Figure 4: Model program *Order*. It imposes a linear order on the execution of bars where execution of bar i has to precede execution of bar j if $i < j$. For example, if the bars are a, b and c, where $a < b < c$, this model program essentially defines the regular expression `Execute`$(a)^*$`Execute`$(b)^*$`Execute`$(c)^*$.

The *Credits* model program in Example 3 illustrates a typical usage of model-programs as protocol-specifications. The actions use parameters, maps and sets are used as state variables and a comprehension expression is used to compute a set. Here the reachability condition is the negated invariant. One of the preconditions is missing (indicated by `bug`). There is a two-action trace leading to a state where the invariant is violated due to this. Asking Z3 with a bound of 2 or more steps (in an incremental mode) produces that trace `Req(0,1),Res(0,0)` in 21ms.

We are also investigating this analysis technique in the context of some embedded real time scheduling problems [23]. In some cases, in particular if the formula is not satisfiable, the solver may stall while trying to exhaust the search space. In this case it may be useful to apply composition to constrain the search space. This is reminiscent to adding user defined lemmas to the theorem prover. A typical example would be the use of a model program that fixes the order of some actions relative to some other actions, tantamount to user controlled partial order reduction. The *Count* example in Figure 3 is a distilled version of the counting aspect of the partiture model from [23]. There are a number of indexed counters that can be decremented. Each index corresponds to an atomic part of a schedule (called a *bar*) and the count for that bar specifies the total number of times that this bar can be executed.

Suppose that there are two bars, 0 and 1, the initial count for both bars is some value n, and that we are interested in finding a sequence of actions that

Table 1: Running times of the bounded reachability checking of the *Count* example in Z3 for different values of the counting limit n and step bound k.

Model program	**Step bound**	**Verdict**	**Time (in seconds)**
Count(5)	10	Sat	0.14
Count(5) $\oplus$ *Order*	10	Sat	0.14
Count(5)	9	Unsat	1.5
Count(5) $\oplus$ *Order*	9	Unsat	0.16
Count(8)	16	Sat	2.2
Count(8) $\oplus$ *Order*	16	Sat	1.4
Count(8)	15	Unsat	152
Count(8) $\oplus$ *Order*	15	Unsat	1

exhausts all the counters, i.e. the reachability condition φ is '`counter` is the empty map'.

If the step bound k is smaller than $2n$ then $\mathit{Reach}(\mathit{Count}(n), \varphi, k)$ is clearly unsatisfiable. The size of the search space of the theorem prover grows exponentially in k in this case (see Table 1). In this simplified example we can use the knowledge that the order of decrementing the different counters is irrelevant and fix such an order using another model program *Order* shown in Figure 4.

8 Related work

The *unbounded* reachability problem for model programs without comprehensions and with parameterless actions is shown to be undecidable in [15], where it is called the hyperstate reachability problem.

General reachability problems for transition systems are discussed in [37] where the main results are related to guarded assignment systems. A guarded assignment system is a union of guarded assignments or update rules. In the proof of Theorem 2, s is a *shifted pairing* [20] of a valid computation. The case when $\mathcal{B} = \mathcal{P}$ in Theorem 3 is related to decidable extensions of $\mathcal{P}$ that are discussed in [5].

The definition of $\mathit{TS}(\mathcal{B})_{\prec}$ in the proof of Theorem [**?**] is equivalent to the following construction in the case when all tuples are required to be flat. Let L_0 be the language of $\mathcal{B}$ and let $\mathcal{B}_0 = \mathcal{B}$. Given L_i and $\mathcal{B}_i$, create L_{i+1} and $\mathcal{B}_{i+1}$ as follows: expand L_i with a relation symbol R_φ of arity n for each L_i-formula $\varphi(x_1, \ldots, x_n)$ and add the definition $\forall \overline{x}(R_\varphi(\overline{x}) \leftrightarrow \varphi(\overline{x}))$ to $\mathcal{B}_i$. Now $\mathit{TS}(\mathcal{B})_{\prec}$ corresponds to $\bigcup_i \mathcal{B}_i$ as follows. Due to the well-founded ordering, each set variable v with the definition $v = \{\langle \overline{x} \rangle \mid_{\overline{x}} \varphi(\overline{x})\}$ corresponds to a relation symbol R_φ. Given a formula φ in $\mathit{TS}(\mathcal{B})_{\prec}$, it corresponds thus to a formula φ_k in $\mathcal{B}_k$ for some k. The statement follows by using the theorem of the existence of definitional expansions [24, Theorem 2.6.4] to reduce φ_{i+1} in L_{i+1} to an equivalent φ_i in L_i.

The full fragment $\mathit{TS}(\mathcal{P})$ is also part of the data structures that are allowed in the Jahob verification system [9]. The translation scheme described in [9, Appendix B] can be used to perform the transformations described in steps 3–7 of the proof of Theorem 3. In Jahob, the translation leads to a first-order formula that can be proven by a resolution theorem prover. Here the purpose of the translation was to show the reduction from $\mathit{TS}(\mathcal{B})_{\prec}$ to $\mathcal{B}$; the verification problem is more specific here, it addresses only bounded reachability by using SMT.

The decidable fragment BAPA [30] is an extension of Boolean algebra with $\mathcal{P}$. The sets in BAPA are finite and bounded by a maximum size and the cardinality operator is allowed, which unlike for $\mathit{TS}(\mathcal{P})_{\prec}$, does not enable encoding of multiplication. 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 [35] where summation constraints can be used to express set cardinality (without using bag cardinality that is also included in the fragment). A related fragment of integer linear arithmetic with a star operator is considered in [36].

In [10] a decision procedure for an array fragment is introduced and in [41] it is shown that this decision procedure can be applied to the bounded reachability problem of a subclass of model programs. However, the fragment in [10] does not allow expressions that include universally quantified variables, other than the variable itself, to occur in array read operations. Consequently, comprehensions where the comprehension expression is not invertible are not covered in [41]. In [22] another fragment of arrays is considered that allows universal variables in array read expressions that relate consecutive elements or talk about periodic properties.

A technique for translating common comprehension expressions (such as *sum* and *count*) into verification conditions is presented in [31] within the Spec# verification system that uses Boogie to generate verification conditions for SMT solvers [4]. The system does not support arbitrary set comprehension expressions as terms but allows axioms that enable explicit definitions of sets.

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 [28] for constructing interpolants for these theories. This work is based on the results of [29], where it is shown that the quantifier-free theories of arrays, sets and multisets can be reduced to quantifier-free theories of uninterpreted symbols with equality, constructors and Presburger arithmetic.

Using SAT for bounded reachability of transition systems was introduced in in [6] and the extension to SMT was introduced in [1]. Besides Z3 [12], other SMT solvers that support arrays are described in [2, 40]. The formula encoding we use [41] into SMT follows the same scheme but does not unwind comprehensions and makes the action label explicit. The explicit use of the action label is needed to compose model programs [43], that can be used for scenario oriented verification [41]. This composition is somewhat related to composition of modules in SAL 2 [13].

Our quantifier elimination scheme is inspired by [10], and refines it by using model-checking to implement an efficient incremental saturation procedure on top of the SMT solver. The work here extends the work in [41] through support for set comprehensions with multiple comprehension variables and non-invertible comprehension expressions, as well as bag (multi-set) axioms. A recent application of the quantifier elimination scheme has been pursued by [27] in the context of railway control systems.

The following problems have not been addressed yet. Bounded reachability of model programs that use nested comprehensions, including for example sets and bags, is interesting for analysis of general purpose algorithms, see e.g. [21]. Given the (computational) complexity of $\mathcal{B}$, what is the complexity of $\mathit{TS}(\mathcal{B})_{\prec}$? It seems that a $\mathit{TS}(\mathcal{B})_{\prec}$ formula can be exponentially more succinct than the corresponding $\mathcal{B}$ formula. So, the complexity of $\mathit{TS}(\mathcal{P})_{\prec}$ could thus be $2^{2^{2^{cn}}}$, since the complexity of $\mathcal{P}$ is $2^{2^{cn}}$ [14]. The proper instantiation of array indices and avoidance of false models generated by an SMT solver, due to the inherent incompleteness of the triggering mechanism of universally quantified axioms, is an important open problem in the general case.

References

[1] 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.

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

[3] AsmL. http://research.microsoft.com/fse/AsmL/.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[5] A. Bès. A survey of arithmetical definability, A tribute to Maurice Boffa, Special Issue of Belg. Math. Soc., pages 1–54, 2002.

[6] 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.

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

[8] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[9] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. On using first-order 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.

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

[11] 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.

[12] 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.

[13] L. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th Int. Conf., (CAV 2004)*, volume 3114 of *LNCS*, pages 496–500. Springer, 2004.

[14] M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAMAMS: Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, pages 27–41, 1974.

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

[16] W. Grieskamp and N. Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In *5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM)*, 2006.

[17] 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.

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

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

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

[21] Y. Gurevich, M. Veanes, and C. Wallace. Can abstract state machines be useful in language theory? *Theor. Comput. Sci.*, 376(1):17–29, 2007.

[22] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about arrays? In R. Amadio, editor, *Proc. of the 11th Int. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'08)*, LNCS. Springer, 2008.

[23] J. Helander, R. Serg, M. Veanes, and P. Roy. Adapting futures: Scalability for real-world computing. In *Proceedings Real-Time Systems Symposium (RTSS 2007)*, pages 105–116. IEEE, 2007.

[24] W. Hodges. *Model theory.* Cambridge Univ. Press, 1995.

[25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

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

[27] S. Jacobs and V. Sofronie-Stokkermans. Applications of hierarchical reasoning in the verification of complex systems. *ENTCS*, 174(8):39–54, 2007.

[28] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE 2006)*, pages 105–116. ACM, 2006.

[29] D. Kapur and C. G. Zarba. A reduction approach to decision procedures, 2006.

[30] V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean algebra with Presburger arithmetic. In R. Nieuwenhuis, editor, *CADE 2005*, volume 3632 of *LNAI*, pages 260–277. Springer, 2005.

[31] R. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *9th Workshop on Formal Techniques for Java-like Programs, FTfJP 2007*, Berlin, Germany, July 2007.

[32] Y. V. Matiyasevich. *Hilbert's tenth problem.* MIT Press, 1993.

[33] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

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

[35] R. Piskac and V. Kuncak. Decision procedures for multisets with cardinality constraints. In F. Logozzo, D. Peled, and L. D. Zuck, editors, *VMCAI*, volume 4905 of *LNCS*, pages 218–232. Springer, 2008.

[36] R. Piskac and V. Kuncak. On Linear Arithmetic with Stars. Technical Report LARA-REPORT-2008-005, EPFL, 2008.

[37] 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.

[38] SMB2. http://msdn2.microsoft.com/en-us/library/cc246482.aspx, 2008.

[39] Spec Explorer. http://research.microsoft.com/specexplorer.

[40] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS'01*, pages 29–37. IEEE, 2001.

[41] 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.

[42] 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.

[43] M. Veanes, C. Campbell, and W. Schulte. Composition of model programs. In J. Derrick and J. Vain, editors, *FORTE'07*, volume 4574 of *LNCS*, pages 128–142. Springer, 2007.

[44] M. Veanes and W. Schulte. Protocol modeling with model program composition. In *FORTE'08*, volume 5048 of *LNCS*, pages 324–339. Springer, 2008.

[45] Z3. http://research.microsoft.com/projects/z3, released September 2007.