An SMT Approach to Bounded Reachability Analysis of Model Programs

Margus Veanes[1], Nikolaj Bjørner[1], and Alexander Raschke[2⋆]

[1] Microsoft Research, Redmond, WA, USA
{margus,nbjorner}@microsoft.com
[2] University of Ulm, Ulm, Germany
alexander.raschke@uni-ulm.de

Abstract. Model programs represent transition systems that are used to specify expected behavior of systems at a high level of abstraction. The main application area is application-level network protocols or protocol-like aspects of software systems. 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 analysis of a fragment of model programs. 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 the comprehension expressions.

1 Introduction

Model programs [20] are used to describe protocol-like behavior of systems at a high level of abstraction, the main application area being application-level network protocols. Model programs typically use abstract data types such as sets and maps, and comprehensions to express complex state updates. Protocols are abundant; we rely on the reliable sending and receiving of email, multimedia, and business data. But protocols, such as the Windows network file protocol SMB (Server Message Block), can be very complex and hard to get right. They require careful design to guarantee reliability and failure resilience; they require careful and efficient implementations; and they require careful documentation and interoperability testing, so that different vendors understand the same protocol. The use of model programs to model such complex protocols is an emerging practice in the software industry [14].

⋆ Part of this work was done during the authors visit at Microsoft Research.

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 incremental SMT solving to perform bounded reachability analysis of a fragment of model programs. We define the formal framework and describe the implementation to Z3 [28, 11]. The translation from a model program into a verification condition of Z3 is and involves lazy elimination of quantifiers that result from the comprehension expressions.

The use of SMT solvers for automatic software analysis has recently been introduced [2] as an extension of SAT-based bounded model checking [5]. 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 [22]. The formula is generated after preprocessing of the program. The preprocessing yields a normalized program where all loops have been eliminated by unwinding the loops up to a fixed bound.

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 paper 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. Allowing arbitrary comprehensions quickly leads to undecidability. We identify a fragment of model programs using the array property fragment [7] 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 [26] can be used for scenario oriented or user directed analysis.

2 Model programs

The semantics of model programs in their full generality builds on the abstract state machine (ASM) theory [15]. Model programs are primarily used in model-based testing tools like Spec Explorer [1, 25] where one of the supported input

languages is the abstract state machine language AsmL [4, 16]. The NModel tool [23, 20] and Spec Explorer 2007 [14] use plain C# for describing model programs. Spec Explorer 2007 uses, in addition, a coordination language Cord for scenario control [13] and model composition. Typically, a model program makes use of a rich background theory [6] $\mathcal{T}$, 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.

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. [7]), 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:

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

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 $\overline{s}$ *defines* s by (5).

$$\forall y(y \in \overline{s} \leftrightarrow \exists x(y = t[x] \wedge x \in r \wedge \varphi[x])). \tag{5}$$

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.)

$$\forall y(y \in \overline{s} \leftrightarrow t^{-1}[y] \in r \wedge \varphi[t^{-1}[y]] \wedge \psi_t[y]) \tag{6}$$

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.

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 $\forall y(y \in \overline{s} \leftrightarrow y - m \in \{1..c\})$, which reduces to $\forall y(y \in \overline{s} \leftrightarrow 1 \leq y - m \wedge y - m \leq c)$.

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 $\forall y(y \in \overline{s} \leftrightarrow y/2 \in \{1..c\} \wedge \mathit{Divisible}(y, 2))$, or equivalently $\forall y(y \in \overline{s} \leftrightarrow 2 \leq y \wedge y \leq 2 \cdot c \wedge \mathit{Divisible}(y, 2))$.

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* [7] 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 [18].

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*. The interpretation of a value term t is uniform in all models of $\mathcal{T}$ and is denoted by $[\![t]\!]$, i.e., $[\![t]\!] = \{s : s =_{\mathcal{T}} t\}$. As the *universe of values* we consider the set of all $[\![t]\!]$ for value terms t.

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. A term $t = f(x_1, \ldots, x_n)$ where $x_i \in \Sigma^{\text{var}}_{S_i}$, for $1 \leq i \leq n$, is called a *signature term for* f.

An *action* is value term $f(t_1, \ldots, t_n)$ where f is an action symbol. We also say *action* for $[\![f(t_1, \ldots, t_n)]\!] = f([\![t_1]\!], \ldots, [\![t_n]\!])$

2.4 Update rules

As update rules we consider basic ASMs [15] enriched with $\mathcal{T}$. Thus, update rules are built using: the empty update rule or skip; simple assignment of a term to a state variable; conditional update rule; parallel update rule; update rule with a local let-binding. As the concrete language in this paper we use the corresponding fragment of AsmL [16].

2.5 Model program definition

Intuitively, a model program describes a transition system with a set of states and transitions labeled by actions where the transition relation is induced by the action rules of the model program.

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

- V_P is a finite subset of Σ^{var}, called the *state variables of* P;
- A_P is a finite subset of $\Sigma_{\mathbb{A}}$, called the *action symbols of* P;
- I_P is a formula over $\Sigma^{\text{static}} \cup V_P$, called the *initial state condition of* P;
- R_P is a family $\{R^f_P\}_{f \in A_P}$ of *action rules* $R^f_P = (F^f_P, G^f_P, U^f_P)$, where
  - F^f_P is a signature term for f called the *action signature term of* R^f_P.
  - G^f_P is a formula called the *guard* or *enabling condition of* R^f_P;
  - U^f_P is an update rule called the *update rule of* R^f_P.

  It is required that all symbols that occur in R^f_P are in $\Sigma^{\text{static}} \cup V_P \cup \Sigma(F^f_P)$.

Let P be a fixed model program. Let $\Sigma(P)$ stand for $\Sigma^{\text{static}} \cup V_P$. A *$P$-state* is a first-order $\Sigma(P)$-structure that models $\mathcal{T}$. Given a P-state S, an extension of S with parameters $\{x_i \mapsto v_i\}_{1 \leq i \leq n}$ is denoted by $(S; \{x_i \mapsto v_i\}_{1 \leq i \leq n})$. Given a first-order structure S, the *reduction* of S to a sub-signature X is denoted by $S \restriction X$.

Definition 2. Let $f \in A_P$, let S be a P-state and let $f(x_1, \ldots, x_n) = F^f_P$. An action $f(t_1, \ldots, t_n)$ is *enabled* in S if $(S; \{x_i \mapsto [\![t_i]\!]\}_{1 \leq i \leq n}) \models G^f_P$.

We use the notion of *firing* of an update rule U in a state S [15], denoted here by $\mathit{Fire}(S, U)$, that yields the updated state.

Definition 3. Let S_1 be a P-state and let $a = f(t_1, \ldots, t_n)$ be an action that is enabled in S_1. Let $f(x_1, \ldots, x_n) = F^f_P$ and let

$$S_2 = \mathit{Fire}((S; \{x_i \mapsto [\![t_i]\!]\}_{1 \leq i \leq n}), U^f_P) \restriction \Sigma(P).$$

Then *a causes a transition from S_1 to S_2*, or S_2 is the result of *executing a* from state S_1.

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 4. 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 $a \in L$ 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 $(s_i, a_i, s_{i+1})_{i<k}$ in $[\![P]\!]$ where s_0 is an initial state of $[\![P]\!]$. A run may be empty.

2.6 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 update rules are the parallel compositions [15], denoted by '$\|$', of the update rules of the component model programs. The formal definition is a simplification of the parallel composition of model programs from [26].

In order to avoid parameter renaming, it is convenient to assume that action rules that are composed, use fixed formal parameter names, i.e. the signature term for each action symbol is fixed and can be omitted from the definition of an action rule.

Definition 5. Let P and Q be model programs such that $A = A_P = A_Q$. The *composition* $P \oplus Q$ is $(V_P \cup V_Q, A, I_P \wedge I_Q, (G_P^f \wedge G_Q^f, U_P^f \parallel U_Q^f)_{f \in A})$.

Composition can be used to do scenario oriented modeling [26]. In Section 5 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 $\Sigma(P)$-formula. The main problem we are addressing is whether φ is reachable in P within a given bound.

Definition 6. 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 [15].

3.1 Step formula creation

The basic idea of generating a reachability formula for bounded model checking and to use SAT to check this formula was introduced in [5]. 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. During the translation all set comprehension terms are replaced by constants that define them as described above. If a comprehension term is normalizable, the generated definition has the form as shown in (6).

A translation from an update rule U to a step formula for step nr i, denoted by $U[i]$, is defined by induction over the structure of update rules. For an assignment update rule '$x := t$', $(x := t)[i]$ is the equality $x[i+1] = t[i]$. If U is a conditional update rule '*if* φ *then* U_1 *else* U_2' let $X_j \subseteq V_P$ be the state variables assigned in U_k but not in U_j, for $\{j,k\} = \{1,2\}$. The translation of $U[i]$ is

$$(\varphi[i] \wedge U_1[i] \wedge_{x \in X_1} x[i+1] = x[i]) \vee (\neg\varphi[i] \wedge U_2[i] \wedge_{x \in X_2} x[i+1] = x[i])$$

For a parallel update rule, $(U_1 \parallel U_2)[i]$ is $U_1[i] \wedge U_2[i]$.

Consider an action symbol $f \in A_P$. Let $X \subseteq V_P$ be the state variables not assigned in f. The step formula for step i generated for the action rule R_P^f is:

$$R_P^f[i] \quad \stackrel{\text{def}}{=} \quad G_P^f[i] \wedge U_P^f[i] \wedge \bigwedge_{x \in X} x[i+1] = x[i]$$

Intuitively this means that the updates can take place provided that the action is enabled and all state variables not assigned by the action rule preserve their old values.

There is a variable $\mathit{action}[i]$ of sort $\mathbb{A}$ for each step nr i. Let $\mathit{skip} = \mathit{default}_{\mathbb{A}}$ be the action that "skips" a step. Let $\mathit{Skip}[i]$ be the formula:

$$\mathit{Skip}[i] \quad \stackrel{\text{def}}{=} \quad \bigwedge_{x \in V_P} x[i+1] = x[i]$$

Finally, the step formula $P[i]$ for P is:

$$P[i] \quad \stackrel{\text{def}}{=} \quad (\mathit{action}[i] = \mathit{skip} \wedge \mathit{Skip}[i]) \vee \bigvee_{f \in A_P} (\mathit{action}[i] = F_P^f[i] \wedge R_P^f[i])$$

The translation assumes that the signatures of all signature terms of all actions are pairwise disjoint. In other words, each action uses unique parameter names for its parameters.

3.2 Reachability

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

$$Reach(P, \varphi, k) \quad \stackrel{\text{def}}{=} \quad I_P \wedge \Big(\bigwedge_{0 \leq i < k} P[i] \Big) \wedge \Big(\bigvee_{0 \leq i \leq k} \varphi[i] \Big) \tag{8}$$

Recall from above, that during the creation of $P[i]$ comprehension terms are given explicit definitions and replaced by corresponding Skolem constants in $P[i]$, thus the formula $P[i]$ is quantifier free (provided that quantifiers are not used in conditions of conditional update rules or in *if-then-else* terms). Recall also the assumption that these definitions are part of $\mathcal{T}_S$ for the corresponding set sort S. Introduce the function *RemoveSkips* which removes all the *Skip* actions from the trace. We can state the following theorem that follows from the construction of $P[i]$ and the definition of a model program.

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

3.3 Array model programs and quantifier elimination

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 [7]. An example of a model program in this fragment is the Credits model program in Figure 1. The model program is explained in detail in [27].

Lemma 1. *Let P be an array model program and assume that all set comprehension definitions of P are normalizable 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 [7, 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 [7, Definition 9].

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

```
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 ∈ window and c > 0
  requests := requests.Add(m, c)
  window := window − {m}

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

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

Fig. 1. *Credits* model program. Specifies how a client and a server need to use message ids, based on a sliding window protocol.

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 $\psi = \mathit{Reach}(P, \varphi, k)$. 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 Figure 1, 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 4 explains the way we use Z3 lazily.

4 Implementation using Z3

Z3 [11, 28] 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*). Enumerations are translated into integers, and for maps whose range consists of non-negative integers we assign *default* to a negative number.

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:

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

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

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

such that the set operations can be encoded using:

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

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

Map comprehensions, on the other hand, are not supported over Z3's API. As explained before, we are therefore using a reduction in the style of [7] in order to handle comprehensions. Our reduction, however, remains hybrid in two respects. First, our reduction does not require eliminating *write*, which would be necessary to follow the approach in [7] literally, instead we use the built-in support for extensional array constructs, together with *write-set*. Second, we are using the API of Z3 to supply an incremental decision procedure for comprehensions. We will explain how this is achieved in the following.

Z3's API exposes the method `AssertCnstr` - to assert a logical formula, and the method `CheckAndGetModel` - to check for satisfiability of the asserted constraints and return a model if the constraints are satisfiable, `Push`, `Pop` - to create logical contexts using a stack discipline. The life-time of an asserted formula follows the scoping indicated by `Push`/`Pop`. We use these facilities to

implement theory specific extensions on top of Z3. Our implementation introduces axioms based on a potential partial index set explained in Section 3.3. These axioms are asserted to Z3 together with the input path constraint. Models returned by `CheckAndGetModel` are checked according to the semantics of the set comprehensions. If the current model can be extended to a model satisfying the comprehensions we are done, if not, additional assertions are added to the current scope, and the updated logical context is re-checked. The model-checking loop furthermore ensures that our reduction that retains *write* and *write-set* constructors in the input does not miss checking array indices that are introduced during Z3's search. For example, Z3 internally introduces Skolem constants for array disequalities. These constants should for completeness be counted into the index set in the SAT_A reduction, these indices are extracted lazily during model checking. Figure 2 illustrates a model refinement loop around Z3 (using the .NET managed API calls with F#). The model refinement loop is iterated with additional assertions as long as Z3 returns a satisfying model which does not satisfy the `model_check` (not shown here) test on the set of extracted indices. The function `model_check` uses another API exposed by Z3 to evaluate terms in the context of a model $\mathcal{M}$. To describe the functionality of `model_check` by example, when encountering a subterm of the form $\mathit{Range}(l, u)$ of a formula φ, we call the evaluation function with the two formulas $i \in \mathit{Range}(l, u)$ and $l \leq i \wedge i \leq u$ for every i in the supplied index set. If their evaluations disagree on a given index i (one version evaluates to *true*, the other to *false*), we add the axiom $i \in \mathit{Range}(l, u) \leftrightarrow (l \leq i \wedge i \leq u)$. Note that Z3 supports quantifiers and therefore allows to add axioms such as $\forall i, l, u\{i \in \mathit{Range}(l, u)\}\ i \in \mathit{Range}(l, u) \leftrightarrow (l \leq i \wedge i \leq u)$, where $\{i \in \mathit{Range}(l, u)\}$ is a pattern. However, relying on such axioms is incomplete as they are only expanded if search explicitly builds a subterm that matches the pattern.

In the context of checking model-based programs we have an alternative way of checking and refining models produced by the SMT solver, Z3. We simply run the model program on the trace returned by the SMT solver. If the run deviates from the model by violating comprehensions on certain indices, we may augment the path constraint by the corresponding index constraints.

5 Experiments

As the concrete input language of model programs we use a subset of AsmL [4] that captures the fragment of ASMs described in Section 2. Model programs have the same meaning as in the Spec Explorer tool [25] or in NModel [23]. 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.

```
let check formula =
    z3.Push();
    z3.AssertCnstr formula;
    let indices = get_indices formula in
    let m = ref (null : Model) in
    let rec refine_model() =
        if !m <> null then
            ((!m).Dispose(); m := null);
        if LBool.True = z3.CheckAndGetModel(m) then
            match model_check (!m) indices formula with
                | None -> ()
                | Some violated_comprehension ->
                z3.AssertCnstr violated_comprehension;
                refine_model()
    in
    refine_model();
    z3.Pop();
    if !m <> null then Some (!m) else None
```

Fig. 2. Model refinement loop with Z3.

The *Credits* model program in Figure 1 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 pre-conditions 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 [19]. 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 [19]. 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 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

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

Fig. 3. *Count*(n) model program.

```
var current as Integer
[Action]
Execute(bar as Integer)
    require current ≤ bar
    current := bar
```

Fig. 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 $\texttt{Execute}(a)^*\texttt{Execute}(b)^*\texttt{Execute}(c)^*$.

(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.

6 Related and future work

The unrolling of transition systems into SAT was introduced in [5] and the extension to SMT was introduced in [2] that also compares the SMT approach to other related program verification work. SMT solvers that support arrays are described in [3, 24].

Our formula encoding into SMT [28, 11, 9, 10] 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 [26]. The composition by using actions and identifying an action signature is somewhat different from composition of modules through shared state variables as in SAL 2 [12], although it can be encoded by introducing a special shared action variable. However, in this case special projection functions need to be used in the semantics to eliminate the action, because in a labeled transition system the action label is not part of the state, i.e. the same target state can be reached through distinct actions. Compositional modeling and verification of physical layer protocols involving real time is done in [8] using SAL 2.

Our quantifier elimination scheme builds on [7], but refines it by using model-checking to implement an efficient incremental saturation procedure on top of the SMT solver of our choice. A recent application of the quantifier elimination

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

scheme has been pursued by [21] in the context of railway control systems. Several areas have been left for future work. In particular model-programs use data structures that we are not yet handling with the SMT solver. For instance, a proper encoding of bags (multi-sets) has been left to future work. The class of array model programs is too restrictive for analysis of more general algorithms, see e.g. [17].

References

1. Spec Explorer. http://research.microsoft.com/specexplorer.
2. 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.
3. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.
4. AsmL. http://research.microsoft.com/fse/AsmL/.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
6. 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.
7. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation: 7^{th} International Conference, (VMCAI'06)*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
8. G. M. Brown and L. Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proc. of the 12th Int. Conf. on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *LNCS*, pages 58–72. Springer, 2006.
9. L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *21st International Conference on Automated Deduction, (CADE'07)*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
10. L. de Moura and N. Bjørner. Model-based theory combination. In *5th International Workshop on Satisfiability Modulo Theories, (SMT'07)*, pages 46–57, Berlin, Germany, July 2007.

11. 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.
12. L. M. 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.
13. 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.
14. 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.
15. Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, 1995.
16. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
17. Y. Gurevich, M. Veanes, and C. Wallace. Can abstract state machines be useful in language theory? *Theor. Comput. Sci.*, 376(1):17–29, 2007.
18. 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.
19. 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.
20. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
21. S. Jacobs and V. Sofronie-Stokkermans. Applications of hierarchical reasoning in the verification of complex systems. *Electr. Notes Theor. Comput. Sci.*, 174(8):39–54, 2007.
22. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
23. NModel. http://www.codeplex.com/NModel, released May 2007.
24. 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.
25. 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.
26. M. Veanes, C. Campbell, and W. Schulte. Composition of model programs. In J. Derrick and J. Vain, editors, *FORTE 2007*, volume 4574 of *LNCS*, pages 128–142. Springer, 2007.
27. M. Veanes and W. Schulte. Protocol modeling with model program composition. In *FORTE'08*, LNCS. Springer, 2008. In this volume.
28. Z3. http://research.microsoft.com/projects/z3, released September 2007.