Automatic Verification of Abstract State Machines

(Extended Abstract)

Marc Spielmann

LuFG MGdI, RWTH Aachen, D-52056 Aachen, Germany
spielmann@informatik.rwth-aachen.de

Abstract. Abstract state machines (ASMs) provide the basis of a successful methodology for specification and verification of software and hardware systems. Nevertheless, computer aided verification of ASM-programs has not yet been well-developed. In this paper we try to shed some light on the limits of automatic verifiability of ASM-programs.

We introduce a class of restricted ASM-programs, which are called nullary programs, and provide an algorithm that decides whether a given nullary program satisfies a given correctness property (expressible in a CTL*-like temporal logic) on all inputs. Our decision algorithm runs in PSPACE and we show that this is optimal. We also show that straightforward generalizations of nullary programs cannot be verified algorithmically, as some basic verification problems become undecidable.

1 Introduction

Abstract state machines (ASMs) [Gur95, Gur97], formerly known as *evolving algebras*, provide the formal foundation of a method to design and analyze complex hardware and software systems. When designing such a system one usually starts with a high-level description of the system and, by stepwise refining intermediate stages, eventually obtains a low-level description which is close to executable code. The *ASM-method* proposes to describe each stage of the refinement process in terms of ASM-programs. (That ASM-programs really suffice to express all levels of abstraction of a dynamic system is witnessed by many large-scale applications of the ASM-method [BH98].) The advantage of this approach is that ASM-programs are close to logic (see Theorem 5 in Section 3), which makes them easily accessible for well-understood mathematical methods. Essentially this mathematical foundation of ASM-programs supports the formal verification of systems designed by means of the ASM-method. For an introduction to the ASM-method the reader is referred to [Bör95]. Although there do exist numerous verification examples in the ASM-literature [BH98], one can hardly find an example where all or part of the verification process is mechanized. That is, computer aided verification of ASM-programs has not yet been well-developed. In this paper we investigate the problem of verifying ASM-programs automatically.

In its full generality, *automatic verification of programs* (not necessarily in ASM-syntax) is the following decision problem. Given a program Π and a correctness property φ (expressed in some appropriate specification formalism), decide whether for every input I the computation of Π on I satisfies φ. Obviously, decidability of this problem crucially depends on the expressiveness of the programming language and the specification formalism one has in mind.

Here, we present a class of restricted ASM-programs and a specification formalism resembling the branching-time logic CTL* [CES86, Eme90] for which the above decision problem is decidable, i.e., which can be verified automatically. We call our programs *nullary programs* because the main restriction we impose on ASM-programs is that every dynamic function must have arity 0. (Roughly speaking, a nullary dynamic function v is nothing but a program variable in the usual sense. During a computation step the value of v, i.e., the interpretation of the function symbol v, may change. This corresponds to assigning a new value to the 'program variable' v.) As a possible field of application for nullary programs we suggest the high-level ASM-descriptions that naturally occur when designing a complex dynamic system via the ASM-method. The decision algorithm we provide can then be used to verify such high-level ASM-descriptions.

Aside from possible applications we think that the technique underlying our decision algorithm is also of independent interest, as in some sense our algorithm performs symbolic model checking of software. By software we here mean programs that get a priori unbounded input and whose computations depend on a 'non-trivial' part of the input. Nullary programs are software in this sense. For example, one can write a nullary program Π_{R} that solves the reachability problem for all finite graphs. Given an arbitrary finite graph with two distinguished nodes *source* and *target* as input, Π_{R} decides whether *target* is reachable from *source* (see Example 3 in Section 3). Π_{R} also indicates that nullary programs go beyond the scope of finite state systems. One can hardly imagine a finite state system which 'faithfully' represents all computations of a reachability algorithm on *all* possible input graphs.

To make our verification technique more precise let us reconsider model checking. Can we use model checking for automatic verification of programs (again, not necessarily in ASM-syntax)? That is, is it possible to model-check whether a given program Π satisfies a given correctness property φ for all possible inputs? The answer is yes if there are only finitely many inputs to be checked and the space (or time) complexity of Π is bounded by some function in the size of the input. For instance, repeat the following steps for each input I. First run Π on I and this way obtain the *computation graph of* Π *on* I, i.e., the graph whose nodes are the reachable configurations of Π on I and whose edges represent transitions from one configuration to a successor configuration. This graph is clearly finite and can be viewed as a Kripke structure whose labels are complete descriptions of configurations of Π. Using standard techniques one can (model-) check whether Π satisfies φ on I. Since there are only finitely many inputs we can indeed decide whether Π satisfies φ on every input. From the theoretical point of view there is no principle difference between finite states systems and resource-bounded programs running on a finite number of inputs.

'Real' programs, however, are supposed to be correct for infinitely many inputs, e.g., for all finite graphs. In this case a naive application of model checking fails simply because one cannot construct for all inputs the corresponding computation graphs. The main idea in this paper is to avoid an explicit construction of the computation graphs by translating a given program Π into a logical formula which can be seen as a symbolic representation of *all* computation graphs of Π (independent of a particular input). Combining this formula with the correctness property φ to be checked, one can reduce the problem of (model-) checking whether Π satisfies φ on *all* inputs to the problem of deciding finite validity of a logical formula.

We demonstrate the new technique for nullary programs and correctness properties definable in a specification logic called CGL* – a straightforward adaption of CTL* for reasoning about computation graphs. It turns out that the one-step semantics of a given nullary program Π can be expressed in terms of an existential first-order formula. Employing a translation of CTL* into transitive closure logic (FO+TC) by Immerman and Vardi [IV97], one can combine this existential formula with an arbitrary CGL*-formula φ so that the resulting (FO+TC)-formula is finitely valid iff all computation graphs of Π satisfy φ. The latter means that Π satisfies φ on all inputs. We then observe that finite validity (resp. finite satisfiability) of the obtained (FO+TC)-formula is decidable in PSPACE if Π takes relational input and φ is an existential (resp. universal) CGL*-formula. Hence, in order to decide whether Π satisfies φ on all inputs our algorithm first turns the instance (Π, φ) of the verification problem into a (FO+TC)-formula and then decides finite validity of this formula.

After showing this positive result about nullary programs with relational input we prove that for nullary programs with functions in their input most basic verification problems (like reachability of a safe state and being constantly in safe states) become undecidable. This even holds for very simple nullary programs. Also, the situation does not change when we restrict attention to relational input and instead increase the computational power of nullary programs (e.g., by allowing first-order quantifiers in guards or dynamic functions of arity > 0).

2 Preliminaries

A *vocabulary* is a set Υ of relation and function symbols each associated with an arity. Nullary function symbols are usually referred to as constant symbols. All vocabularies we consider here are finite and contain at least the two constant symbols 0 and 1 (which we usually do not include explicitly). A *Υ-structure* $\mathcal{A}$ consists of a set A, called the *universe* of $\mathcal{A}$, an interpretation $R^{\mathcal{A}} \subseteq A^k$ for each k-ary relation symbol $R \in \Upsilon$, and an interpretation $f^{\mathcal{A}} : A^k \to A$ for each k-ary function symbol $f \in \Upsilon$. We will always assume that $0^{\mathcal{A}} \neq 1^{\mathcal{A}}$. $\mathrm{Fin}(\Upsilon)$ denotes the set of all finite Υ-structures.

A *k-ary query* on $\mathrm{Fin}(\Upsilon)$ is a mapping Q that assigns to every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ a k-ary relation $Q^{\mathcal{A}} \subseteq A^k$ such that the following holds: every isomorphism between $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}, \mathcal{B} \in \mathrm{Fin}(\Upsilon)$, is also an isomorphism between $(A, Q^{\mathcal{A}})$ and $(B, Q^{\mathcal{B}})$. In the special case $k = 0$ we call Q a *boolean query* and view Q as a subset of $\mathrm{Fin}(\Upsilon)$ closed under isomorphism. As an example, recall that every first-order formula $\varphi(x_1, \ldots, x_k)$ over Υ (where all free variables of φ occur among $x_1, \ldots, x_k$) defines a k-ary query on $\mathrm{Fin}(\Upsilon)$ mapping $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ to $\varphi^{\mathcal{A}} := \{(a_1, \ldots, a_k) \in A^k : \mathcal{A} \models \varphi[a_1, \ldots, a_k]\}$.

Transitive closure logic, (FO+TC), is the closure of first-order logic under the transitive closure operator TC. More formally, (FO+TC)(Υ) is the set of all Υ-formulas derivable from the usual formula-formation rules of first-order logic (with equality) and the following rule.

(TC) If φ is a formula, $\bar{x}$ and $\bar{x}'$ are two k-tuples of variables, and $\bar{t}$ and $\bar{t}'$ are two k-tuples of terms, then $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t}, \bar{t}')$ is a formula.

The meaning of $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi](\bar{t}, \bar{t}')$ is as follows. Regard $[\mathrm{TC}_{\bar{x},\bar{x}'}\,\varphi]$ as a new $2k$-ary relation symbol whose interpretation is the transitive, reflexive closure of the image of the $2k$-ary query defined by $\varphi(\bar{x}, \bar{x}')$. If, e.g., $\varphi(x, y) := Exy \vee Eyx$ and $G = (V, E)$ is a directed graph with two distinguished nodes s and t, then $(G, s, t) \models [\mathrm{TC}_{x,y}\,\varphi](s, t)$ iff there is an undirected path in G connecting s and t. For a formal definition of the semantics of (FO+TC) see, e.g., [EF95].

The *existential fragment of* (FO+TC), (E+TC), is the set of all (FO+TC)-formulas without occurrence of a universal quantifier and where all negated subformulas are quantifier-free.

Deciding *finite validity* and *finite satisfiability* of (E+TC)-formulas over a fixed vocabulary will be of particular interest for us. For every vocabulary Υ, FINVAL_Υ(E+TC) (resp. FINSAT_Υ(E+TC)) is the following decision problem. Given a sentence $\varphi \in$(E+TC)(Υ), decide whether $\mathcal{A} \models \varphi$ for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ (resp. for some $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$).

Theorem 1. *Let Υ be a vocabulary that contains relation and constant symbols only. Then both* FINVAL_Υ(E+TC) *and* FINSAT_Υ(E+TC) *are* PSPACE*-complete.*

3 Nullary Programs

In this section we introduce *nullary programs.* Basically, a nullary program is a *nondeterministic basic ASM-program* (in the sense of [Gur97]) where every dynamic function (i.e., a function that can be redefined during a computation) is nullary. We show that nullary programs have the same expressive power as the logic (E+TC). An immediate consequence is that on ordered input structures they compute exactly all NLOGSPACE computable functions.

Let Υ be a finite vocabulary. A *program vocabulary that extends* Υ, denoted Υ_{P}, is obtained from Υ by adding some new constant symbols $v_1, \ldots, v_k$ to Υ. Nullary programs over Υ_{P} (which we will define below) take finite Υ-structures as input; we frequently refer to Υ as the *input vocabulary.* Each v_i will play the role of a program variable. The value, i.e., the interpretation, of v_i may change during a computation step of a nullary program. We call v_i a *dynamic* (abbreviating the official ASM-term "nullary dynamic function symbol").

Definition 2. Let $\Upsilon_{\mathrm{P}} = \Upsilon \,\dot{\cup}\, \{v_1, \ldots, v_k\}$ be a program vocabulary. *Nullary programs over* Υ_{P} are defined inductively:

1. **Update:** For every dynamic v_i and every Υ_{P}-term t the assignment $v_i := t$ is a nullary program.
2. **Conditional:** If φ is a quantifier-free Υ_{P}-formula and Π a nullary program, then (`if` φ `then` Π) is a nullary program (with *guard* φ).
3. **Parallel execution:** If Π_0 and Π_1 are nullary programs, then $\Pi_0||\Pi_1$ is a nullary program. (For readability, $\Pi_0||\Pi_1$ is sometimes written as $\genfrac{}{}{0pt}{}{\Pi_0}{\Pi_1}$).
4. **Choice:** Let $\bar{z}$ be a tuple of variables, φ a quantifier-free Υ_{P}-formula, and Π a nullary program. If $\exists\bar{z}\varphi$ is finitely valid, i.e., if it holds in all finite Υ_{P}-structures for all interpretations of the free variables of $\exists\bar{z}\varphi$, then (`choose` $\bar{z}$: φ Π) is a nullary program (with *guard* φ).

(Intuitively, the semantics of (`choose` $\bar{z} : \varphi$ Π) is as follows. Choose nondeterministically values for the variables in $\bar{z}$ so that the guard φ is satisfied. Finite validity of $\exists\bar{z}\varphi$ guarantees the existence of such values. The actual program to be executed is then obtained from Π by replacing every occurrence of z_i in Π with the value chosen for z_i. Note that in many cases $\varphi := \mathit{true}$ suffices as guard. However, if for your favorite guard φ, $\exists\bar{z}\varphi$ is not finitely valid, you can often replace φ with $\varphi' := \varphi \vee \bar{z} = \bar{0}$ and sort out 'invalid' choices of $\bar{0}$ inside Π.)

A nullary program is *deterministic* if it is derivable from the above rules without using the choice rule. □

The free and bound variables of a nullary program are defined in the obvious way. For instance, in the nullary program (`choose` $\bar{z}$: φ Π) each variable in $\bar{z}$ occurs bounded. We can restrict attention to nullary programs without free variables if we substitute every free variable by a new constant symbol. For simplicity we will do so from now on.

The semantics of ASM-programs is usually given by means of *update sets* [Gur97, Gur95]. We define the semantics of nullary programs in a different way, which will be more convenient for our purposes. Nevertheless, our semantics coincide with the standard semantics.

Semantics of Nullary Programs. Consider a nullary program Π over Υ_{P}, $\Upsilon_{\mathrm{P}} = \Upsilon \mathbin{\dot\cup} \{v_1, \dots, v_k\}$. Π takes finite Υ-structures as *input.* A *state* of Π on an input $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ is a finite Υ_{P}-structure $(\mathcal{A}, a_1, \dots, a_k)$, where $a_1, \dots, a_k$ are the interpretations (or values) of the dynamics $v_1, \dots, v_k$, respectively. The *initial state* of Π on $\mathcal{A}$ is $(\mathcal{A}, 0, \dots, 0)$. As with a Turing machine program, the 'program text' Π is viewed as a description of how to modify the current state of Π in order to obtain a possible successor state. Formally, Π induces a $2k$-ary relation $\Pi^{\mathcal{A}} \subseteq A^k \times A^k$ so that $(\bar{a}, \bar{a}') \in \Pi^{\mathcal{A}}$ means that if Π is currently in state $(\mathcal{A}, \bar{a})$ then in the next step it may change to state $(\mathcal{A}, \bar{a}')$.

To the definition of $\Pi^{\mathcal{A}}$. By induction on the construction of Π, we define an existential Υ-formula $\varphi_\Pi(\bar{x}, \bar{x}')$, all of whose free variables occur among $\bar{x} = x_1, \dots, x_k$ and $\bar{x}' = x_1', \dots, x_k'$. For a better understanding assume that the interpretation of x_i (resp. x_i') equals the current value of v_i (resp. the value of v_i in a successor state). We write $[\bar{v}/\bar{x}]$ to denote the substitution of every occurrence of v_i by x_i.

If $\Pi = v_i := t$ let $\varphi_\Pi := x'_i = t[\bar{v}/\bar{x}]$.
If $\Pi =$ `if` φ `then` Π_0 let $\varphi_\Pi := \varphi[\bar{v}/\bar{x}] \to \varphi_{\Pi_0}$.
If $\Pi = \Pi_0 || \Pi_1$ let $\varphi_\Pi := \varphi_{\Pi_0} \wedge \varphi_{\Pi_1}$.
If $\Pi =$ `choose` $\bar{z} : \varphi\ \Pi_0$ let $\varphi_\Pi := \exists \bar{z}(\varphi[\bar{v}/\bar{x}] \wedge \varphi_{\Pi_0})$.

Consider two states $(\mathcal{A}, \bar{a})$ and $(\mathcal{A}, \bar{a}')$ of Π. Intuitively, $\mathcal{A} \models \varphi_\Pi[\bar{a}, \bar{a}']$ means that, if Π is currently in state $(\mathcal{A}, \bar{a})$ and $(v_i := t)$ is an update in Π (possibly occurring in the scope of guards all of which are satisfied in $(\mathcal{A}, \bar{a})$), then $a'_i = t^{(\mathcal{A},\bar{a})}$ is the new value of v_i in successor state $(\mathcal{A}, \bar{a}')$. φ_Π describes all those updates which *must* be performed in the next step.

φ_Π is not yet the desired definition of $\Pi^{\mathcal{A}}$. This is because φ_Π does not say that dynamics not effected by any update *must not* change – which is the intended meaning of Π. (Suppose, e.g., that v_i does not occur in Π. Then $\mathcal{A} \models \varphi_\Pi[\bar{a}, \bar{a}']$ may hold even when $a_i \neq a'_i$.) We fix this as follows. For every $\Gamma \subseteq \{x'_1 = x_1, \ldots, x'_k = x_k\}$ let $\varphi_{\Pi,\Gamma}(\bar{x}, \bar{x}') := \varphi_\Pi \wedge \bigwedge \Gamma$ (where, by convention, $\bigwedge \varnothing \equiv$ *true*). Call Γ *maximal w.r.t. to state* $(\mathcal{A}, \bar{a})$ if $\mathcal{A} \models \exists \bar{x}' \varphi_{\Pi,\Gamma}[\bar{a}, \bar{x}']$ and there is no Γ^*, $\Gamma \subsetneq \Gamma^*$, such that $\mathcal{A} \models \exists \bar{x}' \varphi_{\Pi,\Gamma^*}[\bar{a}, \bar{x}']$. Finally, let $(\bar{a}, \bar{a}') \in \Pi^{\mathcal{A}}$ iff either

- there exists a Γ maximal w.r.t. $(\mathcal{A}, \bar{a})$ such that $\mathcal{A} \models \varphi_{\Pi,\Gamma}[\bar{a}, \bar{a}']$, or
- $\bar{a} = \bar{a}'$ and $\mathcal{A} \not\models \exists \bar{x}' \varphi_\Pi[\bar{a}, \bar{x}']$.

In the latter case we say that Π is *inconsistent* in state $(\mathcal{A}, \bar{a})$. If $(\bar{a}, \bar{a}') \in \Pi^{\mathcal{A}}$ then $(\mathcal{A}, \bar{a}')$ is called a *successor state* of $(\mathcal{A}, \bar{a})$. Notice that every state has at least one successor state. If Π is deterministic then every state has a unique successor state.

A *run of* Π *on* $\mathcal{A}$ is an infinite sequence of states such that the first state in the sequence is the initial state of Π on $\mathcal{A}$ and the $(i+1)^{\text{th}}$ state is a successor of the i^{th} state. Every run of Π on $\mathcal{A}$ can be embedded in the *computation graph of* Π *on* $\mathcal{A}$, denoted $C_\Pi(\mathcal{A})$, which is the finite graph (S, R, s_0) consisting of

- state set $S := \{(\mathcal{A}, \bar{a}) : \bar{a} \in A^k\}$,
- reachability relation $R := \{((\mathcal{A}, \bar{a}), (\mathcal{A}, \bar{a}')) : (\bar{a}, \bar{a}') \in \Pi^{\mathcal{A}}\}$, and
- initial state $s_0 := (\mathcal{A}, \bar{0})$.

Assume that Υ_{P} contains the distinguished dynamic *accept*. We say that Π *accepts* $\mathcal{A}$ if in $C_\Pi(\mathcal{A})$ there exists a path from s_0 to a state where the value of the dynamic *accept* is 1. Π *computes a boolean query* $Q \subseteq \mathrm{Fin}(\Upsilon)$ if for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$, Π accepts $\mathcal{A}$ iff $\mathcal{A} \in Q$.

Example 3. Consider the following decision problem known as REACHABILITY. Given a finite directed graph $G = (V, E)$ and two nodes s and t in G, decide whether there exists a path from source s to target t in G. REACHABILITY can be seen as a boolean query on finite structures of the from (G, s, t). We present a nullary program Π_{R} that computes this boolean query.

The input vocabulary of Π_{R} is $\Upsilon := \{E, s, t\}$, where E denotes the binary edge relation of the input graph and s and t the source and the target, respectively. (Recall that by our general assumption we also have $0, 1 \in \Upsilon$.)

Π_{R} as defined below is a nullary program over the program vocabulary $\Upsilon_{\mathrm{P}} := \Upsilon \,\dot\cup\, \{mode, pebble, accept\}$; it employs the three dynamics *mode*, *pebble*, and *accept*. (For readability we use a slightly relaxed syntax and omit parentheses.)

```
Π_R := if mode = 0 then
          pebble := s
          mode := 1

       if mode = 1 then
          if pebble ≠ t then
             choose z : true
                if E(pebble, z) then pebble := z
          else
             accept := 1
```

On an input $\mathcal{A} = (G, s, t)$, the states of Π_{R} are Υ_{P}-structures of the form $(\mathcal{A}, a_m, a_p, a_a)$, where a_m, a_p, and a_a are the values of *mode*, *pebble*, and *accept*, respectively. Initially, Π_{R} is in state $(\mathcal{A}, 0, 0, 0)$. In the first step, Π_{R} moves to state $(\mathcal{A}, 1, s, 0)$. Then, as long as the value of *pebble* does not equal t, Π_{R} chooses a node a in G, checks whether $(pebble, a)$ is an edge in G, and updates *pebble* with a if so; otherwise it performs no update. If *pebble* is ever updated with t, Π_{R} accepts by updating *accept* with 1. In this case Π_{R} becomes idle; it repeats the accepting state infinitely often. □

Lemma 4. *For every nullary program Π over Υ_{P}, $\Upsilon_{\mathrm{P}} = \Upsilon \,\dot\cup\, \{v_1, \ldots, v_k\}$, there is an existential first-order formula $\chi_\Pi(\bar{x}, \bar{x}')$ over Υ with $2k$ free variables $\bar{x}, \bar{x}'$ such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and all $\bar{a}, \bar{a}' \in A^k$, $\mathcal{A} \models \chi_\Pi[\bar{a}, \bar{a}']$ iff $(\bar{a}, \bar{a}') \in \Pi^{\mathcal{A}}$. χ_Π can be obtained from Π in time polynomial in the size of Π.*

One can view χ_Π in the previous lemma as a symbolic representation of the reachability relations of all possible computation graphs of Π (independent of a specific input). In fact, for every input $\mathcal{A}$, there exists a path from s_0 to $(\mathcal{A}, \bar{a})$ in $C_\Pi(\mathcal{A})$ iff $\mathcal{A} \models [\mathrm{TC}_{\bar{x},\bar{x}'}\, \chi_\Pi(\bar{x}, \bar{x}')][\bar{0}, \bar{a}]$.

Theorem 5. *A boolean query Q is computable by a nullary program iff Q is definable in the logic* (E+TC)*.*

Proof. (Sketch.) Suppose Π computes Q. Let $\chi_\Pi(\bar{x}, \bar{x}')$ be obtained from Π according to Lemma 4, where x_i (resp. x'_i) represent the value of *accept*. Then $\exists \bar{x}'([\mathrm{TC}_{\bar{x},\bar{x}'}\, \chi_\Pi(\bar{x}, \bar{x}')](\bar{0}, \bar{x}') \wedge x'_i = 1)$ defines Q. For the other direction assume that the sentence $\varphi \in (\mathrm{E{+}TC})(\Upsilon)$ defines Q. There exists a quantifier-free formula $\psi(\bar{x}, \bar{x}')$ such that φ is equivalent to $[\mathrm{TC}_{\bar{x},\bar{x}'}\, \psi(\bar{x}, \bar{x}')](\bar{0}, \bar{1})$ (see, e.g., [GM96]). Redefine Π_{R} in Example 3 by replacing *pebble*, z, s, t, and $E(pebble, z)$ with $\bar{p}$, $\bar{z}$, $\bar{0}$, $\bar{1}$, and $\psi(\bar{p}, \bar{z})$, respectively, where $\bar{p}$ is now a sequence of dynamics. The obtained program is a nullary program over $\Upsilon \cup \{mode, \bar{p}, accept\}$ and computes Q. □

Immerman [Imm87] showed that on ordered structures a boolean query Q is NLogspace computable iff Q is definable in (E+TC). This gives us the first part of the next corollary. The second part follows from a result in [GS99].

Corollary 6. *Let Q be a boolean query on ordered structures. (1) Q is computable by a nullary program iff Q is* NLOGSPACE *computable. (2) Q is computable by a deterministic nullary program iff Q is* LOGSPACE *computable.*

4 Verifying Nullary Programs

Verification of nullary programs only makes sense in the context of a specification formalism suitable to express correctness properties of nullary programs. Since all runs of a nullary program Π on an input $\mathcal{A}$ are embedded in $C_\Pi(\mathcal{A})$ it is reasonable to express correctness properties of nullary programs as properties of their computation graphs. Below we present a straightforward adaption of the branching-time logic CTL* [CES86, Eme90] to the computation graph setting. The new logic is called CGL* (*computation graph logic 'star'*), alluding to CTL*.

Definition 7. Let Υ_{P} be a program vocabulary. *State formulas over* Υ_{P} and *path formulas over* Υ_{P} are defined by simultaneous induction:

(S1) Every sentence in $(\mathrm{E{+}TC})(\Upsilon_{\mathrm{P}})$ is a state formula.
(S2) If α is a path formula, then $\mathbf{E}\alpha$ is a state formula.
(P1) Every state formula is also a path formula.
(P2) If α and β are path formulas, then so are $\alpha \vee \beta$, $\alpha \wedge \beta$, and $\neg\alpha$.
(P3) If α and β are path formulas, then so are $\mathbf{X}\alpha, \alpha\mathbf{U}\beta$, and $\alpha\mathbf{B}\beta$.

An *existential state formula* is a state formula which can derived from the above rules without using in rule **(P2)** the clause to form negated formulas.

$\mathrm{CGL}^*(\Upsilon_{\mathrm{P}})$ (resp. $\mathrm{ECGL}^*(\Upsilon_{\mathrm{P}})$) is the set of all state formulas (resp. existential state formula) over Υ_{P}. □

The intuitive meaning of the existential path quantifier $\mathbf{E}$ and the temporal operators $\mathbf{X}$ and $\mathbf{U}$ is as in CTL*. $\alpha\mathbf{B}\beta$ stands for "α holds *before* β fails" [IV97]. A formal definition of the semantics of CGL* follows. Let $C = (S, R, s_0)$ be the computation graph of some nullary program over Υ_{P}. A *run in* C is a mapping ρ from the natural numbers to S such that $(\rho(i), \rho(i+1)) \in R$ for all i. Let $\rho|i$ denote the run ρ' defined by $\rho'(j) := \rho(i+j)$. Consider a state formula φ and a path formula α, both over Υ_{P}. Similar to CTL* one defines $(C, \mathcal{A}) \models \varphi$ for every state $\mathcal{A} \in S$ and $(C, \rho) \models \alpha$ for every run ρ in C by simultaneous induction on the construction of φ and α. The only new cases are

$$\begin{array}{llll} \textbf{(S1)} & (C,\mathcal{A}) \models \varphi & :\Leftrightarrow & \mathcal{A} \models \varphi \\ \textbf{(P3)} & (C,\rho) \models \alpha\mathbf{B}\beta & :\Leftrightarrow & \forall i((C,\rho|i) \models \neg\beta \Rightarrow \exists j(j < i \wedge (C,\rho|j) \models \alpha)). \end{array}$$

For every $\varphi \in \mathrm{CGL}^*(\Upsilon_{\mathrm{P}})$ let $C \models \varphi$ iff $(C, s_0) \models \varphi$.

To give an example of a meaningful CGL*-formula, let us express correctness of the nullary program Π_{R} in Example 3 in terms of CGL*. More precisely, we will display a state formula φ_{R} over the program vocabulary of Π_{R}, such that Π_{R} is correct (i.e., Π_{R} computes the boolean query REACHABILITY) iff $C_{\Pi_{\mathrm{R}}}(\mathcal{A}) \models \varphi_{\mathrm{R}}$ for every input $\mathcal{A}$. The following definition of φ_{R} is justified by two observations: (1) Π_{R} is correct iff for every input $\mathcal{A}$, $\mathcal{A} \in$ REACHABILITY iff $C_{\Pi_{\mathrm{R}}}(\mathcal{A}) \models \mathbf{EF}(\mathit{accept} = 1)$ (where $\mathbf{F}\beta := \mathit{true}\mathbf{U}\beta$). (2) $\mathcal{A} \in$ REACHABILITY iff $\mathcal{A} \models [\mathrm{TC}_{x,x'}\, E(x,x')](s,t)$ iff $C_{\Pi_{\mathrm{R}}}(\mathcal{A}) \models \mathbf{E}([\mathrm{TC}_{x,x'}\, E(x,x')](s,t))$.

$$\varphi_{\mathrm{R}} := \mathbf{E}([\mathrm{TC}_{x,x'}\, E(x,x')](s,t)) \leftrightarrow \mathbf{EF}(\mathit{accept} = 1).$$

Hence, one can prove correctness of Π_{R} by verifying $C_{\Pi_{\mathrm{R}}}(\mathcal{A}) \models \varphi_{\mathrm{R}}$ for every input $\mathcal{A}$.

Verifying Nullary Programs w.r.t. CGL*-Properties. Let L be a sublogic of CGL*. *Verifying nullary programs w.r.t.* L means solving the decision problem:

VERIFY(L): Given a nullary program Π and a state formula $\varphi \in L$, both over the same program vocabulary Υ_{P} (that extends some input vocabulary Υ), does $C_{\Pi}(\mathcal{A}) \models \varphi$ hold for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$?

Let $\mathrm{VERIFY}_{\Upsilon}(L)$ denote the corresponding problem where the input vocabulary Υ is a priori fixed (the program vocabulary Υ_{P}, however, may still vary).

The complexity of the latter problem is more significant for applications than that of VERIFY(L). For instance, assume that in order to solve a computational problem a nullary program Π was put forward which happens not to satisfy some correctness property $\varphi \in L$. In that case, one usually has to rewrite Π (and possibly modify some correctness properties), rather than changing the computational problem itself (and thus the input vocabulary Υ).

Notice that deciding $\mathrm{VERIFY}_{\Upsilon}(\mathrm{CGL}^*)$ subsumes symbolic model checking of CTL*-properties. Every Kripke structure $\mathcal{K}$ (given symbolically in terms of boolean formulas) and every CTL*-formula p (appropriate for $\mathcal{K}$) can easily be turned into a nullary program $\Pi_{\mathcal{K}}$ and a CGL*-formula φ_p such that $\mathcal{K} \models p$ iff $(\Pi_{\mathcal{K}}, \varphi_p) \in \mathrm{VERIFY}_{\{0,1\}}(\mathrm{CGL}^*)$.

Recall that ECGL* denotes the existential fragment of CGL* and let ACGL* be the set of all negated ECGL*-formulas. Our main positive result is:

Theorem 8. *Let Υ be a vocabulary that contains relation and constant symbols only. Then both* $\mathrm{VERIFY}_{\Upsilon}(\mathrm{ECGL}^*)$ *and* $\mathrm{VERIFY}_{\Upsilon}(\mathrm{ACGL}^*)$ *are* PSPACE*-complete. In other words, given a nullary program Π and a correctness property $\varphi \in$* ECGL**, both over the same program vocabulary that extends the fixed Υ, deciding whether Π satisfies φ (or $\neg\varphi$) for all inputs is a* PSPACE*-complete problem.*

The restriction to relational input vocabularies in the theorem is essential. In the next section we will see that neither of the two verification problems is decidable if the input vocabulary contains a unary function symbol.

Proof. (Sketch.) PSPACE-hardness of both problems is shown via a reduction from the satisfiability problem for quantified boolean formulas. To prove containment we reduce $\mathrm{VERIFY}_{\Upsilon}(\mathrm{ECGL}^*)$ to $\mathrm{FINVAL}_{\Upsilon}(\mathrm{E{+}TC})$ and $\mathrm{VERIFY}_{\Upsilon}(\mathrm{ACGL}^*)$ to $\mathrm{FINSAT}_{\Upsilon}(\mathrm{E{+}TC})$. The assertion is then implied by Theorem 1. Most of the reduction work has already been done by Immerman and Vardi ([IV97], Theorem 9) who defined a translation of CTL* into (FO+TC). If we replace in this translation $R(y, y')$ (the reachability relation of a given Kripke structure) with $\chi_\Pi(\bar{y}, \bar{y}')$ (the 'reachability relation' induced by Π according to Lemma 4) and replace every variable y (representing a state of the Kripke structure) with a tuple $\bar{y}$ of variables (representing the dynamic part of a state of Π), then we immediately obtain:

Fact 9 ([IV97]). *For every nullary program Π and every $\varphi \in$ ECGL*, both over same program vocabulary Υ_P, $\Upsilon_\mathrm{P} = \Upsilon \,\dot\cup\, \{v_1, \ldots, v_k\}$, there exists a formula $\chi_{\Pi,\varphi}(\bar{y}) \in$ (E+TC)(Υ) such that for every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ and all $\bar{a} \in A^k$, $\mathcal{A} \models \chi_{\Pi,\varphi}[\bar{a}]$ iff $(C_\Pi(\mathcal{A}), (\mathcal{A}, \bar{a})) \models \varphi$.*

It follows that $(\Pi, \varphi) \in$ VERIFY$_\Upsilon$(ECGL*) iff $\chi_{\Pi,\varphi}(\bar{0}) \in$ FINVAL$_\Upsilon$(E+TC) and that $(\Pi, \varphi) \notin$ VERIFY$_\Upsilon$(ACGL*) iff $\chi_{\Pi,\neg\varphi}(\bar{0}) \in$ FINSAT$_\Upsilon$(E+TC). One can modify the translation by Immerman and Vardi (by introducing new variables) so that it becomes polynomial-time computable. □

The space complexity of VERIFY$_\Upsilon$(ECGL*) and VERIFY$_\Upsilon$(ACGL*) grows exponentially in the sum of the arities of relation symbols in Υ. In particular, VERIFY(ECGL*) and VERIFY(ACGL*) are in EXPSPACE for (non-fixed) relational input vocabularies with constants. As already pointed out, this complexity bound is more of theoretical interest since for most applications the number of input relations as well as their arities will be fixed.

Although ECGL* and ACGL* are only small fragments of CGL*, they still suffice to express many useful correctness properties. For example, for every linear-time formula α (i.e, a path-formula without path-quantifiers) we have $\mathbf{E}\alpha \in$ ECGL* and $\mathbf{A}\alpha \in$ ACGL*. Especially common fairness properties like "impartiality", "weak fairness", and "strong fairness" can be expressed in these fragments (see, e.g., [EL87] and references there). Observe though that the formula φ_R expressing correctness of Π_R in Example 3 is neither in ECGL* nor in ACGL*. Nevertheless, there are formulas definable in ACGL* which imply partial correctness of Π_R.

5 On Input with Functions

A minimal requirement on any automatic verifier for nullary programs is that, when given a nullary program Π, it should be able to decide whether Π *reaches only 'safe' states* on every input, or, equally desirable, whether Π *can reach a 'safe' state* on every input. Here, safety for a state could mean that a designated dynamic in Π does or does not assume a particular value. This motivates the definition of two simple verification problems which any automatic verifier for nullary programs should be able to solve:

ALWAYS SAFE: Given a nullary program Π and a dynamic v in Π, does $C_\Pi(\mathcal{A}) \models \mathbf{AG}(v = 0)$ hold for every input $\mathcal{A}$?

SOMETIMES SAFE: Given a nullary program Π and a dynamic v in Π, does $C_\Pi(\mathcal{A}) \models \mathbf{EF}(v \neq 0)$ hold for every input $\mathcal{A}$?

The next theorem states our main negative result. We call a dynamic v in a nullary program Π *boolean* if every update of v in Π has either the form $v := 0$ or $v := 1$.

Theorem 10. *For nullary programs whose input vocabulary contains two non-nullary symbols, one of which is a function symbol,* ALWAYS SAFE *and* SOMETIMES SAFE *are undecidable.* ALWAYS SAFE *is already undecidable for deterministic such programs with two non-boolean dynamics.*

Proof. (Sketch.) Consider a sentence $\varphi \in (\mathrm{E+TC})(\Upsilon)$ and let Q_φ denote the boolean query defined by φ. By Theorem 5 there exists a nullary program Π_φ computing Q_φ. Obviously, φ is finitely valid iff $Q_\varphi = \mathrm{Fin}(\Upsilon)$ iff Π_φ accepts every $\mathcal{A} \in \mathrm{Fin}(\Upsilon)$ iff $(\Pi_\varphi, \mathit{accept}) \in$ SOMETIMES SAFE. This establishes a reduction of $\text{FINVAL}_\Upsilon(\mathrm{E+TC})$ to SOMETIMES SAFE. A similar argument reduces $\text{FINSAT}_\Upsilon(\mathrm{E+TC})$ to ALWAYS SAFE. The first assertion is now implied by:

Lemma 11. *If Υ contains two non-nullary symbols, one of which is a function symbol, then both* $\text{FINSAT}_\Upsilon(\mathrm{E+TC})$ *and* $\text{FINVAL}_\Upsilon(\mathrm{E+TC})$ *are undecidable.*

The proof of Lemma 11 is by reduction of two undecidable problems for deterministic finite automata with two input heads (namely the emptiness problem and its dual – the totality problem) to $\text{FINSAT}_\Upsilon(\mathrm{E+TC})$ and $\text{FINVAL}_\Upsilon(\mathrm{E+TC})$, respectively. A straightforward adaption of the first reduction yields the second assertion of the theorem. □

Theorem 10 essentially says that nullary programs which assume (arbitrarily defined) functions in their input cannot be verified algorithmically. But what if we stick to relational input and increase the computational power of nullary programs? Following the general ASM-framework we may allow first-order quantifiers in guards or dynamic functions of arity > 0. (A unary dynamic function f, e.g., can occur in an update of the form $f(t) := s$, meaning that in the next state the value of f at argument t will be updated to s.) The proof of the next corollary is similar to that of the second assertion of Theorem 10.

Corollary 12. *If the definition of nullary programs is relaxed in one of the following two ways and the input vocabulary contains a relation symbol of arity ≥ 2, then* ALWAYS SAFE *is undecidable. (1) Allow a single first-order quantifier to occur in one guard. (2) Allow the usage of one unary dynamic function.*

6 Conclusions and Future Work

We have introduced nullary programs – a class of restricted abstract state machine programs – and investigated the problem of verifying them automatically. On the one hand, automatic verification of nullary programs with relational input (against CTL*-like correctness properties) is PSPACE-complete. On the other hand, most basic verification problems become undecidable when we admit arbitrarily defined functions in the input or increase the computational power of nullary programs in a straightforward manner. Altogether this might suggest that with nullary programs we are approaching the limit of automatic verifiability of ASM-programs.

There are several directions for future work. (1) The decision procedures underlying Theorem 1 form the core of our verification algorithm. Both procedures perform a semi-naive exhaustive search and hence are not efficient. The question is whether they can be improved so that we obtain a reasonable performance in realistic settings. (2) Identify other fragments L of CGL* for which VERIFY(L) is decidable. To this end investigate finite validity and finite satisfiability of formulas obtained by Fact 9 when φ varies in L. (3) Extend CGL* with counting constructs. Notice that properties like "φ holds in all even moments" are expressible in (E+TC).

Acknowledgements. I am grateful to Erich Grädel for bringing the subject of model checking ASMs to my attention and to Eric Rosen for many fruitful discussions and valuable suggestions.

References

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of the EATCS*, 64:105–127, February 1998.

[Bör95] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In *Proceedings of SOFSEM '95*, volume 1012 of *LNCS*, pages 236–271. Springer Verlag, 1995.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Trans. on Prog. Lang. and Sys.*, 8(2):244–263, April 1986.

[EF95] H. D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.

[EL87] E.A. Emerson and C.L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–11072. Elsevier Science Publishers B.V., 1990.

[GM96] E. Grädel and G. McColm. Hierarchies in Transitive Closure Logic, Stratified Datalog and Infinitary Logic. *Annals of Pure and Applied Logic*, 77:166–199, 1996.

[GS99] E. Grädel and M. Spielmann. Logspace Reducibility via Abstract State Machines. Submitted for publication, 1999.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan, May 1997.

[Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[IV97] N. Immerman and M.Y. Vardi. Model Checking and Transitive Closure Logic. In *Proceedings of CAV '97*, volume 1254 of *LNCS*, pages 291–302. Springer-Verlag, 1997.