Theoretical Computer Science 560 (2014) 364–379

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Program verification using symbolic game semantics

Aleksandar S. Dimovski

Faculty of Information–Communication Tech., FON University, Skopje, 1000, The Former Yugoslav Republic of Macedonia

ARTICLE INFO

Available online 18 January 2014

Keywords:
Algorithmic game semantics
Symbolic automata
Program verification
Predicate abstraction

ABSTRACT

We introduce a new symbolic representation of algorithmic game semantics, and show how it can be applied for efficient verification of open (incomplete) programs. The focus is on an Algol-like programming language which contains the core ingredients of imperative and functional languages, especially on its second-order recursion-free fragment with infinite data types. We revisit the regular-language representation of game semantics of this language fragment. By using symbolic values instead of concrete ones, we generalize the standard notions of regular-language and automata representations of game semantics to those of corresponding symbolic representations. In this way programs with infinite data types, such as integers, can be expressed as finite-state symbolic-automata although the standard automata representation is infinite-state, i.e. the standard regular-language representation has infinite summations. Moreover, in this way significant reductions of the state space of game semantics models are obtained. This enables efficient verification of programs by our prototype tool based on symbolic game models, which is illustrated with several examples.

1. Introduction

Game semantics [1,2,21] is a technique for compositional modelling of programming languages, which gives fully abstract models. This means that the generated models are both sound and complete with respect to observational equivalence of programs. In game semantics, types are interpreted by *games* (or arenas) between a Player, which represents the term being modelled, and an Opponent, which represents the environment in which the term is used. The two participants strictly alternate to make moves, each of which is either a question (a demand for information) or an answer (a supply of information). Computations (executions of terms) are interpreted as *plays* of a game, while terms are expressed as *strategies*, i.e. sets of plays, for a game. It has been shown that game semantics model can be given certain kinds of concrete automata-theoretic representations [11,16,18], and so it can serve as a basis for software model checking and program analysis. Several features of game semantics make it very promising for software model checking. The model is very precise and compositional, i.e. generated inductively on the structure of programs, which is the key feature for achieving scalability. Also there exists a model for any term-in-context (program fragment) with undefined identifiers, such as calls to library functions. However, the main limitation of the model checking technique in general is that it can be applied only if a finite-state model is available. In our case, this problem with infinite-state models arises when we want to handle terms with infinite data types.

Regular-language representation of game semantics of second-order recursion free Idealized Algol with finite data types provides algorithms for automatic verification of a range of properties, such as observational-equivalence, approximation, and safety. It has the disadvantage that in the presence of infinite integer data types the obtained automata become infinite state, i.e. regular-languages have infinite summations, thus losing their algorithmic properties. Similarly, large finite data types are likely to make the state-space of the obtained automata so big that it will be practically infeasible for automatic

E-mail address: aleksandar.dimovski@fon.edu.mk.

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2014.01.016

verification. For example, let us consider how we can model the successor function of type $\mathbb{N} \to \mathbb{N}$. One characteristic play in the strategy for this function looks like this:

$$\begin{array}{lrrr} \mathsf{succ} : \mathbb{N}^{\langle 1\rangle} & \Rightarrow & \mathbb{N}^{\langle 2\rangle} & \\ & & q & O \\ & q & & P \\ & n & & O \\ & & n+1 & P \end{array}$$

The play starts by Opponent (O) asking for the value of output with the question move q, and Player (P) responds by asking for input. When Opponent provides an input n which can be any value from $\mathbb{N}$, Player supplies $n+1$ as output. The model of the successor function consists of all possible plays of the above form. So, it is given by the following regular language: $\sum_{n\in\mathbb{N}}(q^{\langle 2\rangle} \cdot q^{\langle 1\rangle} \cdot n^{\langle 1\rangle} \cdot (n+1)^{\langle 2\rangle})$, which has infinite summation when $\mathbb{N}$ is an infinite data type. Note that moves are tagged with superscripts $\langle 1\rangle$ and $\langle 2\rangle$ to distinguish from which type component, input or output, they originate from.

In this paper we redefine the (standard) regular-language representation [16] at a more abstract level so that terms with infinite data types can be represented as finite automata, and so various program properties can be checked over them. The idea is to transfer attention from the standard form of automata to what we call symbolic automata. The representation of values constitutes the main difference between these two formalisms. In symbolic automata, instead of assigning concrete values to identifiers occurring in terms, they are left as symbols. Operations involving such identifiers will also be left as symbols. Some of the symbols will be guarded by boolean expressions, which indicate under which conditions these symbols can be performed. Also some of the words accepted by symbolic automata will be guarded by boolean expressions, called conditions, which indicate whether a word is feasible or not. Infeasible words have inconsistent (unsatisfiable) conditions, and we will use SMT solvers, such as Yices [15], to check consistency of these conditions. For example, symbolic representation of the successor function will be given by the following word: $q^{\langle 2\rangle} \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot (Z+1)^{\langle 2\rangle}$, where a new symbol Z is used to encode the value of the input argument. This word is unguarded, i.e. its condition is '*true*', and so it is feasible.

This paper represents an extended and revised version of [13]. It is structured as follows. The language we consider here is introduced in Section 2. Symbolic representation of algorithmic game semantics is defined in Section 3. Correctness of the symbolic representation and its suitability for verification of safety properties are shown in Section 4. In Section 5 we discuss some extensions of the language, such as arrays, and show how they can be represented in the symbolic model. A prototype tool, which implements this translation, as well as some examples are described in Section 6. In Section 7, we conclude and present some ideas for future work.

1.1. Related work

By representing game semantic models as symbolic automata, we obtain a predicate abstraction [19,8] alike method for verification. In [3] it was also developed a predicate abstraction from game semantics. This was enabled by extending the models produced using game semantics such that the state (store) is recorded explicitly in the model by using so-called stateful plays. The state is then abstracted by a set of predicates giving rise to pa (predicate abstraction)-plays. However, in our work we achieved predicate abstraction in a more natural way without changing the game semantic models, and also for terms with infinite data types.

Symbolic techniques, in which data is not represented explicitly but symbolically, have found a number of applications in theoretical computer science. Some interesting examples are symbolic execution and verification of programs [6], symbolic program analysis [7,5], symbolic operational semantics of process algebras [20], parameterized verification of data independent systems [23,24], etc.

SMT (Satisfiability Modulo Theories) [4] is concerned with checking the satisfiability of formulas with respect to some background (first-order) theories, which fix interpretations of certain predicates and functions. In recent years, many powerful SMT solvers have been developed in academia and industry. They have been successfully applied in many modern program analysis and program verifications systems. For example, SMT solvers are used by interactive theorem provers, such as Isabelle and PVS, software model checkers, such as SLAM and BLAST, static verifiers, such as Boogie and ESC/Java 2, etc.

2. The language

Idealized Algol (IA) [27] is a well studied language which combines call-by-name λ-calculus with the fundamental imperative features and locally-scoped variables. In this paper we work with its second-order recursion-free fragment (IA_2 for short).

The data types D are integers and booleans ($D ::= \mathsf{int} \mid \mathsf{bool}$). The base types B are expressions, commands, and variables ($B ::= \mathsf{exp}\, D \mid \mathsf{com} \mid \mathsf{var}\, D$). We consider only first-order function types T ($T ::= B \mid B \to T$).