Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Verifying annotated program families using symbolic game semantics

Aleksandar S. Dimovski

IT University of Copenhagen, Copenhagen, Denmark

ARTICLE INFO

Article history:
Received 30 November 2016
Received in revised form 15 September 2017
Accepted 27 September 2017
Available online xxxx
Communicated by P. Stevens

Keywords:
Algorithmic game semantics
Symbolic automata
Program families
Model checking algorithms

ABSTRACT

Many software systems are today built as program families. They permit users to derive a custom program (variant) by selecting suitable configuration options at compile time according to their requirements. Many such program families are safety critical. However, most existing verification techniques are designed to work on the level of single programs. Their application to program families would require to verify each variant in isolation, in a brute force fashion. This approach does not scale in practice due to the (potentially) huge number of possible variants.

In this paper, we propose an efficient game semantics based approach for verification of open program families, i.e. program families with undefined components (identifiers). We use symbolic representation of algorithmic game semantics, where symbolic values for inputs are used instead of concrete ones. In this way, we can compactly represent program families with infinite integers as so-called (finite state) featured symbolic automata. Specifically designed model checking algorithms are then employed to uniformly verify safety of all programs (variants) from a family at once using a single compact model and to pinpoint those programs that are unsafe (respectively, safe). We present a prototype tool implementing this approach, and we illustrate its practicality with several examples.

1. Introduction

Software Product Line (SPL) [1] is an efficient method for systematic development of a family of related programs, known as *variants* (*valid products*), from a common code base. Each variant is specified in terms of *features* (statically configured options) selected for that particular variant. Although there are different strategies for implementing product lines [1,2], many popular SPLs from system software (e.g. Linux kernel) and embedded software (e.g. cars, phones, avionics, healthcare) domains [2] are implemented using annotative approaches such as *conditional compilation*. They enable a simple form of two staged computation in preprocessor style, by extending the programming language with conditional compilation constructs (e.g. `#ifdef` annotations from C preprocessor [3]). At build time, the program family is first configured and a variant describing a particular product is derived by selecting a set of features relevant for it, and only then the derived variant is compiled or interpreted. One of the advantages of preprocessors is that they are mostly independent of the object language and can be applied across paradigms.

Benefits from using program families (SPLs) are multiple: productivity gains, shorter time to market, greater market coverage, increased software quality, etc. Unfortunately, the complexity created by program families (variability) also leads to problems. The simplest *brute-force approach* to verify such program families is to generate all valid variants of a family

E-mail address: adim@itu.dk.

https://doi.org/10.1016/j.tcs.2017.09.029
0304-3975/

using a preprocessor, and then apply an existing single-program verification technique to each resulting variant. However, this approach is very costly and often infeasible in practice since the number of possible variants is exponential in the number of features. All variants will be verified independently one by one despite of their great similarity. This means that one behaviour will be verified as many times as there are variants able to produce it. Therefore, we seek new approaches that rely on finding compact mathematical structures, which take the similarity within the family into account, and on which specialized variability-aware verification algorithms can be applied.

In this work, we address the above challenges by using game semantics models. Game semantics [4–7] is a technique for *compositional* modelling of programming languages, which gives models that are fully abstract (sound and complete) with respect to observational equivalence of programs. It has mathematical elegance of denotational semantics, and step-by-step modelling of computation in the style of operational semantics. In the last two decades, a new line of research has been pursued, known as *algorithmic game semantics*, where game semantics models are given concrete representations as formal languages [8–10]. Thus, they can serve as a basis for software model checking and static program analysis. The most distinctive property of game semantics is compositionality, i.e. the models are generated inductively on the structure of programs. This means that the model of a larger program is obtained from the models of its constituting subprograms, using a notion of composition. Moreover, game semantics yields a very accurate model for any open program with undefined identifiers such as calls to library functions. Finally, the model hides the details of local-state manipulation of a program, and only records how the program observationally interacts with its environment. These properties are essential to achieve *modular* verification, where a larger program is broken down into smaller program fragments which can be modelled and verified independently.

A symbolic representation of algorithmic game semantics for 2nd-order Idealized Algol (IA_2) has been proposed in [11]. It redefines the (standard) regular-language representation [8] at a more abstract level by using symbolic values instead of concrete ones. This allows to give a compact representation of programs with infinite integers by using finite-state symbolic automata. A set of concrete behaviours (plays) from the standard model that fulfils some constraint are represented by a single symbolic behaviour in the symbolic model. Here, we extend the symbolic representation of game semantics models to IA_2 enriched with `#ifdef` constructs, obtaining so-called *featured symbolic automata*, where every symbolic behaviour of a program family is associated with a set of variants able to produce it. This enables us to verify a behaviour only once regardless of how many variants can produce it. Hence, we can use featured symbolic automata to compactly represent and efficiently verify safety properties of program families.

In this paper, we make the following contributions:

- **(C1)** We define algorithmic game semantics of so-called *annotative* program families. That is, program families which are implemented by annotating program parts that vary using preprocessor directives. We also introduce a compact symbolic representation, called *featured symbolic automata*, to represent game semantics models of program families. We use a single model to represent all instances of a family in order to exploit the similarities between them.
- **(C2)** We propose specifically designed family-based model checking algorithms for exploring featured symbolic automata that represent program families. This allows us to uniformly verify safety for all variants of a family using a *single* compact model, and to pinpoint the variants that are unsafe along with the corresponding counter-examples.
- **(C3)** We describe a prototype tool implementing the above algorithms, and we perform an evaluation to demonstrate the improvements of our technique over the brute-force approach where all valid variants are verified independently one by one.

This work is an extended and revised version of [12]. Compared to the earlier work, we make the following extensions here. We expand and elaborate the examples as well as the definition of algorithmic game semantics for `#ifdef` commands. We also extend the description of family-based model checking algorithms and the formal results supporting them. We provide formal proofs for all main results. In addition, we provide more cases for evaluation and support our claims by some practical results.

We proceed by giving a motivating example for family-based model checking based on game semantics in Section 2. In Section 3 we introduce the languages for writing single programs and program families. Their symbolic game semantics models are presented in Section 4 for single programs and in Section 5 for program families. The family-based model checking problems and algorithms for solving them are studied in Section 6. Section 7 describes the prototype tool implementing this approach and evaluates its effectiveness by several examples. Finally, we discuss the related work and conclude.

2. Motivating example

To better illustrate the issues we are addressing in this work, we now present a motivating example. Consider the following program family M:

$$
\begin{aligned}
n : \exp\mathsf{int}^n, abort : \mathsf{com}^{abort} \vdash_{\{A,B\}} \mathsf{new}_{int}\, x := 0 \,\mathsf{in}\\
\#\mathtt{if}\,(A)\,\mathsf{then}\, x := x + n;\\
\#\mathtt{if}\,(B)\,\mathsf{then}\, x := x - n;\\
\mathsf{if}\,(x = 1)\,\mathsf{then}\, abort\,\mathsf{else}\,\mathsf{skip} : \mathsf{com}
\end{aligned}
$$

Please cite this article in press as: A.S. Dimovski, Verifying annotated program families using symbolic game semantics, Theoret. Comput. Sci. (2017), https://doi.org/10.1016/j.tcs.2017.09.029