Theoretical Computer Science 706 (2018) 35–53

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 4 October 2017
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/© 2017 Elsevier B.V. All rights reserved.

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:

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

Table 1
Variants derived from the program family M.

$n : \mathsf{exp\,int}^n, abort : \mathsf{com}^{abort} \vdash$ $\mathsf{new_{int}}\ x := 0\ \mathsf{in}$ $x := x + n;$ $x := x - n;$ $\mathsf{if}\ (x = 1)\ \mathsf{then}\ abort\ \mathsf{else\ skip} : \mathsf{com}$ (a) Variant for $A \wedge B$.	$n : \mathsf{exp\,int}^n, abort : \mathsf{com}^{abort} \vdash$ $\mathsf{new_{int}}\ x := 0\ \mathsf{in}$ $\mathsf{if}\ (x = 1)\ \mathsf{then}\ abort\ \mathsf{else\ skip} : \mathsf{com}$ (b) Variant for $\neg A \wedge \neg B$.
$n : \mathsf{exp\,int}^n, abort : \mathsf{com}^{abort} \vdash$ $\mathsf{new_{int}}\ x := 0\ \mathsf{in}$ $x := x + n;$ $\mathsf{if}\ (x = 1)\ \mathsf{then}\ abort\ \mathsf{else\ skip} : \mathsf{com}$ (c) Variant for $A \wedge \neg B$.	$n : \mathsf{exp\,int}^n, abort : \mathsf{com}^{abort} \vdash$ $\mathsf{new_{int}}\ x := 0\ \mathsf{in}$ $x := x - n;$ $\mathsf{if}\ (x = 1)\ \mathsf{then}\ abort\ \mathsf{else\ skip} : \mathsf{com}$ (d) Variant for $\neg A \wedge B$.

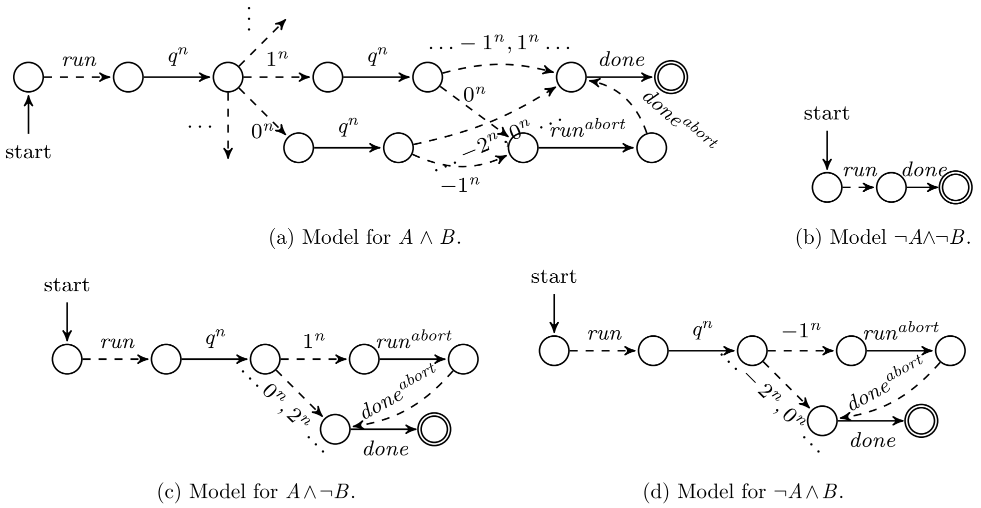

Fig. 1. Automata for all variants of M.

where n is an undefined integer expression and *abort* is a special undefined command. The set of (Boolean) features in the above family M is $\mathbb{F} = \{A, B\}$, and we assume the set of valid configurations is $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. The family M contains two `#if` commands, which increase and decrease the local variable x by the value of the input expression n, depending on which features from $\mathbb{F}$ are enabled. For each valid configuration a different single program can be generated by appropriately resolving `#if` commands. For example, the variant corresponding to the valid configuration $A \wedge B$ will have both features A and B enabled (set to true), which will make both assignment commands in `#if`-s appear in the variant. Programs for $A \wedge \neg B$ and for $\neg A \wedge B$ are different in one assignment command only, the earlier has the feature A enabled and the command $x := x + n$, whereas the latter has the feature B enabled and the command $x := x - n$. Variants corresponding to all configurations are illustrated in Table 1. Thus, if we use existing single-program verification techniques to verify the family M we would need to build and analyze models of four distinct, but very similar, programs.

We show in Fig. 1, the standard regular-language representation of game semantics for these four programs where concrete data values for the inputs (that is, n) are used [8]. We can see that we obtain regular-languages with infinite summations (i.e. infinite-state automata), since we use infinite integers as data type. Hence, they can be used for automatic verification only if the attention is restricted to finite data types [13]. Note that in all models the dashed edges indicate moves of the environment (Opponent) and solid edges moves of the term (Player). They serve only as a visual aid to the reader. Accepting states are designated using an interior circle. For example, the model for the variant $A \wedge \neg B$ in Fig. 1c illustrates the observable interactions of this term with its environment consisting of undefined identifiers n and *abort*. So in the model are only represented moves associated with types of n and *abort* (which are tagged with superscripts n and *abort*, respectively) as well as moves associated with the top-level type com of this term. The environment (Opponent) starts the execution of the term by playing the move *run*; when the term (Player) asks for the value of n with the move q^n, the environment can provide any integer as answer. If the answer is 1, the *abort* is run; otherwise the term terminates successfully by reaching the accepting state. The model for variant $\neg A \wedge \neg B$ is very simple. The environment (Opponent) initiates executing this term by the move *run*, and the term (Player) immediately answers with the move *done* to signal successful termination. This model is equal to the model of the command skip. Therefore, from the full abstraction result for game semantics [8], it follows that the variant $\neg A \wedge \neg B$ shown in Table 1b and skip are observationally equivalent. In the model for the variant $A \wedge B$ in Fig. 1a, the term (Player) asks for the value of n two times. When the first value of n provided by the environment (Opponent) is greater by one than the second value of n, the *abort* is run. In game semantics,

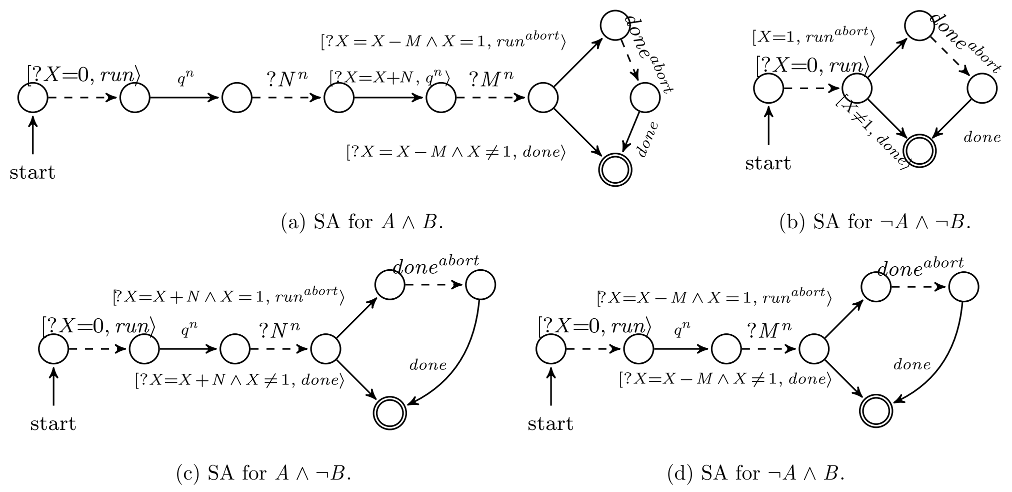

Fig. 2. Symbolic automata for all variants of M.

we consider all possible environments (contexts) in which a term can be placed. Therefore, the undefined expression n may obtain different values at each call in the above variant [8,13]. Note that each move in a model represents an observable action that a term of a given type can perform. Thus, for commands we have a move *run* to initiate a command and a move *done* to signal successful termination of a command, whereas for expressions we have a move q to ask for the value of an expression and an integer move to answer the question q.

If we represent the data at a more abstract level and use symbolic values instead of concrete ones, the game models of these four programs can be represented more compactly by finite-state symbolic automata (SA) as shown in Fig. 2. Symbols are placeholders that can take on any value in some data domain (e.g. integers, booleans). Every letter (label of transition) contains a move and a Boolean condition which represents a constraint that needs to be fulfilled in order the corresponding move to be performed. Note that so-called *input symbols* of the form $?N$ are used for generating new fresh symbolic names, which bind all occurrences of the symbol N that follow in the play until a new input symbol $?N$ is met. The symbol X is used to keep track of the current value of the local variable x. For example, the answer to the question q^n asked by the term for $A \wedge \neg B$ in Fig. 2c now is a newly instantiated symbol from $?N$, say N. If the value of N is 1, the *abort* command is run. Consider the play (trace) in Fig. 2c:

$$[?X{=}0, run\rangle \cdot q^n {\cdot} ?N^n \cdot [?X{=}X{+}N{\wedge}X{=}1, run^{abort}\rangle \cdot done^{abort} \cdot done$$

After instantiating its input symbols with fresh symbolic names (the first occurrence of $?X$ is instantiated with X_1, the second occurrence of $?X$ with X_2, and $?N$ with N), we obtain the following symbolic play:

$$[X_1 = 0, run\rangle \cdot q^n \cdot N^n \cdot [X_2 = X_1{+}N{\wedge}X_2 = 1, run^{abort}\rangle \cdot done^{abort} \cdot done$$

The constraint "$X_1 = 0 \wedge X_2 = X_1{+}N \wedge X_2 = 1$" is called the play condition of the above play. We say that a play is *feasible*, only if its play condition is satisfiable (i.e. there exist concrete assignments to symbols that make that condition true). This can be checked by calling a Satisfiability Modulo Theories (SMT) solver. For example, the above symbolic play is feasible since its play condition is satisfiable for $X_1{=}0, X_2{=}1, N{=}1$, thus yielding the corresponding concrete play: $run \cdot q^n \cdot 1^n \cdot run^{abort} \cdot done^{abort} \cdot done$ (see the concrete model in Fig. 1c). The following play for the variant $\neg A \wedge \neg B$ in Fig. 2b: $[?X = 0, run\rangle \cdot [X = 1, run^{abort}\rangle \cdot done^{abort} \cdot done$ is infeasible, since its play condition "$X_1{=}0{\wedge}X_1{=}1$" is unsatisfiable.

Now, by further enriching letters with feature expressions (propositional formulae defined over the set of features $\mathbb{F}$), we can give more compact single representation of the above related programs to exploit the similarities between them. The feature expression associated with a letter denotes for which valid configurations that letter is feasible. Thus, we can represent all variants of the family M by one compact featured symbolic automaton (FSA) as shown in Fig. 3, which is variability-aware extension of the symbolic automata in Fig. 2. From this model, by exploring all states we can determine for each valid variant whether it is possible or not an unsafe behaviour (one that contains *abort* moves) to occur. If we find such an unsafe play for a valid variant, then we need to check that the found play is feasible. If its play condition is satisfiable, the SMT solver will return concrete assignments to symbols which make that condition true (tt). In this way, we will generate a concrete counter-example (play) for a valid variant. In our example, we can determine that the variant $\neg A{\wedge}\neg B$ is safe, whereas variants $A{\wedge}B$, $A{\wedge}\neg B$, and $\neg A{\wedge}B$ are unsafe with concrete counterexamples: $run {\cdot} q^n {\cdot} 1^n {\cdot} q^n {\cdot} 0^n {\cdot} run^{abort} {\cdot} done^{abort} {\cdot} done$, $run {\cdot} q^n {\cdot} 1^n {\cdot} run^{abort} {\cdot} done^{abort} {\cdot} done$, and $run {\cdot} q^n {\cdot} {-}1^n {\cdot} run^{abort} {\cdot} done^{abort} {\cdot} done$, respectively. Note that, in general there may be many solutions for a play condition which will correspond to various concrete plays instantiated from the given symbolic play. For example, the play condition of the unsafe symbolic play for variant $A{\wedge}B$ in Fig. 2a is: $X_1{=}0{\wedge}X_2{=}X_1{+}N{\wedge}X_3{=}X_2{-}M{\wedge}X_3{=}1$. Its solutions are: $N{=}1{\wedge}M{=}0$, $N{=}2{\wedge}M{=}1$, $N{=}3{\wedge}M{=}2$, etc. However, in practice we need only one solution for a given play condition in order to generate one concrete play (counter-example) corresponding to the found unsafe symbolic play.

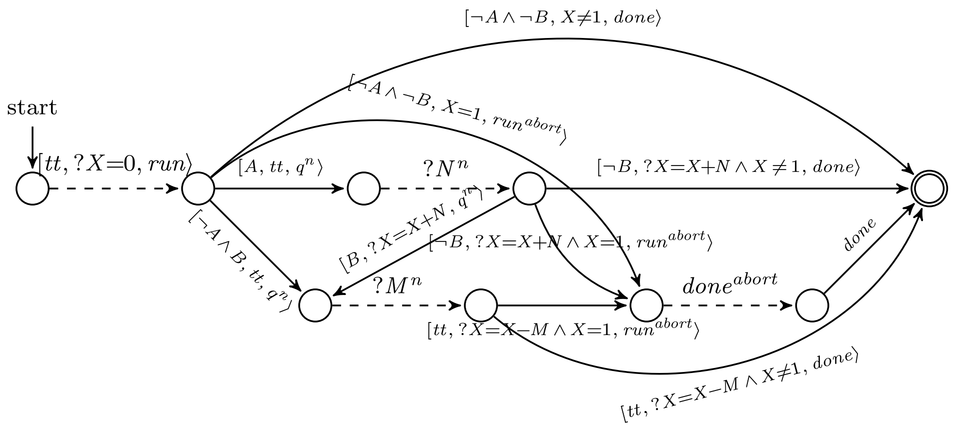
start
$[tt, ?X{=}0, run\rangle$
$[A, tt, q^n\rangle$
$?N^n$
$[\neg A \wedge B, tt, q^n\rangle$
$[B, ?X{=}X{+}N, q^n\rangle$
$?M^n$
$[\neg A \wedge \neg B, X{\neq}1, done\rangle$
$[\neg A \wedge \neg B, X{=}1, run^{abort}\rangle$
$[\neg B, ?X{=}X{+}N \wedge X \neq 1, done\rangle$
$[\neg B, ?X{=}X{+}N \wedge X{=}1, run^{abort}\rangle$
$[tt, ?X{=}X{-}M \wedge X{=}1, run^{abort}\rangle$
$done^{abort}$
$done$
$[tt, ?X{=}X{-}M \wedge X{\neq}1, done\rangle$

Fig. 3. Featured symbolic automaton for the program family M.

Remark. Alternatively, a program family can be verified by generating a so-called family *simulator* [14–16], which is a single program where `#if` commands (compile-time variability) are replaced with normal `if` commands and available features are encoded as free (undefined) identifiers. Then the classical single-system game-based model checking algorithms [17,11] can be used to verify the generated simulator, since it represents a single program. If the simulator is safe, we can conclude that all variants in the family are safe as well. But in case of violation, we will obtain a single counter-example that corresponds to some unsafe variants. However, this answer is incomplete (limited) for program families since there might be some safe variants and also there might be other unsafe variants with different counter-examples. Hence, no conclusive results for all variants in a family will be reported using this approach. For example, the simulator for the family M is:

$$\begin{aligned} n : \mathsf{exp\,int}, abort : \mathsf{com}, A, B : \mathsf{exp\,bool} \vdash\ & \mathsf{new_{int}}\, x := 0 \ \mathsf{in} \\ & \mathsf{if}\,(A)\,\mathsf{then}\, x := x + n; \\ & \mathsf{if}\,(B)\,\mathsf{then}\, x := x - n; \\ & \mathsf{if}\,(x = 1)\,\mathsf{then}\, abort\ \mathsf{else\ skip} : \mathsf{com} \end{aligned}$$

If we generate a (concrete or symbolic) game model for this term [8,11] and verify it using algorithms in [17,11], we will obtain the shortest counter-example corresponding only to the variant $A \wedge \neg B$. No conclusive results for the other variants are reported in this way.

This leads us to propose an approach that solves the general family-based model checking problem: determine for each variant whether or not it is safe, and provide a counter-example for each unsafe variant. Hence, this new family-based approach will report the same complete answers for all variants as the answers obtained from the brute-force approach, but in a much more efficient way.

3. The language for program families

The usage of meta-languages is very common in the semantics community. The semantic model is defined for a meta-language, and a real programming language (C, ML, etc.) can be studied by translating it into this meta-language and using the induced model. We begin this section by presenting a meta-language for which algorithmic game semantics is defined, and then we introduce static variability into it.

Writing single programs. We consider Idealized Algol (IA), a language introduced by Reynolds [18]. It is a compact language which combines call-by-name typed λ-calculus with the fundamental imperative features and locally-scoped variables. We work with its second-order recursion-free fragment (IA_2 for short), because game semantics of this fragment has good algorithmic properties.

The data types D are integers and booleans ($D ::= \mathsf{int} \mid \mathsf{bool}$). We have base types B ($B ::= \mathsf{exp}D \mid \mathsf{com} \mid \mathsf{var}D$) and first-order function types T ($T ::= B \mid B \to T$). The *syntax* of the language is given by:

$$\begin{aligned} M ::= & x \mid v \mid \mathsf{skip} \mid \mathsf{diverge} \mid M\, \mathsf{op}\, M \mid M;M \mid \mathsf{if}\, M\, \mathsf{then}\, M\, \mathsf{else}\, M \mid \mathsf{while}\, M\, \mathsf{do}\, M \\ & \mid M := M \mid !M \mid \mathsf{new}_D\, x := v\ \mathsf{in}\ M \mid \mathsf{mkvar}_D MM \mid \lambda x.M \mid MM \end{aligned}$$

where x ranges over a countable set of identifiers, and v ranges over constants of type D, which includes integers (n) and booleans (tt, ff). The standard arithmetic-logic operations op are employed, as well as the usual imperative constructs:

sequential composition (;), conditional (if), iteration (while), assignment (:=), de-referencing (!),[1] a "do-nothing" command (skip), and a divergence command (diverge). Block-allocated local variables are introduced by a new construct, which initializes a variable and makes it local to a given block. They are also called "good" (storage) variables since what is read from a variable is the last value written into it. The construct mkvar is used for creating so-called "bad" variables, which do not behave like genuine storage variables [6]. There are also standard functional constructs for function definition and application. *Well-typed terms* are given by typing judgements of the form $\Gamma \vdash M : T$, where $\Gamma = x_1 : T_1, \dots, x_k : T_k$ is a *type context* consisting of a finite number of typed free identifiers. Typing rules are given in [5,6].

The *operational semantics* is defined by a big-step reduction relation:

$$\Gamma \vdash M, \mathsf{s} \Longrightarrow V, \mathsf{s}'$$

where $\Gamma \vdash M : T$ is a term in which all free identifiers from Γ are variables, i.e. $\Gamma = x_1 : \mathsf{var}D_1, \dots, x_k : \mathsf{var}D_k$, and s, s′ represent the *state* before and after reduction. The state is a function assigning data values to the variables in Γ. Canonical forms (values) are defined by: $V ::= x \mid v \mid \lambda x.M \mid \mathsf{skip} \mid \mathsf{mkvar}_D MN$. Reduction rules are standard (see [5,6] for details). Given a closed term $\vdash M : \mathsf{com}$, which has no free identifiers, we say that M *terminates* if $\vdash M, \emptyset \Longrightarrow \mathsf{skip}, \emptyset$. We define a *program context* $C[-] : \mathsf{com}$ to be a term with zero or more holes $[-]$ in it, such that if $\Gamma \vdash M : T$ is a term of the same type as the hole then $C[M]$ is a well-typed closed term of type com, i.e. $\vdash C[M] : \mathsf{com}$. We say that a term $\Gamma \vdash M : T$ is an *approximate* of a term $\Gamma \vdash N : T$, written $\Gamma \vdash M \sqsubseteq N$, if and only if for all contexts $C[-] : \mathsf{com}$, such that $\vdash C[M] : \mathsf{com}$ and $\vdash C[N] : \mathsf{com}$, if $C[M]$ terminates then $C[N]$ terminates. If two terms approximate each other they are considered *observationally-equivalent*, denoted by $\Gamma \vdash M \cong N$.

Writing program families. We use a simple form of two-staged computation to lift IA_2 from describing single programs to $\overline{\mathsf{IA}_2}$ for describing program families. The first stage is controlled by a *configuration* k, which describes the set of features that are enabled in the build process. A finite set of Boolean variables describes the available features $\mathbb{F} = \{A_1, \dots, A_n\}$. A configuration k is a truth assignment (a mapping from $\mathbb{F}$ to $\mathsf{bool} = \{tt, ff\}$) which gives a truth value to any feature. If a feature $A \in \mathbb{F}$ is enabled (included) for the configuration k, then $k(A) = tt$. Any configuration k can also be encoded as a conjunction of propositional formulae: $k(A_1) \cdot A_1 \wedge \ldots \wedge k(A_n) \cdot A_n$, where $tt \cdot A = A$ and $ff \cdot A = \neg A$. We write $\mathbb{K}$ for the set of all *valid configurations* defined over $\mathbb{F}$ for a program family. The set of valid configurations is typically described by a feature model [1], but in this work we disregard syntactic representations of the set $\mathbb{K}$.

We add a new syntactic category in $\overline{\mathsf{IA}_2}$ of feature expressions, denoted $FeatExp(\mathbb{F})$, as the set of well-formed propositional logic formulae over $\mathbb{F}$ defined as:

$$\phi ::= A \in \mathbb{F} \mid \neg\phi \mid \phi \wedge \phi$$

We will use $\phi \in FeatExp(\mathbb{F})$ to write presence conditions over features $\mathbb{F}$ in $\overline{\mathsf{IA}_2}$. The language $\overline{\mathsf{IA}_2}$ extends IA_2 with a new compile-time conditional term for encoding multiple variations of a program, i.e. different valid variants. The new term "`#if` ϕ `then` M `else` M'" contains a presence condition ϕ over features $\mathbb{F}$, such that if ϕ is satisfied by a configuration $k \in \mathbb{K}$ then M will be included in the resulting variant, otherwise M' will be included. The new syntax is:

$$M ::= \ldots \mid \texttt{\#if}\ \phi\ \texttt{then}\ M\ \texttt{else}\ M'$$

Well-typed term families are given by typing judgements of the form $\Gamma \vdash_{\mathbb{F}} M : T$, where $\mathbb{F}$ is a set of available features.[2] Typing rules are those of IA_2 extended with a rule for the new construct:

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash_{\mathbb{F}} M : T} \qquad \frac{\Gamma \vdash_{\mathbb{F}} M : T \qquad \Gamma \vdash_{\mathbb{F}} M' : T \qquad \phi : FeatExp(\mathbb{F})}{\Gamma \vdash_{\mathbb{F}} \texttt{\#if}\ \phi\ \texttt{then}\ M\ \texttt{else}\ M' : T}$$

The semantics of $\overline{\mathsf{IA}_2}$ has two stages: first, given a configuration k compute a single IA_2 term without `#if`-s; second, evaluate the IA_2 term using the standard IA_2 semantics. The first stage of computation (also called *projection*) is a simple preprocessor from $\overline{\mathsf{IA}_2}$ to IA_2 specified by the projection function π_k mapping an $\overline{\mathsf{IA}_2}$ term family into a single IA_2 term corresponding to the configuration $k \in \mathbb{K}$. The projection π_k copies all basic terms of $\overline{\mathsf{IA}_2}$ that are also IA_2 terms, and recursively pre-processes all sub-terms of compound terms. For example, $\pi_k(\mathsf{skip}) = \mathsf{skip}$ and $\pi_k(M;M') = \pi_k(M);\pi_k(M')$. The interesting case is for the compilation-time conditional term, where one of the two alternative branches is included in the generated valid product depending on whether the configuration k satisfies (entails) the feature expression ϕ, denoted as $k \models \phi$. We have:

$$\pi_k(\texttt{\#if}\ \phi\ \texttt{then}\ M\ \texttt{else}\ M') = \begin{cases} \pi_k(M) & \text{if } k \models \phi \\ \pi_k(M') & \text{if } k \not\models \phi \end{cases}$$

The variant of a term family $\Gamma \vdash_{\mathbb{F}} M : T$ corresponding to the configuration $k \in \mathbb{K}$ is defined as: $\Gamma \vdash \pi_k(M) : T$.

[1] Note that the de-referencing operator ! is used for reading the value stored in a variable. For clarity of presentation, we omit to write ! when reading from variables in our examples.

[2] For the work in this paper, we assume that the set of features $\mathbb{F}$ is fixed and all features are globally scoped.

4. Symbolic game models for single programs

We now give an overview of symbolic representation of the algorithmic game semantics for IA$_2$ introduced in [11].

Let Sym be a countable set of symbolic names, ranged over by upper case letters X, Y, Z. For any finite $W \subseteq Sym$, the function $new(W)$ returns a minimal symbolic name which does not occur in W, and sets $W := W \cup \{new(W)\}$. A minimal symbolic name not in W is the one which occurs earliest in a fixed enumeration of all possible symbolic names. Let Exp be a set of expressions, ranged over by e, generated by data values ($v \in D$), symbols ($X \in Sym$) of data type D, and standard arithmetic-logic operations. We use a to range over arithmetic expressions ($AExp$) and b over boolean expressions ($BExp$).

Let $\mathcal{A}_{[\![D]\!]}$ be an alphabet of data values from D. We have $\mathcal{A}_{[\![\mathsf{int}]\!]} = \mathbb{Z}$ and $\mathcal{A}_{[\![\mathsf{bool}]\!]} = \{tt, ff\}$. We define a *symbolic alphabet* $\mathcal{A}^{sym}_{[\![D]\!]}$ induced by $\mathcal{A}_{[\![D]\!]}$ as follows:

$$\mathcal{A}^{sym}_{[\![D]\!]} = \mathcal{A}_{[\![D]\!]} \cup \{?X, e \mid X \in Sym, e \in Exp\}$$

where data values in e and symbols X are of data type D. The letters of the form $?X$ are called *input symbols*. They represent a mechanism for generating new symbolic names, i.e. $?X$ means $\mathsf{let}\, X = new(W)\, \mathsf{in}\, X$. More specifically, $?X$ creates a stream of fresh symbolic names, binding X to the next symbol from its stream whenever $?X$ is evaluated (met). We use α to denote a symbolic letter. Given an arbitrary symbolic alphabet $\mathcal{A}^{sym}$, we define a *guarded alphabet* $\mathcal{A}^{gu}$ induced by $\mathcal{A}^{sym}$ as the set of pairs of boolean conditions and symbolic letters:

$$\mathcal{A}^{gu} = \{[b, \alpha\rangle \mid b \in BExp, \alpha \in \mathcal{A}^{sym}\}$$

A guarded letter $[b, \alpha\rangle$ is α only if b evaluates to true otherwise it is the constant $\emptyset$ (the language of $\emptyset$ is $\emptyset$), i.e. $if\,(b = tt)\,then\,\alpha\,else\,\emptyset$. We use β to denote a guarded letter. We will often write only α for the guarded letter $[tt, \alpha\rangle$. A word $[b_1, \alpha_1\rangle \cdot [b_2, \alpha_2\rangle \dots [b_n, \alpha_n\rangle$ over $\mathcal{A}^{gu}$ can be represented as a pair $[b, w\rangle$, where $b = b_1 \wedge b_2 \wedge \dots \wedge b_n$ is a boolean condition and $w = \alpha_1 \cdot \alpha_2 \dots \alpha_n$ is a word of symbolic letters.

Now, we show how IA$_2$ terms in beta-normal form are interpreted by symbolic regular languages and automata, which will be specified by extended regular expressions R. They are defined inductively over finite guarded alphabets $\mathcal{A}^{gu}$ using the following operations [11]:

$$\emptyset \quad \varepsilon \quad \beta \quad R \cdot R' \quad R^* \quad R + R' \quad R \cap R' \quad R[R'/w] \quad R' \circ R$$

where R, R' range over extended regular expressions, $\beta \in \mathcal{A}^{gu}$, and $w \in \mathcal{A}^{gu*}$. Constants $\emptyset$, ε, and β, concatenation $R \cdot R'$, Kleene star R^*, union $R + R'$ and intersection $R \cap R'$ are the standard operations. Substitution $R[R'/w]$ is the language of R where all occurrences of the subword w have been replaced by the words of R'. Composition ($\circ$) of regular expressions R' defined over $\mathcal{A}^{gu\langle 1\rangle} + \mathcal{B}^{gu\langle 2\rangle}$[3] and R over $\mathcal{B}^{gu\langle 2\rangle} + \mathcal{C}^{gu\langle 3\rangle}$ is defined as "parallel composition followed by hiding" in CSP style [5]. The parallel composition is matching (synchronizing) of the letters in the shared alphabets (here $\mathcal{B}^{gu\langle 2\rangle}$), whereas hiding is deleting of all letters from the shared alphabets. Conditions of the shared (interacting) letters are conjoined, along with the condition that their symbolic letters are equal if they contain symbols from Sym. The obtained conjunction is propagated in the composition through the letters from $\mathcal{A}^{gu\langle 1\rangle}$. More specifically, we have:

$$\begin{aligned} R' \circ R = \{w\big[[b \wedge b_1 \wedge b_2 \wedge b'_1 \wedge b'_2 \wedge cond, s\rangle^{\langle 1\rangle}/[b_1, \alpha_1\rangle^{\langle 2\rangle} \cdot [b_2, \alpha_2\rangle^{\langle 2\rangle}\big] \mid \\ w \in R, [b'_1, \alpha'_1\rangle^{\langle 2\rangle} \cdot [b, s\rangle^{\langle 1\rangle} \cdot [b'_2, \alpha'_2\rangle^{\langle 2\rangle} \in R'\} \end{aligned}$$

where R' is a set of words of form $[b'_1, \alpha'_1\rangle^{\langle 2\rangle} \cdot [b, s\rangle^{\langle 1\rangle} \cdot [b'_2, \alpha'_2\rangle^{\langle 2\rangle}$, such that $[b'_1, \alpha'_1\rangle^{\langle 2\rangle}, [b'_2, \alpha'_2\rangle^{\langle 2\rangle} \in \mathcal{B}^{gu\langle 2\rangle}$ and $[b, s\rangle^{\langle 1\rangle}$ contains only letters from $\mathcal{A}^{gu\langle 1\rangle}$. If α_1 and α'_1 (resp., α_2 and α'_2) do not contain symbols then they have to be equal, otherwise we add the conjunct $\alpha_1 = \alpha'_1$ (resp., $\alpha_2 = \alpha'_2$) to $cond$. Thus, all letters of $\mathcal{B}^{gu\langle 2\rangle}$ are removed from the composition, which is defined over the alphabet $\mathcal{A}^{gu\langle 1\rangle} + \mathcal{C}^{gu\langle 3\rangle}$.

Each type T is interpreted by a guarded alphabet of moves $\mathcal{A}^{gu}_{[\![T]\!]}$ induced by $\mathcal{A}^{sym}_{[\![T]\!]}$, which is defined as follows[4]:

$$\begin{aligned} &\mathcal{A}^{sym}_{[\![\mathsf{exp}D]\!]} = \{q\} \cup \mathcal{A}^{sym}_{[\![D]\!]}, \quad \mathcal{A}^{sym}_{[\![\mathsf{var}D]\!]} = \{write(a), read, ok, a \mid a \in \mathcal{A}^{sym}_{[\![D]\!]}\}, \\ &\mathcal{A}^{sym}_{[\![\mathsf{com}]\!]} = \{run, done\}, \quad \mathcal{A}^{sym}_{[\![B_1^{\langle 1\rangle} \to \dots \to B_k^{\langle k\rangle} \to B]\!]} = \sum_{1 \le i \le k} \mathcal{A}^{sym\,\langle i\rangle}_{[\![B_i]\!]} + \mathcal{A}^{sym}_{[\![B]\!]} \end{aligned}$$

Function types are tagged by a superscript $\langle i\rangle$ to keep record from which type each move (letter) comes from. The letters in the alphabet $\mathcal{A}_{[\![T]\!]}$ represent *moves* (observable actions) that a term of type T can perform. Each move is either a *question* (a demand for information) or an *answer* (a supply of information). For expressions in $\mathcal{A}_{[\![\mathsf{exp}D]\!]}$, there is a *question* move q to ask for the value of the expression, and values from $\mathcal{A}_{[\![D]\!]}$ to *answer* the question. For variables, there are *question* moves for writing to the variable, *write*(a), which are acknowledged by the *answer* move *ok*; and there is a *question* move *read* for reading from the variable, which is *answered* by a value from $\mathcal{A}_{[\![D]\!]}$. For commands, there is a *question* move *run* to initiate a command, and an *answer* move *done* to signal successful termination of a command.

[3] Alphabets are tagged by a superscript $\langle i\rangle$ to keep record from which component of the disjoint union each letter comes from.

[4] Here, $+$ denotes a disjoint union of alphabets.

Table 2
Symbolic representations of some language constructs.

$[\![\mathsf{op} : \mathsf{exp}D_1^{\langle 1\rangle} \times \mathsf{exp}D_2^{\langle 2\rangle} \to \mathsf{exp}D]\!] = q \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot q^{\langle 2\rangle} \cdot ?Z'^{\langle 2\rangle} \cdot (Z \,\mathsf{op}\, Z')$
$[\![; : \mathsf{com}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \to \mathsf{com}]\!] = run \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done$
$[\![\mathsf{if} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \times \mathsf{com}^{\langle 3\rangle} \to \mathsf{com}]\!] = [tt, run\rangle \cdot [tt, q^{\langle 1\rangle}\rangle \cdot [tt, ?Z^{\langle 1\rangle}\rangle \cdot \big([Z, run^{\langle 2\rangle}\rangle \cdot [tt, done^{\langle 2\rangle}\rangle + [\neg Z, run^{\langle 3\rangle}\rangle \cdot [tt, done^{\langle 3\rangle}\rangle\big) \cdot [tt, done\rangle$
$[\![\mathsf{while} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \to \mathsf{com}]\!] = [tt, run\rangle \cdot [tt, q^{\langle 1\rangle}\rangle \cdot [tt, ?Z^{\langle 1\rangle}\rangle \cdot \big([Z, run^{\langle 2\rangle}\rangle \cdot [tt, done^{\langle 2\rangle}\rangle \cdot [tt, q^{\langle 1\rangle}\rangle \cdot [tt, ?Z^{\langle 1\rangle}\rangle\big)^* \cdot [\neg Z, done\rangle$
$[\![:= : \mathsf{var}D^{\langle 1\rangle} \times \mathsf{exp}D^{\langle 2\rangle} \to \mathsf{com}]\!] = run \cdot q^{\langle 2\rangle} \cdot ?Z^{\langle 2\rangle} \cdot write(Z)^{\langle 1\rangle} \cdot ok^{\langle 1\rangle} \cdot done$
$[\![! : \mathsf{var}D^{\langle 1\rangle} \to \mathsf{exp}D]\!] = q \cdot read^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot Z$
$\mathsf{cell}_v^{\langle x\rangle} = ([?X = v, read^{\langle x\rangle}\rangle \cdot X^{\langle x\rangle})^* \cdot \big(write(?X)^{\langle x\rangle} \cdot ok^{\langle x\rangle} \cdot (read^{\langle x\rangle} \cdot X^{\langle x\rangle})^*\big)^*$

For any (beta-normal) term, we define a symbolic regular-language which represents its game semantics, i.e. its set of complete symbolic plays. Every complete symbolic play represents the observable effects of a set of completed computations (execution paths, concrete complete plays) of the given term. It is given as a guarded word $[b, w\rangle$, where b is also called the *play condition*. Assumptions about a play to be feasible are recorded in its play condition. For infeasible plays, the play condition is inconsistent (unsatisfiable), thus no assignment of concrete values to symbolic names exists that makes the play condition true. We now formally define the set of concrete instantiations of symbolic plays by means of an evaluation $\rho : W \to \mathcal{A}_{[\![int]\!]} \cup \mathcal{A}_{[\![bool]\!]}$, which is a map that assigns a data value from $\mathcal{A}_{[\![D]\!]}$ to a symbol of type D. $Eval(W)$ denotes the set of all evaluations for symbols from W. Given a symbolic word w, let $\rho(w)$ be a word where every symbol X in w is replaced by the corresponding concrete value $\rho(X)$. Given a guarded word $[b, w\rangle$, we define:

$$\rho([b, w\rangle) = \begin{cases} \rho(w) & \text{if } \rho(b) = tt \\ \emptyset & \text{if } \rho(b) = ff \end{cases}$$

Let $[b, w\rangle$ represent a symbolic play, then the set of concrete plays instantiated from it is $\{\rho([b, w\rangle) \mid \rho \in Eval(W)\}$.

The regular expression for $\Gamma \vdash M : T$, denoted as $[\![\Gamma \vdash M : T]\!]$, is defined over the guarded alphabet:

$$\mathcal{A}^{gu}_{[\![\Gamma \vdash T]\!]} = \big(\textstyle\sum_{x:T' \in \Gamma} \mathcal{A}^{gu\,\langle x\rangle}_{[\![T']\!]}\big) + \mathcal{A}^{gu}_{[\![T]\!]}$$

where moves corresponding to types of free identifiers are tagged with their names.

The representation of constants is standard:

$$[\![\Gamma \vdash v : \mathsf{exp}D]\!] = q \cdot v, \quad [\![\Gamma \vdash \mathsf{skip} : \mathsf{com}]\!] = run \cdot done, \quad [\![\Gamma \vdash \mathsf{diverge} : \mathsf{com}]\!] = \emptyset$$

Free identifiers are represented by the so-called copy-cat regular expressions, which contain all possible behaviours of terms of that type. For example,

$$[\![\Gamma, x : \mathsf{exp}D_1^{\langle x,1\rangle} \to \ldots \mathsf{exp}D_k^{\langle x,k\rangle} \to \mathsf{exp}D^{\langle x\rangle} \vdash x : \mathsf{exp}D_1^{\langle 1\rangle} \to \ldots \mathsf{exp}D_k^{\langle k\rangle} \to \mathsf{exp}D]\!] = \\ q \cdot q^{\langle x\rangle} \cdot \big(\textstyle\sum_{1 \le i \le k} q^{\langle x,i\rangle} \cdot q^{\langle i\rangle} \cdot ?Z^{\langle i\rangle} \cdot Z^{\langle x,i\rangle}\big)^* \cdot ?X^{\langle x\rangle} \cdot X$$

When a call-by-name non-local function x is called, it may evaluate any of its arguments, zero or more times, in an arbitrary order and then it returns any allowable answer from its result type. Recall that the input symbol $?Z$ creates a stream of fresh symbolic names for each instantiation of $?Z$. Thus, whenever $?Z$ is met in a play, the mechanism for fresh symbol generation is used to dynamically instantiate it with a new fresh symbolic name from its stream, which binds all occurrences of Z that follow in the play until a new $?Z$ is met which overrides the previous symbolic name with the next symbolic name taken from its stream. For example, consider a non-local function $f : \mathsf{expint}^{\langle 1\rangle} \to \mathsf{expint}$. Its symbolic model is:

$$q \cdot q^{\langle f\rangle} \cdot \big(q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot Z^{\langle f,1\rangle}\big)^* \cdot ?X^{\langle f\rangle} \cdot X$$

The play corresponding to f which evaluates its argument two times after instantiating its input symbols is given as: $q \cdot q^{\langle f\rangle} \cdot q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot Z_1^{\langle 1\rangle} \cdot Z_1^{\langle f,1\rangle} \cdot q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot Z_2^{\langle 1\rangle} \cdot Z_2^{\langle f,1\rangle} \cdot X^{\langle f\rangle} \cdot X$, where Z_1 and Z_2 are two different symbolic names used to denote values of the argument when it is evaluated the first and the second time, respectively. Therefore, we are using the streaming symbol $?Z$ to create different symbolic names so that we can produce distinct values (independent from one another) if $?Z$ is evaluated multiple times during the execution. Note that letters tagged with $\langle f\rangle$ represent the actions of calling and returning from the function f, while letters tagged with $\langle f, 1\rangle$ are the actions caused by evaluating the first argument of f.

The representations of some language constructs are given in Table 2. Note that language constructs can be also given in a functional form, e.g. we have "$;(M, M') \equiv M\,; M'$", "$\mathsf{if}(B, M, M') \equiv \mathsf{if}\,(B)\,\mathsf{then}\,M\,\mathsf{else}\,M'$", etc. Observe that letter conditions different than tt occur only in plays corresponding to "if" and "while" constructs. In the case of "if" command, when the value of the first argument given by the symbol Z is true then its second argument is run, otherwise if $\neg Z$ is true then its third argument is run. A composite term $\mathsf{c}(M_1,\dots,M_k)$ built out of a language construct "c" and subterms $M_1,\dots,M_k$ is interpreted by composing the regular expressions for $M_1,\dots,M_k$ and the regular expression for "c". For example,

$$[\![\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ M' : \mathsf{com}]\!] = [\![\Gamma \vdash B]\!] \circ [\![\Gamma \vdash M]\!] \circ [\![\Gamma \vdash M']\!] \circ [\![\mathsf{if}]\!] \qquad (1)$$

where $[\![\mathsf{if}]\!]$ is defined in Table 2. The $\mathsf{cell}_v^{\langle x\rangle}$ expression in Table 2 is used to impose the good variable behaviour on a local variable x introduced using $\mathsf{new}_D\, x{:=}v\,\mathsf{in}\, M$. Note that v is the initial value of x, and X is a symbol used to track the current value of x. The $\mathsf{cell}_v^{\langle x\rangle}$ plays the most recently written value in x in response to *read*, or if no value has been written yet then answers *read* with v. The model $[\![\mathsf{new}_D\, x{:=}v\ \mathsf{in}\, M]\!]$ is obtained by constraining the model of M, $[\![\mathsf{var}_D\, x \vdash M]\!]$, only to those plays where x exhibits good variable behaviour, and then by deleting (hiding) all moves associated with x since x is a local variable and so not visible outside of the term [11]. Notice that all symbols used in Table 2 are of data type D, except the symbol Z in if and while constructs, which is of data type *bool*.

The following formal results are proved in [11]. We define an *effective alphabet* of a regular expression to be the set of all letters that appear in the regular language denoted by that regular expression. The effective alphabet of a regular expression representing any term $\Gamma \vdash M : T$ contains only a *finite subset* of letters from $\mathcal{A}^{gu}_{[\![\Gamma\vdash T]\!]}$, which includes all constants, symbols, and expressions used for interpreting free identifiers, language constructs, and local variables in M.

Theorem 1. *For any IA$_2$ term, the set $[\![\Gamma \vdash M : T]\!]$ is a symbolic regular-language without infinite summations defined over its effective finite alphabet. Moreover, a finite-state symbolic automaton $\mathcal{A}[\![\Gamma \vdash M : T]\!]$ which recognizes it is effectively constructible.*

Example 2. Consider the term:

$$f : \mathsf{exp\,int}^{f,1} \to \mathsf{exp\,bool}^{f} \vdash \mathsf{new_{int}}\, x{:=}0\, \mathsf{in\,while}\, \big(f(x)\big)\, \mathsf{do}\, x{:=}x+1 : \mathsf{com}$$

The symbolic regular expression corresponding to this term is[5]:

$$[?X{=}0, run\rangle \cdot q^f \cdot (q^{f,1} \cdot X^{f,1})^{*} {\cdot} ?Z^f \cdot \\ \big([Z{\wedge}?X{=}X{+}1, q^f\rangle \cdot (q^{f,1} \cdot X^{f,1})^{*} {\cdot} ?Z^f\big)^{*} \cdot [\neg Z, done\rangle$$

which can be represented as a (finite-state) automaton. In the model, we can see only observable interactions of this term with the environment consisting of the non-local call-by-name function f. When the term asks for the value returned from $f(x)$ with the move q^f, the environment may evaluate the argument of f, zero or more times, and then provides a new symbol Z as an answer to the question q^f. If Z is *tt* (true), the local variable x is increased and the term again asks for the value of $f(x)$. The term may continue to ask for the value returned from $f(x)$, until this value Z becomes *ff* (false), in which case the term terminates successfully with *done*. □

Suppose that there is a special free identifier *abort* of type com. We say that a term $\Gamma \vdash M$ is *safe*[6] iff $\Gamma \vdash M[\mathsf{skip}/abort] \sqsubseteq\!\!\!\!\sim M[\mathsf{diverge}/abort]$; otherwise we say that a term is *unsafe*. Hence, a safe term has no computation that leads to running *abort*. Let $[\![\Gamma \vdash M : T]\!]^{CR}$ denotes the (concrete) regular-language representation of game semantics for a term M obtained as in [8], where concrete values are used. Since this representation is fully abstract, and so there is a close correspondence with the operational semantics, the following holds [19].

Proposition 3. *A term $\Gamma \vdash M : T$ is safe, if and only if, $[\![\Gamma \vdash M : T]\!]^{CR}$ does not contain any play with moves from $\mathcal{A}^{\langle abort\rangle}_{[\![\mathsf{com}]\!]}$, which we call unsafe plays.*

Let $\rho[\![\Gamma{\vdash}M{:}T]\!] = \{\rho([b,w\rangle) \mid [b,w\rangle \in [\![\Gamma{\vdash}M{:}T]\!], \rho \in Eval(W)\}$. It has been shown in [11] that $\rho[\![\Gamma{\vdash}M{:}T]\!] = [\![\Gamma{\vdash}M{:}T]\!]^{CR}$. As a corollary, we obtain the following result which confirms that symbolic regular-languages $[\![\Gamma{\vdash}M{:}T]\!]$ can be used for establishing safety of terms.

Theorem 4. *$[\![\Gamma{\vdash}M{:}T]\!]$ is safe (all plays are safe), if and only if, $[\![\Gamma{\vdash}M{:}T]\!]^{CR}$ is safe.*

For example, $[\![abort{:}\,\mathsf{com}^{abort} \vdash \mathsf{skip}\,; abort : \mathsf{com}]\!] = run \cdot run^{abort} \cdot done^{abort} \cdot done$, so this term is unsafe since its model contains an unsafe play.

Since symbolic automata are finite state, we can use model-checking techniques to verify safety of IA$_2$ terms with integers. The verification procedure proposed in [11] searches for unsafe plays in the symbolic automata representing a term. If an unsafe play is found, it calls an external SMT solver (Yices) to check consistency (satisfiability) of its play condition. If the condition is consistent, then the corresponding concrete counter-example is reported. We showed in [11] that the procedure is correct and semi-terminating (terminates for unsafe terms, but may diverge for safe terms) under assumption that constraints generated by any program can be checked for consistency by some (SMT) solver.

[5] For simplicity, in examples we omit to write angle brackets $\langle , \rangle$ in superscript tags of moves.

[6] $M[N/x]$ denotes the capture-free substitution of N for x in M.

Example 5. Consider the term family M from Introduction. The symbolic model for the term $A \wedge B$ is given in Fig. 2a. It contains one unsafe play: $[?X=0, run\rangle \cdot q^n \cdot ?N^n \cdot [?X=X+N, q^n\rangle \cdot ?M^n \cdot [?X=X-M \wedge X=1, run^{abort}\rangle \cdot done^{abort} \cdot done$, which after instantiating its input symbols with fresh names becomes:

$$[X_1=0, run\rangle \cdot q^n \cdot N^n \cdot [X_2=X_1+N, q^n\rangle \cdot M^n \cdot \\ [X_3=X_2-M \wedge X_3=1, run^{abort}\rangle \cdot done^{abort} \cdot done$$

An SMT solver will inform us that its play condition is satisfiable, yielding a possible assignment of concrete values to symbols: $X_1=0$, $N=1$, $X_2=1$, $M=0$, and $X_3=1$. Thus, the corresponding concrete counter-example will be: $run \cdot q^n \cdot 1^n \cdot q^n \cdot 0^n \cdot run^{abort} \cdot done^{abort} \cdot done$. This play is present in the concrete model of the term $A \wedge B$ in Fig. 1a. Similarly, concrete counter-examples for terms $A \wedge \neg B$ and $\neg A \wedge B$ can be generated; and it can be verified that the term $\neg A \wedge \neg B$ is safe. □

Remark. The semantics $[\![\Gamma \vdash M : T]\!]$ contains only convergent behaviours of M, and so it is suitable for verifying safety (partial correctness) properties. This means that any non-terminating (divergent) term is considered as safe, since its game semantics is an empty set of complete plays. For example, $[\![abort : \mathsf{com} \vdash abort ; \mathsf{diverge}]\!] = \emptyset$. If we want to take into account liveness (total correctness) properties as well, then the semantics must be enriched to contain also possible divergent behaviours of terms (see [6,20] for details).

5. Symbolic game models for program families

We now discuss how the symbolic game models obtained in Section 4 can be effectively lifted to work on the level of program families from $\overline{\text{IA}_2}$. We extend the definition of guarded alphabet $\mathcal{A}^{gu}$ as the set of triples of feature expressions, boolean conditions and symbolic letters:

$$\mathcal{A}^{gu+f} = \{[\phi, b, \alpha\rangle \mid \phi \in FeatExp(\mathbb{F}), b \in BExp, \alpha \in \mathcal{A}^{sym}\}$$

The meaning of a guarded letter $[\phi, b, \alpha\rangle$ is that α is triggered only if b evaluates to true in valid configurations $k \in \mathbb{K}$ that satisfy ϕ. That is, every letter is *labelled* with a feature expression that defines variants able to perform the letter. As before, we write only α for $[tt, tt, \alpha\rangle$. A word $[\phi_1, b_1, \alpha_1\rangle \cdot [\phi_2, b_2, \alpha_2\rangle \ldots [\phi_n, b_n, \alpha_n\rangle$ over $\mathcal{A}^{gu+f}$ can be written as a triple $[\phi_1 \wedge \ldots \wedge \phi_n, b_1 \wedge \ldots \wedge b_n, \alpha_1 \cdot \ldots \cdot \alpha_n\rangle$. Its meaning is that the word $\alpha_1 \cdot \ldots \cdot \alpha_n$ is feasible only if the condition $b_1 \wedge \ldots \wedge b_n$ is satisfiable, and only for valid configurations that satisfy $\phi_1 \wedge \ldots \wedge \phi_n$. Regular languages and automata defined over $\mathcal{A}^{gu+f}$ are called *featured symbolic*, and they are specified using extended regular expressions R defined over finite $\mathcal{A}^{gu+f}$. All operations defining regular expressions over $\mathcal{A}^{gu}$ can be easily extended. We give the extended definition of the composition operator.

$$R' \circ R = \{w\big[[\phi \wedge \phi_1 \wedge \phi_2 \wedge \phi_1' \wedge \phi_2', b \wedge b_1 \wedge b_2 \wedge b_1' \wedge b_2' \wedge cond, s\rangle^{\langle 1 \rangle} / \\ [\phi_1, b_1, \alpha_1\rangle^{\langle 2 \rangle} \cdot [\phi_2, b_2, \alpha_2\rangle^{\langle 2 \rangle}\big] \mid w \in R, [\phi_1', b_1', \alpha_1'\rangle^{\langle 2 \rangle} \cdot [\phi, b, s\rangle^{\langle 1 \rangle} \cdot [\phi_2', b_2', \alpha_2'\rangle^{\langle 2 \rangle} \in R'\}$$

Feature expressions associated with the shared moves are conjoined, and the obtained conjunction is propagated in the resulting composition.

We can straightforwardly extend the symbolic representation of all IA$_2$ terms in the new setting by extending all guarded letters with the value tt for the first (feature expression) component. Now, we are ready to give representation of the compile-time conditional term:

$$[\![\Gamma \vdash_{\mathbb{F}} \texttt{\#if}\ \phi\ \texttt{then}\ M\ \texttt{else}\ M']\!] = [\![\Gamma \vdash M]\!] \circ [\![\Gamma \vdash M']\!] \circ [\![\texttt{\#if}]\!]^{\phi} \tag{2}$$

where $\circ$ is the composition operator, and the interpretation of the compile-time conditional construct parameterized by the feature expression ϕ is:

$$[\![\texttt{\#if} : \mathsf{com}^{\langle 1 \rangle} \times \mathsf{com}^{\langle 2 \rangle} \to \mathsf{com}]\!]^{\phi} = \\ run \cdot \big([\phi, tt, run^{\langle 1 \rangle}\rangle \cdot done^{\langle 1 \rangle} + [\neg\phi, tt, run^{\langle 2 \rangle}\rangle \cdot done^{\langle 2 \rangle}\big) \cdot done$$

That is, the first argument of `#if` is run for those variants that satisfy ϕ, whereas the second argument of `#if` is run for variants satisfying $\neg\phi$.

Let us compare definitions of $[\![\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ M']\!]$ given in Eq. (1) and $[\![\Gamma \vdash_{\mathbb{F}} \texttt{\#if}\ \phi\ \texttt{then}\ M\ \texttt{else}\ M']\!]$ given in Eq. (2). We can see that in the case of if, the condition B is first evaluated and depending on its (symbolic) value Z, we subsequently evaluate either M (if Z is true) or M' (if Z is false). In the case of `#if`, the presence condition ϕ is not evaluated at all, but ϕ is propagated directly in the resulting model. More specifically, we associate ϕ with the moves of M, whereas we associate $\neg\phi$ with the moves of M'.

Again, the effective alphabet of $[\![\Gamma \vdash_{\mathbb{F}} M : T]\!]$ for any $\overline{\text{IA}_2}$ term is a finite subset of $\mathcal{A}^{gu+f}_{[\![\Gamma \vdash_{\mathbb{F}} T]\!]}$. Therefore, for any $\overline{\text{IA}_2}$ term, the set $[\![\Gamma \vdash_{\mathbb{F}} M : T]\!]$ is a (featured symbolic) regular-language without infinite summations defined over its effective finite alphabet. Similarly as for IA$_2$ (see Theorem 1, for more details on the construction of the automaton the reader is referred to our earlier work [11]), the automaton corresponding to $[\![\Gamma \vdash_{\mathbb{F}} M : T]\!]$ is effectively constructible, and we call it *featured symbolic automaton* (FSA). Basically, an FSA is an SA augmented with transitions labelled (guarded) with feature expressions.

We denote it as $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] = (Q, i, \delta, F)$, where Q is a set of states, i is the initial state, δ is a transition function, and $F \subseteq Q$ is the set of accepting (final) states. The *purpose of an FSA* is to model behaviours (computations) of the entire program family and *link* each computation to the exact set of variants able to execute it. From an FSA, we can obtain the model of one particular variant through *projection*. This transformation is entirely syntactical and consists in removing all transitions (moves) linked to feature expressions that are not satisfied by a configuration $k \in \mathbb{K}$.

Definition 6. The projection of $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] = (Q, i, \delta, F)$ to a configuration $k \in \mathbb{K}$, denoted as $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] \mid_k$, is the symbolic automaton $A = (Q', i, \delta', F)$, where we keep transitions admitted by the configuration k: $\{(q_1, [b, a\rangle, q_2) \mid (q_1, [\phi, b, a\rangle, q_2) \in \delta \wedge k \models \phi\}$. All other transitions are removed, as well as unreachable states and transitions.

Theorem 7 *(Correctness).* $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] \mid_k = \mathcal{A}[\![\Gamma \vdash \pi_k(M) : T]\!]$.

Proof. By induction on the structure of M. Apart from `#if`-term, for all other terms the proof is immediate from definitions of $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!]$, $\mathcal{A}[\![\Gamma \vdash M : T]\!]$ and π_k.

Consider the case of `#if` ϕ `then` M `else` M'.

Let $k \in \mathbb{K}$. Then by definition of π_k, $\mathcal{A}[\![\Gamma \vdash \pi_k(\texttt{#if } \phi \texttt{ then } M \texttt{ else } M')]\!]$ represents either the automaton for $\mathcal{A}[\![\Gamma \vdash \pi_k(M)]\!]$ if $k \models \phi$, or the automaton for $\mathcal{A}[\![\Gamma \vdash \pi_k(M')]\!]$ if $k \models \neg\phi$. On the other hand, consider $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} \texttt{#if } \phi \texttt{ then } M \texttt{ else } M']\!] \mid_k$. If $k \models \phi$, all moves associated with M' (those tagged with $\langle 2 \rangle$) will be discarded since they contain $\neg\phi$ in the first component and $k \not\models \neg\phi$. Thus, we obtain $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} \texttt{#if } \phi \texttt{ then } M \texttt{ else } M']\!] \mid_k = \mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k$ if $k \models \phi$. By IH, we have $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k = \mathcal{A}[\![\Gamma \vdash \pi_k(M)]\!]$. Similarly, we obtain $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} \texttt{#if } \phi \texttt{ then } M \texttt{ else } M']\!] \mid_k = \mathcal{A}[\![\Gamma \vdash \pi_k(M')]\!]$ if $k \models \neg\phi$. It follows that $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} \texttt{#if } \phi \texttt{ then } M \texttt{ else } M']\!] \mid_k = \mathcal{A}[\![\Gamma \vdash \pi_k(\texttt{#if } \phi \texttt{ then } M \texttt{ else } M')]\!]$. □

Example 8. Consider the term family M from Introduction. Its FSA is given in Fig. 3. Letters represent triples, where the first component indicates which valid configurations can enable the corresponding move. For example, we can see that the unsafe symbolic play obtained after instantiating its input symbols with fresh names:

$$[tt, X_1 = 0, run\rangle \cdot [A, tt, q^n\rangle \cdot N^n \cdot [B, X_2 = X_1 + N, q^n\rangle \cdot M^n \cdot$$
$$[tt, X_3 = X_2 - M \wedge X_3 = 1, run^{abort}\rangle \cdot done^{abort} \cdot done$$

is feasible when $N = 1$ and $M = 0$ for the configuration $A \wedge B$, yielding the following concrete counterexample for $A \wedge B$: $run \cdot q^n \cdot 1^n \cdot q^n \cdot 0^n \cdot run^{abort} \cdot done^{abort} \cdot done$. □

6. Family-based model checking algorithms

The *general* family-based model checking problem for program families consists in determining which variants in the family are safe, and which are not safe. The goal is also to report a counter-example for each unsafe variant. Of course, one counter-example may be shared by several variants.

A straightforward but rather naive brute-force approach to solve the above problem is to check all valid variants individually. That is, compute the projection of each valid configuration, generate its model, and verify it using standard algorithms. This approach is very inefficient or even infeasible for large program families with huge configuration spaces. Indeed, all variants (in the worst case exponentially many in the number of features) will be explored in spite of their great similarity. We now propose an alternative algorithm, which explores the set of reachable states in the FSA of a program family rather than the individual models of all its variants. We aim to take advantage of the compact structure of FSAs in order to solve the above general family-based model checking problem.

Safe reachability in FSAs. A model checker is meant to perform a search in the state space of the FSA $\mathcal{F} = (Q, i, \delta, F)$ and to indicate safe and unsafe variants. This boils down to checking if an 'unsafe' state $q' \in Q$ with $q \xrightarrow{[\phi, b, run^{abort}\rangle} q'$ is reachable in the FSA. All states reachable from an unsafe state are also unsafe. We say that a state $q \in Q$ is 'safe' if it is not unsafe.

We define a *reachability relation* of an FSA $\mathcal{F}$ to be a function $R : Q \to \mathcal{P}(\mathbb{K})$, such that for all $q \in Q$ and $k \in R(q)$, q is a state reachable in $\mathcal{F} \mid_k$. Computing R efficiently is the key of our model checking algorithms. Initially, the reachable state is $Init = \{(i, \mathbb{K})\}$. The successors of a state $q \in Q$ reachable by variants in $px \in \mathcal{P}(\mathbb{K})$ are

$$Post(q, px) = \{(q', px') \mid q \xrightarrow{[\phi, b, \alpha\rangle} q' \in \delta \wedge px' = \{k \in px \mid k \models \phi\}\}$$

We can now extend the definition of $Post$ for reachability relations as $Post(R)$ is $\bigcup_{q \in dom(R)} Post(q, R(q))$. The reachability relation of an FSA, R, is given by a least fixed point:

$$R = \mu X. Init \cup Post(X) \tag{3}$$

The least fixed point R can be computed via the Kleene's fixed point theorem by repeated application of $Post$ to $Init$.

To check safety of an FSA, a new reachability relation R_{safe} is computed where only the successors of safe states are considered. That is,

```
Input: $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] = (Q, i, \delta, F)$ and valid configurations $\mathbb{K}$
Output: true if $M$ is safe; else false plus a set of counter-examples
1. $R := \{(i, \mathbb{K})\}$
2. $Queue := [(i, \mathbb{K})]$
3. $unsafe := \emptyset$, $\mathbb{K}_{unsafe} := \emptyset$
4. while $(Queue \neq [])$ do
5.    $(q, px) := remove(Queue)$, where $px \not\subseteq \mathbb{K}_{unsafe}$
6.    if ($q$ is UNSAFE) then
7.        $(e, px') := complete(q, px, trace(q))$
8.        $unsafe := unsafe \cup \{(e, px')\}$
9.        $\mathbb{K}_{unsafe} := \mathbb{K}_{unsafe} \cup px'$
10.   else
11.       $new := \Big\{(q', px' \backslash R(q')) \,|\, q \xrightarrow{[\phi,b,m\rangle} q', px' = \{k \in px \,|\, k \models \phi, k \notin \mathbb{K}_{unsafe}\}, px' \backslash R(q') \neq \emptyset\Big\}$
12.       while $(new \neq [])$ do
13.           Remove $(q', px') \in new$
14.           $R(q') := R(q') \cup px'$
15.           $put((q', px'), Queue)\}$
16.       end_while
17.   end_if
18. end_while
19. return $(unsafe = \emptyset)$, $unsafe$
```

Fig. 4. Family-based $BFS(\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!], \mathbb{K})$.

$$R_{safe} : Init_{safe} = \{(i, \mathbb{K})\},\ Post_{safe}(R_{safe}) = \bigcup_{\substack{q \in dom(R) \\ q \text{ is safe}}} Post(q, R_{safe}(q)) \tag{4}$$

Once an unsafe state q' is found by R_{safe}, a new reachability relation $R_{q'}$ is computed, defined as:

$$R_{q'} : Init_{q'} = \{(q', R_{safe}(q'))\},\ Post_{q'} = Post \tag{5}$$

The violating variants are given either by $R_{q'}(f)$ if f is an accepting state reached from q', or an empty set if no such accepting state can be reached.

Family-based breadth-first search. We use a family-based variant of Breadth-First Search (BFS) to compute R_{safe}. This algorithm serves as the basis for the subsequent verification procedure. BFS encounters all states in the FSA that are reachable from the initial state and checks whether one of them is 'unsafe'. In this way, the BFS finds the shortest unsafe symbolic play (run^{abort} occurs earliest in it) for any variant. The algorithm is shown in Fig. 4. It maintains: a *reachability relation* $R \subseteq Q \times \mathcal{P}(\mathbb{K})$ that stores a set of pairs (q, px) where q is marked as a visited state for the variants from $px \subseteq \mathbb{K}$; a *queue* $Queue$ that keeps track of all states that still have to be visited (explored); and a *set of counterexamples* $unsafe$. R is first initialized by the initial state $i \in Q$ that is reachable for all valid variants, i.e. $(i, \mathbb{K}) \in R$. For $(q, px) \in R$, we write $R(q)$ for the set of variants px. $Queue$ supports two operations: $remove$ which returns and deletes the first element of $Queue$, and put which inserts a new element at the back of $Queue$. In $Queue$ along with each state q, we store a trace, $trace(q)$, that shows how q is reached from the initial state i. For each visited state, it is checked whether that state is unsafe (line 6). Each time an unsafe state q is reached, the pair $(e, px') = complete(q, px, trace(q))$ is added to $unsafe$ where: "e" is an unsafe counter-example, which is a complete play that starts in the initial state and ends in an accepting state; and "px'" is the corresponding set of unsafe variants. The play "e" is generated by looking at the trace kept on $Queue$ along with q, that is $trace(q)$, and by finding the shortest trace from q to an accepting state by performing an embedded BFS. Note that if there exists no such trace from q to an accepting state, then we have not found an unsafe counter-example and we proceed with the search. At each iteration, the BFS calculates the set new of unvisited successors of the current safe state, filtering out states and variants that are already visited in R (line 11). Assuming that we have a transition $q \xrightarrow{[\phi,b,\alpha\rangle} q'$ and the source state q is reachable by variants in px, the target state q' is reachable for variants in $\{k \in px \mid k \models \phi\}$. If all successors were visited (new is empty), the algorithm backtracks (line 5). Given an FSA as input, the algorithm in Fig. 4 calculates all safe reachable states from the initial state i. When the search finishes and $unsafe$ is empty, the algorithm returns $true$; otherwise it returns $false$ and the set $unsafe$.

Compared to the standard BFS, where visited states are marked with Boolean $visited$ flags and no state is visited twice, in our algorithm visited states are marked with sets of variants (for which those states are visited) and a state can be visited multiple times. This is due to the fact that when the BFS arrives at a state s for the second time, such that $R(q) = px$, $(q, px') \in new$, and $px' \not\subseteq px$, then s although already visited, has to be re-explored since transitions that were disallowed for px during the first visit of q might be now allowed for px'.

An important *optimization* of the above algorithm is the possibility to ignore variants that are already known to be unsafe. Given the set of counter-examples maintained by the algorithm $unsafe$, the set of unsafe variants is $\mathbb{K}_{unsafe} = \cup_{(e,px) \in unsafe} px$. Any state q with variants $px \subseteq \mathbb{K}_{unsafe}$ can be ignored, since any violation found due to the exploration of q would be for variants that are already known to be unsafe. In the BFS, this can be achieved by filtering out states with variants $px \subseteq \mathbb{K}_{unsafe}$ as part of the calculation of new (line 11). This can only eliminate newly discovered states, not those

The procedure checks safety of a given term family $\Gamma \vdash_{\mathbb{F}} M : T$.

1 $BFS(\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!], \mathbb{K})$ from Fig. 4 is called.
2 If no unsafe play is found, terminate with answer SAFE for all variants in $\mathbb{K}$.
3 Otherwise, find $\mathbb{K}_{unsafe} = \cup_{(e,px) \in unsafe} px$. For all variants in $\mathbb{K} \backslash \mathbb{K}_{unsafe}$ report that are SAFE. The mechanism for fresh symbol generation is used to instantiate all input symbols in the unsafe symbolic plays e, and their play conditions are tested for consistency.
4 If the condition of some play e is consistent, report the corresponding variants px as UNSAFE with the concrete counter-example instantiated from e. Otherwise, let $E = \bigcup_{\substack{(e,\,px) \in unsafe \\ e \text{ inconsistent}}} \{e\}$ and $\mathbb{K}' = \bigcup_{\substack{(e,\,px) \in unsafe \\ e \text{ inconsistent}}} px$. We discard inconsistent unsafe plays from $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!]$ thus generating the model $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!]' = \mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \backslash E$, and generate $\mathbb{K}' \subseteq \mathbb{K}_{unsafe}$ that contains all variants associated with those inconsistent unsafe plays. Then call $BFS(\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!]', \mathbb{K}')$ and go to Step 2.

Fig. 5. Verification procedure (VP).

that are already on the queue. States on the queue can be filtered out by ignoring elements (q, px) for which $px \subseteq \mathbb{K}_{unsafe}$ without further exploring them (line 5). Hence, we only explore those (q, px) from $Queue$ for which $px \not\subseteq \mathbb{K}_{unsafe}$.

Theorem 9 *(Correctness of BFS). The reachability relation R computed by the family-based BFS is equivalent to R_{safe} in Eq. (4). The reachability relation computed by the embedded BFS 'complete' that starts at an unsafe state q' is equivalent to $R_{q'}$ in Eq. (5).*

Proof. Initially, R contains $Init = \{(i, \mathbb{K})\}$, which is equivalent to the initial iteration of R_{safe}, that is $Post^0_{safe}(Init)$. In every iteration of BFS, we perform an equivalent operation to $Post_{safe}(q, px)$ by adding all successors of the pair (q, px) for q safe that is already in R and unexplored so far. BFS terminates when all possible pairs (q, px) for q safe are explored, which corresponds to the least fixed point R_{safe}.

The embedded BFS '*complete*' starts with $Init_{q'} = \{(q', R_{safe}(q'))\}$, where q' is an unsafe state found by R_{safe}. It explores all successor states without any restrictions, that is using $Post$. If an accepting state f is reached, then the set of violating variants $px' = R_{q'}(f)$ is reported. □

Verification procedure. The complete verification procedure for checking safety of term families is described in Fig. 5. In each iteration, it calls the BFS from Fig. 4 and finds some safe variants, unsafe variants for which consistent (genuine) counter-examples are reported, and variants for which inconsistent (spurious) counter-examples are found. To prevent the model checker to consider again those variants for which conclusive results are previously found (either safe or unsafe with genuine counter-examples), the BFS in the next iteration is called with updated arguments, i.e. we consider a model where all found inconsistent traces are discarded and we take into account only those configurations with no conclusive results. Note that the updated model where all inconsistent unsafe traces are discarded is implemented by maintaining a list of found inconsistent traces in the original model. Whenever the called BFS finds an unsafe trace that belongs to the above list, the found trace is ignored and the search proceeds.

We first show that the projection π_k commutes with our "lifted" verification procedure, which is applied directly at the level of program families.

Theorem 10 *(Correctness). $\Gamma \vdash \pi_k(M)$ is safe, if and only if, $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k$ is safe.*

Proof. By induction on the structure of M. For all cases except the `#if`-term, the proof is immediate corollary from Proposition 3, Theorem 4, and Theorem 7.

Consider the case `#if` ϕ `then` M `else` M'. $\Gamma \vdash \pi_k($`#if` ϕ `then` M `else` $M')$ is safe iff either $\Gamma \vdash \pi_k(M)$ is safe when $k \models \phi$, or $\Gamma \vdash \pi_k(M')$ is safe when $k \not\models \phi$. By IH, we have $\Gamma \vdash \pi_k(M)$ is safe iff $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k$ is safe, and $\Gamma \vdash \pi_k(M')$ is safe iff $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M']\!] \mid_k$ is safe. By Definition 6 and similar arguments as in the proof of Theorem 7, we have $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}}$ `#if` ϕ `then` M `else` $M']\!] \mid_k$ is equal either to $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k$ if $k \models \phi$, or to $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M']\!] \mid_k$ if $k \not\models \phi$. Thus, we obtain $\Gamma \vdash \pi_k($`#if` ϕ `then` M `else` $M')$ is safe iff $\mathcal{F}[\![\Gamma \vdash_{\mathbb{F}}$ `#if` ϕ `then` M `else` $M']\!] \mid_k$ is safe. □

Under assumption that all constraints generated from a term family can be handled by an underlying SMT solver, we show that the VP in Fig. 5 is correct and semi-terminating.

Theorem 11. *The VP is correct and terminates for unsafe term families.*

Proof. As a corollary of Theorems 4, 7, 10 we obtain that the VP returns correct answers for all variants from $\mathbb{K}$.

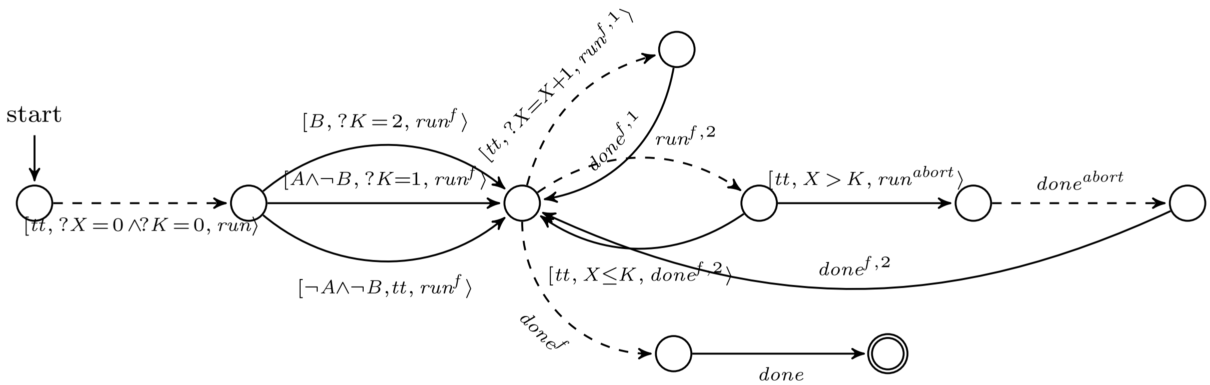
start
$[tt,\, ?X=0 \wedge ?K=0,\, run\rangle$
$[B,\, ?K=2,\, run^f\rangle$
$[A\wedge\neg B,\, ?K{=}1,\, run^f\rangle$
$[\neg A\wedge\neg B, tt,\, run^f\rangle$
$[tt,\, ?X{=}X{+}1,\, run^{f,1}\rangle$
$done^{f,1}$
$run^{f,2}$
$[tt,\, X>K,\, run^{abort}\rangle$
$done^{abort}$
$[tt,\, X\leq K,\, done^{f,2}\rangle$
$done^{f,2}$
$done^f$
$done$

Fig. 6. The FSA for `Proc2` family.

Let $t \in \mathcal{F}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k$ be the shortest unsafe play for the variant $k \in \mathbb{K}$. The VP uses BFS to locate the shortest unsafe plays for all unsafe variants from $\mathbb{K}$ (some of them may be inconsistent in general). The VP will find t after finite number of calls to the BFS, that will first find all inconsistent unsafe plays shorter than t (which must be finitely many [11]: the number of traces with the same length is finite in a finite-state automaton). □

However, the VP may diverge for safe variants, producing in each next iteration longer and longer unsafe symbolic plays with inconsistent conditions. For example, the term

$$\mathsf{new}_{\mathsf{int}}\, x := 0\, \mathsf{in\ while}\, (x < x)\, \mathsf{do}\, \{x := x+1;\, abort\}$$

is safe since the guard of while is always false. Yet, our VP will diverge producing unsafe inconsistent symbolic plays in each iteration.

7. Implementation

We have extended the prototype tool developed in [11] to implement the VP in Fig. 5. That tool converts any single IA_2 term into a symbolic automaton representing the associated game semantics, and then explores the automaton for (consistent) unsafe symbolic plays. The extended tool takes as input a term family, and generates the corresponding FSA, which is then explored based on the VP described in Fig. 5. The tool is implemented in Java along with its own library for working with featured symbolic automata. We have also implemented algorithms for elimination of (empty) ϵ-letters and for removing all unreachable states in the setting of featured symbolic automata. The tool calls an external SMT solver Yices [7] to determine consistency of play conditions. We now illustrate our tool with several examples. The tool, further examples and detailed reports how they execute are available from: `https://aleksdimovski.github.io/symbolicgc.html` (version for program families).

7.1. A procedural term family

Consider the term family `Proc2`:

$$\begin{aligned}
&f : \mathsf{com}^{f,1} \to \mathsf{com}^{f,2} \to \mathsf{com}^{f},\ abort : \mathsf{com}^{abort} \vdash_{\{A,B\}}\\
&\quad \mathsf{new}_{int}\, x := 0\ \mathsf{in\ new}_{int}\, k := 0\ \mathsf{in}\\
&\quad \texttt{\#if}\,(A)\ \mathsf{then}\, k := 1;\\
&\quad \texttt{\#if}\,(B)\ \mathsf{then}\, k := 2;\\
&\quad f\big(x := x+1,\ \mathsf{if}\,(x > k)\ \mathsf{then}\, abort\ \mathsf{else\ skip}\big)\ : \mathsf{com}
\end{aligned}$$

where x and k are two local variables, and f is an undefined procedure with two command arguments. There are two features, A and B, which are used for suitable initialization of k. A variant derived from the above family is not safe, when f calls its first argument $k+1$ times and then f calls its second argument.

The symbolic model for this term family is given in Fig. 6. The local variable k is initialized to 0 for the variant $\neg A \wedge \neg B$, k is 1 for $A \wedge \neg B$, and k is initialized to 2 for variants satisfying B. The undefined procedure f may call its two arguments, zero or more times, in any order and then it terminates successfully. If the first argument of f is called the variable x is increased by one, whereas if the second argument of f is called then $abort$ is run when the current value of x is bigger

[7] http://yices.csl.sri.com.

than k. In the first iteration, the VP reports the shortest counter-example corresponding to the execution when f calls its second argument:

$$[tt, ?X{=}0 \wedge ?K{=}0, run\rangle \cdot [\neg A \wedge \neg B, tt, run^{f}\rangle \cdot run^{f,2} \cdot [tt, X > K, run^{abort}\rangle \cdot \\ done^{abort} \cdot done^{f,2} \cdot done^{f} \cdot done$$

However, its play condition "$X{=}0 \wedge K{=}0 \wedge X{>}K$" is inconsistent, so this play is discarded. In the second iteration, we obtain a counter-example for the variant $\neg A \wedge \neg B$, where f calls its first argument once, and then its second argument

$$[tt, ?X{=}0 \wedge ?K{=}0, run\rangle \cdot [\neg A \wedge \neg B, tt, run^{f}\rangle \cdot [tt, ?X{=}X{+}1, run^{f,1}\rangle \cdot \\ done^{f,1} \cdot run^{f,2} \cdot [tt, X > K, run^{abort}\rangle \cdot done^{abort} \cdot done^{f,2} \cdot done^{f} \cdot done$$

This counterexample is genuine for $X_1{=}0$, $K{=}0$, $X_2{=}1$. In the following iterations, we also obtain genuine counterexamples for the variant $A \wedge \neg B$ (resp., B), where f calls its first argument two (resp., three) times before calling its second argument.

7.2. A linear search term family

Consider the following version of the linear search algorithm, $\texttt{Linear}_3$:

```
$x[k] : \mathsf{var\,int}^{x[-]},\ y : \mathsf{exp\,int}^{y},\ abort : \mathsf{com}^{abort} \vdash_{\{A,B,C\}}$
    $\mathsf{new}_{int}\ i := 0\ \mathsf{in}\ \mathsf{new}_{int}\ j := 0\ \mathsf{in}$
    $\texttt{\#if}\,(A)\ \mathsf{then}\ j := j + 1;$
    $\texttt{\#if}\,(B)\ \mathsf{then}\ j := j - 1;$
    $\texttt{\#if}\,(C)\ \mathsf{then}\ j := j + 2;$
    $\mathsf{new}_{int}\ p := y\ \mathsf{in}$
    $\mathsf{while}\ (i < k)\ \mathsf{do}\ \{$
        $\mathsf{if}\ (x[i] = p)\ \mathsf{then}\ \{\mathsf{if}\ (j = 1)\ \mathsf{then}\ abort\ \mathsf{else}\ j := j - 1;\ \}$
        $i := i + 1$
    $\}\ : \mathsf{com}$
```

The term family contains three features A, B, and C, and hence 8 variants can be produced. In the above, first depending on which features are enabled some value from -1 to 3 is assigned to j, and then the input expression y is copied into the local variable p. The non-local array x is searched for an occurrence of the value stored in p. If the search succeeds j-times (for $j > 0$), *abort* is executed. So various variants correspond to terms with different initial values of j, ranging from -1 to 3.

The arrays are implemented in the symbolic representation by using a special symbol (e.g., k with an initial constraint $k > 0$) to represent the length of an array. A new symbol (e.g., I) is also used to represent the index of the array element that needs to be de-referenced or assigned to (see [11] for details). If we also want to check for array-out-of-bounds errors, we can implicitly include in the representation of arrays plays that perform *abort* moves when $I \geq k$. For simplicity, we do not take into consideration the array-out-of-bounds errors in this example.

The FSA for the above term family is shown in Fig. 7. If the value read from the environment for y occurs j-times (for $j > 0$) in the array x, then an unsafe behaviour is reported. Hence, all variants for which the value assigned to j is less than 1 are safe: $\neg A \wedge \neg B \wedge \neg C$, $\neg A \wedge B \wedge \neg C$, and $A \wedge B \wedge \neg C$. All other variants are unsafe. For example, for variants $A \wedge B \wedge C$ and $\neg A \wedge \neg B \wedge C$ (for which j is set to 2) the tool reports a counter-example that corresponds to a term with an array of size $k = 2$, where the values read from the environment for $x[0]$, $x[1]$, and y are the same, i.e. the following counter-example is generated: $run \cdot q^{y} \cdot 0^{y} \cdot read^{x[0]} \cdot 0^{x[0]} \cdot read^{x[1]} \cdot 0^{x[1]} \cdot run^{abort} \cdot done^{abort} \cdot done$. This counter-example is obtained after two iterations of the VP, and it corresponds to a computation which runs the body of 'while' two times. In the first iteration of the VP, an inconsistent unsafe symbolic play is found whose play condition contains $J = 2 \wedge J = 1$, where the symbol J tracks the current value of the variable j. Here, the first constraint $J = 2$ corresponds to the initialization of j, whereas the second constraint $J = 1$ corresponds to evaluating the guard of if that leads to running *abort*. A consistent genuine counter-example is obtained in the first iteration for products $A \wedge \neg B \wedge \neg C$ and $\neg A \wedge B \wedge C$ (for which j is 1), whereas for $A \wedge \neg B \wedge C$ (j is assigned to 3) in the third iteration. The tool diverges for safe terms, producing longer and longer inconsistent unsafe symbolic plays in each next iteration.

7.3. Evaluation

We ran our tool on a 64-bit Intel®Core^{TM} i5 CPU and 8 GB memory. All times are reported as averages over five independent executions. For our experiments, we use six term families: $\texttt{Intro}$ is the family from Table 1 in Section 2; $\texttt{Proc}_2$ is the procedural term family from Section 7.1; $\texttt{Proc}_3$ is an extended version of $\texttt{Proc}_2$ with one more feature C and initialization command $\texttt{\#if}\,(C)\,\mathsf{then}\,k := 3$; $\texttt{Linear}_3$ is the linear search term family from Section 7.2; $\texttt{Linear}_4$ is an extended version of $\texttt{Linear}_3$ with one more feature D and command: $\texttt{\#if}\,(D)\,\mathsf{then}\,j := j + 3$; and $\texttt{Linear}_5$ is $\texttt{Linear}_4$ extended with one additional feature E and command: $\texttt{\#if}\,(E)\,\mathsf{then}\,j := j - 2$. We restrict our tool to work with bounded

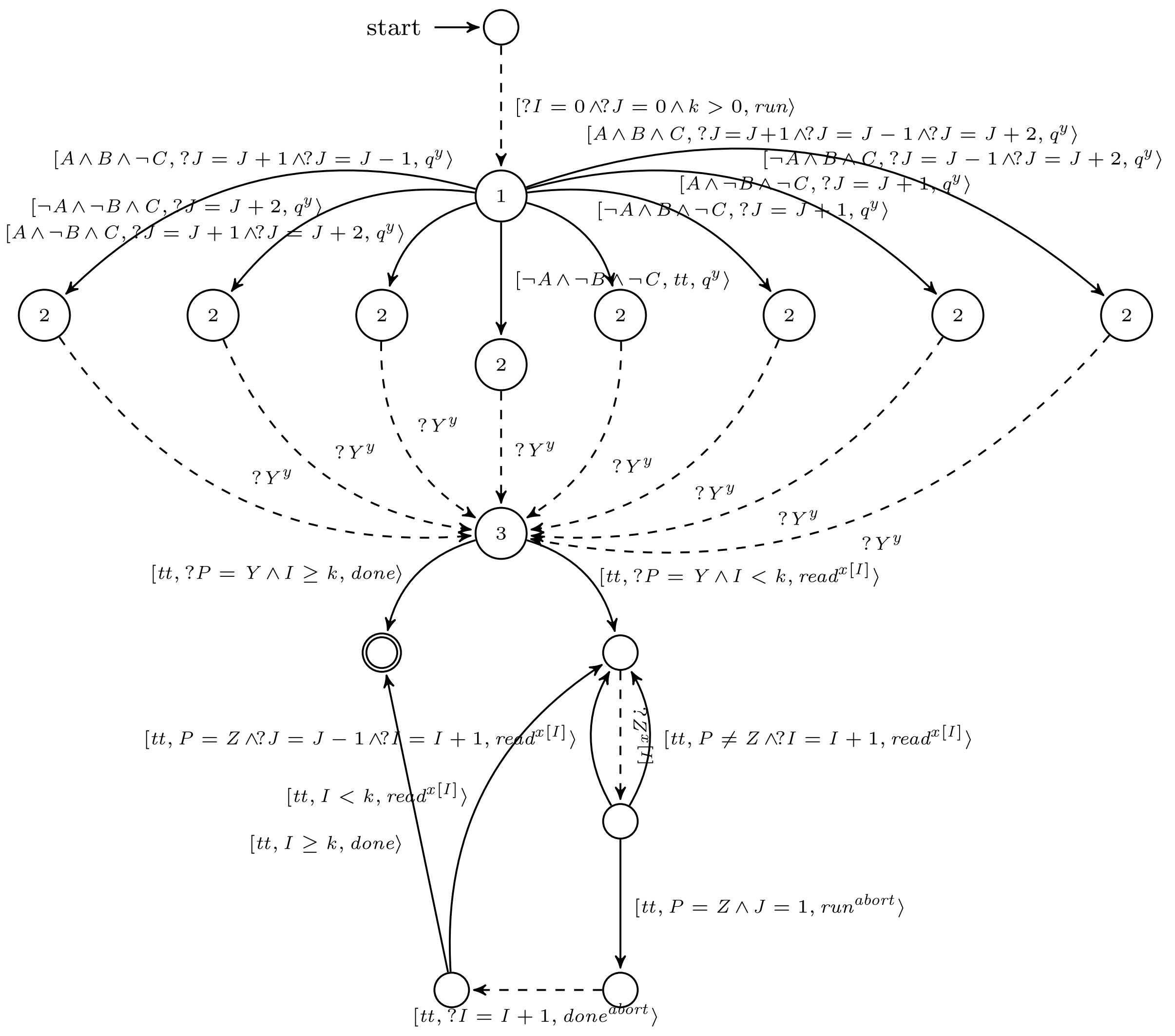
start
$[?I = 0 \wedge ?J = 0 \wedge k > 0, run\rangle$
$[A \wedge B \wedge C, ?J = J+1 \wedge ?J = J - 1 \wedge ?J = J + 2, q^y\rangle$
$[\neg A \wedge B \wedge C, ?J = J - 1 \wedge ?J = J + 2, q^y\rangle$
$[A \wedge \neg B \wedge \neg C, ?J = J + 1, q^y\rangle$
$[\neg A \wedge B \wedge \neg C, ?J = J + 1, q^y\rangle$
$[A \wedge B \wedge \neg C, ?J = J + 1 \wedge ?J = J - 1, q^y\rangle$
$[\neg A \wedge \neg B \wedge C, ?J = J + 2, q^y\rangle$
$[A \wedge \neg B \wedge C, ?J = J + 1 \wedge ?J = J + 2, q^y\rangle$
$[\neg A \wedge \neg B \wedge \neg C, tt, q^y\rangle$
1
2
3
$?Y^y$
$[tt, ?P = Y \wedge I \geq k, done\rangle$
$[tt, ?P = Y \wedge I < k, read^{x[I]}\rangle$
$[tt, P = Z \wedge ?J = J - 1 \wedge ?I = I + 1, read^{x[I]}\rangle$
$?Z^{x[I]}$
$[tt, P \neq Z \wedge ?I = I + 1, read^{x[I]}\rangle$
$[tt, I < k, read^{x[I]}\rangle$
$[tt, I \geq k, done\rangle$
$[tt, P = Z \wedge J = 1, run^{abort}\rangle$
$[tt, ?I = I + 1, done^{abort}\rangle$

Fig. 7. FSA for the linear search family. Note that all states denoted as 2 are merged into only one state in the model, but we represent them separately here in order to aid in the readability of transition labels going from the state 1 to 2. Also, there is only one transition going from the state 2 to 3.

Bench.	$\lvert\mathbb{K}\rvert$	Family-based approach			Brute-force approach		
		Time	Max	Fin	Time	Max	Fin
`Intro`	4	0.34 s	25	9	0.78 s	68	28
`Proc2`	4	0.67 s	39	11	1.71 s	140	44
`Proc3`	8	1.86 s	43	11	3.91 s	280	88
`Linear3`	8	1.34 s	51	9	3.86 s	336	72
`Linear4`	16	2.34 s	57	9	7.28 s	720	144
`Linear5`	32	4.50 s	63	9	16.27 s	1482	288

Fig. 8. Performance comparison for verifying program families.

number of iterations (10 in this case) since the VP diverges for safe terms from the linear search family. All variants from `Proc2` and `Proc3` are unsafe, and the VP reports concrete counter-examples for each of them. For Linear$_4$ the tool reports 13 unsafe variants with corresponding counter-examples, whereas for Linear$_5$ 21 unsafe variants are found. Fig. 8 compares the effect (in terms of Time, the number of states in the maximal model generated during analysis Max, and the number of states in the final model Fin) of verifying benchmarks using our family-based approach vs. using brute-force approach. In the latter case, we first compute all variants, generate their models, and verify them one by one by using the tool for single programs [11]. In this case we report the sum of number of states for the corresponding models in all individual variants. We can see that the family-based approach is between 2.1 and 3.6 times faster (using considerably less space) than the brute-force approach. We expect even bigger efficiency gains for families with higher number of variants.

7.4. Discussion

We have studied our technique on various realistic and expressive, but still ultimately academic examples. In order to tackle larger real case studies from languages such as C, ML, OCaml enriched with `#ifdef`-s, we need to cope with other features as well: pointers, call-by-value, higher-order functions, etc. Still, game semantics based model checking is currently focused on compositionality and handling open program fragments with undefined identifiers. Compositionality is achieved in a clean and theoretically solid way and we believe that using compositional techniques are the best guarantee of scalability to very large systems. The current approach shows how to handle open program families with undefined identifiers by generating small models with maximum level of abstraction. Also, the main aim of this approach is to handle complexity stemming from the size of the configuration space, not only the size of the code. Evaluation results confirm that our approach achieves significant improvements for families with middle sized configuration spaces ($|\mathbb{K}|$ ranges between 4 and 32). It would be also interesting to see how this technique scales to larger families with much bigger configuration spaces.

8. Related work

Recently, many so-called lifted (family-based) techniques have been proposed, which lift existing single-program analysis techniques to work on the level of program families (see [2] for a survey). This includes family-based syntax checking [21], family-based type checking [22], family-based model checking [23,24], family-based static program analysis [25–27], etc. We divide our discussion of related work into two categories: family-based model checking, and game semantics based model checking.

Family-based model checking. Classen et al. [23] have proposed featured transition systems (FTSs) as the foundation for behavioural specification and verification of variational systems. An FTS, which is a variability-aware extension of the standard transition systems, superimposes the behaviour of all instances of a variational system in a single model. In FTSs, transitions are enriched with guard predicates over features that specify for which variants a transition is enabled. Therefore, the proposed family-based model checking algorithms for FTSs check an execution behaviour only once, no matter in how many variants it is enabled. The algorithms for verifying FTSs against LTL properties are implemented in the SNIP family-based model checker [24]. The input language to SNIP is fPromela, which is a variability-aware extension of the well-known SPIN's language Promela [28]. In order to make this approach more scalable, the works [29–31] propose applying abstractions on FTSs. In particular, a calculus of so-called variability abstractions is devised [29,30] for deriving abstract family-based model checking of FTSs. Such variability abstractions enable deliberate trading of precision for speed in family-based model checking by reducing the size of the configuration space. Dimovski and Wasowski [31] propose an automatic abstraction refinement procedure for family-based model checking, which works until a genuine counterexample is found or the property satisfaction is shown for all variants in the family. The application of variability abstractions for verifying real-time variational systems has been described in [32]. In this work, we also propose special family-based model checking algorithms. However, they are not applied on high-level models of variational systems (fPromela and FTSs), but on game semantics models extracted directly from concrete second-order program fragments with `#ifdef`-s. In simulator-based approaches for family-based software model checking [14–16], program families are first translated into single programs by transforming features into variables which are nondeterministically initialized. Then, an existing off-the-shelf single-program model checker is used to detect violations. This approach cannot report conclusive results for all variants since the model checker stops after a first violating variant is found. In contrast, we consider here specialized family-based model checking algorithms which report conclusive results for all variants in the family.

Game semantics based model checking. The first application of game semantics to model checking was proposed by Ghica and McCusker [8]. They show how game semantics of IA_2 with finite data types can be represented in a very simple form by regular-languages. A verification tool based on this regular-language representation was also developed [13]. Subsequently, several verification algorithms have been proposed for model checking IA_2 with infinite (data types) integers using game models [19,33,11]. The work [19] presents a counter-example guided abstraction refinement procedure, which iteratively refines data abstractions of all integer types in a given input program until either a genuine counterexample is found or safety of the program is shown. The work in [33] develops a predicate abstraction method for verification from game semantics models. It extends the game models by recording the state (store) explicitly in the model using so-called stateful plays. The state is then abstracted by a set of predicates giving rise to (predicate) abstracted plays and models. By using symbols instead of concrete data for inputs, it was shown [11] how to generate (finite) symbolic automata representation of game semantics and how the obtained automata can be used for verification by calling SMT solvers to determine satisfiability of play conditions. The symbolic game models are used for performing probabilistic analysis of open programs [34], where a model counting tool (Latte) is called, instead of an SMT solver, to determine the number of solutions to play conditions. The automata-theoretic representation of game semantics have been also extended to programs with various other features. Murawski and Walukiewicz [10] show that game semantics of third-order IA terms with finite data types can be represented using visibly pushdown automata, whereas Ghica and Murawski [35] give an algorithmic representation of (finite) parallel IA terms that contain shared-variable concurrency and binary semaphores. Algorithmic game semantics for nondeterministic programs is defined in [36], which is fully abstract with respect to two complementary notions of observational equivalence: may-termination and must termination. Algorithmic game semantics for probabilistic IA have been studied as well. In [37] probabilistic equivalence and refinement have been explored in the context of game semantics, whereas an automated equivalence checker for probabilistic IA_2 terms has been developed in [38]. An algorithmic study of program equivalence for an imperative object calculus called Interface Middleweight Java has been also introduced [39]. Game semantics for this language is captured using visibly pushdown fresh-register automata, and then the equivalence problem is reduced to the emptiness problem of the obtained automata. In this work, we describe an application of algorithmic game semantics for efficient verification of program families implemented with preprocessor `#ifdef`-s.

9. Conclusion

In this work we introduce the featured symbolic automata (FSA), a formalism designed to describe the combined game semantics models of a whole program family. A specifically designed model checking technique allows to verify safety of an FSA. Thereby, we can verify all variants of a family at once using a single compact model and pinpoint the unsafe variants. An evaluation shows that our approach outperforms the brute force approach where all variants are individually verified. The proposed approach can be extended to support multi-features (implemented by arrays) and numeric (non-Boolean) features. Another interesting direction for extension would be to apply the variability abstractions from [29,26] to define abstract family-based model checking in the context of game semantics models.

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison–Wesley, 2001.
[2] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (1) (2014) 6:1–6:45.
[3] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: 30th International Conference on Software Engineering, ICSE 2008, ACM, 2008, pp. 311–320.
[4] S. Abramsky, R. Jagadeesan, P. Malacaria, Full abstraction for PCF, Inform. Sci. 163 (2) (2000) 409–470.
[5] S. Abramsky, G. McCusker, Linearity, sharing and state: a fully abstract game semantics for idealized Algol with active expressions, Electron. Notes Theor. Comput. Sci. 3 (1996) 2–14.
[6] R. Harmer, G. McCusker, A fully abstract game semantics for finite nondeterminism, in: 14th Annual IEEE Symposium on Logic in Computer Science, LICS 1999, 1999, pp. 422–430.
[7] J.M.E. Hyland, C.L. Ong, On full abstraction for PCF: I, II, and III, Inform. Sci. 163 (2) (2000) 285–408.
[8] D.R. Ghica, G. McCusker, The regular-language semantics of second-order idealized Algol, Theoret. Comput. Sci. 309 (1–3) (2003) 469–502.
[9] A. Dimovski, R. Lazic, Compositional software verification based on game semantics and process algebra, STTT 9 (1) (2007) 37–51.
[10] A.S. Murawski, I. Walukiewicz, Third-order Idealized Algol with iteration is decidable, in: Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, in: LNCS, vol. 3441, Springer, 2005, pp. 202–218.
[11] A.S. Dimovski, Program verification using symbolic game semantics, Theoret. Comput. Sci. 560 (2014) 364–379.
[12] A.S. Dimovski, Symbolic game semantics for model checking program families, in: Model Checking Software – 23rd International Symposium, SPIN 2016, Proceedings, in: LNCS, vol. 9641, Springer, 2016, pp. 19–37.
[13] S. Abramsky, D.R. Ghica, A.S. Murawski, C.L. Ong, Applying game semantics to compositional software modeling and verification, in: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings, in: LNCS, vol. 2988, Springer, 2004, pp. 421–435.
[14] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: case studies and experiments, in: 35th International Conference on Software Engineering, ICSE '13, IEEE Computer Society, 2013, pp. 482–491.
[15] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, S. Apel, Variability encoding: from compile-time to load-time variability, J. Log. Algebraic Methods Program. 85 (1) (2016) 125–145.
[16] A.F. Iosif-Lazar, J. Melo, A.S. Dimovski, C. Brabrand, A. Wasowski, Effective analysis of C programs by rewriting variability, Program. J. 1 (1) (2017) 1.
[17] A. Dimovski, D.R. Ghica, R. Lazic, A counterexample-guided refinement tool for open procedural programs, in: Model Checking Software, 13th International SPIN Workshop, Proceedings, in: LNCS, vol. 3925, Springer, 2006, pp. 288–292.
[18] J.C. Reynolds, The essence of Algol, in: P.W. O'Hearn, R.D. Tennent (Eds.), Algol-Like Languages, Birkhaüser, 1997.
[19] A. Dimovski, D.R. Ghica, R. Lazic, Data-abstraction refinement: a game semantic approach, in: 12th International Symposium on Static Analysis, SAS '05, in: LNCS, vol. 3672, Springer, 2005, pp. 102–117.
[20] A. Dimovski, A compositional method for deciding equivalence and termination of nondeterministic programs, in: 8th International Conference on Integrated Formal Methods, IFM'10, in: LNCS, vol. 6396, Springer, 2010, pp. 121–135.
[21] C. Kästner, P.G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. Berger, Variability-aware parsing in the presence of lexical macros and conditional compilation, in: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, ACM, 2011, pp. 805–824.
[22] S. Chen, M. Erwig, E. Walkingshaw, An error-tolerant type system for variational lambda calculus, in: ACM SIGPLAN International Conference on Functional Programming, ICFP'12, ACM, 2012, pp. 29–40.
[23] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, J. Raskin, Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking, IEEE Trans. Softw. Eng. 39 (8) (2013) 1069–1089.
[24] A. Classen, M. Cordy, P. Heymans, A. Legay, P. Schobbens, Model checking software product lines with SNIP, STTT 14 (5) (2012) 589–612.
[25] J. Midtgaard, A.S. Dimovski, C. Brabrand, A. Wasowski, Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. 105 (2015) 145–170.
[26] A.S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions: trading precision for speed in family-based analyses, in: 29th European Conference on Object-Oriented Programming, ECOOP 2015, in: LIPIcs, vol. 37, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015, pp. 247–270.
[27] A. Dimovski, C. Brabrand, A. Wasowski, Finding suitable variability abstractions for family-based analysis, in: FM 2016: Formal Methods – 21st International Symposium, Proceedings, in: LNCS, vol. 9995, 2016, pp. 217–234.
[28] G.J. Holzmann, The SPIN Model Checker – Primer and Reference Manual, Addison–Wesley, 2004.

[29] A.S. Dimovski, A.S. Al-Sibahi, C. Brabrand, A. Wasowski, Family-based model checking without a family-based model checker, in: Model Checking Software – 22nd International Symposium, SPIN 2015, Proceedings, in: LNCS, vol. 9232, Springer, 2015, pp. 282–299.
[30] A.S. Dimovski, A.S. Al-Sibahi, C. Brabrand, A. Wasowski, Efficient family-based model checking via variability abstractions, STTT 19 (5) (2017) 585–603.
[31] A.S. Dimovski, A. Wasowski, Variability-specific abstraction refinement for family-based model checking, in: Fundamental Approaches to Software Engineering – 20th International Conference, FASE 2017, Proceedings, in: LNCS, vol. 10202, 2017, pp. 406–423.
[32] A.S. Dimovski, A. Wasowski, From transition systems to variability models and from lifted model checking back to UPPAAL, in: Models, Algorithms, Logics and Tools – Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, in: LNCS, vol. 10460, Springer, 2017, pp. 249–268.
[33] A. Bakewell, D.R. Ghica, Compositional predicate abstraction from game semantics, in: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Proceedings, 2009, pp. 62–76.
[34] A.S. Dimovski, Probabilistic analysis based on symbolic game semantics and model counting, in: Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, in: EPTCS, vol. 256, 2017, pp. 1–15.
[35] D.R. Ghica, A.S. Murawski, Compositional model extraction for higher-order concurrent programs, in: 12th International Conference TACAS 2006, in: LNCS, vol. 3920, Springer, 2006, pp. 303–317.
[36] A.S. Murawski, Reachability games and game semantics: comparing nondeterministic programs, in: Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, IEEE Computer Society, 2008, pp. 353–363.
[37] A.S. Murawski, J. Ouaknine, On probabilistic program equivalence and refinement, in: Concurrency Theory, 16th International Conference, CONCUR 2005, Proceedings, in: LNCS, vol. 3653, Springer, 2005, pp. 156–170.
[38] A. Legay, A.S. Murawski, J. Ouaknine, J. Worrell, On automated verification of probabilistic programs, in: 14th International Conference TACAS 2008, in: LNCS, vol. 4963, Springer, 2008, pp. 173–187.
[39] A.S. Murawski, S.J. Ramsay, N. Tzevelekos, Game semantic analysis of equivalence in IMJ, in: Automated Technology for Verification and Analysis – 13th International Symposium, ATVA 2015, Proceedings, in: LNCS, vol. 9364, Springer, 2015, pp. 411–428.