Symbolic Game Semantics for Model Checking Program Families

Aleksandar S. Dimovski

IT University of Copenhagen, Denmark

Abstract. Program families can produce a (potentially huge) number of related programs from a common code base. Many such programs are safety critical. However, most verification techniques are designed to work on the level of single programs, and thus are too costly to apply to the entire program family. In this paper, we propose an efficient game semantics based approach for verifying open program families, i.e. program families with free (undefined) identifiers. We use symbolic representation of algorithmic game semantics, where concrete values are replaced with symbolic 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 verify safety of all programs from a family at once and pinpoint those programs that are unsafe (respectively, safe). We present a prototype tool implementing this approach, and we illustrate it with several examples.

1 Introduction

Software Product Line (SPL) [5] 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. While there are different implementation strategies, many popular SPLs from system software (e.g. Linux kernel) and embedded software (e.g. cars, phones) domains [18] are implemented using a simple form of two staged computation in preprocessor style, where the programming language is extended with *conditional compilation* constructs (e.g. `#ifdef` annotations from C preprocessor). 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, and greater market coverage. Unfortunately, the complexity created by program families (variability) also leads to problems. The simplest *brute-force approach* to verify such program families is to use a preprocessor to generate all valid products of an SPL, and then apply an existing single-program verification technique to each resulting product. However, this approach is very

costly and often infeasible in practice since the number of possible products is exponential in the number of features. Therefore, we seek for new approaches that rely on finding compact mathematical structures, which take the variability within the family into account, and on which specialized variability-aware verification algorithms can be applied.

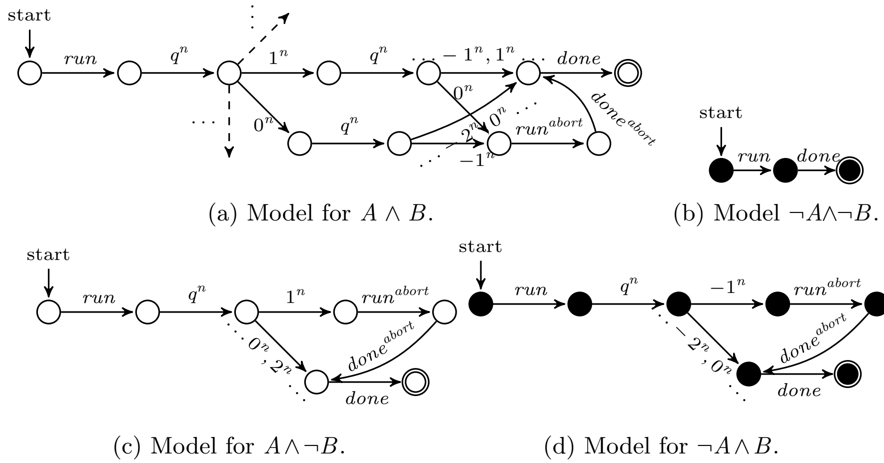
In this work, we address the above challenges by using game semantics models. Game semantics [1,14] is a technique for *compositional* modelling of programming languages, which gives models that are fully abstract (sound and complete) with respect to observational equivalence of programs. It has mathematical elegance of denotational semantics, and step-by-step modelling of computation in the style of operational semantics. In the last decade, a new line of research has been pursued, known as *algorithmic game semantics*, where game semantics models are given certain kinds of concrete automata-theoretic representations [12,8,16]. Thus, they can serve as a basis for software model checking and program analysis. The most distinctive property of game semantics is compositionality, i.e. the models are generated inductively on the structure of programs. This is the key to achieve *scalable (modular)* verification, where a larger program is broken down into smaller program fragments which can be modeled and verified independently. Moreover, game semantics yields a very accurate model for any open program with free (undefined) identifiers such as calls to library functions.

In [9], a symbolic representation of algorithmic game semantics has been proposed for second-order Idealized Algol (IA_2). It redefines the (standard) regular-language representation [12] at a more abstract level by using symbolic values instead of concrete ones. This allows us to give a compact representation of programs with infinite integers by using finite-state symbolic automata. Here, we extend the symbolic representation of game semantics models, obtaining so-called *featured symbolic automata*, which are used to compactly represent and verify safety properties of program families.

Motivating Example. To better illustrate the issues we are addressing in this work, we now present a motivating example. Table 1 shows a simple program family M that contains two `#if` commands. They increase and decrease the local variable x by the value of a non-local expression n, depending on the enabled features. The program uses features $\mathbb{F} = \{A, B\}$ and we assume it has the following set of valid configurations $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. For each valid configuration a different single program can be generated by appropriately resolving the `#if` commands. For example, the single program 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 to be present in the program. 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 $x := x - n$. Programs corresponding to all valid configurations are illustrated in Table 1. Thus, to verify our family M we need to build and analyze models of four distinct, but very similar, programs.

Program family M:	Config. $A \wedge B$:
n : exp int^n, $abort$: com^{abort} $\vdash_{\{A,B\}}$ new$_{\text{int}}$ $x := 0$ in #if (A) then $x := x + n$; #if (B) then $x := x - n$; if $(x = 1)$ then abort : com	n : exp int^n, $abort$: com^{abort} $\vdash$ new$_{\text{int}}$ $x := 0$ in $x := x + n$; $x := x - n$; if $(x = 1)$ then abort : com
Configs. $A \wedge \neg B$ $(\neg A \wedge B)$: n : exp int^n, $abort$: com^{abort} $\vdash$ new$_{\text{int}}$ $x := 0$ in $x := x + n$; $(x := x - n;\ )$ if $(x = 1)$ then abort : com	Config. $\neg A \wedge \neg B$: n : exp int^n, $abort$: com^{abort} $\vdash$ new$_{\text{int}}$ $x := 0$ in if $(x = 1)$ then abort : com

Table 1: Motivating example: the program family M and its valid products



(a) Model for $A \wedge B$.

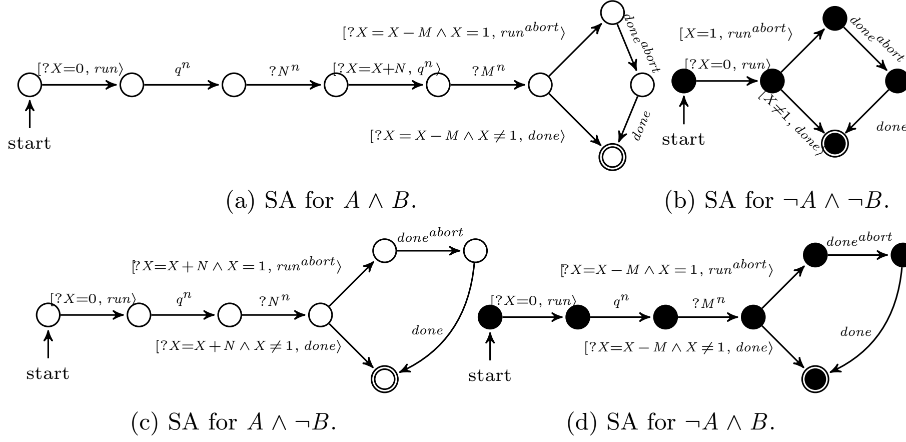
(b) Model $\neg A \wedge \neg B$.

(c) Model for $A \wedge \neg B$.

(d) Model for $\neg A \wedge B$.

Fig. 1: Automata for valid products of M.

We show in Fig. 1, the standard regular-language representation of game semantics for these four programs where concrete values are used [12]. 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. For example, the model for the product $A \wedge \neg B$ in Fig. 1c illustrates the observable interactions of this term of type com with its environment consisting of free 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 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



(a) SA for $A \wedge B$. (b) SA for $\neg A \wedge \neg B$.

(c) SA for $A \wedge \neg B$. (d) SA for $\neg A \wedge B$.

Fig. 2: Symbolic automata for valid products of M.

term terminates successfully by reaching the accepting state (shown as double circle in the model). Note that each move 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. Every letter (label of transition) contains a move and a Boolean condition which represents a constraint that needs to be fulfilled in order for 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 N. If the value of N is 1, the *abort* command is run. We say that "$X_1 = 0 \wedge X_2 = X_1 + N \wedge X_2 = 1$" is a play condition for the play in Fig. 2c: $[X_1 = 0, run\rangle \cdot q^n \cdot N^n \cdot [X_2 = X_1{+}N \wedge X_2 = 1, run^{abort}\rangle \cdot done^{abort} \cdot done$. This play is obtained from: $[?X = 0, run\rangle \cdot q^n \cdot ?N^n \cdot [?X = X{+}N \wedge X = 1, run^{abort}\rangle \cdot done^{abort} \cdot done$, after instantiating its input symbols with fresh symbolic names. We say that one 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 an SMT solver.

Now, by further enriching letters with feature expressions (propositional formulae defined over the set of features), we can give a 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

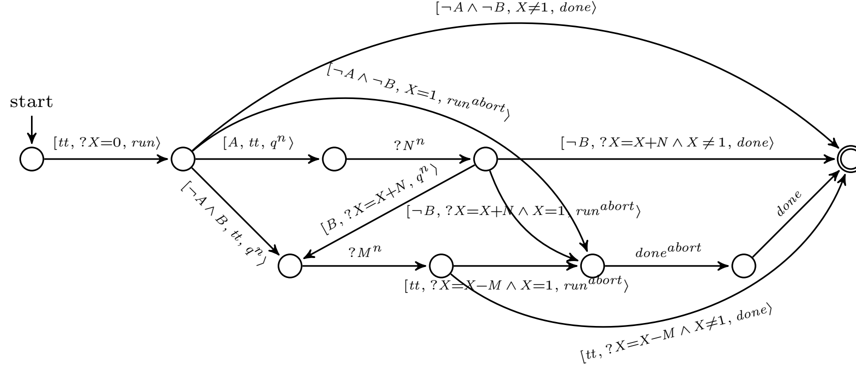


Fig. 3: Featured symbolic automaton for the program family M.

configurations that letter (in fact, the corresponding move) is feasible. Thus, we can represent all products of 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 product whether an unsafe behaviour (one that contains *abort* moves) can be exercised. If we find such an unsafe play for a valid product, then we need to check that the play is feasible. If its play condition is satisfiable, the SMT solver will return concrete assignments to symbols which make that condition *true*. In this way, we will generate a concrete counterexample for a valid product. In our example, we can determine that the product $\neg A \wedge \neg B$ is safe, whereas products $A \wedge B$, $A \wedge \neg B$, 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.

Remark. Alternatively, a program family can be verified by generating a so-called family *simulator* [2], 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) model checking algorithms [8,9] can be used to verify the generated simulator, since it represents a single program. In case of violation, we will obtain a single counterexample that corresponds to some unsafe products. However, this answer is incomplete (limited) for program families since there might be some safe products and also there might be other unsafe products with different counterexamples. Hence, no conclusive results for all products in a family are reported using this approach. For example, the simulator for the family M is:

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

If we generate a (game) model for this term and verify it using algorithms in [8,9], we will obtain a counterexample corresponding only to the product $A \wedge \neg B$.

This leads us to propose an approach that solves the general family-based model checking problem: determine for each product whether or not it is safe, and provide a counterexample for each unsafe product.

Contributions. In this paper, we make the following contributions:

- We introduce a compact symbolic representation, called *featured symbolic automata*, which represent 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 propose specifically designed (family-based) model checking algorithms for verifying featured symbolic automata that represent program families. This allows us to verify safety for all products of a family at once (in a single execution), and to pinpoint the products that are unsafe (resp., safe).
- We describe a prototype tool implementing the above algorithms, and we perform an evaluation to demonstrate the improvements over the brute-force approach where all valid products are verified independently one by one.

2 The Language for Program Families

The standard approach in semantics community is to use meta-languages for the description of certain kinds of computational behaviour. 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 can be defined, and then we introduce static variability into it.

Writing Single Programs. We consider the meta-language: Idealized Algol (IA) introduced by Reynolds in [17]. 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 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:

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

where v ranges over constants of type D, which includes integers (n) and boolean ($tt, f\!f$). The standard arithmetic-logic operations op are employed, as well as the usual imperative and functional constructs. *Well-typed terms* are given by typing judgements of the form $\Gamma \vdash M : T$, where Γ is a type *context* consisting of a finite number of typed free identifiers. Typing rules are given in [1,17].

The *operational semantics* is defined by a big-step reduction relation: $\Gamma \vdash M, \mathrm{s} \Longrightarrow V, \mathrm{s}'$, where $\Gamma \vdash M : T$ is a term in which all free identifiers from Γ are

variables, 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 [1,17] for details). If M is a closed term (with no free identifiers) of type com, then we abbreviate the relation $M, \emptyset \Longrightarrow \mathsf{skip}, \emptyset$ with $M \Downarrow$. 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 terms-with-hole $C[-] : \mathsf{com}$, such that $\vdash C[M] : \mathsf{com}$ and $\vdash C[N] : \mathsf{com}$ are well-typed closed terms of type com, if $C[M] \Downarrow$ then $C[N] \Downarrow$. 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 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 $\mathsf{bool} = \{tt, f\!f\}$) 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 formulas: $k(A_1) \cdot A_1 \wedge \dots \wedge k(A_n) \cdot A_n$, where $tt \cdot A = A$ and $f\!f \cdot A = \neg A$. 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 [5,18], but in this work we disregard syntactic representations of the set $\mathbb{K}$.

The language $\overline{\mathrm{IA}_2}$ extends IA_2 with a new compile-time conditional term for encoding multiple variations of a program, i.e. different valid products. The new term "$\mathtt{\#if}\ \phi\ \mathtt{then}\ M\ \mathtt{else}\ M'$" 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 product, otherwise M' will be included. The new syntax is:

$$M ::= \dots \mid \mathtt{\#if}\ \phi\ \mathtt{then}\ M\ \mathtt{else}\ M' \qquad\qquad \phi ::= A \in \mathbb{F} \mid \neg\phi \mid \phi \wedge \phi$$

We add a new syntactic category of feature expressions (i.e. propositional logic formulae over $\mathbb{F}$), $\mathit{FeatExp}(\mathbb{F})$, ranged over by ϕ, to write compile-time conditions over features $\mathbb{F}$. *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 [1]. Typing rules are those of IA_2 extended with a rule for the new construct:

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

The semantics of $\overline{\mathrm{IA}_2}$ 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 $\overline{\mathrm{IA}_2}$ to IA_2 specified by the projection function π_k mapping

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

an $\overline{\text{IA}_2}$ 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 $\overline{\text{IA}_2}$ 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 $k \in \mathbb{K}$ can now be defined as: $\Gamma \vdash \pi_k(M) : T$.

3 Symbolic Representation of Game Semantics

In this section we first recall symbolic representation of algorithmic game semantics for IA_2 [9], and then we extend this representation for $\overline{\text{IA}_2}$.

3.1 Symbolic Models of $\mathbf{IA_2}$

Let *Sym* be a countable set of symbolic names, ranged over by upper case letters X, Y, Z. For any finite $W \subseteq \mathit{Sym}$, the function $\mathit{new}(W)$ returns a minimal symbolic name which does not occur in W, and sets $W := W \cup \mathit{new}(W)$. A minimal symbolic name not in W is the one which occurs earliest in a fixed enumeration $X_1, X_2, \ldots$ of all possible symbolic names. Let *Exp* be a set of expressions, ranged over by e, inductively generated by using data values ($v \in D$), symbols ($X \in \mathit{Sym}$), and standard arithmetic-logic operations (op). We use a to range over arithmetic expressions (*AExp*) and b over boolean expressions (*BExp*).

Let $\mathcal{A}$ be an alphabet of letters. We define a *symbolic alphabet* $\mathcal{A}^{sym}$ induced by $\mathcal{A}$ as follows: $\mathcal{A}^{sym} = \mathcal{A} \cup \{?X, e \mid X \in \mathit{Sym}, e \in \mathit{Exp}\}$. The letters of the form $?X$ are called *input symbols*. They represent a mechanism for generating new symbolic names, i.e. $?X$ means $\mathsf{let}\, X = \mathit{new}(W)\, \mathsf{in}\, X\ \ldots$. We use α to range over $\mathcal{A}^{sym}$. Next we define a *guarded alphabet* $\mathcal{A}^{gu}$ induced by $\mathcal{A}$ as the set of pairs of boolean conditions and symbolic letters: $\mathcal{A}^{gu} = \{[b, \alpha\rangle \mid b \in \mathit{BExp}, \alpha \in \mathcal{A}^{sym}\}$. A guarded letter $[b, \alpha\rangle$ means that α occurs only if b evaluates to true, i.e. *if* $(b = tt)$ *then* α *else* $\emptyset$. We use β to range over $\mathcal{A}^{gu}$. We will often write only α for the guarded letter $[tt, \alpha\rangle$. A word $[b_1, \alpha_1\rangle \cdot [b_2, \alpha_2\rangle \ldots [b_n, \alpha_n\rangle$ over $\mathcal{A}^{gu}$ can be represented as a pair $[b, w\rangle$, where $b = b_1 \wedge b_2 \wedge \ldots \wedge b_n$ is a boolean condition and $w = \alpha_1 \cdot \alpha_2 \ldots \alpha_n$ is a word of symbolic letters.

Now, we show how IA_2 terms in β-normal form are interpreted by symbolic regular languages and automata, which will be specified by extended regular expressions R. Each type T is interpreted by a guarded alphabet of moves $\mathcal{A}^{gu}_{[\![T]\!]}$ induced by $\mathcal{A}_{[\![T]\!]}$, which is defined as follows:

$$\mathcal{A}_{[\![\mathsf{exp} D]\!]} = \{q\} \cup \mathcal{A}_{[\![D]\!]}, \mathcal{A}_{[\![\mathsf{com}]\!]} = \{\mathit{run}, \mathit{done}\}, \mathcal{A}_{[\![\mathsf{var} D]\!]} = \{\mathit{write}(a), \mathit{read}, \mathit{ok}, a \mid a \in \mathcal{A}_{[\![D]\!]}\}$$
$$\mathcal{A}^{gu}_{[\![B_1^{\langle 1 \rangle} \to \ldots \to B_k^{\langle k \rangle} \to B]\!]} = \sum_{1 \leq i \leq k} \mathcal{A}^{gu\ \langle i \rangle}_{[\![B_i]\!]} + \mathcal{A}^{gu}_{[\![B]\!]}$$

where $\mathcal{A}_{[\![\mathsf{int}]\!]} = \mathbb{Z}$, $\mathcal{A}_{[\![\mathsf{bool}]\!]} = \{tt, f\!f\}$, and + denotes a disjoint union of alphabets. Function types are tagged by a superscript $\langle i \rangle$ to keep record from which type, i.e. which component of the disjoint union, each move comes from. The letters in the alphabet $\mathcal{A}_{[\![T]\!]}$ represent the *moves* (observable actions) that a term of type T can perform. Each of moves 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 commands, there is a *question* move *run* to initiate a command, and an *answer* move *done* to signal successful termination of a command. 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 any (β-normal) term, we define a (symbolic) regular-language which represents its game semantics, i.e. its set of complete plays. Every complete play represents the observable effects of a completed computation of the given term. It is given as a guarded word $[b, w\rangle$, where b is also called *play condition*. Assumptions about a play (computation) 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. If the play condition is inconsistent, this play is discarded from the final model of the corresponding term. The regular expression for $\Gamma \vdash M : T$, denoted as $[\![\Gamma \vdash M : T]\!]$, is defined over the guarded alphabet: $\mathcal{A}^{gu}_{[\![\Gamma \vdash T]\!]} = \big(\sum_{x:T' \in \Gamma} \mathcal{A}^{gu\,\langle x \rangle}_{[\![T']\!]}\big) + \mathcal{A}^{gu}_{[\![T]\!]}$, where moves corresponding to types of free identifiers are tagged with their names.

The representation of constants is standard:

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

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

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

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. Note that whenever an input symbol $?X$ ($\mathsf{let}\, X = new(W) \,\mathsf{in}\, X \ldots$) is met in a play, the mechanism for fresh symbol generation is used to instantiate it with a new fresh symbolic name, which binds all occurrences of X that follow in the play until a new $?X$ is met which overrides the previous one. 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.

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

Table 2: Symbolic representations of some language constructs

The representations of some language constructs are given in Table 2. Note 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 $\mathsf{c}(M_1, \ldots, M_k)$ built out of a language construct "c" and subterms $M_1, \ldots, M_k$ is interpreted by composing the regular expressions for $M_1, \ldots, M_k$ and the regular expression for "c". Composition of regular expressions ($\fatsemi$) is defined as "parallel composition plus hiding in CSP" [1]. Conditions of the shared (interacting) guarded letters in the composition are conjoined, along with the condition that their symbolic letters are equal [9]. The $\mathsf{cell}_v^{\langle x\rangle}$ regular 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 the initial value v. The model $[\![\mathsf{new}_D\ x := v \ \mathsf{in}\ M]\!]$ is obtained by constraining the model of M, $[\![\mathsf{new}_D\ x \vdash M]\!]$, only to those plays where x exhibits good variable behaviour described by $\mathsf{cell}_v^{\langle x\rangle}$, and then by deleting (hiding) all moves associated with x since x is a local variable and so not visible outside of the term [9].

The following formal results are proved in [9]. We define an *effective alphabet* of a regular expression to be the set of all letters that appear in the 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 $\mathcal{L}[\![\Gamma \vdash M : T]\!]$ is a symbolic regular-language over its effective (finite) alphabet. Moreover, a finite-state symbolic automata $\mathcal{A}[\![\Gamma \vdash M : T]\!]$ which recognizes it is effectively constructible.*

Suppose that there is a special free identifier abort of type com. We say that a term $\Gamma \vdash M$ is *safe* iff $\Gamma \vdash M[\mathsf{skip}/\mathsf{abort}] \sqsubseteq M[\mathsf{diverge}/\mathsf{abort}]$; otherwise we say that a term is *unsafe*. Hence, a safe term has no computation that leads to

running abort. Let $\mathcal{L}[\![\Gamma \vdash M : T]\!]^{CR}$ denotes the (standard) regular-language representation of game semantics for a term M obtained as in [12], 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 [7].

Proposition 1. *A term $\Gamma \vdash M : T$ is safe iff $\mathcal{L}[\![\Gamma \vdash M : T]\!]^{CR}$ does not contain any play with moves from $\mathcal{A}^{\langle abort \rangle}_{[\![\mathsf{com}]\!]}$, which we call unsafe plays.*

The following result [9] confirms that symbolic automata (models) can be used for establishing safety of terms.

Theorem 2. *$\mathcal{L}[\![\Gamma \vdash M : T]\!]$ is safe (all plays are safe) iff $\mathcal{L}[\![\Gamma \vdash M : T]\!]^{CR}$ is safe.*

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

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 [9] 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 a concrete counterexample is reported. We showed in [9] 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.

Example 1. Consider the term family M from Table 1 in Section 1. The symbolic model for the term $A \wedge B$ is given in Figure 2a. The term asks for a value of the non-local expression n with the move q^n two times, and the environment provides as answers symbols N and M. When the difference $N - M$ is 1, then *abort* command is run. The symbolic model in Fig. 2a 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 $(X_1 = 0 \wedge X_2 = X_1 + N \wedge X_3 = X_2 - M \wedge X_3 = 1)$ 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 counterexample will be: $run \cdot q^n \cdot 1^n \cdot q^n \cdot 0^n \cdot run^{abort} \cdot done^{abort} \cdot done$. Similarly, concrete counterexamples 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. □

3.2 Symbolic Models of $\overline{\mathbf{IA_2}}$

We extend the definition of guarded alphabet $\mathcal{A}^{gu}$ as the set of triples of feature expressions, boolean conditions and symbolic letters:

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

Thus, a guarded letter $[\phi, b, \alpha\rangle$ means 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 products 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*.

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:

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

where $\fatsemi$ is the composition operator, and the interpretation of the compile-time conditional construct parameterized by the feature expressions 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 configurations that satisfy ϕ, whereas the second argument of `#if` is run for configurations satisfying $\neg\phi$.

Again, the effective alphabet of $[\![\Gamma \vdash_{\mathbb{F}} M : T]\!]$ for any $\overline{\mathrm{IA}_2}$ term is a finite subset of $\mathcal{A}^{gu+f}_{[\![\Gamma \vdash_{\mathbb{F}} T]\!]}$. Hence, the automata corresponding to $[\![\Gamma \vdash_{\mathbb{F}} M : T]\!]$ is effectively constructible, and we call it *featured symbolic automata* (FSA). Basically, an FSA is a SA augmented with transitions labelled (guarded) with feature expressions. We denote it as $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] = (Q, i, \delta, F)$, 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 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 products able to execute it. From an FSA, we can obtain the model of one particular product 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 1. *The projection of $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] = (Q, i, \delta, F)$ to a configuration $k \in \mathbb{K}$, denoted as $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] \mid_k$, is the symbolic automaton $A = (Q, i, \delta', F)$, where $\delta' = \{(q_1, [b, a\rangle, q_2) \mid (q_1, [\phi, b, a\rangle, q_2) \in \delta \wedge k \models \phi\}$.*

Theorem 3 (Correctness). $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M : T]\!] \mid_k = \mathcal{A}[\![\Gamma \vdash \pi_k(M) : T]\!]$.

Example 2. 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 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 $A \wedge B$. □

4 Model Checking Algorithms

The *general* model checking problem for program families consists in determining which products in the family are safe, and which are not safe. The goal is to report all products that are not safe and to provide a counterexample for each.

A straightforward but rather naive algorithm to solve the above problem is to check all valid programs individually. That is, compute the projection of each valid configuration, generate its model, and verify it using standard algorithms. This is so-called brute force approach, and it is rather inefficient. Indeed, all programs (exponentially many in the worst case) 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 products. We aim to take advantage of the compact structure of FSAs in order to solve the above general model checking problem.

A model checker is meant to perform a search in the state space of the FSA (Q, i, δ, F) and to indicate safe and unsafe products. This boils down to checking if an 'unsafe' state $q' \in Q$ with $q \stackrel{[\phi, b, run^{abort}\rangle}{\longrightarrow} q'$ is reachable in the FSA. This can be accomplished with a Breadth-First Search (BFS) in the FSA that encounters all states that are reachable from the initial state and checks whether one of them is 'unsafe'. In this way, the BFS finds the shortest unsafe play for any product. 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 valid products 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 products, i.e. $(i, \mathbb{K}) \in R$. For $(q, px) \in R$, we write $R(q)$ for the set of products px. *Queue* supports the 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 a complete counterexample generated by looking at the trace kept on *Queue* along with q and by finding the shortest trace from q to an accepting state (by performing an embedded BFS); and px' is the corresponding set of unsafe products. Note that e represents the shortest unsafe play for the products in px'. At each iteration, the BFS calculates the set *new* of unvisited successors of the current state, filtering out states and products that are already visited in R. Assuming that we have a transition $q \stackrel{[\phi, b, \alpha\rangle}{\longrightarrow} q'$ and the source state q is reachable by products in px, the target state q' is reachable for products in $\{k \in px \mid k \models \phi\}$. Given an FSA as input, the algorithm in Fig. 4 calculates all 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*.

After having found an unsafe state, our algorithm will continue exploration until the entire model is explored. Since the aim is to identify violating prod-

Input: $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M]\!] = (Q, i, \delta, F)$ and valid configs. $\mathbb{K}$
Output: *true* if M is safe; otherwise *false* plus a set of counterexamples
1. $R := \{(i, \mathbb{K})\}$
2. $\mathit{Queue} := [(i, \mathbb{K})]$
3. $\mathit{unsafe} := \emptyset, \mathbb{K}_{unsafe} := \emptyset$
4. **while** $(\mathit{Queue} \neq [])$ **do**
5. $\quad (q, px) := \mathit{remove}(\mathit{Queue})$
6. $\quad$ **if** (q is UNSAFE) **then**
7. $\quad\quad (e, px') := \mathit{complete}(q, px, \mathit{trace}(q))$
8. $\quad\quad \mathit{unsafe} := \mathit{unsafe} \cup (e, px'), \mathbb{K}_{unsafe} := \mathbb{K}_{unsafe} \cup px'$
9. $\quad\quad \mathit{Queue} := \mathit{Queue} - \{(q, px) \in \mathit{Queue} \mid px \subseteq \mathbb{K}_{unsafe}\}$
10. $\quad$ **else** $\mathit{new} := \Big\{(q', px' \backslash R(q')) \,|_{q \stackrel{[\phi,b,m\rangle}{\longrightarrow} q', px' = \{k \in px | k \models \phi, k \notin \mathbb{K}_{unsafe}\}, px' \backslash R(q') \neq \emptyset}\Big\}$
11. $\quad\quad$ **while** $(\mathit{new} \neq \emptyset)$ **do**
12. $\quad\quad\quad (q', px') := \mathit{remove}(\mathit{new})$
13. $\quad\quad\quad R(q') := R(q') \cup px'$
14. $\quad\quad\quad \mathit{put}((q', px'), \mathit{Queue})\}$
15. $\quad\quad$ **end**
16. $\quad$ **end**
17. **end**
18. return$(\mathit{unsafe} \neq \emptyset), \mathit{unsafe}$

Fig. 4: Model Checking Algorithm for verifying safety based on Specialized BFS

ucts, it can ignore products that are already known to violate, $\mathbb{K}_{unsafe}$, that is $\cup_{(e,px) \in unsafe} px$. In the BFS, this can be achieved by filtering out states with products $px \subseteq \mathbb{K}_{unsafe}$ as part of the calculation of *new*. This can only eliminate newly discovered states, not those that are already on the queue. States on the queue can be filtered out by removing elements (q, px) for which $px \subseteq \mathbb{K}_{unsafe}$ (line 9).

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 products (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 $R(q) = px$, $(q, px') \in \mathit{new}$, and $px' \nsubseteq px$, 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'.

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 products and (unsafe) products for which a genuine (consistent) counterexample is reported. To prevent the model checker to consider these products (for which conclusive results are previously found), the BFS in the next iteration is called with updated arguments, i.e. only for configurations with no conclusive results. We first show that the projection π_k commutes with our "lifted" verification procedure, which is applied directly on the level of program families.

Theorem 4 (Correctness). *$\Gamma \vdash \pi_k(M)$ is safe iff $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M]\!] \mid_k$ is safe.*

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

1 The BFS from Fig. 4 is called with arguments: $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M]\!]$ and $\mathbb{K}$.
2 If no unsafe play is found, terminate with answer SAFE.
3 Otherwise, find $\mathbb{K}_{unsafe} = \cup_{(e,px) \in unsafe} px$. For all products in $\mathbb{K} \backslash \mathbb{K}_{unsafe}$ report that they are SAFE. The mechanism for fresh symbol generation is used to instantiate all input symbols in the unsafe plays e, and their conditions are tested for consistency.
4 If the condition of some play e from $(e, px) \in unsafe$ is consistent, report the corresponding products px as UNSAFE with counterexample e. Otherwise generate $\mathbb{K}' \subseteq \mathbb{K}_{unsafe}$ that contains all products associated with inconsistent plays. Then go to Step 1, i.e. call BFS with arguments: $\mathcal{FSA}[\![\Gamma \vdash_{\mathbb{F}} M]\!]$ without all inconsistent unsafe plays and $\mathbb{K}'$.

Fig. 5: Verification procedure (VP)

As a corollary of Theorems 2, 3, 4 we obtain that the VP in Fig. 5 returns correct answers for all products. Moreover, it terminates for all unsafe products by generating the corresponding unsafe plays. The VP will find the shortest consistent unsafe play t for each unsafe product after finite number of calls to the BFS, that will first find all inconsistent unsafe plays shorter than t (which are finitely many [9]). However, the VP may diverge for safe products, producing in each next iteration longer and longer unsafe plays with inconsistent conditions.

5 Implementation

We have extended the prototype tool developed in [9] to implement the VP in Fig. 5. That tool [9] converts any single (IA_2) term into a symbolic automata representing its game semantics, and then explores the automata for unsafe plays. The extended tool takes as input a term family, and generates the corresponding FSA, which is then explored based on the procedure described in Fig. 5. The tool is implemented in Java along with its own library for working with featured symbolic automata. The tool calls an external SMT solver Yices to determine consistency of play conditions. We now illustrate our tool with an example. The tool, further examples and detailed reports how they execute are available from: `http://www.itu.dk/~adim/symbolicgc.htm`. Consider the following version of the linear search algorithm, $\mathtt{Linear}_3$.

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

Bench.	$\mid \mathbb{K} \mid$	*family-based approach* Time	Max	Final	*brute-force approach* Time	Max	Final
`Intro`	4	0.34 s	25	9	0.78 s	68	28
`Linear`$_3$	8	1.34 s	51	9	3.86 s	336	72
`Linear`$_4$	16	2.34 s	57	9	7.28 s	720	144
`Linear`$_5$	32	4.50 s	63	9	16.27 s	1482	288

Fig. 6: Performance comparison for verifying program families.

The term family contains three features A, B, and C, and hence 8 products 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.

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 [9] for details). If we also want to check for array-out-of-bounds errors, we can include in the representation of arrays plays that perform abort moves when $I \geq k$.

If the value read from the environment for y occurs j-times (for $j > 0$) in the array x, then an unsafe behaviour is found in the FSA of the above term family. Hence, all products 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 products are unsafe. For example, for products $A \wedge B \wedge C$ and $\neg A \wedge \neg B \wedge C$ (for which j is set to 2) the tool reports a counterexample 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 counterexample 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 counterexample is obtained after 2 iterations of the VP, and it corresponds to a computation which runs the body of 'while' two times. In the first iteration, an inconsistent unsafe play is found (its condition contains $J = 2 \wedge J = 1$, where the symbol J tracks the current value of j). A consistent counterexample 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 plays in each next iteration.

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 four families: `Intro` is the family from Table 1 in Section 1; `Linear`$_3$ is the above family for linear search with three features; `Linear`$_4$ is an extended version of `Linear`$_3$ with one more feature D and command: `#if` (D) then $j := j + 3$; and `Linear`$_5$ is `Linear`$_4$ extended with one additional feature E and command: `#if` (E) then $j := j - 2$. We restrict our tool to work with bounded number of iterations (10 in this case) since the VP loops for safe terms. For Linear$_4$ the

tool reports 13 unsafe products with corresponding counterexamples, whereas for Linear_5 21 unsafe products are found. Fig 6 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 FINAL) of verifying benchmarks using our family-based approach vs. using brute-force approach. In the latter case, we first compute all products, generate their models, and verify them one by one by using the tool for single programs [9]. In this case we report the sum of number of states for the corresponding models in all individual products. We can see that the family-based approach is between 2.3 and 3.6 times faster (using considerably less space) than the brute-force. We expect even bigger efficiency gains for families with higher number of products.

6 Related work and Conclusion

Recently, many so-called lifted techniques have been proposed, which lift existing single-program analysis techniques to work on the level of program families (see [18] for a survey). This includes lifted type checking [3], lifted model checking [4], lifted data-flow analysis [11], etc. Classen et al. have proposed featured transition systems (FTSs) in [4] as the foundation for behavioural specification and verification of variational systems. An FTS, which is feature-aware extension of the standard transition systems, represents the behaviour of all instances of a variational system. They show how the family-based model checking algorithms for verifying FTSs against fLTL properties are implemented in the SNIP model checker. The input language to SNIP is fPromela, which is a feature-aware extension of Promela. In this work, we also propose special family-based model checking algorithms. However, they are not applied on models of variational systems, but on game semantics models extracted from concrete program fragments with `#ifdef`-s.

The first application of game semantics to model checking was proposed by Ghica and McCusker in [12]. They show how game semantics of IA_2 with finite data-types can be represented in a remarkably simple form by regular-languages. Subsequently, several algorithms have been proposed for model checking IA_2 with infinite data types [7,9]. The automata-theoretic representation of game semantics have been also extended to programs with various features: concurrency [13], third-order functions [16], probabilistic constructs [15], nondeterminism [6], etc.

To conclude, 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 us to verify safety of an FSA. The proposed approach can be extended to support multi-features and numeric features. So-called variability abstractions [10,11] can also be used to define abstract family-based model checking.

References

1. Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor.*

Comput. Sci., 3:2–14, 1996.
2. Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th Intern. Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.
3. Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40. ACM, 2012.
4. Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013.
5. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
6. Aleksandar Dimovski. A compositional method for deciding equivalence and termination of nondeterministic programs. In *8th International Conference on Integrated Formal Methods, IFM'10*, volume 6396 of *LNCS*, pages 121–135. Springer, 2010.
7. Aleksandar Dimovski, Dan R. Ghica, and Ranko Lazic. Data-abstraction refinement: A game semantic approach. In *12th International Symposium on Static Analysis, SAS '05*, volume 3672 of *LNCS*, pages 102–117. Springer, 2005.
8. Aleksandar Dimovski and Ranko Lazic. Compositional software verification based on game semantics and process algebra. *STTT*, 9(1):37–51, 2007.
9. Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014.
10. Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Family-based model checking without a family-based model checker. In *22nd International SPIN Workshop on Model Checking of Software, SPIN '15*, volume 9232 of *LNCS*, pages 282–299. Springer, 2015.
11. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP '15*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
12. Dan R. Ghica and Guy McCusker. The regular-language semantics of second-order idealized algol. *Theor. Comput. Sci.*, 309(1-3):469–502, 2003.
13. Dan R. Ghica and Andrzej S. Murawski. Compositional model extraction for higher-order concurrent programs. In *12th International Conference TACAS 2006*, volume 3920 of *LNCS*, pages 303–317. Springer, 2006.
14. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.
15. Axel Legay, Andrzej S. Murawski, Joël Ouaknine, and James Worrell. On automated verification of probabilistic programs. In *14th International Conference TACAS 2008*, volume 4963 of *LNCS*, pages 173–187. Springer, 2008.
16. Andrzej S. Murawski and Igor Walukiewicz. Third-order idealized algol with iteration is decidable. In *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005*, pages 202–218, 2005.
17. John C. Reynolds. *The essence of Algol*. In: O'Hearn, P.W., Tennent, R.D. (eds), *Algol-like languages*, Birkhaüser, 1997.
18. Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.