Probabilistic Analysis Based On Symbolic Game Semantics and Model Counting

Aleksandar S. Dimovski
Computer Science Department, IT University of Copenhagen, Copenhagen, Denmark
adim@itu.dk

Probabilistic program analysis aims to quantify the probability that a given program satisfies a required property. It has many potential applications, from program understanding and debugging to computing program reliability, compiler optimizations and quantitative information flow analysis for security. In these situations, it is usually more relevant to quantify the probability of satisfying/violating a given property than to just assess the possibility of such events to occur.

In this work, we introduce an approach for probabilistic analysis of open programs (i.e. programs with undefined identifiers) based on game semantics and model counting. We use a symbolic representation of algorithmic game semantics to collect the symbolic constraints on the input data (context) that lead to the occurrence of the target events (e.g. satisfaction/violation of a given property). The constraints are then analyzed to quantify how likely is an input to satisfy them. We use model counting techniques to count the number of solutions (from a bounded integer domain) that satisfy given constraints. These counts are then used to assign probabilities to program executions and to assess the probability for the target event to occur at the desired level of confidence. Finally, we present the results of applying our approach to several interesting examples and illustrate the benefits they may offer.

1 Introduction

In order to understand program behaviour better, apart from finding out whether a behaviour (execution) can successfully terminate or not, we often need to know how *likely* a behaviour is to occur. In particular, we want to distinguish between what is possible behaviour (even with extremely low probability) and what is likely behaviour (possible with higher probability). In this work, we show how to calculate the probability of behaviours and estimate the reliability of programs by using a combination of (symbolic) game semantics and model counting.

Game semantics [1, 17] is a technique for building models of programs that are *fully abstract*, i.e. sound and complete with respect to observational equivalence. The notion of observational equivalence relies on comparing the outcomes of placing programs in all possible syntactic contexts (environments). Its algorithmic subarea [15, 8, 7, 18] aims to apply game semantics models to software verification by providing concrete automata-based representations for them. The key characteristics of game semantics models are the following. They provide precise and compact summaries of observable (input and output) program behaviour, without showing the explicit reference to a state (state manipulations are hidden). There is a model for any open program with free (undefined) identifiers such as calls to library functions. Finally, the models are generated inductively (compositionally) on the structure of programs, which is often essential for the *modular* analysis of larger programs. Symbolic representation of game semantics models [10] extends the (standard) regular-language representation [15] by using symbolic data values instead of concrete ones for the inputs. This allows us to obtain compact models of programs by using finite-state symbolic automata. Each *complete symbolic play* (accepting word) in the model corresponds

Submitted to:
GandALF 2017

to a program execution (path), and it is guarded by a conjunction of constraints on the symbols, known as *play condition*, which indicate under what conditions this play (word, execution) is feasible. If the play condition is satisfied by some concrete values for symbols, then they represent input values that will allow the execution to follow the specific path through the code. For the generation of symbolic game models where each play is associated with a play condition we use the SYMBOLIC GAMECHECKER [1] tool [10]. *Model counting* is the problem of determining the number of solutions of a given constraint (formula). The LATTE [2] tool [20] implements state-of-the-art algorithms for computing volumes, both real and integral, of convex polytopes as well as integrating functions over those polytopes. In particular, we use model counting techniques and the LATTE tool to estimate algorithmically the exact number of points of a bounded (possibly very large) discrete domain that satisfy given linear constraints.

In this paper, we describe a method based on symbolic game models and model counting for performing a specific type of quantitative analysis – the calculation of play probabilities and the program reliability. Calculating the probability of a symbolic play (path) involves counting the number of solutions to the play condition (by using model counting), and dividing it by the total space of values of the inputs (context). We assume that the input values are *uniformly distributed* within their finite discrete domain. We label each (complete) symbolic play with either *success* or *failure* depending on whether a designated abort command is executed or not. Since the set of play conditions produced by the symbolic game model is a complete partition of the given finite input domain, we can compute the reliability of the program as the probability of satisfying any of the successful play conditions. To account for cycles (infinite behaviours) in the model, we use bounded analysis. For a "while" command, a bound is set for the exploration depth (i.e. the number of re-visited states). For an undefined (first-order) function, we restrict the number of times the function can call its arguments when placed in the given bounded context, thus obtaining a finite input domain.

The main contributions of this work are: (1) A demonstration of how to add path probabilities using (symbolic) algorithmic game semantics and model counting; (2) An application of our approach to calculate the reliability of open programs; (3) A prototype implementation as part of SYMBOLIC GAMECHECKER.

2 Programming Language

The use 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. Here we consider Idealized Algol (IA), a well studied meta-language introduced by Reynolds [22]. IA enables functional (typed call-by-name λ-calculus) and imperative programming. For the purpose of obtaining an automata-based representation of game semantics, we shall consider its second-order recursion-free fragment (IA$_2$ for short). Its types are:

$$D ::= \mathsf{int} \mid \mathsf{bool} \qquad B ::= \mathsf{exp}D \mid \mathsf{com} \mid \mathsf{var}D \qquad T ::= B \mid B \to T$$

where D, B, and T stand for data types, base types, and first-order function types, respectively. The *syntax* of the language is:

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

[1]`https://aleksdimovski.github.io/symbolicgc.html`.
[2]`http://www.math.ucdavis.edu/~latte`. UC Davis, Mathematics.

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 operator (!) which is used for reading the value stored in a variable, 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 [16]. 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 [1, 22].

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, 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 [1, 22] 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 $\Gamma \vdash M \cong N$. In general, observational equivalence is very difficult to reason about due to the universal quantification over all syntactic contexts $C[-]$ in which the terms can be placed.

3 Symbolic Game Models

We now give a brief overview of symbolic representation of the algorithmic game semantics for IA$_2$ [10]. Let *Sym* be a countable set of symbolic names, ranged over by 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 of all possible symbolic names. Let *Exp* be a set of expressions, ranged over by e, generated by data values ($v \in D$), symbols ($X \in \mathit{Sym}$) of data type D, and arithmetic-logic operations (op). We use a to range over arithmetic expressions (*AExp*) and b over boolean expressions (*BExp*).

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

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

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 dynamically generating new symbolic names. More specifically, $?X$ creates a stream of fresh symbolic names, binding X to the next symbol from its stream, $\mathit{new}(W)$, whenever $?X$ is evaluated (met). We use α to denote a symbolic letter. Given an arbitrary symbolic alphabet $\mathscr{A}^{sym}$, we define a *guarded alphabet* $\mathscr{A}^{gu}$ induced by $\mathscr{A}^{sym}$ as the set of pairs of boolean conditions and symbolic letters:

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

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. $\mathit{if}\,(b = tt)\,\mathit{then}\,\alpha\,\mathit{else}\,\emptyset$. We use β to denote a guarded letter. We will often write only α for the guarded letter $[tt, \alpha\rangle$. A word $[b_1, \alpha_1\rangle \cdot [b_2, \alpha_2\rangle \ldots [b_n, \alpha_n\rangle$ over $\mathscr{A}^{gu}$ 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.

We now describe how IA$_2$ terms can be translated into symbolic regular languages and symbolic automata. Each type T is interpreted by a guarded alphabet of moves $\mathscr{A}^{gu}_{[\![T]\!]}$ induced by $\mathscr{A}^{sym}_{[\![T]\!]}$, which is defined as follows: [3]

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

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 $\mathscr{A}^{sym}_{[\![T]\!]}$ represent the *moves*, i.e. 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 $\mathscr{A}^{sym}_{[\![\mathsf{exp}D]\!]}$, there is a *question* move q to ask for the value of the expression, and values from $\mathscr{A}^{sym}_{[\![D]\!]}$ 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 $\mathscr{A}^{sym}_{[\![D]\!]}$.

For any term, we define a (symbolic) regular-language which represents its game semantics, i.e. its set of complete symbolic plays. A *play* is a sequence of moves played by two players in turns: **P** (Player) which represents the term being modeled, and **O** (Opponent) which represents its context. Every (complete) symbolic play represents the observable effects of a completed execution (path) 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 symbolic play to be feasible are recorded in its play condition. For infeasible plays, the play condition is unsatisfiable, thus no assignment of concrete values to symbolic names exists that makes the play condition true. The regular expression for $\Gamma \vdash M : T$, denoted as $[\![\Gamma \vdash M : T]\!]$, is defined over the guarded alphabet:

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

where moves corresponding to types of free identifiers are tagged with their names to indicate the origin of moves. Hence, $[\![\Gamma \vdash M : T]\!]$ contains only observable moves associated with types of free identifiers from Γ (suitably tagged) as well as moves of the top-level type T.

The representation of constants is standard:

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

For example, an integer or boolean constant v is modeled by a play where the initial question q (“what is the value of this expression?”) is answered by the value of that constant v.

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

$$\begin{array}{l}
[\![\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 \\
\qquad\qquad\qquad \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 \\
\qquad\qquad\qquad \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)^*
\end{array}$$

Table 1: Symbolic representations of some language constructs

Free identifiers are represented by the so-called copy-cat regular expressions, which contain all possible behaviours of terms of that type, thus providing the most general context for an open term. Thus,

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

When a call-by-name non-local function x with k arguments is called, it may evaluate any of its arguments, zero or more times, in an arbitrary order (hence, the Kleene closure *) and then it returns any allowable answer X 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 the term $f : \mathsf{expint}^{\langle f,1\rangle} \to \mathsf{expint}^{\langle f,2\rangle} \to \mathsf{expint}^{\langle f\rangle} \vdash f : \mathsf{expint}^{\langle 1\rangle} \to \mathsf{expint}^{\langle 2\rangle} \to \mathsf{expint}$, where f is an undefined function with two arguments. Its symbolic model is:

$$q \cdot q^{\langle f\rangle} \cdot \big(q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot ?Z_1^{\langle 1\rangle} \cdot Z_1^{\langle f,1\rangle} + q^{\langle f,2\rangle} \cdot q^{\langle 2\rangle} \cdot ?Z_2^{\langle 2\rangle} \cdot Z_2^{\langle f,2\rangle}\big)^* \cdot ?X^{\langle f\rangle} \cdot X \qquad (2)$$

The play corresponding to function "f" which evaluates its first argument two times, after instantiating its input symbols $?Z_1$ and $?X$ is given as: $q \cdot q^{\langle f\rangle} \cdot q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot Z_{1,1}^{\langle 1\rangle} \cdot Z_{1,1}^{\langle f,1\rangle} \cdot q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot Z_{1,2}^{\langle 1\rangle} \cdot Z_{1,2}^{\langle f,1\rangle} \cdot X^{\langle f\rangle} \cdot X$, where $Z_{1,1}$ and $Z_{1,2}$ are two different symbolic names used to denote values of the first argument when it is evaluated the first and the second time, respectively. Therefore, we are using the streaming symbol $?Z_1$ to create different symbolic names so that we can produce distinct values (independent from one another) if $?Z_1$ 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 $\langle f,1\rangle$ (resp. $\langle f,2\rangle$) are the actions caused by evaluating the first (resp. second) argument of f.

The representations of some language constructs "c" are given in Table 1. Observe that letter conditions different than tt occur only in plays corresponding to "if" and "while" constructs. In the case of "if" construct, 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". For example, we have:

$$[\![\Gamma \vdash \mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, M' : \mathsf{com}]\!] = [\![\Gamma \vdash B : \mathsf{expbool}^{\langle 1\rangle}]\!] \,{}_{9}^{\circ}\, [\![\Gamma \vdash M : \mathsf{com}^{\langle 2\rangle}]\!] \,{}_{9}^{\circ}\, [\![\Gamma \vdash M' : \mathsf{com}^{\langle 3\rangle}]\!] \,{}_{9}^{\circ}\, [\![\mathsf{if}]\!]$$

where $[\![\mathsf{if} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \times \mathsf{com}^{\langle 3\rangle} \to \mathsf{com}]\!]$ is defined in Table 1. Composition of regular expressions ($\fatsemi$) is defined as "parallel composition followed by hiding" in CSP style [1]. The parallel composition is matching (synchronizing) of the moves in the shared types, whereas hiding is deleting of all moves from the shared types. Conditions of the shared (interacting) moves (guarded letters) in the composition are conjoined, along with the condition that their symbolic letters are equal [10]. The $\mathsf{cell}_v^{\langle x\rangle}$ regular expression in Table 1 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 $\mathsf{cell}_v^{\langle x\rangle}$ behaves as a storage cell and 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, $[\![\mathsf{var}_D\, x \vdash M]\!]$, 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 local and so not visible outside of the term [10].

The following formal results are proved before [10]. 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 $\mathscr{A}^{gu}_{[\![\Gamma \vdash T]\!]}$, which includes all constants, symbols, and expressions used for interpreting free identifiers, constructs, and local variables in M.

Proposition 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 automata $\mathscr{A}[\![\Gamma \vdash M : T]\!]$ 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* [4] iff $\Gamma \vdash M[\mathsf{skip}/\mathsf{abort}] \sqsubseteq M[\mathsf{diverge}/\mathsf{abort}]$; otherwise we say that a term is *unsafe*. We say that one play is *safe* if it does not contain moves from $\mathscr{A}^{\langle abort\rangle}_{[\![\mathsf{com}]\!]}$; otherwise we say that the play is *unsafe*.

Proposition 2 *A term $\Gamma \vdash M : T$ is safe iff all plays in $[\![\Gamma \vdash M : T]\!]$ are safe.*

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

Example 3 *Consider the term M:*

$$n : \mathsf{expint}^n, \mathsf{abort} : \mathsf{com}^{abort} \vdash\ \mathsf{new}_{\mathsf{int}}\, x := 0 \text{ in } \mathsf{while}\,(!x < n)\ \mathsf{do}\ x := !x + 1; \\ \mathsf{if}\,(!x > 1)\ \mathsf{then}\ \mathsf{abort} : \mathsf{com}$$

The model for this term is given in Fig. 1 [5]*. The dashed edges indicate moves of the environment (**O**) and solid edges moves of the term (**P**). They serve only as a visual aid to the reader. Accepting states are designated by an interior circle. Observe that the term communicates with its environment using non-local identifiers n and* abort*. So in the model will only be represented actions associated with n and* abort *as well as with the top-level type* com*. The input symbol ?X is used to keep track of the current value of the local variable x (note that X occurs only in conditional part of plays). Each time the term (**P**) asks for a value of n with the move q^n, the environment (**O**) provides a new fresh symbol ?N for it. Note that we consider all possible environments (contexts) in which a term can be placed. Therefore, the undefined expression n may obtain different value at each call in the above term [15]. At this point, the term (**P**) has three possible options depending on the current values of symbols N and X: it can terminate successfully with done; it can execute* abort *and terminate; or it can run the assignment $x := x + 1$ and ask for a new value of n.*

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

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

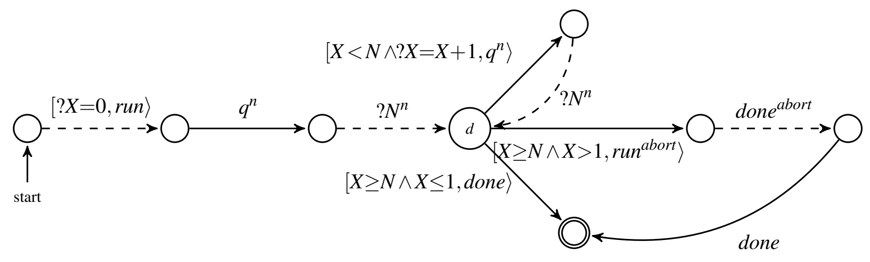


Figure 1: The symbolic game model for M.

4 Calculating Success and Failure Probabilities

In this section, we define the success and failure probability of terms, and show how they can be automatically calculated using symbolic game models and model counting. We also show how to cope with cases that introduce infinite behaviours.

4.1 Definition

We define the *success probability* as the probability that a term terminates successfully without hitting any failure, such as running the abort command. On the other hand, the *failure probability* is the probability that a term hits a failure during its execution. The resulting symbolic game model is a set of symbolic plays (words), each with a play condition. Some of these plays are unsafe (i.e. lead to a failure, abortion); whereas some of them are safe (i.e. lead to a successful termination without abortion). The plays are therefore classified in two sets: P^s which contains safe plays, and P^f which contains unsafe plays.

Our discussion focusses on the case of computing probabilities for terms that have *finite input domains* for all their plays (executions). This is achieved by constraining all identifiers from Γ to be of types in which only finite sets of basic data values D are used. For example, we may consider only the basic types over bool and $\mathsf{int}_k = [0,k) = \{0,\dots,k-1\}$ for any $k > 0$. We also need to bound the input domain when undefined (first-order) functions are used. This case is handled separately in Section 4.2. Finally, we restrict our attention on play conditions expressed as *linear integer arithmetic* (LIA) constraints over symbols whose values are *uniformly distributed* over their finite input domain.

Given a symbolic play $p \in [\![\Gamma \vdash M : T]\!]$, let ID_p be the total space of possible values in its finite input domain and let pc_p be its play condition (constraint). We now show how to calculate the probability of p occurring, denoted $Pr(p)$. We use the LATTE tool to compute the number of elements of ID_p that satisfy pc_p, denoted $\#(pc_p)$. The size of ID_p, denoted $\#(ID_p)$, is the product of domain's sizes of all symbols instantiated in p, which correspond to all calls of free identifiers of types in which data values D are used. Thus, we have: $\#(ID_p) = \prod_{Z \in p} \mid dom(Z) \mid$ and $Pr(p) = \#(pc_p)/\#(ID_p)$, where $\mid dom(Z) \mid = k$ if Z is a symbol that represents a value from the finite domain int_k. Note that the size of the input domain (context) ID_p for each play p can be different, and depends on how many symbols have been instantiated in p that correspond to the data type D. The play conditions associated with plays from P^s and P^f define disjoint input sets and cover the whole finite input domain, thus defining a complete partition of the finite input domain. Finally, we define the *success probability* (resp., *failure probability*) as the probability of evaluating the term $\Gamma \vdash M : T$ within a context (input) that enables all safe (resp., unsafe) plays:

$$Pr^s(\Gamma \vdash M : T) = \sum_{p \in P^s} \frac{\#(pc_p)}{\#(ID_p)}, \quad Pr^f(\Gamma \vdash M : T) = \sum_{p \in P^f} \frac{\#(pc_p)}{\#(ID_p)} \tag{3}$$

Note that $Pr^s(\Gamma \vdash M : T) + Pr^f(\Gamma \vdash M : T) = 1$.

Example 4 *Consider the term M':*

$$n : \mathsf{expint}^n_{10}, abort : \mathsf{com}^{abort} \vdash \mathtt{if}\ (n \geq 5)\ \mathtt{then}\,\mathsf{skip}\ \mathtt{else}\,\mathsf{abort}\colon \mathsf{com}$$

Its symbolic game model is:

$$run \cdot q^n \cdot ?N^n \cdot \big([N \geq 5, done\rangle + [N < 5, run^{abort}\rangle \cdot done^{abort} \cdot done\big)$$

Suppose that $n \in [0,10)$ and that the possible values for n are independently and uniformly distributed across this range. Thus, after instantiation of the input symbol ?N, there are one safe play ($run \cdot q^n \cdot N^n \cdot [N \geq 5, done\rangle$) and one unsafe play ($run \cdot q^n \cdot N^n \cdot [N < 5, run^{abort}\rangle \cdot done^{abort} \cdot done$). The safe (resp., unsafe) play condition is: $N \geq 5$ (resp., $N < 5$). Thus, we obtain $Pr^s(M') = 5/10\,(50\%)$ and $Pr^f(M') = 5/10\,(50\%)$.

We use model counting and the LATTE tool [20] to determine the number of solutions of a given constraint. LATTE accepts LIA constraints expressed as a system of linear inequalities each of which defines a hyperplane encoded as the matrix inequality: $Ax \leq B$, where A is an $m \times n$ matrix of coefficients and B is an $n \times 1$ column vector of constants. Most LIA constraints can easily be converted into the form: $a_1x_1 + \ldots + a_nx_n \leq b$. For example, $\geq$ and $>$ can be flipped by multiplying both sides by -1, and strict inequalities $<$ can be converted by decrementing the constant b. In LATTE equalities $=$ can be expressed directly. If we have disequalities $\neq$, they can be handled by counting a set of constraints that encode all possible solutions. For example, the constraint $\alpha \wedge (x_1 \neq x_2)$ is handled by finding the sum of solutions for $\alpha \wedge (x_1 \leq x_2 - 1)$ and $\alpha \wedge (x_1 \geq x_2 + 1)$. For a system $Ax \leq B$, where A is an $m \times n$ matrix and B is an $n \times 1$ column vector, the input LATTE file is:

$$\begin{array}{cc} m & n{+}1 \\ B & -A \end{array}$$

For example, the constraint "$N < 5$" from Example 4 results in the following (hyperplane) H-representation for LATTE:

$$\begin{array}{cc} 3 & 2 \\ 9 & -1 \\ 0 & 1 \\ 4 & -1 \end{array}$$

where the first line indicates the matrix size: the number of inequalities by the number of variables plus one. The next two inequalities encode the max and min values for the symbol N based on its data type. The last inequality expresses the constraint: $N \leq 4$ (i.e. $N < 5$). LATTE reports that there are exactly 5 points that satisfy the above inequalities ($N \leq 9 \wedge N \geq 0 \wedge N < 5$).

4.2 Bounded Analysis

The presence of "while" command and free identifiers of function type (i.e. undefined functions) introduce infinite behaviors, a cycle, in our model. Hence, convenient analysis strategies are required for handling them in order to compute the success and failure probabilities. In the case of the "while" command, the source of infinite behaviour is the *term* being modeled, but the context is still finite. On the other hand, in the case of undefined (first-order) functions, the source of infinite behaviour is the *context* in which that function can be placed (e.g. the function may call its arguments infinitely many times), so the context is unbounded in this case. This is the reason why we have two different strategies to cope with "while" and undefined functions.

The while **command.** The solution is based on bounded exploration: a (user-defined) *bound* $d \in \mathbb{N}$ is set for the search depth (i.e. the number of times a state can be re-visited). When the bound is reached the search backtracks. Intuitively, the bound $d \in \mathbb{N}$ represents the number of iterations of the while-loop and so we have the following bounded definition for while (instead of the one in Table 1):

$$[\![\mathsf{while} : \mathsf{expbool}^{\langle 1 \rangle} \times \mathsf{com}^{\langle 2 \rangle} \to \mathsf{com}]\!] = [tt, run\rangle \cdot [tt, q^{\langle 1 \rangle}\rangle \cdot [tt, ?Z^{\langle 1 \rangle}\rangle \cdot \\ \sum_{k=0}^{d} \big([Z, run^{\langle 2 \rangle}\rangle \cdot [tt, done^{\langle 2 \rangle}\rangle \cdot [tt, q^{\langle 1 \rangle}\rangle \cdot [tt, ?Z^{\langle 1 \rangle}\rangle\big)^k \cdot [\neg Z, done\rangle$$

In this setting the search is no longer complete, and besides safe and unsafe plays, a new set of plays is collected for traces interrupted before completing the search. We call this set of plays *grey* and label it as P^g. We can define $Pr^g(\Gamma \vdash M : T)$ analogously to the other sets as shown in Eqn. (3). The three sets of play conditions associated with plays in P^s, P^f, and P^g are disjoint and constitute a complete partition of the entire finite input domain. Hence, $Pr^s(\Gamma \vdash M : T) + Pr^f(\Gamma \vdash M : T) + Pr^g(\Gamma \vdash M : T) = 1$. The intuitive meaning of $Pr^g(\Gamma \vdash M : T)$ is to quantify the plays of $[\![\Gamma \vdash M : T]\!]$ for which neither safety nor unsafety have been revealed at the current exploration depth. This information is a measure of the *confidence* we can put on our success (resp., failure) estimation obtained within the given exploration bound: $Confidence = 1 - Pr^g(P)$. $Confidence = 1$ means that the search is complete, i.e. for each input we can state if it leads to a safe or an unsafe execution. Increasing the exploration depth, the confidence grows revealing more accurate safe (resp., unsafe) predictions.

Example 5 *Let us reconsider the term M from Example 3. Suppose that n is of type* expint_{10}. *We will now calculate the values of* Pr^s, Pr^f, Pr^g, *and Confidence, for different exploration depths d. Let* $d = 0$. *This means the state* ⓓ *from its symbolic model given in Fig. 1 can be visited only once (i.e.* ⓓ *cannot be re-visited). Let N be the symbol name instantiated for ?N. In this case, there is one unsafe play:* $[X{=}0, run\rangle \cdot q^n \cdot N^n \cdot [X{\geq}N \wedge X{>}1, run^{abort}\rangle \cdot done^{abort} \cdot done$, *and one safe play:* $[X{=}0, run\rangle \cdot q^n \cdot N^n \cdot [X \geq N \wedge X \leq 1, done\rangle$. *The condition of the unsafe play is unsatisfiable (note* $X{=}0 \wedge X{>}1$*) and so* $Pr^f(M) = 0$*; whereas the condition of the safe play is satisfiable with only one solution for* $N = 0$ *and so* $Pr^s(M) = 1/10\,(10\%)$. *For* $N \in [1, 10)$*, the state* ⓓ *needs to be re-explored so* $Pr^g(M) = 9/10$ *and* $Confidence = 1/10\,(10\%)$.

Let $d = 1$. *This means the state* ⓓ *in Fig. 1 can be re-visited once. Let* N_1 *and* N_2 *be the symbol names instantiated when ?N is evaluated the first and the second time, respectively. In this case, there are two unsatisfiable unsafe plays and two safe plays. The first safe play is from the previous iteration corresponding to* $N_1 = 0$ *with probability* $1/10$. *The second safe play is:* $run \cdot q^n \cdot N_1^n \cdot q^n \cdot N_2^n \cdot done$, *with the condition:* $X_1 = 0 \wedge X_1 < N_1 \wedge X_2 = X_1 + 1 \wedge X_2 \geq N_2 \wedge X_2 \leq 1$, *which has 18 solutions: for* $N_1 \in [1, 10)$ *and* $N_2 \in [0, 2)$. *Thus,* $Pr^s(M) = 1/10 + (9/10) \cdot (2/10) = 28/100\,(28\%)$*;* $Pr^f(M) = 0$*;* $Pr^g(M) = 72/100$*; and* $Confidence = 28/100\,(28\%)$.

Let $d = 2$ *and let* N_1, N_2, N_3 *be the symbol names instantiated the first, the second, and the third time when ?N is met, respectively. We obtain unsafe plays when* $N_1 \in [1, 10)$, $N_2 \in [2, 10)$, *and* $N_3 \in [0, 3)$*, and so we have* $Pr^f(M) = (9/10) \cdot (8/10) \cdot (3/10) = 21.6\%$, $Pr^s(M) = 28\%$, $Pr^g(M) = 50.4\%$, *and* $Confidence = 49.6\%$. *For* $d = 3$, *we have* $Pr^f(M) = 41.76\%$, $Pr^s(M) = 28\%$, $Pr^g(M) = 30.24\%$, *and* $Confidence = 69.76\%$.

Undefined functions. Recall the definition of undefined functions in Eqn. (1). The 'generic behaviour' of a call-by-name function is, when called by its context, to perform some sequence of calls to its arguments, and then to return a result. Since the number of times the function's arguments are called can be arbitrary (even infinite, see the Kleene closure in Eqn. (1)), the corresponding input domain is not finite. One solution is to place numeric bounds on the number of times an undefined function can call its arguments. For any integer $d > 0$, we define $\Gamma \vdash_d M : T$ as a term which can be placed into contexts where any of its first-order free identifiers from Γ can call its arguments at most $d-1$ times.

For example, the interpretation of $f : \mathsf{expint}^{\langle f,1\rangle} \to \mathsf{expint}^{\langle f,2\rangle} \to \mathsf{expint}^{\langle f\rangle} \vdash f : \mathsf{expint}^{\langle 1\rangle} \to \mathsf{expint}^{\langle 2\rangle} \to \mathsf{expint}$ now becomes:

$$q \cdot q^{\langle f\rangle} \cdot \textstyle\sum_{k=0}^{d-1} \left(q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot ?Z_1^{\langle 1\rangle} \cdot Z_1^{\langle f,1\rangle} + q^{\langle f,2\rangle} \cdot q^{\langle 2\rangle} \cdot ?Z_2^{\langle 2\rangle} \cdot Z_2^{\langle f,2\rangle}\right)^k \cdot ?X^{\langle f\rangle} \cdot X \qquad (4)$$

Thus, we now use the bound d instead of the Kleene closure *, which is used in the general case given in Eqn. (2). Let us calculate the sizes of input domains corresponding to individual plays from the above model in Eqn. (4). Assume that we work with the finite integer domain $\mathsf{int}_{10} = [0,10)$. The play $p_0 = q \cdot q^{\langle f\rangle} \cdot X^{\langle f\rangle} \cdot X$ corresponds to a function "f" which does not evaluate its arguments at all (a non-strict function), and so there are 10 different instantiations of p_0 since $X \in [0,10)$. Note that if the play condition is $\mathtt{true}$, which means that all instantiations of p_0 are feasible then $\#(\mathtt{true}_{p_0}) = 10$. If "$f$" evaluates its arguments once, then we have two plays: $p_{1,1} = q \cdot q^{\langle f\rangle} \cdot q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot Z_1^{\langle 1\rangle} \cdot Z_1^{\langle f,1\rangle} \cdot X^{\langle f\rangle} \cdot X$ ("f" evaluates its first argument) with 10^2 different instantiations corresponding to $Z_1, X \in [0,10)$, and $p_{1,2} = q \cdot q^{\langle f\rangle} \cdot q^{\langle f,2\rangle} \cdot q^{\langle 2\rangle} \cdot Z_2^{\langle 2\rangle} \cdot Z_2^{\langle f,2\rangle} \cdot X^{\langle f\rangle} \cdot X$ ("f" evaluates its second argument) with 10^2 different instantiations corresponding to $Z_2, X \in [0,10)$. For a function "f" that calls its arguments $d-1$ times in any order, we have 2^{d-1} plays each of which with 10^d different instantiations. The total number of symbolic plays is $1 + 2^1 + 2^2 + \ldots + 2^{d-1} = 2^d - 1$.

In general, for a play $p \in [\![\Gamma \vdash_d M : T]\!]$ where M contains $m \geq 0$ calls to an undefined function with n arguments, we have:

$$\#(ID_p) = (1 + n + n^2 + \ldots + n^{d-1})^m \cdot \textstyle\prod_{Z \in p} |\,dom(Z)\,|, \quad \#(pc_p) \leq \prod_{Z \in p} |\,dom(Z)\,| \qquad (5)$$

Note that if the undefined function has 1 argument, then the total number of symbolic plays is $1 + 1^2 + \ldots + 1^{d-1} = d$. When the play condition is $\mathtt{true}$, then $\#(\mathtt{true}_p) = \prod_{Z \in p} |\,dom(Z)\,|$.

Example 6 *Consider the term:*

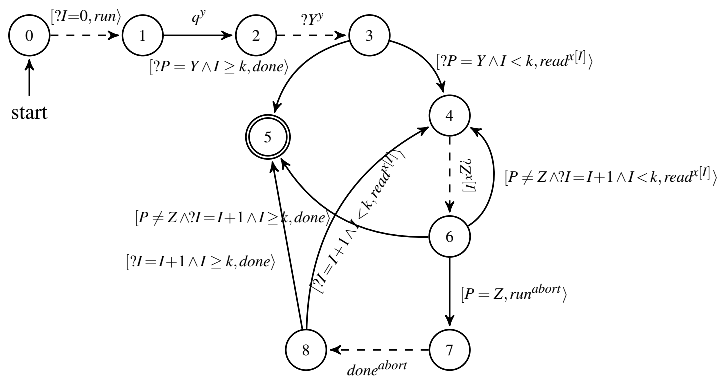
$$f : \mathsf{com}^{f,1} \to \mathsf{expint}_{10}^{f}, \mathsf{abort} : \mathsf{com}^{abort} \vdash_5 \ \mathsf{new}_{\mathsf{int}}\, x := 0 \ \mathsf{in}$$
$$\mathsf{if}\ (f(x := !x + 1) + !x > 3)\ \mathsf{then\ skip\ else\ abort} : \mathsf{com}$$

where we bound the size of context on definitions of "f" which can call its argument at most 4 times. Note that "f" has 1 argument and is called once in the above term. The symbolic model of the above term is:

$$[?X{=}0, run\rangle \cdot q^f \cdot \textstyle\sum_{k=0}^{4} \left([?X{=}X{+}1, run^{f,1}\rangle \cdot done^{f,1}\right)^k \cdot ?Z^f \cdot$$
$$\left([Z{+}X{>}3, done\rangle + [Z{+}X{\leq}3, run^{abort}\rangle \cdot done^{abort} \cdot done\right)$$

For the contexts corresponding to "f" which does not call its argument at all ($X = 0$), the unsafe behaviour is exercised when the value returned from f is $Z \in [0,4)$, i.e. the failure probability is $(1/5) \cdot (4/10)$. When the function "f" calls its argument once, the variable x is incremented once ($X = 1$) and so the failure probability is $(1/5) \cdot (3/10)$. For the contexts when "f" calls its argument twice ($X = 2$), abort *is run with the likelihood $(1/5) \cdot (2/10)$; when "f" calls its argument three times ($X = 3$) the failure probability is $(1/5) \cdot (1/10)$; whereas when "f" calls its argument four times ($X = 4$), the failure probability is 0% ($Z + X \leq 3$ is unsatisfiable). Therefore, for $d = 5$, the failure probability is $(4/50) + (3/50) + (2/50) + (1/50) + (0/50) = 10/50\,(20\%)$; whereas the success probability is $40/50\,(80\%)$.*

When $d = 6$, the failure probability is $10/60\,(16.7\%)$, and the success is 83.3%. For $d = 10$, the failure is $10/100\,(10\%)$, and the success is 90%.



0
1
2
3
4
5
6
7
8
start
[?I=0, run⟩
q^y
?Y^y
[?P = Y ∧ I ≥ k, done⟩
[?P = Y ∧ I < k, read^{x[I]}⟩
?Z^{x[I]}
[P ≠ Z ∧ ?I=I+1 ∧ I<k, read^{x[I]}⟩
[P ≠ Z ∧ ?I=I+1 ∧ I ≥ k, done⟩
[?I=I+1 ∧ I ≥ k, done⟩
[?I=I+1 ∧ I<k, read^{x[I]}⟩
[P = Z, run^{abort}⟩
done^{abort}

Figure 2: The model for the linear search term.

5 Implementation

We have extended the SYMBOLIC GAMECHECKER tool [10] to implement our approach for performing probabilistic analysis of open terms. The basic tool [10] converts any IA_2 term into a symbolic automaton representing its game semantics, and then explores the automaton for unsafe traces (plays). It calls an external SMT solver, Yices [12], to determine satisfiability of play conditions. The extended tool performs a bounded probabilistic analysis on the obtained symbolic automaton in order to determine the success and failure probabilities of the input term. Instead of an SMT solver, the extended tool calls a model counter, LATTE [20], to determine the number of solutions to play conditions. We now illustrate our tool with an example. The tool, further examples and reports on how they execute are available from: `https://aleksdimovski.github.io/symbolicgc.html` (version for probabilistic analysis).

Consider the following version of the linear search algorithm:

```
$x[k]$ : $\mathsf{varint}_n^{x[-]}$, $y$ : $\mathsf{expint}_n^{y}$, abort : $\mathsf{com}^{abort}$ $\vdash$
    $\mathsf{new}_{int}\, i := 0$ in
    $\mathsf{new}_{int}\, p := y$ in
    while $(i < k)$ do {
        if $(x[i] = p)$ then abort;
        $i := i + 1$;
    } : com
```

The meta variable $k > 0$ represents the size of array x, and $n > 0$ represents the domain size of input expressions y and $x[0], \ldots, x[k-1]$. Both k and n will be replaced by several different values. In the above term, first the input expression y is copied into the local variable p. Then the non-local array x is searched for an occurrence of the value stored in p. If the search succeeds, abort is executed. We assume that y and all elements of the array x can take one uniform value from the range $[0,n)$, i.e. their type is expint_{10}.

The symbolic model for this term is given in Fig. 2. The array $x[k]$ is given a symbolic representation [10], where the array size k and the index of the array elements represent symbols. We use symbols I and P to track the current values of local variables i and p, respectively. The symbol I is used to represent the index of an array element that needs to be de-referenced or assigned to. If the value Y read from the environment $\mathbf{O}$ for the expression y is equal to the value Z read from the environment $\mathbf{O}$ for some array element $X[I]$, where $0 \leq I < k$, i.e. the constraint $(P = Y) \wedge (P = Z)$ holds, then an unsafe behaviour is

k	# traces		# states	Analysis Time		
	unsafe	safe		$n = 10$	$n = 256$	$n = 65,536$
1	1	2	35	0.83	0.87	0.99
3	11	4	162	6.43	6.55	6.83
5	57	6	756	52.50	55.52	56.55
6	120	7	1677	166.58	169.02	175.32
7	247	8	3758	607.40	611.35	619.14

Table 2: Performance of the probabilistic analysis of the linear search term for different values of the array size k and the domain size n. Time is in seconds (s).

exercised.

We now present the probabilistic analysis of the above term for $n = 10$ and for various concrete values of k. The exploration bound is $d = k - 1$. For $k = 1$, abort is run only when the values $Y \in [0,10)$ and $Z \in [0,10)$ read from the environment for y and $x[0]$, respectively, are equal. Hence, the failure probability is 10%, and the success probability is 90%. For $k = 2$, we obtain only one feasible safe trace when $Y \neq Z_1$ and $Y \neq Z_2$ for the values $Y, Z_1, Z_2 \in [0,10)$ read from the environment for y, $x[0]$ and $x[1]$, respectively. Therefore, the success probability is $(9 \cdot 9)/(10 \cdot 10) = 81\%$, and the failure probability is 19%. For $k = 3$, we obtain that the success probability is 72.9% and the failure probability is 27.1%. For $k = 5$, we have $Pr^s = 59.05\%$ and $Pr^f = 40.95\%$. For $k = 10$ and $d = 9$, the success probability is 40.7% and the failure probability is 59.3%. We notice that as the array size k grows, the likelihood of running abort grows as well since there are more array elements in this case and the probability that some of them is equal to the input expression y is bigger.

We now report experimental results for performing the probabilistic analysis of the linear search term for different values of k (size of array) and n (domain size of undefined expressions). We ran our tool on a 64-bit Intel®Core^{TM} i5 CPU and 8 GB memory. The performance numbers reported constitute the average runtime of five independent executions.

The symbolic model has 9 states and the total time needed to generate the model is 0.24 sec. Note that the model size and the time needed to generate it are the same for all values of k and n. The results from running probabilistic analysis for this term are shown in Table 2. For different values of k we list: the number of generated unsafe and safe traces, the total number of visited (re-explored) states during the analysis, and the execution time in seconds needed to perform the analysis (search) when $n = 10$, $n = 256$ (1 byte) and $n = 65,536$ (2 bytes). We perform three sets of experiments, the first when the domain size of undefined expressions is $n = 10$, the second when the domain size is $n = 256$, and the third when $n = 65,536$. We only show the different analysis times corresponding to various values of n, since the first three parameters are the same in all cases. We observe that we obtain similar time performance results for $n = 10$, $n = 256$, and $n = 65,536$, mostly due to the fact that LATTE is largely insensitive to those values in terms of time. On the other hand, the analysis time increases for bigger values of k. In those cases, we have more traces to analyze and more complex constraints, which lead to more calls to LATTE.

6 Related work

Traditional formal approaches for probabilistic analysis based on probabilistic model checking [19] require a high-level design of the software. However, such models are difficult to maintain and may abstract important details that impact the chance of property satisfaction in the system. The ultimate goal is to perform probabilistic analysis directly on implementations, not on high-level models. Recent approaches [14, 13, 2] have proposed to use symbolic execution to support probabilistic analysis on the source code. In this work, we define probabilistic analysis in the settings of game semantics. This brings several distinctive features to our approach, such as: very precise models, compositional modelling, models of open second order imperative programs with free identifiers which take into account all possible contexts in which the programs can be considered. In contrast, the works in [14, 13, 2] consider imperative programs that only have some undefined global variables.

Game semantics for full Idealized Algol has been defined before [1]. The applications to software model checking were first proposed in [15], where game semantics models for second-order IA with finite data types were represented as finite automata. By using symbols instead of concrete data for inputs, it was shown [5, 10] how to generate finite symbolic automata for second-order IA with infinite data types. We have shown how to extend symbolic automata in order to represent program families implemented using `#ifdef` annotations [11]. Specifically designed model checking algorithms are then employed to verify safety of all variants of the family at once, and report those variants that are unsafe (resp., safe). Algorithmic game semantics also provides a method [6, 9] for ensuring secure information flow of open programs, i.e. for verifying security properties such as timing leaks, non-interference, and termination leaks. A fully abstract game semantics models for Probabilistic Idealized Algol (PA) has been formally defined in [3]. PA extends IA by allowing (fair) coin-tossing as a valid expression. Algorithmic probabilistic game semantics for PA have been studied as well [21, 18]. In particular, probabilistic equivalence and refinement have been explored in the context of game semantics. Intuitively, two probabilistic programs are equivalent if for each input they give rise to identical probabilistic distributions on the set of possible outputs. An automated equivalence checker for PA is also developed which takes a program as input and returns a probabilistic automaton capturing the game semantics of the program. An interesting direction for future work would be to calculate path probabilities and program reliability for PA terms.

7 Conclusion

In this work, we show how game semantics and model counting can be used to give a specific quantitative analysis of open programs – calculation of program path probabilities. We also apply the obtained analysis results for predicting program reliability.

Our analysis used the IA_2 language in order to stay focused, but a similar approach can be extended to any language for which an algorithmic game semantics exists. The model considered here contains only convergent behaviours of terms, and so it is suitable for verifying safety properties. If we want to take into account liveness properties as well, then the model should be enriched to contain all possible divergent behaviors of terms [16, 4]. For such models, we can calculate probabilities for convergence and divergence of terms similarly as we did here for success and failure.

References

[1] Samson Abramsky & Guy McCusker (1996): *Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions*. *Electr. Notes Theor. Comput. Sci.* 3, pp. 2–14, doi:10.1016/S1571-0661(05)80398-6.

[2] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Pasareanu & Willem Visser (2014): *Compositional solution space quantification for probabilistic software analysis*. In Michael F. P. O'Boyle & Keshav Pingali, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14*, ACM, p. 15, doi:10.1145/2594291.2594329.

[3] Vincent Danos & Russell Harmer (2002): *Probabilistic game semantics*. *ACM Trans. Comput. Log.* 3(3), pp. 359–382, doi:10.1145/507382.507385.

[4] Aleksandar Dimovski (2010): *A Compositional Method for Deciding Equivalence and Termination of Nondeterministic Programs*. In: *8th International Conference on Integrated Formal Methods, IFM'10*, *LNCS* 6396, Springer, pp. 121–135, doi:10.1007/978-3-642-16265-7_10.

[5] Aleksandar Dimovski (2012): *Symbolic Representation of Algorithmic Game Semantics*. In: *Proceedings Third International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2012,*, *EPTCS* 96, pp. 99–112, doi:10.4204/EPTCS.96.8.

[6] Aleksandar Dimovski (2013): *Slot Games for Detecting Timing Leaks of Programs*. In: *Proceedings Fourth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2013,*, *EPTCS* 119, pp. 166–179, doi:10.4204/EPTCS.119.15.

[7] Aleksandar Dimovski, Dan R. Ghica & Ranko Lazic (2005): *Data-Abstraction Refinement: A Game Semantic Approach*. In: *12th International Symposium on Static Analysis, SAS '05*, *LNCS* 3672, Springer, pp. 102–117, doi:10.1007/11547662_9.

[8] Aleksandar Dimovski & Ranko Lazic (2007): *Compositional software verification based on game semantics and process algebra*. *STTT* 9(1), pp. 37–51, doi:10.1007/s10009-006-0005-y.

[9] Aleksandar S. Dimovski (2014): *Ensuring Secure Non-interference of Programs by Game Semantics*. In: *Security and Trust Management - 10th International Workshop, STM 2014. Proceedings*, *LNCS* 8743, Springer, pp. 81–96, doi:10.1007/978-3-319-11851-2_6.

[10] Aleksandar S. Dimovski (2014): *Program verification using symbolic game semantics*. *Theor. Comput. Sci.* 560, pp. 364–379, doi:10.1016/j.tcs.2014.01.016.

[11] Aleksandar S. Dimovski (2016): *Symbolic Game Semantics for Model Checking Program Families*. In Dragan Bosnacki & Anton Wijs, editors: *Proceedings of 23rd International Symposium on Model Checking Software, SPIN'16*, *LNCS* 9641, Springer, pp. 19–37, doi:10.1007/978-3-319-32582-8_2.

[12] Bruno Dutertre (2014): *Yices 2.2*. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[13] Antonio Filieri, Corina S. Pasareanu & Willem Visser (2013): *Reliability analysis in symbolic pathfinder*. In: *35th International Conference on Software Engineering, ICSE '13*, IEEE Computer Society, pp. 622–631, doi:10.1109/ICSE.2013.6606608.

[14] Jaco Geldenhuys, Matthew B. Dwyer & Willem Visser (2012): *Probabilistic symbolic execution*. In Mats Per Erik Heimdahl & Zhendong Su, editors: *International Symposium on Software Testing and Analysis, ISSTA 2012*, ACM, pp. 166–176, doi:10.1145/2338965.2336773.

[15] Dan R. Ghica & Guy McCusker (2003): *The regular-language semantics of second-order Idealized ALGOL*. *Theor. Comput. Sci.* 309(1-3), pp. 469–502, doi:10.1016/S0304-3975(03)00315-3.

[16] Russell Harmer & Guy McCusker (1999): *A Fully Abstract Game Semantics for Finite Nondeterminism*. In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pp. 422–430, doi:10.1109/LICS.1999.782637.

[17] J. M. E. Hyland & C.-H. Luke Ong (2000): *On Full Abstraction for PCF: I, II, and III*. Inf. Comput. 163(2), pp. 285–408, doi:10.1006/inco.2000.2917.

[18] Stefan Kiefer, Andrzej S. Murawski, Joël Ouaknine, Björn Wachter & James Worrell (2013): *Algorithmic probabilistic game semantics - Playing games with automata*. Formal Methods in System Design 43(2), pp. 285–312, doi:10.1007/s10703-012-0173-1.

[19] Marta Z. Kwiatkowska, Gethin Norman & David Parker (2002): *Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach*. In Joost-Pieter Katoen & Perdita Stevens, editors: Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Proceedings, LNCS 2280, Springer, pp. 52–66, doi:10.1007/3-540-46002-0_5.

[20] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer & Ruriko Yoshida (2004): *Effective lattice point counting in rational convex polytopes*. J. Symb. Comput. 38(4), pp. 1273–1302, doi:10.1016/j.jsc.2003.04.003.

[21] Andrzej S. Murawski & Joël Ouaknine (2005): *On Probabilistic Program Equivalence and Refinement*. In: CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, Proceedings, LNCS 3653, Springer, pp. 156–170, doi:10.1007/11539452_15.

[22] John C. Reynolds (1997): *The essence of Algol*. In: O'Hearn, P.W., Tennent, R.D. (eds), *Algol-like languages*, Birkhaüser, doi:10.1007/978-1-4612-4118-8_4.