Program Verification using Symbolic Game Semantics

Aleksandar S. Dimovski

Faculty of Information-Communication Tech., FON University, Skopje, 1000, MKD

Abstract

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

Keywords: Algorithmic Game Semantics, Symbolic Automata, Program Verification, Predicate Abstraction

1. Introduction

Game semantics [1, 2, 19] is a technique for compositional modelling of programming languages, which gives fully abstract models. This means that the generated models are both sound and complete with respect to observational equivalence of programs. In game semantics, types are interpreted by *games* (or arenas) between a Player, which represents the term being modelled, and an Opponent, which represents the environment in which the term is used. The two participants strictly alternate to make moves, each of which is either a question (a demand for information) or an answer (a supply of information). Computations (executions of terms) are interpreted as *plays* of a game, while terms are expressed as *strategies*, i.e. sets of plays, for a game. It has been shown

Email address: `aleksandar.dimovski@fon.edu.mk` (Aleksandar S. Dimovski)

that game semantics model can be given certain kinds of concrete automata-theoretic representations [10, 14, 16], and so it can serve as a basis for software model checking and program analysis. Several features of game semantics make it very promising for software model checking. The model is very precise and compositional, i.e. generated inductively on the structure of programs, which is the key feature for achieving scalability. There is a model for any term-in-context (program fragment) with undefined identifiers, such as calls to library functions. However, the main limitation of model checking technique in general is that it can be applied only if a finite-state model is available. This problem arises when we want to handle terms with infinite data types.

Regular-language representation of game semantics of second-order recursion free Idealized Algol with finite data types provides algorithms for automatic verification of a range of properties, such as observational-equivalence, approximation, and safety. It has the disadvantage that in the presence of infinite integer data types the obtained automata become infinite state, i.e. regular-languages have infinite summations, thus losing their algorithmic properties. Similarly, large finite data types are likely to make the state-space of the obtained automata so big that it will be practically infeasible for automatic verification. For example, let us consider how we can model the successor function of type $\mathbb{N} \to \mathbb{N}$. One characteristic play in the strategy for this function looks like this:

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

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

In this paper we redefine the (standard) regular-language representation [14] at a more abstract level so that terms with infinite data types can be represented as finite automata, and so various program properties can be checked over them. The idea is to transfer attention from the standard form of automata to what we call symbolic automata. The representation of values constitutes the main difference between these two formalisms. In symbolic automata, instead of assigning concrete values to identifiers occurring in terms, they are left as symbols. Operations involving such identifiers will also be left as symbols. Some of the symbols will be guarded by boolean expressions, which indicate under which conditions these symbols can be performed. For example, symbolic representation of the successor function will be given by the following word: $q^{\langle 2 \rangle} \cdot q^{\langle 1 \rangle} \cdot ?Z^{\langle 1 \rangle} \cdot (Z+1)^{\langle 2 \rangle}$, where a new symbol Z is used to encode the value of the input argument.

This paper represents an extended version of [12]. It is organised as follows. The language we consider here is introduced in Section 2. Symbolic representation of algorithmic game semantics is defined in Section 3. Correctness of the symbolic representation and its suitability for verification of safety properties are shown in Section 4. In Section 5 we discuss some extensions of the language, such as arrays, and how they can be represented in the symbolic model. A prototype tool, which implements this translation, as well as some examples are described in Section 6. In Section 7, we conclude and present some ideas for future work.

1.1. *Related work*

By representing game semantic models as symbolic automata, we obtain a predicate abstraction [17, 7] based method for verification. In [3] it was also developed a predicate abstraction from game semantics. This was enabled by extending the models produced using game semantics such that the state (store) is recorded explicitly in the model by using so-called stateful plays. The state is then abstracted by a set of predicates giving rise to pa (predicate abstraction)-plays. However, in our work we achieved predicate abstraction in a more natural way without changing the game semantic models, and also for terms with infinite data types.

Symbolic techniques, in which data is not represented explicitly but symbolically, have found a number of applications in theoretical computer science. Some interesting examples are symbolic execution and verification of programs [5], symbolic program analysis [6, 4], symbolic operational semantics of process algebras [18], parameterized verification of data independent systems [21, 22], etc.

2. The Language

Idealized Algol (IA) [25] is a well studied language which combines call-by-name λ-calculus with the fundamental imperative features and locally-scoped variables. In this paper we work with its second-order recursion-free fragment (IA_2 for short).

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

Terms are formed by the following grammar:

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

where v ranges over constants of type D. Expression constants are infinite integers and booleans. The standard arithmetic-logic operations op are employed. We have the usual imperative constructs: sequential composition (;), conditional (if), iteration (while), assignment (:=), de-referencing (!), "do nothing" command skip, and $\mathsf{diverge}$ command which causes a program to enter an unresponsive state similar to that caused by an infinite loop. Block-allocated local variables are introduced by a *new* construct, which initializes a variable and makes it local to a given block. The constructor mkvar is used for creating "bad" variables. We have the standard functional constructs for function definition and application. *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, i.e. of the form $x_1 : T_1, \ldots, x_k : T_k$. Typing rules of the language are given in [1, 2].

Language constructs can be also given in a functional form: $; : \mathsf{com} \rightarrow B \rightarrow B$, $\mathsf{if} : \mathsf{expbool} \rightarrow B \rightarrow B \rightarrow B$, $\mathsf{while} : \mathsf{expbool} \rightarrow \mathsf{com} \rightarrow \mathsf{com}$, $:= \, : \mathsf{var}D \rightarrow \mathsf{exp}D \rightarrow \mathsf{com}$, $! : \mathsf{var}D \rightarrow \mathsf{exp}D$, $\mathsf{mkvar_D} : (\mathsf{exp}D \rightarrow \mathsf{com}) \rightarrow \mathsf{exp}D \rightarrow \mathsf{var}D$. A term can be presented in either form as is convenient depending on context. For example, $;\ (M, N) \equiv M\ ;\ N$, $\mathsf{if}(M, N_1, N_2) \equiv \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2$, etc.

The operational semantics of our language is given for terms $\Gamma \vdash M : T$, such that all identifiers in Γ are variables, i.e. $\Gamma = x_1 : \mathsf{var}D_1, \ldots, x_k : \mathsf{var}D_k$. It is defined by a big-step reduction relation:

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

where s, s′ represent the *state* before and after reduction. The state is a function assigning data values to the variables in Γ. We denote by V terms in *canonical form* defined by $V ::= x \mid v \mid \lambda\, x.M \mid \mathsf{skip} \mid \mathsf{mkvar}_D MN$. Reduction rules are standard [1] (see [1, 2] for details). The language is deterministic, so every term can be reduced to at most one canonical form.

Given a term $\Gamma \vdash M : \mathsf{com}$, where all identifiers in Γ are variables, we say that M *terminates* in state s, written $M, \mathrm{s} \Downarrow$, if $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'$ for some state s′. If M is a closed term then we abbreviate the relation $M, \emptyset \Downarrow$ with $M \Downarrow$. We say that a term $\Gamma \vdash M : T$ is an *approximate* of a term $\Gamma \vdash N : T$, denoted by $\Gamma \vdash M \sqsubsetapprox N$, 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 $\Gamma \vdash M \cong N$.

3. Symbolic Game Semantics

We start by introducing a number of syntactic categories necessary for construction of symbolic automata. 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.

[1] Note that the $\mathsf{diverge}$ command is not reducible.

A set of expressions Exp, ranged over by e, is defined as follows:

$$\begin{array}{l} e ::= a \mid b \\ a ::= n \mid X^{int} \mid a \,\mathsf{op}\, a \\ b ::= tt \mid ff \mid X^{bool} \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \end{array}$$

where a ranges over arithmetic expressions ($AExp$), and b over boolean expressions ($BExp$). We use superscripts to denote the data type of a symbolic name X. We will often omit to write them, when they are clear from the context.

Let $\mathcal{A}$ be an alphabet of letters. We define a *symbolic alphabet* $\mathcal{A}^{sym}$ induced by $\mathcal{A}$ as follows:

$$\mathcal{A}^{sym} = \mathcal{A} \cup \{?X, e \mid X \in Sym, e \in Exp\}$$

The letters of the form $?X$ are called *input symbols*. They generate new symbolic names, i.e. $?X$ means $\mathsf{let}\, X = new(W)\, \mathsf{in} \ldots$. 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, i.e. we have:

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

A guarded letter $[b, \alpha\rangle$ means that the symbolic letter α occurs only if the boolean b evaluates to true, i.e. $if\ (b = tt)\ then\ \alpha\ else\ \emptyset$. 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 guarded alphabet $\mathcal{A}^{gu}$ 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.

We now show how IA_2 with infinite integers is interpreted by symbolic automata, which will be denoted by extended regular expressions. For simplicity the translation is defined for terms in β-normal form. If a term has β-redexes, it is first reduced to β-normal form syntactically by substitution. In this setting, arenas in which games are played (types) are represented as *guarded alphabets* of moves, plays of a game as *words* over a guarded alphabet, and strategies for a game as *symbolic automata* (symbolic regular languages) over a guarded alphabet. The symbolic automata and regular languages, denoted by $\mathcal{S}(R)$ and $\mathcal{L}(R)$ respectively, are specified using *extended regular expressions* R. They are defined inductively over finite guarded alphabets $\mathcal{A}^{gu}$ using the following operations:

$$\begin{array}{c} \emptyset \quad \varepsilon \quad \beta \quad R \cdot R' \quad R^* \quad R + R' \quad R \cap R' \\ R\mid_{\mathcal{A}'^{gu}} \quad R[R'/w] \quad R^{\langle \alpha \rangle} \quad R' \,{}_{\mathring{9}\mathcal{B}^{gu}}\, R \quad R \bowtie R' \end{array}$$

where R, R' ranges over extended regular expressions, $\mathcal{A}^{gu}, \mathcal{B}^{gu}$ over finite guarded alphabets, $\beta \in \mathcal{A}^{gu}$, $\alpha \in \mathcal{A}^{sym}$, $\mathcal{A}'^{sym} \subseteq \mathcal{A}^{sym}$ and $w \in \mathcal{A}^{gu*}$.

Constants $\emptyset$, ε and β denote the languages $\emptyset$, $\{\varepsilon\}$ and $\{\beta\}$, respectively. Concatenation $R \cdot R'$, Kleene star R^*, union $R + R'$ and intersection $R \cap R'$ are the standard operations. Restriction $R\mid_{\mathcal{A}'^{gu}}$ replaces all symbolic letters from $\mathcal{A}'^{sym}$ with ε in all words of R, but keeps all boolean conditions. Substitution

$R[R'/w]$ is the language of R where all occurrences of the subword w have been replaced by the words of R'. Given two symbols $\alpha \in \mathcal{A}^{sym}$, $\beta \in \mathcal{A}^{gu}$, $\beta^{\langle\alpha\rangle}$ is a new letter obtained by tagging. If a letter is tagged more than once, we write $(\beta^{\langle\alpha_1\rangle})^{\langle\alpha_2\rangle} = \beta^{\langle\alpha_2,\alpha_1\rangle}$. We define the alphabet $\mathcal{A}^{gu\langle\alpha\rangle} = \{\beta^{\langle\alpha\rangle} \mid \beta \in \mathcal{A}^{gu}\}$. Composition of regular expressions R' defined over $\mathcal{A}^{gu\langle 1\rangle} + \mathcal{B}^{gu\langle 2\rangle}$ and R over $\mathcal{B}^{gu\langle 2\rangle} + \mathcal{C}^{gu\langle 3\rangle}$ is given as follows:

$$\begin{aligned} R' \,{}_{9\mathcal{B}^{gu\langle 2\rangle}}\, R = \{ w\big[& [b \wedge b_1 \wedge b_2 \wedge b_1' \wedge b_2' \wedge \alpha_1 = \alpha_1' \wedge \alpha_2 = \alpha_2', s\,\rangle^{\langle 1\rangle} / \\ & [b_1, \alpha_1\rangle^{\langle 2\rangle} \cdot [b_2, \alpha_2\rangle^{\langle 2\rangle}\big] \mid w \in R, [b_1', \alpha_1'\rangle^{\langle 2\rangle} \cdot [b, s\rangle^{\langle 1\rangle} \cdot [b_2', \alpha_2'\rangle^{\langle 2\rangle} \in R' \} \end{aligned}$$

where R' is a set of words of form $[b_1', \alpha_1'\rangle^{\langle 2\rangle} \cdot [b, s\rangle^{\langle 1\rangle} \cdot [b_2', \alpha_2'\rangle^{\langle 2\rangle}$, such that $[b_1', \alpha_1'\rangle^{\langle 2\rangle}$, $[b_2', \alpha_2'\rangle^{\langle 2\rangle} \in \mathcal{B}^{gu\langle 2\rangle}$ and $[b, s\rangle$ contains only letters from $\mathcal{A}^{gu\langle 1\rangle}$. So all letters of $\mathcal{B}^{gu\langle 2\rangle}$ are removed from the composition, which is defined over the alphabet $\mathcal{A}^{gu\langle 1\rangle} + \mathcal{C}^{gu\langle 3\rangle}$. The shuffle operation of two regular languages is defined as $\mathcal{L}(R) \bowtie \mathcal{L}(R') = \bigcup_{w_1 \in \mathcal{L}(R), w_2 \in \mathcal{L}(R')} w_1 \bowtie w_2$, where $w \bowtie \varepsilon = \varepsilon \bowtie w = w$ and $a \cdot w_1 \bowtie b \cdot w_2 = a \cdot (w_1 \bowtie b \cdot w_2) + b \cdot (a \cdot w_1 \bowtie w_2)$. It is a standard result that any extended regular expression obtained from the operations above denotes a regular language [14, pp. 11–12], which can be recognised by a finite (symbolic) automaton [20].

Each type T is interpreted by a guarded alphabet of moves $\mathcal{A}^{gu}_{[\![T]\!]}$ induced by $\mathcal{A}_{[\![T]\!]}$. The alphabet $\mathcal{A}_{[\![T]\!]}$ contains two kinds of moves: *questions* and *answers*. They are defined as follows [2].

$$\begin{aligned} & \mathcal{A}_{[\![\mathsf{int}]\!]} = \{\ldots, -n, -n+1, \ldots, n, n+1, \ldots\} \qquad \mathcal{A}_{[\![\mathsf{bool}]\!]} = \{tt, f\!f\} \\ & \mathcal{A}_{[\![\mathsf{exp}D]\!]} = \{q\} \cup \mathcal{A}_{[\![D]\!]} \qquad \mathcal{A}_{[\![\mathsf{com}]\!]} = \{run, done\} \\ & \mathcal{A}_{[\![\mathsf{var}D]\!]} = \{read, write(a), a, ok \mid a \in \mathcal{A}_{[\![D]\!]}\} \\ & \mathcal{A}^{gu}_{[\![B_1^{\langle 1\rangle} \to \ldots \to B_k^{\langle k\rangle} \to B]\!]} = \sum_{1 \le i \le k} \mathcal{A}^{gu\,\langle i\rangle}_{[\![B_i]\!]} + \mathcal{A}^{gu}_{[\![B]\!]} \end{aligned}$$

Note that function types are tagged by a superscript ($\langle i\rangle$) in order 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 *moves* (observable actions) that a term of type T can perform. For example, 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, in $\mathcal{A}_{[\![\mathsf{com}]\!]}$ there is a *question* move *run* to initiate a command, and an *answer* move *done* to signal successful termination of a command. For variables, we have *question* moves for writing to the variable, $write(a)$, acknowledged by the *answer* move *ok*, and for reading from the variable, a *question* move *read*, and corresponding to it an *answer* from $\mathcal{A}_{[\![D]\!]}$.

For any (β-normal) term, we define a 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

[2]Here + denotes a disjoint union of alphabets.

$$
\begin{aligned}
&[\![\Gamma, x : B_1^{\langle x,1\rangle} \to \dots B_k^{\langle x,k\rangle} \to \mathsf{exp}D^{\langle x\rangle} \vdash x : B_1^{\langle 1\rangle} \to \dots B_k^{\langle k\rangle} \to \mathsf{exp}D]\!] = \\
&\qquad\qquad q \cdot q^{\langle x\rangle} \cdot \big(\textstyle\sum_{1\le i\le k} R_{B_i}^{\langle x,i\rangle}\big)^* \cdot ?X^{\langle x\rangle} \cdot X \\
&[\![\Gamma, x : B_1^{\langle x,1\rangle} \to \dots B_k^{\langle x,k\rangle} \to \mathsf{com}^{\langle x\rangle} \vdash x : B_1^{\langle 1\rangle} \to \dots B_k^{\langle k\rangle} \to \mathsf{com}]\!] = \\
&\qquad\qquad run \cdot run^{\langle x\rangle} \cdot \big(\textstyle\sum_{1\le i\le k} R_{B_i}^{\langle x,i\rangle}\big)^* \cdot done^{\langle x\rangle} \cdot done \\
&[\![\Gamma, x : B_1^{\langle x,1\rangle} \to \dots B_k^{\langle x,k\rangle} \to \mathsf{var}D^{\langle x\rangle} \vdash x : B_1^{\langle 1\rangle} \to \dots B_k^{\langle k\rangle} \to \mathsf{var}D]\!] = \\
&\qquad \big(read \cdot read^{\langle x\rangle} \cdot \big(\textstyle\sum_{1\le i\le k} R_{B_i}^{\langle x,i\rangle}\big)^* \cdot ?Z^{\langle x\rangle} \cdot Z\big) + \\
&\qquad \big(write(?Z') \cdot write(Z')^{\langle x\rangle} \cdot \big(\textstyle\sum_{1\le i\le k} R_{B_i}^{\langle x,i\rangle}\big)^* \cdot ok^{\langle x\rangle} \cdot ok\big) \\
&R_{\mathsf{exp}D}^{\langle x,i\rangle} = q^{\langle x,i\rangle} \cdot q^{\langle i\rangle} \cdot ?Z^{\langle i\rangle} \cdot Z^{\langle x,i\rangle} \\
&R_{\mathsf{com}}^{\langle x,i\rangle} = run^{\langle x,i\rangle} \cdot run^{\langle i\rangle} \cdot done^{\langle i\rangle} \cdot done^{\langle x,i\rangle} \\
&R_{\mathsf{var}D}^{\langle x,i\rangle} = (read^{\langle x,i\rangle} \cdot read^{\langle i\rangle} \cdot ?Z^{\langle i\rangle} \cdot Z^{\langle x,i\rangle}) + \\
&\qquad (write(?Z')^{\langle x,i\rangle} \cdot write(Z')^{\langle i\rangle} \cdot ok^{\langle i\rangle} \cdot ok^{\langle x,i\rangle})
\end{aligned}
$$

Table 1: Free Identifiers

as a guarded word $[b, w\rangle$, where the boolean b is also called *play condition.* Assumptions about a play (computation) to be feasible are recorded in the 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. So it is desirable for any play to check the consistency (satisfiability) of its play condition. If the play condition is found to be inconsistent, this play is discarded from the final model of the corresponding term. The regular expression for $\Gamma \vdash M : T$ is denoted $[\![\Gamma \vdash M : T]\!]$, and it is defined over the guarded alphabet $\mathcal{A}^{gu}_{[\![\Gamma\vdash T]\!]}$ defined as:

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

Free identifiers $x : T \in \Gamma$ are represented by the copy-cat regular expressions given in Table 1, which contain all possible behaviours of terms of that type. They provide the most general closure of an open program term. For example, $x : \mathsf{exp}D^{\langle x\rangle} \vdash x : \mathsf{exp}D$ is modelled by the word $q \cdot q^{\langle x\rangle} \cdot ?X^{\langle x\rangle} \cdot X$. Its meaning is that Opponent starts the play by asking what is the value of this expression with the move q, and Player responds by playing $q^{\langle x\rangle}$ (i.e. what is the value of the non-local expression x). Then Opponent provides the value of x by using a new symbolic name X, which will be also the value of this expression. Languages $R_B^{\langle x,i\rangle}$ in Table 1 contain plays representing evaluation of the i-th argument of a non-local function x. So when a first-order non-local function is called, it may evaluate any of its arguments, zero or more times, and then it returns any allowable answer from its result type.

Note that whenever an input symbol $?X$ is met in a play, a new symbolic name is created, which binds all occurrences of X that follow in the play until a new $?X$ is met. For example, $[\![f : \mathsf{expint}^{\langle f,1\rangle} \to \mathsf{expint}^{\langle f\rangle} \vdash f : \mathsf{expint}^{\langle 1\rangle} \to \mathsf{expint}]\!] = q \cdot q^{\langle f\rangle} \cdot \big(q^{\langle f,1\rangle} \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot Z^{\langle f,1\rangle}\big)^* \cdot ?X^{\langle f\rangle} \cdot X$ is a model for a non-local function f which may evaluate its argument zero or more times. The

$[\![\Gamma \vdash v : \mathsf{exp}D]\!] = q \cdot v$
$[\![\Gamma \vdash \mathsf{skip} : \mathsf{com}]\!] = run \cdot done$
$[\![\Gamma \vdash \mathsf{c}(M_1, \ldots, M_k) : B']\!] = [\![\Gamma \vdash M_1 : B_1^{\langle 1 \rangle}]\!] \,{}_{9}^{\circ}{}_{\mathcal{A}^{gu\,\langle 1 \rangle}_{[\![B_1]\!]}} \cdots$
$\cdots\ [\![\Gamma \vdash M_k : B_k^{\langle k \rangle}]\!] \,{}_{9}^{\circ}{}_{\mathcal{A}^{gu\,\langle k \rangle}_{[\![B_k]\!]}} [\![\mathsf{c} : B_1^{\langle 1 \rangle} \times \ldots B_k^{\langle k \rangle} \to B']\!]$
$[\![\Gamma \vdash MN : T]\!] = [\![\Gamma \vdash N : B^{\langle 1 \rangle}]\!] \,{}_{9}^{\circ}{}_{\mathcal{A}^{gu\,\langle 1 \rangle}_{[\![B]\!]}} [\![\Gamma \vdash M : B^{\langle 1 \rangle} \to T]\!]$
$[\![\Gamma \vdash \mathsf{new}_D\, x := v \,\mathsf{in}\, M : B]\!] = \big([\![\Gamma, x\!:\!\mathsf{var}D \vdash M]\!] \cap (\gamma_v^x \bowtie \mathcal{A}^*_{[\![\Gamma \vdash B]\!]^{gu}})\big)\vert_{\mathcal{A}^{gu\,\langle x \rangle}_{[\![\mathsf{var}D]\!]}}$
$\gamma_v^x = (read^{\langle x \rangle} \cdot v^{\langle x \rangle})^* \cdot \big(write(?Z)^{\langle x \rangle} \cdot ok^{\langle x \rangle} \cdot (read^{\langle x \rangle} \cdot Z^{\langle x \rangle})^*\big)^*$

Table 2: Language terms

play corresponding to f which evaluates its argument two times 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. Note that letters tagged with f represent the actions of calling and returning from the function, while letters tagged with $f.1$ are the actions caused by evaluating the first argument of f.

In Table 2 terms are interpreted by regular expressions describing their sets of complete plays. An integer or boolean constant is modeled by a play where the initial question q is answered by the value of that constant. The only play for skip responds to *run* with *done*. A composite term $\mathsf{c}(M_1, \ldots, M_k)$ consisting 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 a regular expression for 'c'. The representation of language constructs 'c' is given in Table 3. For example, the regular expression for any arithmetic-logic operation op asks for values of the arguments, and after obtaining them by symbolic names Z and Z' responds by performing the operation $(Z\mathsf{op}Z')$. In the case of branching, if the value of the first argument given by Z is true then its second argument is run, otherwise if Z is false then its third argument is run. In the definition for local variables given in Table 2, a 'cell' regular expression γ_v^x is used to impose the good variable behaviour on the local variable x. It responds to each *write*$(-)$ with *ok*, and plays the most recently written value in response to *read*, or if no value has been written yet then answers the *read* with the initial value v. Notice that all symbols used in Tables 1,2,3 are of data type D, except the symbol Z in if and while constructs, which is of data type *bool*.

We define an effective alphabet of a regular expression to be the set of all letters appearing in the language denoted by that regular expression. Then by inspecting all definitions given in Tables 1,2,3, it is easy to show the following result.

Proposition 1. *For any term* $\Gamma \vdash M : T$*, the effective alphabet of* $[\![\Gamma \vdash M : T]\!]$ *is a finite subset of* $\mathcal{A}^{gu}_{[\![\Gamma \vdash T]\!]}$.

Any term $\Gamma \vdash M : T$ from IA_2 with infinite integers is interpreted by

$$
\begin{array}{|l|}
\hline
[\![\mathsf{op} : \mathsf{exp}D_1^{\langle 1\rangle} \times \mathsf{exp}D_2^{\langle 2\rangle} \to \mathsf{exp}D]\!] = q \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot q^{\langle 2\rangle} \cdot ?Z'^{\langle 2\rangle} \cdot (Z \,\mathsf{op}\, Z') \\
[\![; \ : \mathsf{com}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \to \mathsf{com}]\!] = run \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done \\
[\![\mathsf{if} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \times \mathsf{com}^{\langle 3\rangle} \to \mathsf{com}]\!] = [tt, run\rangle \cdot [tt, q^{\langle 1\rangle}\rangle \cdot [tt, ?Z^{\langle 1\rangle}\rangle \cdot \\
\qquad\qquad \big([Z, run^{\langle 2\rangle}\rangle \cdot [tt, done^{\langle 2\rangle}\rangle + [\neg Z, run^{\langle 3\rangle}\rangle \cdot [tt, done^{\langle 3\rangle}\rangle\big) \cdot [tt, done\rangle \\
[\![\mathsf{while} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \to \mathsf{com}]\!] = [tt, run\rangle \cdot [tt, q^{\langle 1\rangle}\rangle \cdot [tt, ?Z^{\langle 1\rangle}\rangle \cdot \\
\qquad\qquad \big([Z, run^{\langle 2\rangle}\rangle \cdot [tt, done^{\langle 2\rangle}\rangle \cdot [tt, q^{\langle 1\rangle}\rangle \cdot [tt, ?Z^{\langle 1\rangle}\rangle\big)^* \cdot [\neg Z, done\rangle \\
[\![:= : \mathsf{var}D^{\langle 1\rangle} \times \mathsf{exp}D^{\langle 2\rangle} \to \mathsf{com}]\!] = run \cdot q^{\langle 2\rangle} \cdot ?Z^{\langle 2\rangle} \cdot write(Z)^{\langle 1\rangle} \cdot ok^{\langle 1\rangle} \cdot done \\
[\![! : \mathsf{var}D^{\langle 1\rangle} \to \mathsf{exp}D]\!] = q \cdot read^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot Z \\
\hline
\end{array}
$$

Table 3: Language constructs

extended regular expression without infinite summations defined over finite alphabet. So the following is immediate.

Theorem 1. *For any IA_2 term, the set $\mathcal{L}[\![\Gamma \vdash M : T]\!]$ is a symbolic regular-language without infinite summations over finite alphabet. Moreover, a finite symbolic automata $\mathcal{S}[\![\Gamma \vdash M : T]\!]$ which recognizes it is effectively constructible.*

Proof. The proof is by induction on the structure of $\Gamma \vdash M : T$.

An automaton is a tuple (Q, i, δ, F) where Q is the finite set of states, $i \in Q$ is the initial state, δ is the transition function, and $F \subseteq Q$ is the set of final states. We now introduce two auxiliary operations. Let $A' = (Q', i', \delta', F')$ be an automaton, then $A = rename(A', tag)$ is defined as:

$Q = Q' \qquad i = i' \qquad F = F'$

$\delta = \{q_1 \xrightarrow{[b,m\rangle} q_2 \in \delta' \mid q_1 \neq i', q_2 \notin F'\} +$
$\quad \{i' \xrightarrow{[b,m^{\langle tag\rangle}\rangle} q \mid i' \xrightarrow{[b,m\rangle} q \in \delta'\} + \{q_1 \xrightarrow{[b,m^{\langle tag\rangle}\rangle} q_2 \mid q_1 \xrightarrow{[b,m\rangle} q_2 \in \delta', q_2 \in F'\}$

Let $A_1 = (Q_1, i_1, \delta_1, F_1)$ and $A_2 = (Q_2, i_2, \delta_2, F_2)$ be two automata, such that all transitions going out of i_2 and going to a state from F_2 are tagged with tag. Define $A = compose(A_1, A_2, tag)$ as follows:

$Q = Q_1 + Q_2 \backslash \{i_2, F_2\} \qquad i = i_1 \qquad F = F_1$

$\delta = \{q_1 \xrightarrow{[b,m\rangle} q_1' \in \delta_1 \mid m \neq n^{\langle tag\rangle}\} + \{q_2 \xrightarrow{[b,m\rangle} q_2' \in \delta_2 \mid m \neq n^{\langle tag\rangle}\} +$
$\quad \{q_1 \xrightarrow{[b_1 \wedge b_2 \wedge m_1 = m_2, \varepsilon\rangle} q_2' \mid q_1 \xrightarrow{[b_1, m_1^{\langle tag\rangle}\rangle} q_1' \in \delta_1, i_2 \xrightarrow{[b_2, m_2^{\langle tag\rangle}\rangle} q_2' \in \delta_2,$
$\quad \{m_1, m_2\}$ are questions$\} + \{q_2 \xrightarrow{[b_1 \wedge b_2 \wedge m_1 = m_2, \varepsilon\rangle} q_1' \mid q_1 \xrightarrow{[b_1, m_1^{\langle tag\rangle}\rangle} q_1' \in \delta_1,$
$\quad q_2 \xrightarrow{[b_2, m_2^{\langle tag\rangle}\rangle} q_2' \in \delta_2,\ q_2' \in F_2, \{m_1, m_2\}$ are answers$\}$

Let A_M, A_N, and $A_{\fatsemi}$ be automata representing $\Gamma \vdash M$, $\Gamma \vdash N$, and construct ; (see Table 3), respectively. The unique automaton representing $\Gamma \vdash M \,;\, N$ is defined as:

$$A_{M\,;\,N} = compose(compose(A_{\fatsemi}, rename(A_M, 1), 1), rename(A_N, 2), 2)$$

The other cases for constructs are similar.

The automaton $A = (Q, i, \delta, F)$ for $[\![\Gamma \vdash \mathsf{new}_D\, x := v \,\mathsf{in}\, M]\!]$ is constructed in two stages. First we eliminate x-tagged symbolic letters from the automaton

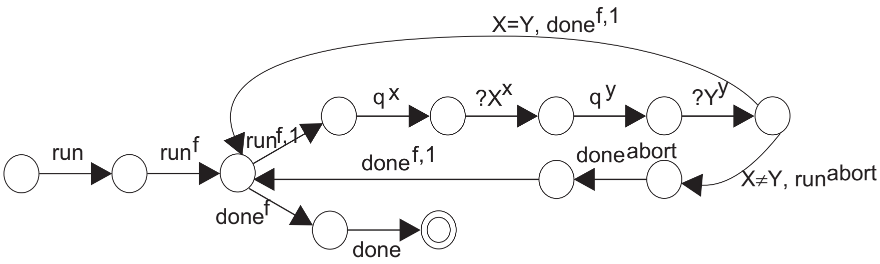

Figure 1: The symbolic representation of the strategy for M_1.

$A_M = (Q_M, i_M, \delta_M, F_M)$, which represents $[\![\Gamma, x : \mathsf{var}D \vdash M]\!]$, by replacing them with ε, thus obtaining $A_\varepsilon = (Q_\varepsilon, i_\varepsilon, \delta_\varepsilon, F_\varepsilon)$. We also introduce a new symbolic name X to keep track of what changes to x are made by each x-tagged move.

$$Q_\varepsilon = Q_M \qquad i_\varepsilon = i_M \qquad F_\varepsilon = F_M$$

$$\begin{aligned}\delta_\varepsilon = &\{ i_M \xrightarrow{[?X = v \wedge b, m\rangle} q \mid i_M \xrightarrow{[b,m\rangle} q \in \delta_M \} + \\ &\{ q_1 \xrightarrow{[b,m\rangle} q_2 \mid q_1 \xrightarrow{[b,m\rangle} q_2 \in \delta_M, m \not\in \{ write(a)^{\langle x \rangle}, ok^{\langle x \rangle}, read^{\langle x \rangle}, a^{\langle x \rangle} \} \} \\ &\{ q_1 \xrightarrow{[?X = a' \wedge b_1 \wedge b_2, \varepsilon\rangle} q_2 \mid \exists\, q. (q_1 \xrightarrow{[b_1, write(a')^{\langle x \rangle}\rangle} q \in \delta_M, q \xrightarrow{[b_2, ok^{\langle x \rangle}\rangle} q_2 \in \delta_M) \} \\ &\{ q_1 \xrightarrow{[a' = X \wedge b_1 \wedge b_2, \varepsilon\rangle} q_2 \mid \exists\, q. (q_1 \xrightarrow{[b_1, read^{\langle x \rangle}\rangle} q \in \delta_M, q \xrightarrow{[b_2, a'^{\langle x \rangle}\rangle} q_2 \in \delta_M) \}\end{aligned}$$

The final automaton is obtained by eliminating ε-letters from A_ε. Note that conditions associated to ε-letters are not removed.

$$Q = Q_\varepsilon \qquad i = i_\varepsilon \qquad F = F_\varepsilon$$

$$\begin{aligned}\delta = &\big(\{ \delta_\varepsilon \backslash \{ q_1 \xrightarrow{[b,\varepsilon\rangle} q_2 \mid q_1, q_2 \in Q_\varepsilon \} \big) + \\ &\{ q_1 \xrightarrow{[b \wedge b_\varepsilon, m\rangle} q_2 \mid \exists\, q' \in Q_\varepsilon . (q_1 \xrightarrow{[b_\varepsilon, \varepsilon\rangle^*} q', q' \xrightarrow{[b,m\rangle} q_2) \}\end{aligned}$$

We write $q_1 \xrightarrow{[b_\varepsilon, \varepsilon\rangle^*} q_2$ if q_2 is reachable from q_1 by a series of ε-transitions $[b_1, \varepsilon\rangle, \ldots, [b_k, \varepsilon\rangle$, where $b_\varepsilon = b_1 \wedge \ldots b_k$. □

Example 1. Consider the term M_1:

$$f : \mathsf{com}^{f,1} \to \mathsf{com}^f, abort : \mathsf{com}^{abort}, x : \mathsf{expint}^x, y : \mathsf{expint}^y \vdash f\big(\mathsf{if}\ (x \neq y)\ \mathsf{then}\ abort\big) : \mathsf{com}$$

in which f is a non-local procedure, and x, y are non-local expressions.

The strategy for this term represented as a finite symbolic automaton is shown in Figure 1. The model illustrates only the possible behaviors of this term: the non-local procedure f may call its argument, zero or more times, then the term terminates successfully with *done*. If f calls its argument, arbitrary values for x and y are read from the environment by using symbols X and Y. If they are different ($X \neq Y$), then the abort command is executed. The standard regular-language representation [14] of M_1, where concrete values are employed, is given in Figure 2. It represents an infinite-state automaton, and so it is not suitable for automatic verification (model checking). Note that, the values for non-local expressions x and y can be any possible integer. □

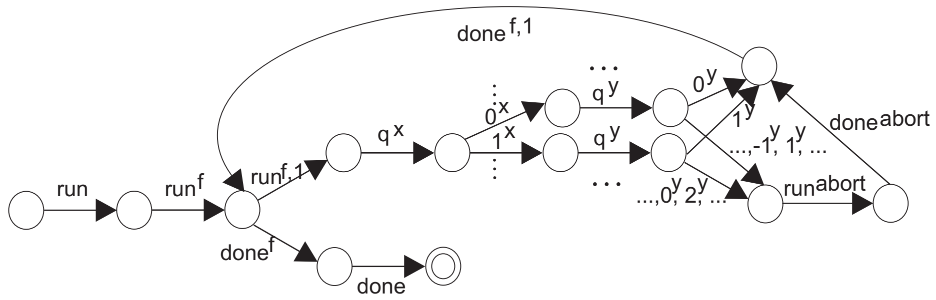

Figure 2: The standard representation of the strategy for M_1.

4. Formal Properties

In [14, pp. 28–32], it was shown the correctness of the standard regular-language representation for finitary IA_2 by showing that it is isomorphic to the game semantics model [1]. As a corollary, it was obtained that the standard regular-language representation is fully abstract.

Let $[\![\Gamma \vdash M : T]\!]^{CR}$ denotes the set of all complete plays in the strategy for a term $\Gamma \vdash M : T$ from IA_2 with infinite integers obtained as in [14], where concrete values in moves and infinite summations in regular expressions are used. Suppose that there is a special free identifier abort of type $\mathsf{com}^{\mathsf{abort}}$. We say that a term $\Gamma \vdash M$ is *safe* iff $\Gamma \backslash \mathsf{abort} \vdash M[\mathsf{skip}/\mathsf{abort}] \sqsubseteq M[\mathsf{diverge}/\mathsf{abort}]$; otherwise we say that a term is *unsafe*. Since the standard regular-language game semantics is fully abstract, the following result is easy to show (see also [8]).

Proposition 2. *A term $\Gamma \vdash M$ is safe iff $[\![\Gamma \vdash M]\!]$ does not contain any play with moves from $\mathcal{A}^{abort}_{[\![\mathsf{com}]\!]}$, which we call unsafe plays.*

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.

Let *Eval* be the set of evaluations, i.e. the set of total functions from W to $\mathcal{A}_{[\![\mathsf{int}]\!]} \cup \mathcal{A}_{[\![\mathsf{bool}]\!]}$. We use ρ to range over *Eval*. So we have $\rho(X^D) \in \mathcal{A}_{[\![D]\!]}$ for any evaluation $\rho \in Eval$ and $X^D \in W$. Given a word of symbolic letters w, let $\rho(w)$ be a word where every symbolic name is replaced by the corresponding concrete value as defined by ρ. Given a guarded word $[b, w\rangle$, define $\rho([b, w\rangle) = \rho(w)$ if $\rho(b) = tt$; otherwise $\rho([b, w\rangle) = \emptyset$ if $\rho(b) = f\!f$. The concretization of a symbolic regular-language over a guarded alphabet is defined as follows: $\gamma\,\mathcal{L}(R) = \{\rho[b, w\rangle \mid [b, w\rangle \in \mathcal{L}(R), \rho \in Eval\}$. Let $[\![\Gamma \vdash M : T]\!]^{SR} = \mathcal{L}[\![\Gamma \vdash M : T]\!]$ be the strategy obtained as in Section 3, where symbols instead of concrete values are used.

Theorem 2. *For any IA_2 term*

$$\gamma\,[\![\Gamma \vdash M : T]\!]^{SR} \;=\; [\![\Gamma \vdash M : T]\!]^{CR}$$

Proof. By induction on the typing rules. Definitions of constants are the same.

Consider the case of free identifiers.

$$\begin{aligned}\gamma[\![x : \mathsf{exp}D^{\langle x\rangle} \vdash x : \mathsf{exp}D]\!]^{SR} &= \gamma\{q \cdot q^{\langle x\rangle} \cdot X^{D\,\langle x\rangle} \cdot X^D\} \\ &= \{q \cdot q^{\langle x\rangle} \cdot \rho(X^D)^{\langle x\rangle} \cdot \rho(X^D) \mid \rho : \{X^D\} \to \mathcal{A}_{[\![D]\!]}\} \\ &= \{q \cdot q^{\langle x\rangle} \cdot v^{\langle x\rangle} \cdot v \mid v \in \mathcal{A}_{[\![D]\!]}\} = [\![x : \mathsf{exp}D^{\langle x\rangle} \vdash x : \mathsf{exp}D]\!]^{CR}\end{aligned}$$

Let us consider the branching construct.

$$\begin{aligned}&\gamma[\![\mathsf{if} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \times \mathsf{com}^{\langle 3\rangle} \to \mathsf{com}]\!]^{SR} = \\ &\quad \gamma\{run \cdot q^{\langle 1\rangle} \cdot Z^{\langle 1\rangle} \cdot \big([Z, run^{\langle 2\rangle}\rangle \cdot done^{\langle 2\rangle} + [\neg Z, run^{\langle 3\rangle}\rangle \cdot done^{\langle 3\rangle}\big) \cdot done\} = \\ &\quad \{run \cdot q^{\langle 1\rangle} \cdot \rho(Z)^{\langle 1\rangle} \cdot \big([\rho(Z), run^{\langle 2\rangle}\rangle \cdot done^{\langle 2\rangle} + [\neg\rho(Z), run^{\langle 3\rangle}\rangle \cdot done^{\langle 3\rangle}\big) \\ &\qquad\qquad \cdot done \mid \rho : \{Z\} \to \{tt, ff\}\} = \\ &\quad \{run \cdot q^{\langle 1\rangle} \cdot v^{\langle 1\rangle} \cdot \big((if\ (v)\ then\ run^{\langle 2\rangle}\ else\ \emptyset) \cdot done^{\langle 2\rangle} + \\ &\qquad (if\ (\neg v)\ then\ run^{\langle 3\rangle}\ else\ \emptyset) \cdot done^{\langle 3\rangle}\big) \cdot done \mid v \in \{tt, ff\}\} = \\ &\quad \{run \cdot \big(q^{\langle 1\rangle} \cdot tt^{\langle 1\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} + q^{\langle 1\rangle} \cdot ff^{\langle 1\rangle} \cdot run^{\langle 3\rangle} \cdot done^{\langle 3\rangle}\big) \cdot done\} = \\ &\quad [\![\mathsf{if} : \mathsf{expbool}^{\langle 1\rangle} \times \mathsf{com}^{\langle 2\rangle} \times \mathsf{com}^{\langle 3\rangle} \to \mathsf{com}]\!]^{CR}\end{aligned}$$

The other cases as well as composition are similar to prove. □

As a corollary we obtain the following result.

Theorem 3. *$[\![\Gamma \vdash M : T]\!]^{SR}$ is safe iff $[\![\Gamma \vdash N : T]\!]^{CR}$ is safe.*

By Proposition 2 and Theorem 3 it follows that a term is safe if its symbolic regular-language semantics is safe. Since symbolic automata are finite state, it follows that we can use model-checking to verify safety of IA_2 terms with infinite data types.

In order to verify safety of a term we need to check whether the symbolic automaton representing a term contains unsafe plays. We use an external SMT solver Yices [3] [13] to determine consistency of the play conditions of the discovered unsafe plays. If some play condition is consistent, i.e. there exists an evaluation ρ that makes the play condition true, the corresponding unsafe play is feasible and it is reported as a genuine counter-example. By replacing symbolic names in the play with the concrete values as defined by ρ, we will obtain a concrete genuine counter-example corresponding to an unsafe computation of the term. If the condition of an unsafe play is found to be inconsistent, then the play is considered as infeasible, and so discarded from the model.

Given a finite-state symbolic model of a term, we use the following procedure to check its safety. The breadth-first search algorithm is applied to find the shortest unsafe play in the model. If its play condition is found to be consistent, the procedure terminates. Otherwise, the next shortest unsafe play is found and tested for consistency.

Example 2. The term M_1 from Example 1 is abort-unsafe, with the following counter-example:

$$run \cdot run^f \cdot run^{f,1} \cdot q^x \cdot X^x \cdot q^y \cdot Y^y \cdot [X \neq Y, run^{abort}\rangle \cdot done^{abort} \cdot done^{f,1} \cdot done^f \cdot done$$

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

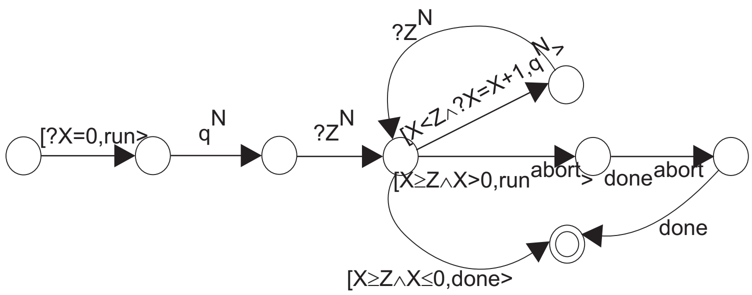

Figure 3: The strategy for M_2.

The consistency of the play condition is established by instructing Yices to check the formula:

$$(\mathit{define}\ X :: \mathit{int})$$
$$(\mathit{define}\ Y :: \mathit{int})$$
$$(\mathit{assert}\ (/ =\ X\ \ Y))$$

The following satisfiable assignments to symbols are reported: $X = 1$ and $Y = 2$, yielding a concrete unsafe play:

$$run \cdot run^f \cdot run^{f,1} \cdot q^x \cdot 1^x \cdot q^y \cdot 2^y \cdot run^{abort} \cdot done^{abort} \cdot done^{f,1} \cdot done^f \cdot done$$

□

Example 3. Consider the term M_2:

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

The strategy for this term (suitably adapted for readability) is given in Figure 3. Observe that the term communicates with its environment using non-local identifiers N and abort. So in the model will only be represented actions of N and abort. Notice that each time the term (Player) asks for a value of N with the move q^N, the environment (Opponent) provides a new fresh value $?Z$ for it. The symbol X is used to keep track of the current value of x. Whenever a new value for N is provided, the term has three possible options depending on the current values of Z 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.

The shortest unsafe play found in the model is:

$$[X = 0, run\rangle \cdot q^N \cdot Z^N \cdot [X \geq Z \wedge X > 0, run^{abort}\rangle \cdot done^{abort} \cdot done$$

But the play condition for it, $X = 0 \wedge X \geq Z \wedge X > 0$, is inconsistent. The next unsafe play is:

$$[X_1 = 0, run\rangle \cdot q^N \cdot {Z_1}^N \cdot [X_1 < Z_1 \wedge X_2 = X_1 + 1, q^N\rangle \cdot {Z_2}^N \cdot$$
$$[X_2 \geq Z_2 \wedge X_2 > 0, run^{abort}\rangle \cdot done^{abort} \cdot done$$

Now Yices reports that the condition for this play is satisfiable, yielding a possible assignment of concrete values to symbols that makes the condition true: $X_1 = 0$, $Z_1 = 1$, $X_2 = 1$, $Z_2 = 0$. So it is a genuine counter-example, such that one corresponding concrete unsafe play is:

$$run \cdot q^N \cdot 1^N \cdot q^N \cdot 0^N \cdot run^{abort} \cdot done^{abort} \cdot done$$

This play corresponds to a computation which runs the body of while exactly once.

Let us modify the M_2 term as follows

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

where $k > 0$ is any positive integer. The model for this modified term is the same as shown in Figure 3, except that conditions associated with letters run^{abort} (resp., $done$) are $X \geq Z \wedge X > k$ (resp., $X \geq Z \wedge X \leq k$). In this case the $(k+1)$-shortest unsafe plays in the model are found to be inconsistent. The first consistent unsafe play corresponds to executing the body of while $(k+1)$-times, and one possible concrete representation of it (as generated by Yices) is:

$$run \cdot q^N \cdot 1^N \cdot q^N \cdot 2^N \cdot \ldots \cdot q^N \cdot (k+1)^N \cdot q^N \cdot 0^N \cdot run^{abort} \cdot done^{abort} \cdot done$$

$\Box$

Note that the procedure described above may diverge for safe terms. As a simple example, we can consider a slightly modified version of the term M_2:

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

It is a safe term, since the guard of ‘if’ statement will always evaluate to false. However, our procedure will continually report unsafe plays, whose play conditions will be inconsistent (unsatisfiable).

5. Handling arrays

We now extend the language with arrays of length $k > 0$. They can be handled in two ways. Firstly, we can introduce arrays of fixed length as syntactic sugar by using existing term formers. An array $x[k]$, where k is a fixed positive integer, is represented as a set of k distinct variables $x[0]$, $x[1]$, …, $x[k-1]$, such that we will use the following abbreviations:

$$\begin{array}{l} \mathsf{new}_D\ x[k] := v \mathsf{\ in\ } M \ \equiv \\ \quad \mathsf{new}_D\ x[0] := v \mathsf{\ in} \\ \quad \ldots \\ \quad\quad \mathsf{new}_D\ x[k-1] := v \mathsf{\ in\ } M \\ x[E] \ \equiv \\ \quad \mathsf{if\ } E = 0 \mathsf{\ then\ } x[0] \mathsf{\ else} \\ \quad \ldots \\ \quad\quad \mathsf{if\ } E = k-1 \mathsf{\ then\ } x[k-1] \mathsf{\ else\ skip\ (abort)} \end{array}$$

If we want to verify whether array out-of-bounds errors are present in the term, i.e. there is an attempt to access elements out of the bounds of an array, we execute abort instead of skip when $E \geq k$. This approach for handling arrays is taken by the standard representation of game semantics [14, 10].

Secondly, since we work with symbols we can have more efficient representation of arrays with unfixed (arbitrary) length. While in the first approach the length of an array k must be a concrete positive integer, in the second approach k can be represented by a symbol with an initial constraint $k > 0$. We use the support that Yices provides for arrays by enabling: function definitions, function updates, and lambda expressions. For each local array $x[k] : \mathsf{var}D$, we can define in Yices a function symbol X of type $int \to D$ as:

$$(\mathit{define}\ X :: (\to int\ \ D))$$

The function symbol X can be initialized and updated as follows:

$$\begin{array}{l}(\mathit{lambda}\ (\mathit{index} :: int)\ \mathit{val}) \\ (\mathit{update}\ X\ (\mathit{index})\ \mathit{val})\end{array}$$

For example, a local array $x[k] : \mathsf{varint}$ initialized to 0 is represented in Yices as:

$$\begin{array}{l}(\mathit{define}\ X :: (\to int\ \ int)) \\ (\mathit{assert}\ (=\ X\ (\mathit{lambda}\ (j :: int)\ 0)))\end{array}$$

In this approach, symbolic representation of a non-local array is as follows.

$$\begin{array}{l}[\![\Gamma, x[k] \vdash x[E] : \mathsf{var}D]\!] = [\![\Gamma \vdash E : \mathsf{expint}^{\langle 1\rangle}]\!] \,{}^{\circ}_{9}{}_{\mathcal{A}^{gu\,\langle 1\rangle}_{[\![\mathsf{expint}]\!]}}\, [\![\Gamma, x[k] \vdash x[-] : \mathsf{var}D]\!] \\ [\![\Gamma, x[k] \vdash x[-] : \mathsf{var}D]\!] = (\mathit{read} \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot [Z < k, \mathit{read}^{\langle x[Z]\rangle}\rangle \cdot ?Z'^{\langle x[Z]\rangle} \cdot Z') \\ \qquad +(\mathit{write}(?Z') \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot [Z < k, \mathit{write}(Z')^{\langle x[Z]\rangle}\rangle \cdot \mathit{ok}^{\langle x[Z]\rangle} \cdot \mathit{ok})\end{array} \tag{1}$$

We can see that a new symbolic name Z is used to represent the index of the array element that needs to be de-referenced or assigned to.

If we also want to check for array out-of-bounds errors, we extend this interpretation by including plays that perform moves associated with abort command when $Z \geq k$. In this case, the symbolic interpretation of arrays will be as follows:

$$\begin{array}{l}[\![\Gamma, x[k] \vdash x[-] : \mathsf{var}D]\!] = \big(\mathit{read} \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot ([Z < k, \mathit{read}^{\langle x[Z]\rangle}\rangle \cdot ?Z'^{\langle x[Z]\rangle} \cdot Z' \\ \qquad\qquad + [Z \geq k, \mathit{run}^{\langle abort\rangle}\rangle \cdot \mathit{done}^{\langle abort\rangle} \cdot 0)\,\big) \\ \qquad +\big(\mathit{write}(?Z') \cdot q^{\langle 1\rangle} \cdot ?Z^{\langle 1\rangle} \cdot ([Z < k, \mathit{write}(Z')^{\langle x[Z]\rangle}\rangle \cdot \mathit{ok}^{\langle x[Z]\rangle} \cdot \mathit{ok} \\ \qquad\qquad + [Z \geq k, \mathit{run}^{\langle abort\rangle}\rangle \cdot \mathit{done}^{\langle abort\rangle} \cdot \mathit{ok})\big)\end{array} \tag{2}$$

So when we have an array in a term, we can interpret it either by using (1) in the case that we do not want to verify array out-of-bounds errors, or by using (2) in the case that we want to check such errors.

The automaton A for $[\![\Gamma \vdash \mathsf{new}_D\, x[k] := v \,\mathsf{in}\, M]\!]$, where A_M represents $[\![\Gamma, x[k] \vdash M]\!]$, is obtained as follows. We first construct A_ε by eliminating x-tagged moves from A_M. Similarly as with local variables, we use a new function

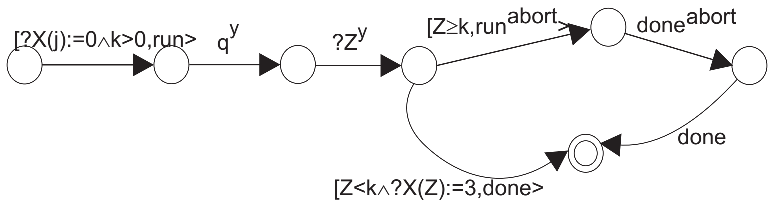

Figure 4: The symbolic model for M_3.

symbol X of type $int \rightarrow D$ to keep track of what changes to array x are made by each x-tagged move.

$$
\begin{aligned}
& Q_\varepsilon = Q_M \qquad i_\varepsilon = i_M \qquad F_\varepsilon = F_M \\
& \delta_\varepsilon = \{ i_M \xrightarrow{[?X(j):=v \wedge k>0 \wedge b, m\rangle} q \mid i_M \xrightarrow{[b,m\rangle} q \in \delta_M \} + \\
& \{ q_1 \xrightarrow{[b,m\rangle} q_2 \mid q_1 \xrightarrow{[b,m\rangle} q_2 \in \delta_M, m \notin \{ write(a)^{\langle x[a'] \rangle}, ok^{\langle x[a'] \rangle}, read^{\langle x[a'] \rangle}, a^{\langle x[a'] \rangle} \} \} \\
& \{ q_1 \xrightarrow{[?X(a'):=a \wedge b_1 \wedge b_2, \varepsilon\rangle} q_2 \mid \exists\, q.(q_1 \xrightarrow{[b_1, write(a)^{\langle x[a'] \rangle}\rangle} q \in \delta_M, q \xrightarrow{[b_2, ok^{\langle x[a'] \rangle}\rangle} q_2 \in \delta_M) \} \\
& \{ q_1 \xrightarrow{[a=X(a') \wedge b_1 \wedge b_2, \varepsilon\rangle} q_2 \mid \exists\, q.(q_1 \xrightarrow{[b_1, read^{\langle x[a'] \rangle}\rangle} q \in \delta_M, q \xrightarrow{[b_2, a^{\langle x[a'] \rangle}\rangle} q_2 \in \delta_M) \}
\end{aligned}
$$

We use $?X(j) := v$ to mean that a new function symbol X is defined and initialized to v for all its arguments, while $?X(a') := a$ means that the current function symbol X is updated at argument a' to a. The final automaton A is generated by removing ε-letters from A_ε, similarly as it was done for the case of new_D in Theorem 1.

Example 4. Consider the term M_3:

$$y : \mathsf{expint}^y, abort : \mathsf{com}^{abort} \vdash \mathsf{new_{int}}\, x[k] := 0 \text{ in } x[y] := 3 : \mathsf{com}$$

The symbolic model of this term is given in Figure 4, where the array x is interpreted by (2). The shortest unsafe play is:

$$[X(j) := 0 \wedge k > 0, run\rangle \cdot q^y \cdot Z^y \cdot [Z \geq k, run^{abort}\rangle \cdot done^{abort} \cdot done$$

Its play condition is satisfiable for the evaluation: $k = 1$, $Z = 1$, yielding an unsafe computation where the value 1 is read from y and the length of the array x is 1. So this represents a computation in which an array out-of-bounds error occurs. □

6. Implementation

We have developed a prototype tool in Java, called Symbolic GameChecker, which automatically converts an IA_2 term with integers into a symbolic automaton which represents its game semantics. The model is then used to verify safety of the term. Further examples as well as detailed reports of how they execute

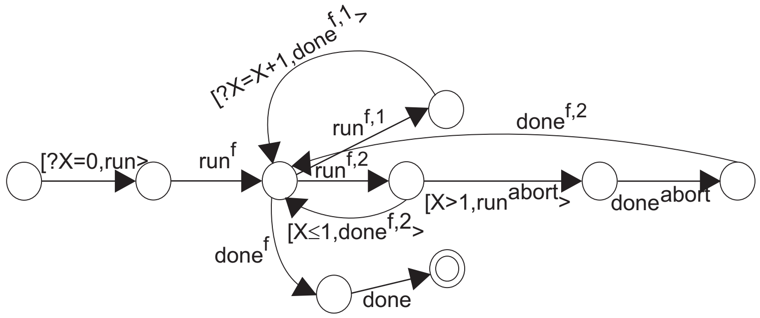

Figure 5: The symbolic model for the procedural term.

on Symbolic GameChecker are available from:
`http://www.dcs.warwick.ac.uk/~aleks/symbolicgc.htm`.

Along with the tool we have also implemented in Java our own library of classes for working with symbolic automata. We could not just reuse some of the existing libraries for finite-state automata, due to the specific nature of symbolic automata we use. The symbolic automata generated by the tool is checked for safety. We use the breadth-first search algorithm to find the shortest unsafe play in the model. Then an external SMT solver Yices is called to check consistency of its condition. If the condition is found to be consistent, the unsafe play is reported; otherwise we search for another unsafe play. If no unsafe play is discovered or all unsafe plays are found to be inconsistent, then the term is deemed safe. The tool also uses a simple forward reachability algorithm to remove all unreachable states of a symbolic automaton.

6.1. A procedural term

Consider the term:

$$f : \mathsf{com}^{f,1} \to \mathsf{com}^{f,2} \to \mathsf{com}^{f}, abort : \mathsf{com}^{abort} \vdash \mathsf{new}_{int}\, x := 0 \,\mathsf{in} \\ f\big(x := x+1, \mathsf{if}\ (x > 1)\ \mathsf{then}\ abort\big) : \mathsf{com}$$

in which f is a non-local procedure.

The symbolic model for this term is given in Figure 5. The non-local procedure f may call its arguments, zero or more times, in any order, and then terminates successfully. The shortest counter-example is:

$$[X = 0, run\rangle \cdot run^{f} \cdot run^{f,2} \cdot [X > 1, run^{abort}\rangle \cdot done^{abort} \cdot done^{f,2} \cdot done^{f} \cdot done$$

Its play condition ($X = 0 \wedge X > 0$) is inconsistent, so it is discarded. The next found counter-example is:

$$[X_1 = 0, run\rangle \cdot run^{f} \cdot run^{f,1} \cdot [X_2 = X_1 + 1, done^{f,1}\rangle \cdot run^{f,2} \cdot \\ [X_2 > 1, run^{abort}\rangle \cdot done^{abort} \cdot done^{f,2} \cdot done^{f} \cdot done$$

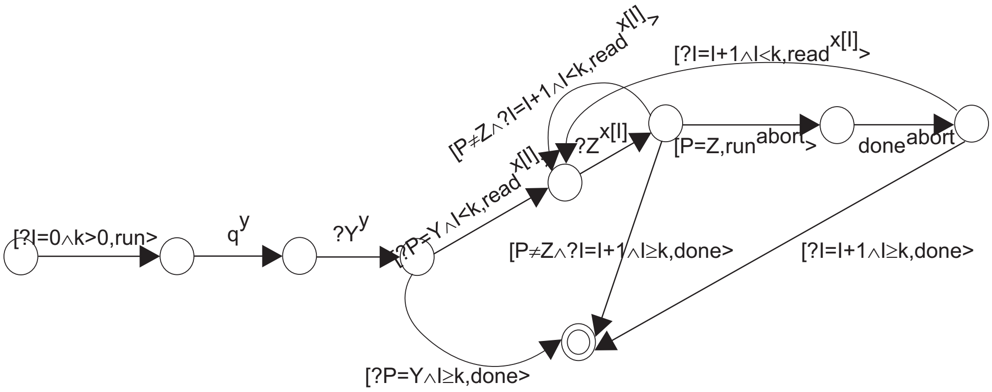

Figure 6: The symbolic model for the linear search.

Yices finds that the condition for this play $(X_1 = 0 \wedge X_2 = X_1 + 1 \wedge X_2 > 0)$ is also found to be unsatisfiable. The next counter-example is:

$$[X_1 = 0, run\rangle \cdot run^f \cdot run^{f,1} \cdot [X_2 = X_1 + 1, done^{f,1}\rangle \cdot run^{f,1} \cdot \\ [X_3 = X_2 + 1, done^{f,1}\rangle \cdot run^{f,2} \cdot [X_3 > 1, run^{abort}\rangle \cdot done^{abort} \cdot \\ done^{f,2} \cdot done^f \cdot done$$

Its play condition is satisfiable, so it is reported as a genuine counter-example. It corresponds to a computation where f uses its first argument two times, then its second argument.

6.2. A linear search term

Let us consider the following implementation of the linear search algorithm.

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

The program first remembers the input expression y into a local variable p. The non-local array x is then searched for an occurrence of the value stored in p. If the search succeeds, then abort is executed.

The symbolic model for this term is shown in Fig. 6, where for simplicity array out-of-bounds errors are not taken in the consideration. If the value read from the environment for y has occurred in x, then an unsafe behaviour of the term exists. So this term is unsafe, and the following counter-example is found:

$$[I_1 = 0 \wedge k > 0, run\rangle \quad q^y \quad Y^y \quad [P = Y \wedge I_1 < k, read^{x[I_1]}\rangle \quad Z^{x[I_1]} \\ [Z = P, run^{abort}\rangle \quad done^{abort} \quad [I_2 = I_1 + 1 \wedge I_2 \geq k, done\rangle$$

	$n = 2$		$n = 3$	
k	Time	Model	Time	Model
1	< 1	11	< 1	13
5	1	43	1	61
10	2	83	2	121
15	5	123	6	181

Table 4: Verification of the linear search with finite data

This play corresponds to a term with an array x of size $k = 1$, where the values read from $x[0]$ and y are equal.

Overall, the symbolic model for linear search term has 9 states and the total time needed to generate the model and test its safety is less than 1 sec. We can compare this approach with the tool in [10], where the standard algorithmic representation of game semantics based on CSP process algebra [26] for terms with finite data types is used. We performed experiments for the linear search term with different sizes of k and all integer types replaced by finite data types. The types of x, y, and p is int_n, i.e. they contain n distinct values $\{0, \ldots n - 1\}$, and the type of the index i is int_{k+1}, i.e. one more than the size of the array. Such term was converted into a CSP process which represents its game semantics, and then the FDR model checker [4] for CSP process algebra was used to generate its model and test its safety. Experimental results are shown in Table 4, where we list the execution time in seconds, and the size of the final model in number of states. The model and the time increase very fast as we increase the sizes of k and n. We ran FDR and Symbolic GameChecker on a Machine AMD Phenom II X4 940 with 4GB RAM. The obtained experimental results confirm efficiency of our symbolic approach.

7. Conclusion

We have shown how to reduce the verification of safety of game-semantics infinite-state models of IA_2 terms with infinite data types to the checking of the more abstract finite-state symbolic automata. The main feature of symbolic automata is that data is not represented explicitly in it, but symbolically.

Counter-example guided abstraction refinement procedures (ARP) [8, 9] can also be used for verification of terms with infinite integers. ARP starts by model-checking the most abstract version of the concrete program, where all infinite integer types are abstracted to the coarsest abstraction that contains only one abstract value. If no counterexample or a genuine one is found, the procedure terminates. Otherwise, it uses a spurious counterexample to gradually refine the abstraction for the next iteration. So ARP finds solutions after performing a few iterations in order to adjust integer identifiers to suitable abstractions.

[4] http://www.fsel.com/

In each iteration, one abstract term is model-checked. If an abstract term needs larger abstractions, then it is likely to obtain a model with very large state space, which is difficult (infeasible) to generate and check automatically. The symbolic approach presented in this paper provides solutions in only one iteration, by checking symbolic models which are significantly smaller than the abstract models in ARP. The possibility to handle arrays with arbitrary length is another important benefit of this approach.

Extensions to nondeterministic [11, 24], concurrent [15, 16], and probabilistic [23] terms can be interesting to consider. In the case of nondeterministic programs, there exists an algorithmic game semantics model [24] which is fully abstract with respect to two complementary notions of observational equivalence of programs: the possibility of termination (may-termination) and the guarantee of termination (must-termination). Apart from containing convergent behaviours of programs, the model also contains divergent behaviours of programs. So its symbolic representation can be used for efficient verification of both safety and liveness properties, such as termination of nondeterministic programs with infinite data types.

Since we consider standard regular-languages to be ultimate meanings of terms, and their symbolic representations to be stepping stones to those meanings, we can regard two symbolic automata to be equal if their corresponding concretization induced by γ are the same. This allows to study and perform transformations of symbolic automata which preserve the equality, and will also enable us to verify properties such as observational-equivalence and approximation. We leave this topic for future research.

References

[1] Abramsky, S., McCusker, G. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In *Electr. Notes Theor. Comput. Sci.* **3**, pp. 2–14, (1996).

[2] Abramsky, S., McCusker, G. Game Semantics. In Proceedings of *the 1997 Marktoberdorf Summer School: Computational Logic* , (1998), 1–56. Springer.

[3] Bakewell, A., Ghica, D. R. Compositional Predicate Abstraction from Game Semantics. In: Kowalewski, S., Philippou A. (eds.) TACAS 2009. LNCS vol. 5505, pp. 62–76. Springer, Heidelberg (2009).

[4] Burgstaller, B., Scholz, B., Blieberger, J. Symbolic Analysis of Imperative Programming Languages. In: Lightfoot, D.E., Szyperski, C.A. (eds.) JMLC 2006. LNCS vol. 4228, pp. 172–194. Springer, Heidelberg (2009).

[5] Clarke, L.A., Richardson, D.J. Symbolic evaluation methods for program analysis. In: Muchnick, S.S., Jones, N.D. (eds.), *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] Dannenberg, R.B., Ernst. G.W. Formal program verification using symbolic execution. In *IEEE Transactions on Software Engineering*, **8(1)**, pp. 43–52, (1982).

[7] Das, S., Dill, D. L., Park, S. Experience with Predicate Abstraction. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS vol. 1633, pp. 160–171. Springer, Heidelberg (1999).

[8] Dimovski, A., Ghica, D. R., Lazić, R. Data-Abstraction Refinement: A Game Semantic Approach. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS vol. 3672, pp. 102–117. Springer, Heidelberg (2005).

[9] Dimovski, A., Ghica, D. R., Lazić, R. A Counterexample-Guided Refinement Tool for Open Procedural Programs. In: Valmari, A. (ed.) SPIN 2006. LNCS vol. 3925, pp. 288–292. Springer, Heidelberg (2006).

[10] Dimovski, A., Lazić, R. Compositional Software Verification Based on Game Semantics and Process Algebras. In *Int. Journal on STTT* **9(1)**, pp. 37–51, (2007).

[11] Dimovski, A. A Compositional Method for Deciding Equivalence and Termination of Nondeterministic Programs. In: Mery, D., Merz, S. (eds.) IFM 2010. LNCS vol. 6396, pp. 121–135. Springer, Heidelberg (2010).

[12] Dimovski, A. Symbolic Representation of Algorithmic Game Semantics. In: Faella, M., Murano, A. (eds.) GandALF 2012. EPTCS vol. 96, pp. 99–112. Open Publishing Association, (2012).

[13] Dutertre, B., de Moura, L. M. A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS vol. 4144, pp. 81–94. Springer, Heidelberg (2006).

[14] Ghica, D. R., McCusker, G. The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science **309** (1–3), pp. 469–502, (2003).

[15] Ghica, D. R., Murawski, A. S: Angelic semantics of fine-grained concurrency. In: Walukiewicz, I. (ed.) FoSSaCS 2004. LNCS vol. 2987, pp. 211–255. Springer, Heidelberg (2004).

[16] Ghica, D. R., Murawski, A. S. Compositional Model Extraction for Higher-Order Concurrent Programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS vol. 3920, pp. 303–317. Springer, Heidelberg (2006).

[17] Graf, S., Saidi, H. Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS vol. 1254, pp. 72–83. Springer, Heidelberg (1997).

[18] Hennessy, M., Lin, H. Symbolic Bisimulations. In *Theoretical Computer Science* **138(2)**, pp. 353–389, (1995).

[19] Hyland, J. M. E., and C.-H. L. Ong, *On Full Abstraction for PCF: I, II, and III*, In Information and Computation **163(2)**, (2000), 285–400.

[20] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* (Addison Wesley Longman, 2001).

[21] Lazić, R. A Semantic Study of Data Independence with Applications to Model Checking. D. Phil. Thesis, Oxford University, 1999.

[22] Lazić, R., and Nowak, D. A Unifying Approach to Data-Independence. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS vol. 1887, pp. 581–595. Springer, Heidelberg (2000).

[23] Murawski, A.S., Ouaknine, J: On Probabilistic Program Equivalence and Refinement. In Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS vol. 3653, pp. 156–170. Springer, Heidelberg (2005).

[24] Murawski, A: Reachability Games and Game Semantics: Comparing Nondeterministic Programs. In: Proceedings of LICS 2008. IEEE, pp. 173–183. IEEE, Los Alamitos (2008).

[25] Reynolds, J. C: The essence of Algol. In: O'Hearn, P.W., Tennent, R.D. (eds), *Algol-like languages.* (Birkhaüser, 1997).

[26] Roscoe, W. A: *Theory and Practice of Concurrency.* Prentice-Hall, 1998.