Parameterized Verification of Open Procedural Programs

Aleksandar S. Dimovski
Faculty of Information-Communication Tech., FON University
Skopje, 1000, MKD
aleksandar.dimovski@fon.edu.mk

ABSTRACT

This paper describes a concrete implementation of a game-semantics based approach for verification of open program terms parameterized by a data type. The programs are restricted to be data-independent with respect to the data type treated as a parameter, which means that the only operation allowed on values of that type is equality testing. The programs can also input, output, and assign such values. This provides a method for verifying a range of safety properties of programs which contain data-independent infinite types. In order to enable verification of programs with arbitrary infinite (integer) types, the proposed method can be extended by combining it with an abstraction refinement procedure. We have developed a tool which implements this method as well as its extension, and we present its practicality by several academic examples.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques; D.3.1 [**Software**]: Programming Languages—*Formal Definitions and Theory*; F.3.2 [**Theory of Computation**]: Logics And Meanings Of Programs—*Specifying and Verifying and Reasoning about Programs*

General Terms

Theory and Verification

Keywords

Software Verification, Data Independence, Game Semantics

1. INTRODUCTION

Automatic software verification is one of the most important problems in computer science today. Software systems should work correctly, and in some application fields, their correct functioning is critical. Model checking [3] has proved to be one of the most effective methods for automatic software verification. In model checking, the system to be verified is interpreted by a model, which is given concrete representation using various automata or process theoretic formalisms. System correctness is then shown by computing that a given property is satisfied by the model, which is done by exhaustively exploring the entire state space of the model. When the model fails to satisfy the desired property, further information is provided in the form of counter-examples, which trace some system executions that violate the property of interest. However, the main limitation of model checking is that it can be used only if a finite-state model is available. This problem arises when we want to verify programs with infinite data types, such as integers.

We say that a system is *data-independent* with respect to a data type, if the only operations it can perform on values of that type are to input, output, assign, and test for equality such values. The data type with respect to which a system is data-independent can be treated as a parameter of the system, i.e. as a data type variable that can be instantiated by any concrete type. Techniques for verifying data-independent systems have been developed for a number of different frameworks [12, 13, 14, 17]. They enable verification for all instantiations of the data type to be reduced, to the verification of finite instantiations. If some infinite type is substituted for the parameter, we obtain a method to address the problem of model checking systems with infinite-state models.

Game semantics [1, 2, 11] is a syntax-independent approach of modeling programs by looking at the ways in which the programs can observably interact with their context (environment). In this approach, computation (execution) of a program is seen as a game between two players, a Player, which represents the program, and an Opponent, which represents the environment in which the program is used. The two take turns to make moves, each of which is either a question (a demand for information) or an answer (a supply of information). The model of a program is given as a strategy for Player to respond to the moves Opponent can make. The game-semantics model is fully abstract (sound and complete), and it is the most precise model we can use for a programming language. Game semantics is also compositional, i.e. defined inductively on the structure of programs, which is essential for modular verification of open (incomplete) programs. It has been shown that for several interesting programming language fragments with finite data types, the game-semantics model can be given certain kinds of concrete automata or process-theoretic representations [6, 9, 10], thus providing direct support for software model checking.

In this paper we extend the previous approaches for model checking based on game semantics [6, 9, 10], such that instead of having programs with finite data types we work with programs with data-independent infinite data types. This is done by considering programs that are data-independent with respect to a given data type, which is treated as a parameter. We use the results obtained in [8] to verify game-semantics models of such parameterized programs. They say that in order to verify a parameterized program for all instantiations of the parameter, it suffices to consider a single appropriate finite instantiation of the data type, such that if a property holds/fails for this instantiation, then it holds/fails for all instantiations of the parameter. This enables us to verify programs with data-independent infinite data types by model checking, such that the infinite type (e.g. integer) is replaced with the appropriate finite data type. We have implemented a tool which compiles a program with the appropriate finite data type substituted for the parameter into a process in the CSP process algebra [17], whose finite traces set represents the game-semantic model of the program. The resulting process is then verified using the FDR tool, which is a model checker for CSP processes. We have also integrated this compiler with an abstraction refinement procedure [4], which runs a sequence of iterations in order to compute an appropriate finite domain for those infinite types that are not data-independent.

The paper is organized in the following way. Section 2 introduces the language considered in this paper, the representation of its game-semantics model in CSP, and several properties of the model which provide direct support for parameterized verification. In Section 3 we present the tool implementing this approach, and evaluate its effectiveness by several examples. Integration of parameterized verification with an abstraction refinement procedure is shown in Section 4. In the end, we conclude and present some ideas for future work.

2. THE LANGUAGE AND ITS GAME SEMANTICS MODEL

We shall be considering open programs (terms) of an expressive programming language, called Idealized Algol (IA for short) [1, 2], which combines the fundamental imperative features, locally-scoped variables, and call-by-name procedures. The data types D are finite integers ($\mathsf{int}_n = \{0, \ldots, n-1\}$), booleans, and a data type variable X.

$$D ::= \mathsf{int}_n \mid \mathsf{bool} \mid X$$

The phrase types are expressions, variables, and commands ($B ::= \mathsf{exp}D \mid \mathsf{var}D \mid \mathsf{com}$), as well as 1st-order function types ($T ::= B \mid B \rightarrow T$). Terms are formed by the following grammar:

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

where v ranges over constants of type D, which includes integers (n), booleans (tt, ff), and data values from the set W which interprets X ($w \in W$). The standard arithmetic-logic operations are employed $\mathsf{op} \in \{+, -, *, /, =, \neq, <, \ldots\}$. We have the usual imperative constructs: sequential composition, conditional, iteration, assignment, de-referencing, and "do nothing" command skip. 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 so called "bad" variables. The standard functional constructs for function definition and application are used.

We write $\Gamma \vdash_W M : T$ to indicate that a term M with free identifiers in $\Gamma = x_1 : T_1, \ldots, x_k : T_k$ has type T, where W is a set of data values used to interpret X. We may omit W, whenever it does not cause ambiguity. Typing rules can be found in [1, 2]), where the rules for arithmetic-logic operations are:

$$\frac{\Gamma \vdash_W M : \mathsf{exp}D \qquad \Gamma \vdash_W N : \mathsf{exp}D}{\Gamma \vdash_W M \,\mathsf{op}\, N : \mathsf{exp}D'} \; D, D' \in \{\mathsf{int_n}, \mathsf{bool}\}$$

$$\frac{\Gamma \vdash_W M : \mathsf{exp}X \qquad \Gamma \vdash_W N : \mathsf{exp}X}{\Gamma \vdash_W M = N : \mathsf{expbool}}$$

So any term can contain only equality operation between values of X. For such terms we say that are *data-independent* with respect to the data type X. We say that a term $\Gamma \vdash_W M : T$ satisfies $(\mathbf{NoEq}_X)$ condition if it contains no equality tests between values of X.

The operational semantics of our language is given for terms $\Gamma \vdash_W M : T$, 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:

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

where s, s$'$ represent the *state* before and after reduction. A state s 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 those of IA [1, 2].

Given a term $\Gamma \vdash_W M : \mathsf{com}$, where all identifiers in Γ are variables, we say that M *terminates* in state s, if $\Gamma \vdash_W M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'$ for some state s$'$. Then, we say that a term $\Gamma \vdash_W M : T$ is an *approximate* of a term $\Gamma \vdash_W N : T$, denoted by $\Gamma \vdash_W M \lessapprox N$, if and only if for any term-with-hole $C[-]$, such that both $C[M]$ and $C[N]$ are closed terms of type com, if $C[M]$ terminates then $C[N]$ terminates. If two terms approximate each other they are considered *observationally-equivalent*, denoted by $\Gamma \vdash_W M \cong N$.

We now show how the game-semantics models of parameterized terms can be represented using the CSP process algebra [17]. This translation is an extension of the one presented in [6], where the considered language is IA. Each type T is interpreted by an *alphabet* of possible events (moves) $\mathcal{A}_{[\![T]\!]}$, which contains two kinds of events: *questions* and *answers*.

$$\mathcal{A}_{[\![\mathsf{int}_n]\!]} = \{0, \ldots, n-1\} \qquad \mathcal{A}_{[\![\mathsf{bool}]\!]} = \{tt, ff\} \qquad \mathcal{A}_{[\![X]\!]} = W$$

$$\mathcal{A}_{[\![\mathsf{exp}D]\!]} = \{q\} \cup \mathcal{A}_{[\![D]\!]} \qquad \mathcal{A}_{[\![\mathsf{com}]\!]} = \{run, done\}$$

$$\mathcal{A}_{[\![\mathsf{var}D]\!]} = \{read, write(v), v, ok \mid v \in \mathcal{A}_{[\![D]\!]}\}$$

$$\mathcal{A}_{[\![B_1^{\langle 1\rangle} \rightarrow \ldots \rightarrow B_k^{\langle k\rangle} \rightarrow B]\!]} = \sum_{1 \le i \le k} \mathcal{A}^{\langle i\rangle}_{[\![B_i]\!]} + \mathcal{A}_{[\![B]\!]}$$

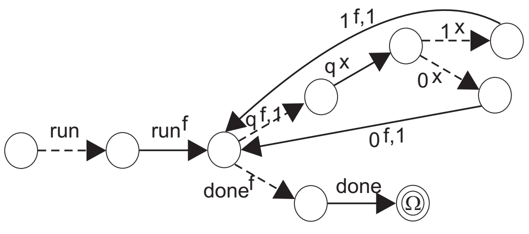

Figure 1: A model as a finite automaton.

Note that in function types we use a superscript to tag type components in order to keep record from which type, i.e. which component of the disjoint union $(+)$, each event comes from. The events (moves) in the alphabet $\mathcal{A}_{[\![T]\!]}$ represent 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, 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 moves for writing to the variable, *write*(v), acknowledged by the move *ok*, and for reading from the variable, a question move *read*, and corresponding to it an answer from $\mathcal{A}_{[\![D]\!]}$.

For any parameterized term $\Gamma \vdash M : T$, we define a parameterized CSP process $[\![\Gamma \vdash M : T]\!]^Y$, such that the set of finite traces $traces([\![\Gamma \vdash M : T]\!]^W)$ represents the game-semantics model of the term $\Gamma \vdash_W M : T$. It is defined over the alphabet:

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

Those processes are defined compositionallly, by induction of the structure of terms. The reader is referred to [6] for details.

Example 1. Consider the term:

$$f : \mathsf{exp}X^{f,1} \rightarrow \mathsf{com}^f, x : \mathsf{exp}X^x \vdash f(x) : \mathsf{com}$$

The model representing this term when $W = \{0, 1\}$ is shown in Fig. 1. The dashed edges indicate moves of the Opponent and solid edges moves of the Player. They serve only as a visual aid to the reader. The model illustrates only the possible behaviors of this term: the non-local procedure f may evaluate its argument, one or more times, and the result will be 0 or 1. The term then terminates successfully with an event *done* leading to a special terminated state Ω. The model does not assume that f uses its argument, or how many times. Note that events tagged with f represent the actions of calling and returning from the function, while events tagged with $f, 1$ are the actions caused by evaluating the first argument of f. Events tagged with x represent requesting and providing a value for the non-local expression x. In general, any finite trace that ends in Ω is called successfully terminated trace, and it represents the observable effects of a completed computation of the given term. □

2.1 Formal Properties

Here we recall some of the properties of the game-semantics model and its CSP representation, which will enable parameterized verification of terms.

Firstly, it was shown in [6] that this game-semantics model is fully abstract for observational-equivalence.

PROPOSITION 1. *$\Gamma \vdash_W M \sqsubseteq N$ iff $traces^{\checkmark}([\![\Gamma \vdash M]\!]^W) \subseteq traces^{\checkmark}([\![\Gamma \vdash N]\!]^W)$, where $traces^{\checkmark}(P)$ is the set of all successfully terminated traces of P.*

Suppose that there is a special free identifier abort of type com. A term is abort-free if it has no occurrence of abort. We say that a term is *safe* if for any abort-free term-with-hole $C[-]$, the term $C[M]$ does not execute the abort command; otherwise we say that a term is *unsafe*. Since the game-semantics model is fully abstract, the following was shown in [4].

PROPOSITION 2. *A term $\Gamma \vdash_W T$ is safe if $[\![\Gamma \vdash M]\!]^W$ does not contain a successfully terminated trace with events from $\mathcal{A}^{\mathsf{abort}}_{[\![\mathsf{com}]\!]}$.*

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

The following results were proved in [8] by using logical relations [15, 16]. They provide sufficient conditions for reducing the verification of a range of safety properties for all instantiations of the parameter X, to the verification of the same properties for finite instantiations of X.

PROPOSITION 3. *Let $\Gamma \vdash M, N : T$ be terms which satisfy (**NoEq**$_X$) and $W_0 = \{0\}$.*
(i) If $\Gamma \vdash_{W_0} M \not\sqsubseteq N$ then $\Gamma \vdash_W M \not\sqsubseteq N$ for all W.
(ii) If $\Gamma \vdash_{W_0} M$ is not safe then $\Gamma \vdash_W M$ is not safe for all W.

PROPOSITION 4. *Let $\Gamma \vdash M, N : T$ be terms and κ be a nonzero cardinal such that $W_\kappa = \{0, \ldots, \kappa\}$.*
(i) If $\Gamma \vdash_{W_\kappa} M \not\sqsubseteq N$ then $\Gamma \vdash_{W_{\kappa'}} M \not\sqsubseteq N$ for all $\kappa' \geq \kappa$.
(ii) If $\Gamma \vdash_{W_\kappa} M$ is not safe then $\Gamma \vdash_{W_{\kappa'}} M$ is not safe for all $\kappa' \geq \kappa$.

3. IMPLEMENTATION

We have extended the tool given in [6], which supports only IA terms, such that it includes a compiler from any parameterized IA term to a parameterized CSP process which represents its game semantics. The resulting CSP process is defined by a script in machine readable CSP [17] which the tool outputs. In the input syntax of terms, we use simple type annotations to indicate what finite sets of integers will be used to model integer free identifiers and local variables (those which are not data-independent). An integer constant n is implicitly defined of type int_{n+1}. An operation between values of types int_{n_1} and int_{n_2} produces a value of type $\mathsf{int}_{max\{n_1,n_2\}}$. The operation is performed modulo $max\{n_1, n_2\}$. The following are more involved three examples.

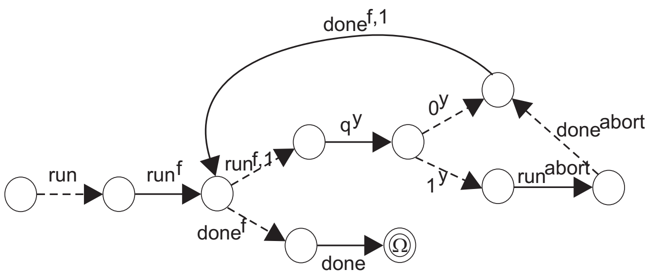

Figure 2: The model for introductory term with $W = \{0, 1\}$.

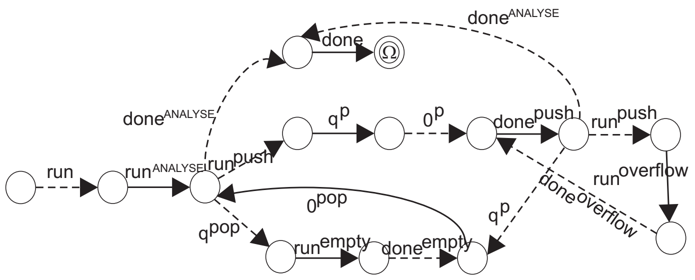

Figure 3: The model for stack term with $n = 1$ and $W = \{0\}$.

3.1 An introductory example

Consider the term:

$$\begin{array}{l} \mathsf{abort} : \mathsf{com}^{abort}, f : \mathsf{com}^{f,1} \to \mathsf{com}^{f}, y : \mathsf{exp}X^{y} \vdash_W \\ \qquad \mathsf{new}_X\ x := 0\ \mathsf{in}\ f\big(\mathsf{if}\ \neg(x = y)\ \mathsf{then}\ abort\big) : \mathsf{com} \end{array}$$

which uses a local variable x, a non-local function f, and a non-local expression y. The model for this term with parameter $W = \{0, 1\}$ is shown in Fig. 2. Notice that no references to the variable x appear in the model because it is locally defined and so not visible from the outside of the term. So in the model are represented only actions of non-local identifiers. When f calls its argument, the term (Player) asks for the value of y by using the event q^y. If the value provided from the environment (Opponent) for y is different from 0 (the value of x), abort is executed.

This term does not satisfy ($\mathbf{NoEq}_X$) condition. It is unsafe for parameter $W = \{0, 1\}$, with the following counter-example:

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

By Proposition 4, we have that this term is unsafe for all $W_{\kappa'}$, $\kappa' \geq 1$. So if int is substituted for X in this term, it will also be unsafe.

3.2 A stack example

Consider the following implementation of a stack of maximum size n (a meta-variable).

```
abort : com^{abort}, p : expX^p,
ANALYSE : com^{push} → expX^{pop} → com^{ANALYSE}  ⊢_W
  new_X buffer[n] in
  new_{int_{n+1}} top := 0 in
  let com push {
    if (top = n) then overflow
      else {buffer[top] := p;  top := top + 1; } } in
  let expint pop {
    if (top = 0) then empty
      else {top := top − 1;  return buffer[top + 1]}} in
  ANALYSE(push, pop) : com
```

After implementing the stack, the non-locally defined function *ANALYSE* is called with arguments *push* and *pop*. Game semantics will therefore give us a model which contains all interleavings of calls to *push* and *pop*, corresponding to all possible behaviours of the non-local identifiers p and *ANALYSE*. The model for the stack with capacity $n = 1$ and $W = \{0\}$ is shown in Figure 3. For clarity, labels push and pop are used instead of ANALYSE, 1 and ANALYSE, 2.

By replacing empty with abort and overflow with abort, we can check separately for 'empty' (reads from empty stacks) and 'overflow' (writes to full stacks) errors. For 'empty' error, the counter-example corresponds to a single first call of pop method; while for 'overflow' error abort is executed after $n + 1$ consecutive calls of push.

This term satisfies ($\mathbf{NoEq}_X$) condition, since no equality test is performed for p and *buffer*. So by Proposition 3, we conclude that the stack is unsafe for all data types W (including int) substituted for X.

Table 1 contains experimental results for checking the stack term. Experiments were performed by converting the term into a CSP process and then letting FDR model checker generate its model and test its safety. For different sizes of n and $W_0 = \{0\}$, we list the execution time in seconds to check any of the two properties ('empty' and 'overflow'), and the size of the final model in number of states. We ran FDR on a Machine AMD Phenom II X4 940 with 4GB RAM.

3.3 Reversal of an array

Consider a term for reversal of an array with size n (a meta-variable).

```
abort : com^{abort}, x[n] : varX^{x[−]} ⊢
    new_X a[n] in
    new_{int_{n+1}} i := 0 in
    while (i < n) do {a[i] := x[i];   i := i + 1; }
    i := 0;
    while (i < n) do {x[i] := a[n − i − 1];   i := i + 1; }
    : com
```

Arrays are introduced in our language as syntactic sugar by using existing term formers. An array $x[n]$ is represented as a set of n distinct variables $x[0]$, $x[1]$, ..., $x[n-1]$, such that abort is executed whenever an array is referenced out of its bounds, i.e.

$$\begin{array}{rcl} x[E] & \equiv & \mathsf{if}\ (E = 0)\ \mathsf{then}\ x[0]\ \mathsf{else}\ \ldots \\ & & \qquad \mathsf{if}\ (E = k - 1)\ \mathsf{then}\ x[k-1]\ \mathsf{else\ abort} \end{array}$$

We can check that array-out-of-bounds errors are not present in this term. The term first copies the input array x into a local array a, which is then copied back into x in reversal order. A model for the term with $n = 2$ and $W = \{0, 1\}$

	stack		linear search					
	W_0		W_1		W_2		W_3	
n	Time	Model	Time	Model	Time	Model	Time	Model
5	1	34	2	36	2	68	2	124
10	3	59	7	66	8	138	10	274
15	7	84	18	96	20	208	39	424
20	14	109	40	126	47	278	160	574

Table 1: Experimental results for the stack and linear search terms.

is shown in Fig. 4. All possible combinations of values are first read from x, then the same values are written back but in reversal order.

This term satisfies ($\mathbf{NoEq}_X$) condition, since there is no equality test between the elements of arrays x and a. We can modify the term by replacing one of the while conditions ($i < n$) with ($i \leq n$), which introduces an out-of-bounds error. Now by Proposition 3 we can establish unsafety of the term for all W (including int) by checking the case when $W_0 = \{0\}$.

4. EXTENSIONS

We can combine our parameterized approach with an abstraction refinement procedure (ARP) [4, 5], since both approaches can be applied to terms which contain infinite (integer) types. The former approach will be used to handle all data-independent integer types, which will be treated as parameters, while the rest of infinite types will be handled by the latter approach.

We now give a brief overview of the ARP. For a full description, see [4, 5]. Instead of finite integers, we introduce a new data type of abstracted integers int_π. The abstractions π range over computable partitionings of the integers. We use the following finitary abstractions:

$$[\,] = \{\mathbb{Z}\}, \quad [n,m] = \{<n, n, \ldots, 0, \ldots, m-1, m, >m\}$$

where $<n = \{n' \mid n' < n\}$, and $>n = \{n' \mid n' > n\}$. Abstractions are refined by *splitting* abstract values: $[\,]$ is refined to $[0,0]$ by splitting $\mathbb{Z}$; $[n,m]$ to $[n-1,m]$ by splitting $<n$, or to $[n,m+1]$ by splitting $>m$.

We check safety of $\Gamma \vdash_W M : T$ (with infinite integer data types) by performing a sequence of iterations. The initial abstracted term $\Gamma_0 \vdash_W M_0 : T_0$ uses the coarsest abstraction $[\,]$ for any integer free identifier or local variable, and the abstraction $[0,n]$ or $[n,0]$ for constants n. Other abstractions (such as those for integer expression subterms) are determined from the former by inference.

The following are the steps of every iteration. The game-semantics model of $\Gamma_i \vdash_W M_i : \theta_i$ is fed to a model checker (the FDR tool in our case). Since our abstractions are safe, any abstracted program is an over-approximation of the concrete program. If no counter-example is found, the procedure terminates with answer safe. Otherwise, the counter-examples are analysed and classified as either *genuine*, which correspond to execution traces in the concrete program, or *potentially spurious*, which can be introduced due to abstraction. If genuine counter-examples exist the program is deemed unsafe, otherwise the spurious counterexamples are used to refine the abstractions. That yields a refined abstracted term $\Gamma_{i+1} \vdash_W M_{i+1} : \theta_{i+1}$, which is passed to the next iteration.

Consider the following implementation of the linear search algorithm.

```
$x[n] : \mathsf{var}X^{x[-]},\ y : \mathsf{exp}X^{y},\ \mathsf{abort} : \mathsf{com}^{abort} \vdash$
    $\mathsf{new}_X\ a[n]$ in
    $\mathsf{new}_X\ z := y$ in
    $\mathsf{new}_{int}\ i := 0$ in
    while $(i < n)$ do { $a[i] := x[i];\ \ i := i+1;$ }
    $\mathsf{new}_{bool}\ present := false$ in
    while $(i < n)$ do {
        if $(\mathbf{a[i]} = \mathbf{z})$ then $present := true;$
        $i := i+1;$ }
    if $(\neg\, present)$ then abort  :  com
```

The program first copies the input array x into a local array a, and the input expression y into a local variable z. Then, the local array is searched for an occurrence of the value stored in z. If the search fails, then abort is executed. The equality is the only operation on the data of type X (see the bold in the code), so this term does not satisfy ($\mathbf{NoEq}_X$) condition.

The ARP needs $n+2$ iterations to automatically adjust the type of the local variable i from $[\,]$ to $[0,n]$. The model for this term with $n = 2$, $W = \{0,1\}$, and i of type $\mathsf{int}_{[0,n]}$ is shown in Fig. 5. For such abstraction of i, we obtain the following genuine counter-example:

$$run\ read^{x[0]}\ 1^{x[0]}\ read^{x[1]}\ 1^{x[1]}\ q^{y}\ 0^{y}\ run^{abort}\ done^{abort} done$$

By Proposition 4, it follows that this term is unsafe for all $W_{\kappa'}$, $\kappa' \geq 1$. Experimental results for checking safety of this term with different sizes of n and W_κ, where the variable i is of type $\mathsf{int}_{[0,n]}$, are shown in Table 1.

We can also combine the ARP with the parameterized verification to check the stack example from Section 3.2. Namely, $n+2$ iterations will be needed to adjust the local variable tracking the top of the stack (top) to the domain $[0,n]$. For such abstraction of top, we obtain genuine counter-examples as described in Section 3.2.

5. CONCLUSION

We presented a compositional approach for verifying safety properties of parameterized programs for all instances of the parameter X. It will be interesting to consider extending the proposed approach to nondeterministic [7] and concurrent

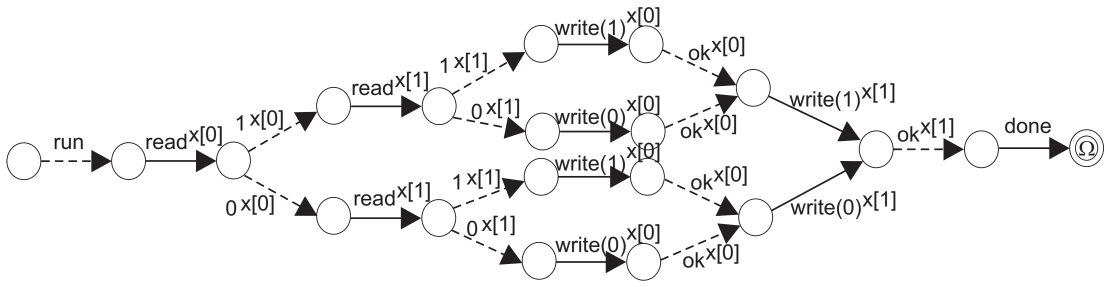

Figure 4: Model for the reversal of an array term with n=2 and $W = \{0, 1\}$.

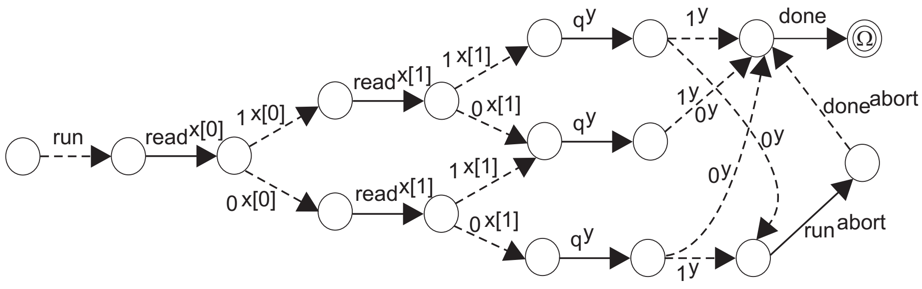

Figure 5: The model for linear search term with n=2, $W = \{0, 1\}$, and i of type $\mathsf{int}_{[0,n]}$.

programs [10], and to verification of some liveness properties, such as termination and may&must equivalence of programs.

6. REFERENCES

[1] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *In: O'Hearn, P.W, and Tennent, R.D. (eds), Algol-like languages*, 1997.

[2] S. Abramsky and G. McCusker. Game semantics. *In Proceedings of the 1997 Marktoberdorf Summer School: Computational Logic*, pages 1–56, 1998.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[4] A. Dimovski, D. R. Ghica, and R. Lazić. Data-abstraction refinement: A game semantic approach. In *Hankin, C., Siveroni, I. (eds.) SAS 2005, LNCS vol. 3672*, pages 102–117. Springer, Heidelberg, 2005.

[5] A. Dimovski, D. R. Ghica, and R. Lazić. A counterexample-guided refinement tool for open procedural programs. In *Valmari, A. (ed.) SPIN 2006, LNCS vol. 3925*, pages 288–292. Springer, Heidelberg, 2006.

[6] A. Dimovski and R. Lazić. Compositional software verification based on game semantics and process algebras. *Int. Journal on STTT*, 9(1):37–51, 2007.

[7] A. Dimovski. A compositional method for deciding equivalence and termination of nondeterministic programs. In *Mery, D., Merz, S. (eds.) IFM 2010, LNCS vol. 6396*, pages 121–135. Springer, Heidelberg, 2010.

[8] A. Dimovski. Logical relations for game semantics and their applications. *Submitted for publication.*

[9] D. R. Ghica and G. McCusker. The regular-language semantics of second-order idealized algol. *Theoretical Computer Science*, 309(1–3):469–502, 2003.

[10] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. In *Diaz, J., Karhumaki, J., Lepisto, A., Sannella, D. (eds.) ICALP 2004, LNCS vol. 3142*, pages 683–694. Springer, Heidelberg, 2004.

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

[12] C. N. Ip and D. L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

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

[14] R. Lazić and D. Nowak. A unifying approach to data-independence. In *Palamidessi, C. (ed.) CONCUR 2000, LNCS vol. 1887*, pages 581–595. Springer, Heidelberg, 2000.

[15] Q.Ma and J. C. Reynolds, Types, Abstraction, and Parametric Polymorphism, Part 2. In *Brookes, S.D., Main, M.G., Melton, A., Mislove, M.W., Schmidt, D.A. (eds.) MFPS 1991, LNCS vol. 598*, pages 1–40, Springer, Heidelberg (1992).

[16] P.W. O'Hearn and R. D. Tennent, Parametricity and Local Variables. *Journal of the ACM* 42(3):658–709, 1995.

[17] W. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1999.