A Compositional Method for Deciding Equivalence and Termination of Nondeterministic Programs

Aleksandar Dimovski

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

Abstract. In this paper we address the problem of deciding may- and must-equivalence and termination of nondeterministic finite programs from second-order recursion-free Erratic Idealized Algol. We use game semantics to compositionally extract finite models of programs, and the CSP process algebra as a concrete formalism for representation of models and their efficient verification. Observational may- and must-equivalence and liveness properties, such as divergence and termination, are decided by checking traces refinements and divergence-freedom of CSP processes using the FDR tool. The practicality of the approach is evaluated on several examples.

1 Introduction

Game semantics is a syntax-independent approach of modeling open programs by looking at the ways in which a program can observably interact with its environment (context). Types are modeled by games (or arenas) between a Player (i.e. program) and an Opponent (i.e. environment), and programs are modeled by strategies on games. It was shown that, for several interesting programming language fragments, their game semantics yield algorithms for model checking. The focus has been on Idealized Algol (IA) [1, 12], which represents a metalanguage combining imperative with higher-order functional features. Game semantics is compositional, i.e. defined recursively on the syntax, which is essential for the modular analysis of larger programs. Previous work on model checking using game semantic models has been mainly concerned with verification of safety properties of sequential, concurrent, and probabilistic programs [9, 10, 13]. All these models say about what a program *may* do, but nothing about what it *must* do. This reflects the deficiencies of the models for reasoning about anything other than safety properties.

In order to take account of liveness properties of nondeterministic programs, the requested model must make distinctions between a reliable program, such as skip, and an unreliable program, such as skip or $\mathsf{diverge}_{\mathsf{com}}$. This means that the model must capture two complementary notions of program equivalence: the possibility of termination (may-termination) and the guarantee of termination (must-termination). To address this issue, the strategy of a program, apart from containing the potential convergent behaviours of the program, must be enriched

with an extra information about the possible divergent behaviours of the program. In [11, 12], it is given a game semantic model for Erratic IA (EIA) which is fully abstract with respect to may &must-termination equivalence. EIA represents a nondeterministic extension of IA, i.e. it is IA enriched with an erratic choice 'or' operator. The full abstraction result means that the model validates all and only correct (may &must-termination) equivalences between programs, i.e. it is sound and complete. Although this model is appropriate for verifying both, safety and liveness, properties, it is complicated and so equivalence and a range of properties are not decidable within it. However, it has been shown in [14] that the game models of second- (resp., third-) order recursion-free finitary EIA can be represented by finite (resp., visibly pushdown) automata. This gives a decision procedure for a range of verification problems to be solved algorithmically, such as: may-equivalence, must-equivalence, may &must-equivalence, termination and other properties.

In this work we propose a verification tool for analyzing nondeterministic programs. We show how for second-order recursion-free EIA with finite data types game semantic models can be represented as CSP processes, i.e. any program is compositionally modeled as a CSP process whose terminated traces and minimal divergences are exactly all the complete plays and divergences of the strategy for the program. This enables observational may- and must-equivalence between any two programs and a range of (safety and) liveness properties, such as termination and divergence, of programs to be decided by checking traces refinements and divergence-freedom of CSP processes by using the FDR tool.

CSP [15] is a particulary convenient formalism for encoding game semantic models. The FDR model checker can be used to automatically check refinements between two processes and a variety of properties of a process, and to debug interactively when a refinement or a property does not hold. FDR has direct support for three different forms of refinement: traces ($\sqsubseteq_T$), failures ($\sqsubseteq_F$), and failures-divergences ($\sqsubseteq_{FD}$); and for the following properties: deadlock, divergence, and determinism. Then, composition of strategies, which is used in game semantics to obtain the strategy for a program from strategies for its subprograms, is represented in CSP by renaming, parallel composition and hiding operators, and FDR is highly optimised for verification of such networks of processes. Finally, FDR builds the models gradually, at each stage compressing the submodels to produce an equivalent process with many fewer states. A number of hierarchical compression algorithms are available in FDR, which can be applied during either model generation or refinement checking.

The paper is organised in the following way. Section 2 introduces the language considered in this paper. The game semantic model of the language is defined in Section 3, and its CSP representation is presented in Section 4. Correctness of the CSP representation, decidability of observational may- and must-equivalence and termination are shown in Section 5. The effectiveness of this approach is evaluated in Section 6. In the end, we conclude and present some ideas for future work.

Related work. Automated verification of liveness properties of programs is an active research topic. The work in [5] presents an abstraction refinement procedure for proving termination of programs. The procedure successively weakens candidate transition invariants and successively refines transition predicate abstractions of the given program. The approach taken in [3, 4] proves termination of programs by generating linear ranking functions, which assign a value from a well-founded domain to each program state. The method represents program invariants and transition relations as polyhedral cones and constructs linear ranking functions by manipulating these cones. Compared to the aforementioned approaches, the main focus of our method is compositionality which is reached in a clean and theoretically firm semantics-based way. Namely, the model of a program is constructed out of the models of its subprograms, which facilitates breaking down the verification of a larger program into verifications of its subprograms. By applying various program analysis techniques, such as predicate abstraction and counter-example guided abstraction refinement [2, 7], the efficiency of our method and its applicability to a broader class of programs can be significantly improved.

2 Programming Language

Erractic Idealized Algol [12] is a nondeterministic imperative-functional language which combines the fundamental imperative features, locally-scoped variables, and full higher-order function mechanism based on a typed call-by-name λ-calculus.

The data types D are a finite subset of the integers (from 0 to $n-1$, where $n > 0$) and the Booleans ($D ::= \mathsf{int}_n \mid \mathsf{bool}$). The phrase types consists of base types (expressions, commands, variables) and function types ($B ::= \mathsf{exp}D \mid \mathsf{var}D \mid \mathsf{com},\ T ::= B \mid T \to T$). Terms are formed by the following grammar:

$$M ::= x \mid v \mid \mathsf{skip} \mid \mathsf{diverge}_B \mid M \,\mathsf{op}\, M \mid M; M \mid \mathsf{if}\, M \,\mathsf{then}\, M \,\mathsf{else}\, M \mid \mathsf{while}\, M \,\mathsf{do}\, M$$
$$\mid M := M \mid !M \mid \mathsf{newvar} D\ x := v \ \mathsf{in}\ M \mid \mathsf{mkvar} MM \mid M \,\mathsf{or}\, M \mid \lambda\, x.M \mid MM \mid YM$$

where v ranges over constants of type D. The language constants are a "do nothing" command skip which always terminates successfully, and for each base type there is a constant $\mathsf{diverge}_B$ which causes a program to enter an unresponsive state similar to that caused by an infinite loop. The usual imperative constructs are employed: sequential composition (;), conditional (if), iteration (while), assignment (:=), and de-referencing (!). Block-allocated local variables are introduced by a *new* construct, which initializes a variable and makes it local to a given block. There are constructs for nondeterminism, function creation and application, as well as recursion.

Well-typed terms are given by typing judgements of the form $\Gamma \vdash M : T$, where Γ is a type *context* consisting of a finite number of typed free identifiers. Typing rules of the language are those of EIA (e.g. [1, 12]), extended with a rule for the $\mathsf{diverge}_B$ constant: $\Gamma \vdash \mathsf{diverge}_B : B$.

The operational semantics of our language is given in terms of *states*. Given a type context $\Gamma = x_1 : \mathsf{var}D_1, \ldots, x_k : \mathsf{var}D_k$ where all identifiers are variables, which is called var-*context*, we define a Γ-*state* s as a (partial) function assigning data values to the variables $\{x_1, \ldots, x_k\}$. The canonical forms are defined by $V ::= x \mid v \mid \lambda\, x : T.M \mid \mathsf{skip} \mid \mathsf{mkvar}MN$. The operational semantics is defined by a big-step reduction relation:

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

where $\Gamma \vdash M : T$ is a term, Γ is a var-context, s, s′ are Γ-states, and V is a canonical form. Reduction rules are those of EIA (see [1, 12] for details). The $\mathsf{diverge}_B$ constant is not reducible.

Since the language is nondeterministic, it is possible that a term may reduce to more than one value. Given a term $\Gamma \vdash M : \mathsf{com}$ where Γ is a var-context, we say that M *may terminate* in state s, written $M, \mathrm{s} \Downarrow^{may}$, if there exists a reduction $\Gamma \vdash M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'$ for some state s′. We say that M *must terminate* in a state s, written $M, \mathrm{s} \Downarrow^{must}$, if all reductions at start state s end with the term skip. If M is a closed term then we abbreviate the relation $M, \emptyset \Downarrow^{may}$ (resp., $M, \emptyset \Downarrow^{must}$) with $M \Downarrow^{may}$ (resp., $M \Downarrow^{must}$). Next, we define a *program context* $C[-] : \mathsf{com}$ *with hole* to be a term with (possibly several occurrences of) a hole in it, such that if $\Gamma \vdash M : T$ is a term of the same type as the hole then $C[M]$ is a well-typed closed term of type com, i.e. $\vdash C[M] : \mathsf{com}$. Then, we say that a term $\Gamma \vdash M : T$ is a *may-approximate* (resp., *must-approximate*) of a term $\Gamma \vdash N : T$, denoted by $\Gamma \vdash M \sqsubsetsim_{may} N$ (resp., $\Gamma \vdash M \sqsubsetsim_{must} N$), if and only if for all program contexts $C[-] : \mathsf{com}$, if $C[M] \Downarrow^{may}$ (resp., $C[M] \Downarrow^{must}$) then $C[N] \Downarrow^{may}$ (resp., $C[N] \Downarrow^{must}$). If two terms may-approximate (resp., must-approximate) each other they are considered *may-equivalent* (resp., *must-equivalent*), denoted by $\Gamma \vdash M \cong_{may} N$ (resp., $\Gamma \vdash M \cong_{must} N$). Combining may- and must-approximation (resp., equivalence) gives rise to *may&must approximation* (resp., *equivalence*). For instance, the following facts hold:

skip or $\mathsf{diverge_{com}} \cong_{may} \mathsf{skip}$ $\qquad$ skip or $\mathsf{diverge_{com}} \cong_{must} \mathsf{diverge_{com}}$

skip or $\mathsf{diverge_{com}} \not\cong_{may\&must} \mathsf{skip}$ $\qquad$ skip or $\mathsf{diverge_{com}} \not\cong_{may\&must} \mathsf{diverge_{com}}$

Example 1. Consider the term from [14]:

$$f : \mathsf{com} \to \mathsf{com} \vdash \mathsf{newint}_2\, x := 0 \,\mathsf{in} \left(f(x := 1)\ ;\ \mathsf{if}\,(x = 1)\,\mathsf{then}\,\mathsf{diverge_{com}}\right) \\ \mathsf{or}\ \left(f(x := 1)\ ;\ \mathsf{if}\,(x = 0)\,\mathsf{then}\,\mathsf{diverge_{com}}\right) : \mathsf{com}$$

in which f is a non-local function. The function-call mechanism is by-name, so every call to the argument of f sets x to 1. In what follows, we will see that this term is may-equivalent to $f(\mathsf{skip})$, must-equivalent to $\mathsf{diverge_{com}}$, and may &must-equivalent to $f(\mathsf{skip})$ or $\mathsf{diverge_{com}}$. □

3 Game Semantics

In this section we give an overview of game semantics for EIA, which is fully abstract with respect to may &must-equivalence. A complete definition can be found in [12, pp. 99–138].

An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$, where M_A is a countable set of *moves*, $\lambda_A : M_A \rightarrow \{\mathsf{O},\mathsf{P}\} \times \{\mathsf{Q},\mathsf{A}\}$ is a labeling function which indicates whether a move is by *Opponent* (O) or *Player* (P), and whether it is a *question* (Q) or an *answer* (A). Then, $\vdash_A$ is a binary relation between $M_A + \{*\}$ ($* \notin M_A$) and M_A, called *enabling* (if $m \vdash_A n$ we say that m enables move n), which satisfies the following conditions: (i) Initial moves (a move enabled by $*$ is called *initial*) are Opponent questions, and they are not enabled by any other moves besides $*$; (ii) Answer moves can only be enabled by question moves; (iii) Two participants always enable each others moves, never their own.

We denote the set of all initial moves in A as I_A. The simplest arena is the empty arena $I = \langle \emptyset, \emptyset, \emptyset \rangle$. The base types are interpreted by arenas where all questions are initial and P-moves answer them.

$$
\begin{aligned}
&[\![\mathsf{expD}]\!] = \langle\ \{q, v \mid v \in \mathsf{D}\}, \{\lambda(q) = \mathsf{OQ},\ \lambda(v) = \mathsf{PA}\}, \{(*, q),\ (q, v) \mid v \in D\}\ \rangle \\
&[\![\mathsf{com}]\!] = \langle\ \{run, done\}, \{\lambda(run) = \mathsf{OQ},\ \lambda(done) = \mathsf{PA}\}, \{(*, run), (run, done)\}\rangle \\
&[\![\mathsf{varD}]\!] = \langle\ \{read, v, write(v), ok \mid v \in \mathsf{D}\}, \{\lambda(read, write(v)) = \mathsf{OQ}, \\
&\qquad \lambda(v, ok) = \mathsf{PA}\}, \{(*, read), (*, write(v)), (read, v), (write(v), ok) \mid v \in D\}\ \rangle
\end{aligned}
$$

Given arenas A and B, we define new arenas $A \times B$, $A \Rightarrow B$ as follows[1]:

$$
\begin{aligned}
&A \times B = \langle\, M_A + M_B,\ [\lambda_A, \lambda_B], \vdash_A + \vdash_B \,\rangle \\
&A \Rightarrow B = \langle\, M_A + M_B,\ [\overline{\lambda}_A, \lambda_B], \vdash_B + \big(I_B \times I_A\big) + \big(\vdash_A \cap (M_A \times M_A)\big)\rangle
\end{aligned}
$$

A *justified sequence* s in arena A is a finite sequence of moves of A together with a pointer from each non-initial move n to an earlier move m such that $m \vdash_A n$. We say that n is (explicitly) justified by m. A *legal play* (or just a play) is a justified sequence with some additional constraints: *alternation* (Opponent and Player moves strictly alternate), *well-bracketed* condition (when an answer is given, it is always to the most recent question which has not been answered), and *visibility* condition (a move to be played depends upon a certain subsequence of the play so far, rather than on all of it). The set of all legal plays in arena A is denoted by L_A. We use meta-variables m, n to range over moves, and s to range over sequences of moves. We also write $s\,m$ or $s \cdot m$ for the concatenation of s and m. The empty sequence is written as ϵ, and $\sqsubseteq$ denotes the prefix ordering on sequences.

A *strategy* σ on an arena A (written as $\sigma : A$) is a pair (T_σ, D_σ), where:

- T_σ is a non-empty set of even-length plays of A, known as the *traces* of σ, satisfying: if $s \cdot m \cdot n \in T_\sigma$ then $s \in T_\sigma$.
- D_σ is a set of odd-length plays of A, known as the *divergences* of σ, satisfying: if $s \cdot m \in D_\sigma$ then $s \in T_\sigma$; and if $s \in T_\sigma$, $s \cdot m \in L_A$, and $s \cdot m \cdot n \notin T_\sigma$ for $\forall\, n$ then $\exists\, d \in D_\sigma . d \sqsubseteq s \cdot m$.

A strategy specifies what options Player has at any given point of a play and it does not restrict the Opponent moves. If Player can not respond at some point of a play then this is reflected by an appropriate divergence sequence in

[1] $\overline{\lambda}_A$ is like λ_A except that it reverses O/P part, and $+$ is a disjoint union.

the strategy. Note that, here we choose a "minimal" representation of divergence, where only the minimal divergences of a strategy, denoted by $div(\sigma)$, are recorded. An alternative "maximal" representation, as is done in CSP, is also possible. It considers the divergences as extension-closed set, which forces the traces set to include all possible sequences after a divergence has been reached. These two representations are equivalent.

Given strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, the composition $\sigma \,;\, \tau = (T_{\sigma;\tau}, D_{\sigma;\tau}) : A \Rightarrow C$ is defined in the following way. Let u be a sequence of moves from A, B, and C. We define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C as well as pointers between them (pointers from/to moves of A are deleted). Similarly define $u \upharpoonright A, B$. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C, but where there was a pointer from a move $m_A \in M_A$ to an initial move $m \in I_B$ extend the pointer to the initial move in C which was pointed to from m. We say that u is an *interaction* of A, B, C if $u \upharpoonright A, B \in L_{A \Rightarrow B}$, $u \upharpoonright B, C \in L_{B \Rightarrow C}$, and $u \upharpoonright A, C \in L_{A \Rightarrow C}$. The set of all such sequences is written as $int(A, B, C)$.

$$T_{\sigma;\tau} = \{u \upharpoonright A, C \mid u \in int(A, B, C) \wedge u \upharpoonright A, B \in T_\sigma \wedge u \upharpoonright B, C \in T_\tau\}$$

So $T_{\sigma;\tau}$ consists of sequences generated by playing T_σ and T_τ in parallel, making them synchronize on moves in B, which are afterwards hidden.

We define an *infinite interaction* of A, B, C to be a sequence $u \in (M_A + M_B + M_C)^\infty$ such that $u \upharpoonright A, C \in L_{A \Rightarrow C}$, and for all $i \in \mathbb{N}_0$, $u_{<i} \upharpoonright A, B \in L_{A \Rightarrow B}$ and $u_{<i} \upharpoonright B, C \in L_{B \Rightarrow C}$, where $u_{\leq i}$ denotes the finite prefix of u with length i. The set of all such sequences is written as $int_\infty(A, B, C)$. Then, a set of *finitely generated* divergences is defined as:

$$\begin{aligned} D_\sigma \swarrow D_\tau = \{u \in int(A, B, C) \mid & \big(u \upharpoonright A, B \in T_\sigma \ \wedge \ u \upharpoonright B, C \in D_\tau\big) \\ & \vee \big(u \upharpoonright A, B \in D_\sigma \ \wedge \ u \upharpoonright B, C \in T_\tau\big)\} \end{aligned}$$

$D_\sigma \swarrow D_\tau$ consists of sequences containing a trace from σ and a divergence from τ, or vice versa. A set of *infinitely generated* divergences is defined as [2]:

$$\begin{aligned} T_\sigma \swarrow T_\tau = \{u \in int_\infty(A, B, C) \mid \forall\, i \in \mathbb{N}_0 . u_{\leq i} \upharpoonright A, B \in T_\sigma \cup dom(\sigma) \\ \wedge\ u_{\leq i} \upharpoonright B, C \in T_\tau \cup dom(\tau)\} \end{aligned}$$

$T_\sigma \swarrow T_\tau$ consists of sequences that have an infinite tail in B. This situation is called *livelock*. We have that: $D_{\sigma;\tau} = \{u \upharpoonright A, C \mid u \in D_\sigma \swarrow D_\tau \vee u \in T_\sigma \swarrow T_\tau\}$.

The *identity strategy*, which is also called *copy-cat*, for an arena A is $Id_A = (id_A, \emptyset)$, where $id_A = \{s \in P_{A \Rightarrow A} \mid \forall\, s' \sqsubseteq^{even} s.\ s' \upharpoonright A_l = s' \upharpoonright A_r\}$. We use the l and r tags to distinguish between the two occurrences of A and $s' \sqsubseteq^{even} s$ means that s' is an even-length prefix of s. So in id_A, a move by Opponent in either occurrence of A is immediately copied by Player to the other occurrence.

In general, plays in a strategy may contain several occurrences of initial moves, which define several different *threads* of the play in the following way: two moves are in the same thread if they are connected via chains of pointers to

[2] The domain of a strategy σ is the set $dom(\sigma) = \{s \cdot m \mid \exists\, n . s \cdot m \cdot n \in T_\sigma\}$.

the same occurrence of an initial move. We consider the class of *single-threaded* strategies whose behaviour depends only on one thread at a time, i.e. any Player response depends solely on the current thread of the play and any divergence is caused by the play in a single thread. We say that a strategy is *well-opened* if all its plays have exactly one initial move. It can be established one-to-one correspondence between single-threaded and well-opened strategies. It is shown in [12] that arenas as objects and single-threaded (well-opened) strategies as arrows constitute a cpo-enriched cartesian closed category, which can be used for construction of semantic models of programming languages. From now on, we proceed to work only with well-opened strategies and plays with exactly one initial move.

A term $\Gamma \vdash M : T$, where $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, is interpreted by a strategy $[\![\Gamma \vdash M : T]\!]$ for the arena $[\![\Gamma \vdash T]\!] = [\![T_1]\!] \times \ldots \times [\![T_n]\!] \Rightarrow [\![T]\!]$. Language constants and constructs are interpreted by strategies and compound terms are modelled by composition of the strategies that interpret their constituents. For example, some of the strategies are [12]: $[\![\mathsf{n} : \mathsf{expint}]\!] = (\{\epsilon, q\, n\}, \emptyset)$, $[\![\mathsf{skip} : \mathsf{com}]\!] = (\{\epsilon, run\, done\}, \emptyset)$, $[\![\mathsf{diverge}_{\mathsf{com}} : \mathsf{com}]\!] = (\{\epsilon\}, \{run\})$, $[\![\mathsf{or} : \mathsf{exp}D_0 \times \mathsf{exp}D_1 \to \mathsf{exp}D_2]\!]$ [3] $= (\{\epsilon, q_2\, q_0, q_2\, q_1, q_2\, q_0\, v_0\, v_2, q_2\, q_1\, v_1\, v_2 \mid v \in D\}, \emptyset)$, free identifiers are interpreted by identity strategies, etc. Using standard game-semantic techniques, it has been shown in [12] that the quotient of this model with respect to the so-called intrinsic preorder is fully abstract for may &must-equivalence. However, the model itself is sound [12], so the must-termination of terms follows from this result.

Proposition 1. $\Gamma \vdash M$ must terminate *iff* $D_{[\![\Gamma \vdash M]\!]} = \emptyset$.

More explicit characterizations of may- and must-approximation (resp., equivalence) are given in [14]. A play is *complete* if all questions occurring in it have been answered. Given a strategy σ, we write $comp(\sigma)$ for the set of its non-empty complete plays.

Proposition 2. $\Gamma \vdash M \sqsubseteq_{may} N$ *iff* $comp([\![\Gamma \vdash M]\!]) \subseteq comp([\![\Gamma \vdash N]\!])$.

In order to capture must-approximation, we define a new relation $\leq_{must}$ on strategies over any arena A: $\sigma \leq_{must} \tau$ iff for any $s \in (T_\tau \cup D_\tau) \backslash (T_\sigma \cup D_\sigma)$ there exists $s' \sqsubseteq^{odd} s$ such that $s' \sqsubseteq d$ for some $d \in D_\sigma$.

Proposition 3. $\Gamma \vdash M \sqsubseteq_{must} N$ *iff* $[\![\Gamma \vdash M]\!] \leq_{must} [\![\Gamma \vdash N]\!]$.

4 CSP Representation

In the rest of the paper, we work with the 2nd-order recursion-free fragment of EIA. In particular, function types are restricted to $T ::= B \mid B \to T$. Without loss of generality, we consider only terms in β-normal form. We now show how the game semantic model of this fragment of EIA can be given a concrete representation using the CSP process algebra. This translation is an extension of the

[3] Every move is tagged with the index of type component where it occurs.

one presented in [6, 9], where the considered language is IA and the model takes account of only safety properties.

CSP (Communicating Sequential Processes) [15] is a language for modelling systems which consist of interacting components. Each component is specified through its behaviour which is given as a *process*. Processes are defined in terms of the *events* that they can perform. The set of all possible events is denoted Σ.

Processes can be given denotational semantics by the following sets of their possible behaviours. The set $\mathsf{traces}(P)$ contains all possible finite sequences of events that the process P can perform. The set $\mathsf{failures}(P)$ consists of all pairs (s, X), where $s \in \mathsf{traces}(P)$ and X is a set of events that P can refuse to do in some stable state after the trace s. And, we define $\mathsf{divergences}(P)$ as the set of traces after which the process can perform an infinite sequence of consecutive internal events called τ. We consider here two semantic models of CSP processes: *traces semantics*, denoted as P_T, and *failures-divergences semantics*, denoted as P_{FD}. We omit the subscripts when they are clear from the context. Traces semantics of a process P is given by the set $\mathsf{traces}(P)$, while failures-divergences semantics of P is given by the pair $(\mathsf{failures}(P), \mathsf{divergences}(P))$. Divergences of a process are not modeled in its traces semantics, but they have "maximal" representation in its failures-divergences semantics. For example, let consider the div process. It represents a special divergent process in CSP which does nothing but diverge. It is equivalent to the recursive process $\mu\, p.p$. We have that $\mathsf{div}_T = \{\epsilon\}$, but $\mathsf{div}_{FD} = (\Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{\checkmark}), \Sigma^{*\checkmark})$, where $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$ [4], and $\Sigma^{*\checkmark} = \Sigma \cup \{s \cdot \checkmark \mid s \in \Sigma^*\}$. *Traces refinement* between processes is defined as:

$$P_1 \sqsubseteq_T P_2 \;\Leftrightarrow\; \mathsf{traces}(P_2) \subseteq \mathsf{traces}(P_1)$$

CSP processes can also be given operational semantics using *labelled transition systems* (LTS). The LTS of a process is a directed graph whose nodes represent process states and whose edges are labelled by events representing what happens when the given event is performed. LTSs have a distinguished start state, and any edge whose label is $\checkmark$ leads to a special terminated state Ω.

With each type T, we associate a set of possible events: an alphabet $\mathcal{A}_{[\![T]\!]}$. It contains a set of events $\mathsf{q} \in \mathrm{Q}_{[\![T]\!]}$, called questions, which are appended to a channel with name Q, and for each question q, there is a set of events $\mathsf{a} \in \mathrm{A}^{\mathsf{q}}_{[\![T]\!]}$, called answers, which are appended to a channel with name A.

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

$$\mathrm{Q}_{[\![\mathsf{exp} D]\!]} = \{q\} \qquad \mathrm{A}^{q}_{[\![\mathsf{exp} D]\!]} = \mathcal{A}_{[\![D]\!]} \qquad \mathrm{Q}_{[\![\mathsf{com}]\!]} = \{run\} \qquad \mathrm{A}^{run}_{[\![\mathsf{com}]\!]} = \{done\}$$

$$\mathrm{Q}_{[\![\mathsf{var} D]\!]} = \{read, write.v \mid v \in \mathcal{A}_{[\![D]\!]}\} \qquad \mathrm{A}^{read}_{[\![\mathsf{var} D]\!]} = \mathcal{A}_{[\![D]\!]} \qquad \mathrm{A}^{write.v}_{[\![\mathsf{var} D]\!]} = \{ok\}$$

$$\mathrm{Q}_{[\![B_1 \to \ldots \to B_k \to B]\!]} = \bigcup_{1 \le i \le k} \{i.\mathsf{q} \mid \mathsf{q} \in \mathrm{Q}_{[\![B_i]\!]}\} \cup \mathrm{Q}_{[\![B]\!]}$$

$$\mathrm{A}^{i.\mathsf{q}}_{[\![B_1 \to \ldots \to B_k \to B]\!]} = \{i.\mathsf{a} \mid \mathsf{a} \in \mathrm{A}^{\mathsf{q}}_{[\![B_i]\!]}\},\ \mathsf{q} \in \mathrm{Q}_{[\![B_i]\!]},\ 1 \le i \le k$$

$$\mathrm{A}^{\mathsf{q}}_{[\![B_1 \to \ldots \to B_k \to B]\!]} = \mathrm{A}^{\mathsf{q}}_{[\![B]\!]},\ \mathsf{q} \in \mathrm{Q}_{[\![B]\!]}$$

[4] *SKIP* is a process that successfully terminates causing the special event $\checkmark$ ($\checkmark \notin \Sigma$).

$$\mathcal{A}_{[\![T]\!]} = Q.\mathrm{Q}_{[\![T]\!]} \cup A. \bigcup_{\mathsf{q} \in \mathrm{Q}_{[\![T]\!]}} \mathrm{A}^{\mathsf{q}}_{[\![T]\!]}$$

For any term $\Gamma \vdash M : T$, we define a CSP process $[\![\Gamma \vdash M : T]\!]$ which represents the strategy for the term. Events of this process are from the alphabet $\mathcal{A}_{[\![\Gamma \vdash T]\!]}$ defined as follows: $\mathcal{A}_{[\![x:T]\!]} = x.\mathcal{A}_{[\![T]\!]}$, $\mathcal{A}_{[\![\Gamma]\!]} = \bigcup_{x:T \in \Gamma} \mathcal{A}_{[\![x:T]\!]}$, and $\mathcal{A}_{[\![\Gamma \vdash T]\!]} = \mathcal{A}_{[\![\Gamma]\!]} \cup \mathcal{A}_{[\![T]\!]}$.

Processes for constants and free identifiers $x : T \vdash x : T$ are defined in Table 1. The process for $\mathsf{diverge}_B$ performs the div process after communicating the initial question event.

$$
\begin{array}{l}
[\![\Gamma \vdash v : \mathsf{exp}D]\!] = Q.q \to A.v \to \mathit{SKIP},\ \ v \in \mathcal{A}_{[\![D]\!]} \\
[\![\Gamma \vdash \mathsf{skip} : \mathsf{com}]\!] = Q.\mathit{run} \to A.\mathit{done} \to \mathit{SKIP} \\
[\![\Gamma \vdash \mathsf{diverge}_B : B]\!] = Q?\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to \mathsf{div} \\
[\![x : \mathrm{exp}D \vdash x : \mathrm{exp}D]\!] = Q.q \to x.Q.q \to x.A?\mathsf{a} : \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} \to A.\mathsf{a} \to \mathit{SKIP} \\
[\![x : \mathsf{com} \vdash x : \mathsf{com}]\!] = Q.\mathit{run} \to x.Q.\mathit{run} \to x.A.\mathit{done} \to A.\mathit{done} \to \mathit{SKIP} \\
[\![x : \mathsf{var}D \vdash x : \mathsf{var}D]\!] = (Q.\mathit{read} \to x.Q.\mathit{read} \to x.A?\mathsf{a} : \mathrm{A}^{\mathit{read}}_{[\![\mathsf{var}D]\!]} \to A.\mathsf{a} \to \mathit{SKIP}) \\
\qquad\qquad \Box\ (Q.\mathit{write}?v : \mathcal{A}_{[\![D]\!]} \to x.Q.\mathit{write}.v \to x.A.\mathit{ok} \to A.\mathit{ok} \to \mathit{SKIP}) \\
[\![x : B_1 \to \ldots \to B_k \to B \vdash x : B_1 \to \ldots \to B_k \to B]\!] = Q?\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to x.Q.\mathsf{q} \to \\
\qquad \mu\, L \bullet \Big(\mathit{SKIP} \,\Box \Big(\Box_{j=1}^{k} (x.Q.j?\mathsf{q}_j : \mathrm{Q}_{[\![B_j]\!]} \to Q.j.\mathsf{q}_j \to A.j?\mathsf{a}_j : \mathrm{A}^{\mathsf{q}_j}_{[\![B_j]\!]} \to \\
\qquad x.A.j.\mathsf{a}_j \to \mathit{SKIP}) \,⨟\, L \Big)\Big) \,⨟\, x.A?\mathsf{a} : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a} \to \mathit{SKIP}
\end{array}
$$

Table 1. Processes for constants and free identifiers

For each language construct 'c', a process P_{c} which corresponds to its strategy is defined in Table 2. For example, P_{or} nondeterministically runs either its first or its second argument. Events of the first (resp., second) argument of 'or' occur on channels tagged with index 1 (resp., 2). Then, for each composite term $\mathsf{c}(M_1, \ldots, M_n)$ consisting of a language construct 'c' and subterms $M_1, \ldots, M_n$, we define $[\![\mathsf{c}(M_1, \ldots, M_n)]\!]$ from the process P_{c} and processes $[\![M_i]\!]$ and $[\![M_i]\!]^*$ [5], using only the CSP operators of renaming, parallel composition and hiding. For example, the process for 'or' is defined as:

$$
\begin{array}{rl}
[\![\Gamma \vdash M_1\, \mathsf{or} M_2 : B]\!] = & ([\![\Gamma \vdash M_1 : B]\!][Q_1/Q, A_1/A] \,\Box\, \mathit{SKIP}) \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\
& \big(([\![\Gamma \vdash M_2 : B]\!][Q_2/Q, A_2/A] \,\Box\, \mathit{SKIP}) \underset{\{\mid Q_2, A_2 \mid\}}{\parallel} \\
& P_{\mathsf{or}} \setminus \{\mid Q_2, A_2 \mid\}\big) \setminus \{\mid Q_1, A_1 \mid\}
\end{array}
$$

After renaming the channels Q, A to Q_1, A_1 in the process for M_1, and to Q_2, A_2 in the process for M_2 respectively, the processes for M_1 and M_2 are composed

[5] P^* is a process which performs the process P arbitrary many times.

with P_{or}. The composition is performed by synchronising the component processes on events occurring on channels Q_1, A_1, Q_2, A_2, which are then hidden. Since one of the processes for M_1 and M_2 will not be run in the composition, *SKIP* is used to enable such empty termination.

$$
\begin{aligned}
&P_{\mathsf{op}} = Q.q \to Q_1.q \to A_1?\mathsf{a}_1 : \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} \to Q_2.q \to A_2?\mathsf{a}_2 : \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} \\
&\qquad \to A.\mathsf{a}_1\ \mathsf{op}\ \mathsf{a}_2 \to \mathit{SKIP} \\
&P_{;} = Q?\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to Q_1.\mathit{run} \to A_1.\mathit{done} \to Q_2.\mathsf{q} \to A_2?\mathsf{a} : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a} \to \mathit{SKIP} \\
&P_{\mathsf{if}} = Q.\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to Q_0.q \to A_0?\mathsf{a}_0 : \mathrm{A}^{q}_{[\![\mathsf{expbool}]\!]} \to \mathsf{if}\ (\mathsf{a}_0)\ \mathsf{then}\ \big(Q_1.\mathsf{q} \to \\
&\qquad A_1?\mathsf{a}_1 : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a}_1 \to \mathit{SKIP}\big)\ \mathsf{else}\ \big(Q_2.\mathsf{q} \to A_2?\mathsf{a}_2 : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a}_2 \to \mathit{SKIP}\big) \\
&P_{\mathsf{while}} = Q.\mathit{run} \to \mu\, p \bullet Q_1.q \to A_1?\mathsf{a}_1 : \mathrm{A}^{q}_{[\![\mathsf{expbool}]\!]} \to \Big(\mathsf{if}\ (\mathsf{a}_1)\ \mathsf{then} \\
&\qquad \big(Q_2.\mathit{run} \to A_2.\mathit{done} \to p\big)\ \mathsf{else}\ \big(A.\mathit{done} \to \mathit{SKIP}\big)\Big) \\
&P_{:=} = Q.\mathit{run} \to Q_1.q \to A_1?\mathsf{a} : \mathrm{A}^{q}_{[\![\mathsf{exp}D]\!]} \to Q_2.\mathit{write}.\mathsf{a} \to A_2.\mathit{ok} \to A.\mathit{done} \to \mathit{SKIP} \\
&P_{!} = Q.q \to Q_1.\mathit{read} \to A_1?\mathsf{a} : \mathrm{A}^{\mathit{read}}_{[\![\mathsf{var}D]\!]} \to A.\mathsf{a} \to \mathit{SKIP} \\
&P_{\mathsf{or}} = \big(Q.\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to Q_1.\mathsf{q} \to A_1?\mathsf{a}_1 : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a}_1 \to \mathit{SKIP}\big)\ \Box \\
&\qquad \big(Q.\mathsf{q} : \mathrm{Q}_{[\![B]\!]} \to Q_2.\mathsf{q} \to A_2?\mathsf{a}_2 : \mathrm{A}^{\mathsf{q}}_{[\![B]\!]} \to A.\mathsf{a}_2 \to \mathit{SKIP}\big) \\
&P_{\mathsf{new}}(x, v) = Q.\mathit{run} \to Q_1.\mathit{run} \to U_D(x, v) \\
&U_D(x, v) = \big(x.Q.\mathit{read} \to x.A.v \to U_D(x, v)\big)\Box \\
&\qquad \big(x.Q.\mathit{write}?v' : \mathcal{A}_{[\![D]\!]} \to x.A.\mathit{ok} \to U_D(x, v')\big)\ \Box\ \big(A_1.\mathit{done} \to A.\mathit{done} \to \mathit{SKIP}\big)
\end{aligned}
$$

Table 2. Processes for constructs

Example 2. Consider the term from Example 1:

$$
\begin{aligned}
f : \mathsf{com} \to \mathsf{com} \vdash \mathsf{newint}_2\ x := 0\ \mathsf{in}\ &\big(f(x := 1)\ ;\ \mathsf{if}\ (x = 1)\ \mathsf{then\ diverge_{com}}\big) \\
\mathsf{or}\ &\big(f(x := 1)\ ;\ \mathsf{if}\ (x = 0)\ \mathsf{then\ diverge_{com}}\big) : \mathsf{com}
\end{aligned}
$$

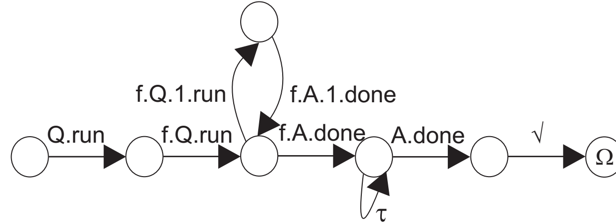

Fig. 1. A strategy as a LTS

The LTS of the CSP process representing this term is shown in Fig. 1. The first argument of 'or' terminates successfully when f does not call its argument; otherwise it diverges. The second argument of 'or' terminates successfully when

f calls its argument, one or more times; otherwise it diverges. The set of divergences is $Q.run \cdot f.Q.run \cdot (f.Q.1.run \cdot f.A.1.done)^* \cdot f.A.done$, while the set of traces that end with ✓ is $Q.run \cdot f.Q.run \cdot (f.Q.1.run \cdot f.A.1.done)^* \cdot f.A.done \cdot A.done \cdot \checkmark$. Notice that no references to the variable x appear in the model because it is locally defined. □

5 Correctness and Formal Properties

We now show that for any term from 2nd-order recursion-free EIA with finite data types, the sets of all terminated traces and divergences of its CSP interpretation are isomorphic to the sets of all complete plays and divergences of its fully abstract game semantic model. Given a term $\Gamma \vdash M : T$, we denote by $[\![\Gamma \vdash M : T]\!]^{GS}$ its game semantic model as described in Section 3, and we denote by $[\![\Gamma \vdash M : T]\!]^{CSP}$ its CSP interpretation as described in Section 4.

Theorem 1. *For any term $\Gamma \vdash M : T$, we have:*

$$\mathsf{traces}^{\checkmark}([\![\Gamma \vdash M : T]\!]^{CSP}_T) \stackrel{\phi}{\equiv} comp([\![\Gamma \vdash M : T]\!]^{GS})$$
$$div(\mathsf{divergences}([\![\Gamma \vdash M : T]\!]^{CSP}_{FD})) \stackrel{\phi}{\equiv} D_{[\![\Gamma \vdash M : T]\!]^{GS}}$$

where $\mathsf{traces}^{\checkmark}(P_T)$ *is the set of all terminated traces of process P that end with ✓ in its traces semantics,* $div(\mathsf{divergences}(P_{FD}))$ *is the set of all minimal divergences of P in its failures-divergences semantics, and ϕ is an isomorphism defined by:*

- *For a type T of the form $B_1 \to \ldots \to B_k \to B$:*
  $\phi(a) = L.j.a$, *for* $a \in M_{[\![B_j]\!]^{GS}}$, $\lambda^{\mathsf{QA}}(a) = L$, $1 \leq j \leq k$
  $\phi(a) = L.a$, *for* $a \in M_{[\![B]\!]^{GS}}$, $\lambda^{\mathsf{QA}}(a) = L$
- *For any* $x : B'_1 \to \ldots \to B'_{k_x} \to B' \in \Gamma$*:*
  $\phi(a) = x.L.i.a$, *for* $a \in M_{[\![B'_i]\!]^{GS}}$, $\lambda^{\mathsf{QA}}(a) = L$, $1 \leq i \leq k_x$
  $\phi(a) = x.L.a$, *for* $a \in M_{[\![B']\!]^{GS}}$, $\lambda^{\mathsf{QA}}(a) = L$

Proof. The proof is by a routine induction on the typing rules. □

Corollary 1.

$$\Gamma \vdash M \underset{\sim}{\sqsubset}_{may} N \;\Leftrightarrow [\![\Gamma \vdash N : T]\!]^{CSP} \,\Box\, RUN_{\mathcal{A}_{[\![\Gamma \vdash T]\!]}} \sqsubseteq_T \; [\![\Gamma \vdash M : T]\!]^{CSP}$$

where $RUN_A = \mu\, p{\bullet}?x : A \to p$*, i.e. it is a process which can perform any event from the set A, but it cannot perform ✓ or any other event not in A.*

Proof. It follows from Proposition 2, Theorem 1, and the traces semantics of the □ operator and the $RUN_{\mathcal{A}_{[\![\Gamma \vdash T]\!]}}$ process. □

The checks performed by FDR terminate only for finite-state processes, i.e. those whose labelled transition systems are finite. It is easy to show that this is the case for the processes interpreting the EIA terms by extending the same result for IA terms in [9]. As a corollary, we have that observational may-approximation is decidable using FDR.

Corollary 2. Observational may-approximation *and* may-equivalence *of EIA terms are decidable by using FDR tool.*

Example 3. Consider the process for the term M from Examples 1 and 2. As shown in Fig. 1, we have that $\mathsf{traces}^{\checkmark}(\llbracket M \rrbracket_T^{CSP}) = Q.run \cdot f.Q.run \cdot (f.Q.1.run \cdot f.A.1.done)^* \cdot f.A.done \cdot A.done \cdot \checkmark$. But, this is the same as the set of all terminated traces of the process for $f : \mathsf{com} \rightarrow \mathsf{com} \vdash f(\mathsf{skip}) : \mathsf{com}$. So, these two terms are may-equivalent. □

By Proposition 3 and Theorem 1, we have that *must-approximation* $\Gamma \vdash M \precsim_{must} N$ can be determined by the following procedure:

(1) Check: $\llbracket \Gamma \vdash M \rrbracket^{CSP} \sqsubseteq_T \llbracket \Gamma \vdash N \rrbracket^{CSP}$, $\llbracket \Gamma \vdash N \rrbracket^{CSP}$ is divergence-free, and $\llbracket \Gamma \vdash M \rrbracket^{CSP}$ is divergence-free. If all three checks hold, then terminate with answer $\Gamma \vdash M \precsim_{must} N$, else go to (2).
(2) Let C_1, C_2, and C_3 be the sets of all minimal counterexamples returned by the above three checks respectively. Set $C := C_1 \cup (C_2 \backslash C_3)$. If $C_3 = \emptyset$, then terminate with answer $\Gamma \vdash M \not\precsim_{must} N$, else go to (3).
(3) For each $c \in C$, check whether there exists $s' \sqsubseteq^{odd} c$, such that $s' \sqsubseteq d$ for some $d \in C_3$. If this is correct, then terminate with $\Gamma \vdash M \precsim_{must} N$, otherwise with $\Gamma \vdash M \not\precsim_{must} N$.

Proposition 4. *The procedure for determining must-approximation is correct.*

Proof. We can check by inspection that all answers returned by the procedure are correct. Let the procedure terminate in Step (1). Then, $\mathsf{traces}(\llbracket \Gamma \vdash N \rrbracket^{CSP}) \subseteq \mathsf{traces}(\llbracket \Gamma \vdash M \rrbracket^{CSP})$, $\mathsf{divergences}(\llbracket \Gamma \vdash N \rrbracket^{CSP}) = \emptyset$, and $\mathsf{divergences}(\llbracket \Gamma \vdash M \rrbracket^{CSP}) = \emptyset$. So, we have that $(T_{\llbracket \Gamma \vdash N \rrbracket^{GS}} \cup D_{\llbracket \Gamma \vdash N \rrbracket^{GS}}) \backslash (T_{\llbracket \Gamma \vdash M \rrbracket^{GS}} \cup D_{\llbracket \Gamma \vdash M \rrbracket^{GS}}) = \emptyset$, which implies that $\Gamma \vdash M \precsim_{must} N$. The other cases are similar. □

Corollary 3. Observational must-approximation *and* must-equivalence *of EIA terms are decidable by using FDR tool.*

Example 4. We can verify that the term M from Examples 1 and 2 is must-equivalent with $\vdash \mathsf{diverge}_{\mathsf{com}}$. Let C_1, C_2, C_3, and C_4 be the minimal counterexamples associated with the following checks: $\llbracket \mathsf{diverge}_{\mathsf{com}} \rrbracket^{CSP} \sqsubseteq_T \llbracket M \rrbracket^{CSP}$, $\llbracket M \rrbracket^{CSP} \sqsubseteq_T \llbracket \mathsf{diverge}_{\mathsf{com}} \rrbracket^{CSP}$, $\llbracket M \rrbracket^{CSP}$ and $\llbracket \mathsf{diverge}_{\mathsf{com}} \rrbracket^{CSP}$ are divergence-free, respectively. Then, $C_1 = \{Q.run \cdot f.Q.run\}$, $C_2 = \emptyset$, $C_3 = \{Q.run \cdot f.Q.run \cdot f.A.done\}$, $C_4 = \{Q.run\}$. By following the previously described procedure for determining must-approximation, it is easy to check that these two terms are must-equivalent. □

In addition to checking observational equivalence of two terms, it is desirable to be able to check properties, safety (see [9, 8] for details) and liveness, of terms. By Proposition 1 and Theorem 1, we have that:

Corollary 4. Must-termination *of a term* $\Gamma \vdash M$ *is decidable using FDR tool by checking one divergence-freedom test:*

$$\llbracket \Gamma \vdash M \rrbracket^{CSP} \text{ is divergence-free} \tag{1}$$

If the test (1) does not hold, then the term *diverges* and one or more counter-examples reported by the FDR debugger can be used to explore the reasons why. Otherwise, the term does not diverge, i.e. it *terminates*.

Example 5. By testing the process for the term while $(true)$ do skip for divergence-freedom, we can verify that the term diverges. The counter-example is: $Q.run$.

We can also verify that the term from Examples 1 and 2 diverges. The obtained counter-example is:

$$Q.run \;\; f.Q.run \;\; f.A.done \qquad \Box$$

6 Applications

We have implemented a tool, which automatically converts a term into a CSP process which represents its game semantics. The resulting CSP process is defined by a script in machine readable CSP [15] which the tool outputs. In the input syntax, we use simple type annotations to indicate what finite sets of integers will be used to model integer free identifiers and local variables. 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 $max\{n_1, n_2\}$.

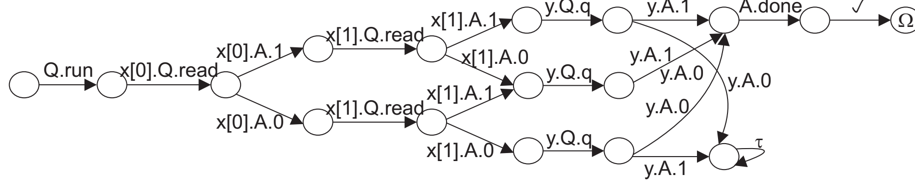

Fig. 2. Model for linear search with k=2.

We now analyse an implementation of the linear search algorithm:

$x[k]$: varint_2, y : expint_2 $\vdash$
newint_2 $a[k] := 0$ in
newint_{k+1} $i := 0$ in
while $(i < k)$ do $\{$ $a[i] := x[i];$ $\;\; i := i + 1;$ $\}$
newint_2 $z := y$ in
newbool $present := false$ in
while (not $present$) do $\{$
    if $(i < k)$ then if $(a[i] = z)$ then $present := true;$
    $i := i + 1;$ $\}$: com

The code includes a meta variable $k > 0$, representing array size, which will be replaced by several different values. The data stored in the arrays and the

expression y is of type int_2, i.e. two distinct values 0 and 1 can be stored, and the type of index i is int_{k+1}, i.e. one more than the size of the array. 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 y. The array being effectively searched, $a[]$, and the variable z, are not visible from the outside of the term because they are locally defined, so only reads from the non-local identifiers x and y are seen in the model of this term.

A labelled transition system of the CSP process for the term with $k = 2$ is shown in Fig. 2. It illustrates the possible behaviours of this term: if the value read from y has occurred in $x[]$ then the term terminates successfully; otherwise the term diverges. If we test this process for divergence-freedom, we obtain the following counter-example:

$$Q.run \;\; x[0].Q.read \;\; x[0].A.1 \;\; x[1].Q.read \;\; x[1].A.1 \;\; y.Q.q \;\; y.A.0$$

So the linear search term diverges when the value read from y does not occur in the array $x[]$ making the while loop forever.

Table 3 shows some experimental results for checking divergence-freedom. The experiment consisted of running the tool on the linear search term with different values of k, and then letting FDR generate its model and test its divergence-freedom. The latter stage involved a number of hierarchical compressions, as described in [9]. For different values of k, we list the execution time in seconds, and the size of the final model. We ran FDR on a Machine AMD Sempron Processor 3500^{+} with 2GB RAM.

Arr size k	Time (sec)	Model states
5	2	35
10	5	65
20	39	125
30	145	185

Table 3. Model generation of linear search

7 Conclusion

We presented a compositional approach for verifying equivalence and liveness properties of nondeterministic sequential programs with finite data types. An interesting direction for extension is to consider infinite integers with all the usual operators. Counter-example guided abstraction refinement procedures [7, 8] for verifying safety properties can be adopted to the specific setting for verifying liveness properties. It is also important to extend the proposed approach to programs with concurrency [10], probabilistic constructs [13], and other features.

References

1. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P.W.O'Hearn and R.D.Tennent, editors, *Algol-like languages.* (Birkhaüser, 1997).
2. A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In Proceedings of *TACAS*, LNCS **4693**, (2008), 78–92.
3. M. A. Colon and H. B. Sipma. Practical Methods for Proving Program Termination. In Proceedings of *CAV*, LNCS **2404**, (2002), 442–454.
4. M. A. Colon and H. B. Sipma. Synthesis of linear ranking functions. In Proceedings of *TACAS*, LNCS **2031**, (2001), 67–81.
5. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In Proceedings of *SAS*, LNCS **3672**, (2005), 86–101.
6. A. Dimovski and R. Lazić. CSP Representation of Game Semantics for Second-order Idealized Algol. In Proceedings of *ICFEM*, LNCS **3308**, (2004), 146–161.
7. A. Dimovski, D. R. Ghica, and R. Lazić. Data-Abstraction Refinement: A Game Semantic Approach. In Proceedings of *SAS*, LNCS **3672**, (2005), 102–117.
8. A. Dimovski, D. R. Ghica, R. Lazić. A Counterexample-Guided Refinement Tool for Open Procedural Programs. In Proc. of *SPIN*, LNCS **3925**, (2006), 288–292.
9. A. Dimovski and R. Lazić. Compositional Software Verification Based on Game Semantics and Process Algebras. In *Int. Journal on STTT* **9(1)**, (2007), 37–51.
10. D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency. In Proceedings of *FoSSaCS*, LNCS **2987**, (2004), 211-255.
11. R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In Proceedings of *LICS*, IEEE, (1999). 422–430.
12. R. Harmer. Games and Full Abstraction for Nondeterministic Languages. Ph. D. Thesis Imperial College, 1999.
13. A. Legay, A. Murawski, J. Ouaknine, and J. Worrell. On Automated Verification of Probabilistic Programs. In Proceedings of *TACAS*, LNCS **4693**, (2008), 173–187.
14. A. Murawski. Reachability Games and Game Semantics: Comparing Nondeterministic Programs. In Proceedings of *LICS*, IEEE, (2008), 173–183.
15. W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.