CSP Representation of Game Semantics for Second-order Idealized Algol*

Aleksandar Dimovski and Ranko Lazić

Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
{aleks,lazic}@dcs.warwick.ac.uk

Abstract. We show how game semantics of an interesting fragment of Idealised Algol can be represented compositionally by CSP processes. This enables observational equivalence and a range of properties of terms-in-context (i.e. open program fragments) to be checked using the FDR tool. We have built a prototype compiler which implements the representation, and initial experimental results are positive.

1 Introduction

Context. One of the main breakthroughs in theoretical computer science in the past decade has been the development of game semantics (e.g. [11, 1]). Types are modelled by games between Player (i.e. term) and Opponent (i.e. context or environment), and terms are modelled by strategies. This has produced the first accurate (i.e. fully abstract and fully complete) models for a variety of programming languages and logical systems.

It has recently been shown that, for several interesting programming language fragments, game semantics yields algorithms for software model checking. The focus has been on Idealised Algol (IA) [14] with active expressions. IA is similar to Core ML. It is a compact programming language which combines the fundamental features of imperative languages with a full higher-order procedure mechanism. For example, simple forms of classes and objects may be encoded in IA.

For second-order recursion-free IA with iteration and finite data types, [9] shows that game semantics can be represented by regular expressions, so that observational equivalence between any two terms can be decided by equality of regular languages. For third order and without iteration, it was established in [13] that game semantics can be represented by deterministic pushdown automata, which makes observational equivalence decidable by equality of deterministic context-free languages. Classes of properties other than observational equivalence can also be checked algorithmically, such as language containment or Hoare triples (e.g. [2]).

* We acknowledge support by the EPSRC (GR/S52759/01). The first author was also supported by the Intel Corporation.

In recent years, software model checking has become an active research area, and powerful tools have been built (e.g. [4]). Compared with other approaches to software model checking, the approach based on game semantics has a number of advantages [3]:

- there is a model for any term-in-context, which enables verification of program fragments which contain free variable and procedure names;
- game semantics is compositional, which facilitates verifying a term to be broken down into verifying its subterms;
- terms are modelled by how they interact with their environments, and details of their internal state during computations are not recorded, which results in small models.

Our contribution. In this paper, we show how game semantics of second-order recursion-free IA with iteration and finite data types can be represented in the CSP process algebra. For any term-in-context, we compositionally define a CSP process whose terminated traces are exactly all the complete plays of the strategy for the term. Observational equivalence between two terms can then be decided by checking two traces refinements between CSP processes.

Compared with the representation by regular expressions (or automata) [9], the CSP representation brings several benefits:

- CSP operators preserve traces refinement (e.g. [17]), which means that a CSP process representing a term can be optimised and abstracted compositionally at the syntactic level (e.g. using process algebraic laws), and its set of terminated traces will be preserved or enlarged;
- the ProBE and FDR tools [7] can be used to explore CSP processes visually, to check traces refinements automatically, and to debug interactively when traces refinements do not hold;
- compositional state-space reduction algorithms in FDR [16] enable smaller models to be generated before or during refinement checking;
- composition of strategies, which is used in game semantics to obtain the strategy for a term from strategies for its subterms, is represented in CSP by renaming, parallel composition and hiding operators, and FDR is highly optimised for verification of such networks of processes;
- parameterised terms (as a simple example, a program which reverses an array of values of an arbitrary data type α) can be interpreted by single parameterised processes, which can then be verified e.g. using techniques from the infinite-state model checking literature.

We have implemented a prototype compiler which, given any IA term-in-context, outputs a CSP process representing its game semantics. We report some initial experimental results, which show that for model generation, FDR outperforms the tool based on the representation by regular expressions [8].

Organisation In the next section, we present the fragment of IA we are addressing. Section 3 contains brief introductions to game semantics, CSP and

FDR. In section 4, we define the CSP representation of game semantics for the IA fragment. Correctness of the CSP model, and decidability of observational equivalence by traces refinement, are shown in section 5. We present the experimental results in section 6. Finally, in section 7, we conclude and discuss future work.

Further proof details and examples can be found in the full paper [5].

2 The programming language

Idealized Algol [14] is a functional-imperative language with usual imperative features as iteration, branching, assignment, sequential composition, combined with a function mechanism based on a typed call-by-name lambda calculus. We consider only the recursion-free second-order fragment of this language. We will only work with finite data sets.

The language has basic data types τ, which are a finite subset of the integers and the booleans. The phrase types of the language are expressions, commands and variables, plus first-order function types.

$$\begin{aligned} \tau &::= \mathit{int} \mid \mathit{bool} \\ \sigma &::= \mathit{exp}[\tau] \mid \mathit{comm} \mid \mathit{var}[\tau] \\ \theta &::= \sigma \mid \sigma \times \sigma \times \cdots \sigma \to \sigma \end{aligned}$$

Terms are introduced using type judgements of the form:

$$\Gamma \vdash M : \theta, \text{ where } \Gamma = \{\iota_1 : \theta_1, \cdots, \iota_k : \theta_k\}$$

For the sake of simplicity, we assume that terms are β-normal, so there is no λ abstractions, and also function application is restricted to free identifiers. The terms of the language and their typing rules are given in Table 1.

For type $\mathit{exp}[\mathit{int}]$, the finitary fragment contains constants n belonging to a finite subset of the set of integers, and for type $\mathit{exp}[\mathit{bool}]$ there are constants *true* and *false*. For type *comm*, there are basic commands *skip*, to do nothing, and *diverge* which causes a program to enter an unresponsive state similar to that caused by an infinite loop. The other commands are assignment to variables, $V := E$, conditional operation, *if* B *then* C *else* C', and while loop, *while* B *do* C. Also, we have sequential composition of commands $C \,⨾\, C'$ as well as sequential composition of a command with an expression or a variable. There are also term formers for dereferencing variables, !V, application of first-order free identifiers to arguments $\iota M_1 \cdots M_k$, and local variable declaration $\mathit{new}[\tau]$ ι *in* C. Finally, we have a function definition *let* constructor.

3 Background

3.1 Game semantics

We give an informal overview of game semantics and we illustrate it with some examples. A more complete introduction can be found in [2].

Table 1. Terms and typing rules

$$\frac{}{\Gamma \vdash true : exp[bool]} \qquad \frac{}{\Gamma \vdash n : exp[int]}$$

$$\frac{}{\Gamma \vdash skip : comm} \qquad \frac{}{\Gamma \vdash diverge : comm}$$

$$\frac{}{\Gamma, \iota : \theta \vdash \iota : \theta} \qquad \frac{\Gamma \vdash V : var[\tau]}{\Gamma \vdash !V : exp[\tau]}$$

$$\frac{\Gamma \vdash E_1 : exp[int] \quad \Gamma \vdash E_2 : exp[int]}{\Gamma \vdash E_1 + E_2 : exp[int]} \qquad \frac{\Gamma \vdash E_1 : exp[int] \quad \Gamma \vdash E_2 : exp[int]}{\Gamma \vdash E_1 = E_2 : exp[bool]}$$

$$\frac{\Gamma \vdash B_1 : exp[bool] \quad \Gamma \vdash B_2 : exp[bool]}{\Gamma \vdash B_1 \ and \ B_2 : exp[bool]} \qquad \frac{\Gamma \vdash B : exp[bool]}{\Gamma \vdash not \ B : exp[bool]}$$

$$\frac{\Gamma \vdash V : var[\tau] \quad \Gamma \vdash E \ : \ exp[\tau]}{\Gamma \vdash V := E : comm} \qquad \frac{\Gamma \vdash C : comm \quad \Gamma \vdash M : \sigma}{\Gamma \vdash C \mathbin{;} M : \sigma}$$

$$\frac{\Gamma \vdash B : exp[bool] \quad \Gamma \vdash M_1 : \sigma \quad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash if \ B \ then \ M_1 \ else \ M_2 : \sigma} \qquad \frac{\Gamma \vdash B : exp[bool] \quad \Gamma \vdash C : comm}{\Gamma \vdash while \ B \ do \ C : comm}$$

$$\frac{\Gamma \vdash \iota : \sigma_1 \times \cdots \times \sigma_k \to \sigma \quad \Gamma \vdash M_i : \sigma_i}{\Gamma \vdash \iota(M_1, M_2, \ldots, M_k) \ : \ \sigma} \qquad \frac{\Gamma, \iota : var[\tau] \vdash C : comm}{\Gamma \vdash new[\tau] \ \iota \ in \ C : comm}$$

$$\frac{\Gamma, \iota : \sigma_1 \times \cdots \times \sigma_k \to \sigma' \vdash M : \sigma \qquad \Gamma, \iota_1 : \sigma_1, \ldots, \iota_k : \sigma_k \vdash N : \sigma'}{\Gamma \vdash let \ \iota \ (\iota_1 : \sigma_1, \ldots, \iota_k : \sigma_k) \ = \ N \ in \ M : \sigma}$$

As the name suggests, game semantics models computation as a certain kind of game, with two participants, called Player(P) and Opponent(O). P represents the term (program), while O represents the environment, i.e. the context in which the term is used. A play between O and P consists of a sequence of moves, governed by rules. For example, O and P need to take turn and every move needs to be justified by a preceding move. The moves are of two kinds, questions and answers.

To every type in the language corresponds a game — the set of all possible plays (sequences of moves). A term is represented as a set of all complete plays in the appropriate game, more precisely as a strategy for that game — a predetermined way for P to respond to O's moves.

For example, in the game for the type $exp[\tau]$, there is an initial move q and corresponding to it a single response to return its value. So a complete play for a constant $\vdash \ c \ : \ exp[\tau]$ is:

O: q (opponent asks for value)

P: c (player answers to the question)

Consider a more complex example $\iota : exp[int] \to \text{exp}[int] \ \vdash \ \iota(2) : exp[int]$, where the identifier ι is some non-locally defined function. A play for this term begins with O asking for the value of the result expression by playing the question move q, and P replies asking for the returned value of the non-local function ι, move q^{ι}. In this situation, the function ι may need to evaluate its argument, represented by O's move $q^{\iota 1}$ — what is the value of the first argument to ι. P

will respond with answer $2^{\iota 1}$. Here, O could repeat the question $q^{\iota 1}$ to represent the function which evaluates its argument more than once. In the end, when O plays the move n^{ι} — the value returned from ι, P will copy this value and answer to the first question with n. A sample complete play for this term, when the function ι evaluates only once its argument, is:

O: q (asks for result value)
P: q^{ι} (P asks for value returned from function ι)
O: $q^{\iota 1}$ (O questions what is the first argument to ι)
P: $2^{\iota 1}$ (P answers: 2)
O: n^{ι} (O supplies the value returned from ι)
P: n (P gives the answer to the first question)

In the game for commands, there is an initial move *run* to initiate a command, and a single response *done* to signal termination of the command. Thus the only complete play for the command $\vdash$ *skip* : *comm* is:

O: *run* (start executing)
P: *done* (terminate command)

Variables are represented as objects with 2 methods: the "read method" for dereferencing, represented by an initial move *read*, with response an element of τ, and the "write method", for assignment, represented by an initial move $write(x)$ for any element x of τ, to which there is only one possible response *ok*. For example, a complete play for the command $v : var[\tau] \vdash v :=!v + 1 : comm$ is:

O: *run*
P: $read_v$ (what is the value of v)
O: 2 (O supplies the value 2)
P: $write(3)_v$ (write 3 into v)
O: ok_v (the assignment is complete)
P: *done*

When P asks to read from v, O can return an arbitrary value, i.e. not necessarily the last value P wrote into v. This is because, in general, the value of v can also be modified by the context into which the term will be placed. However, when a variable is declared, "good variable" behaviour is enforced within the scope of the declaration.

3.2 CSP

CSP (Communicating Sequential Processes) is a language for modelling interacting components. Each component is specified through its behaviour which is given as a process. This section only introduces the CSP notation and the ideas used in this paper. For a fuller introduction to the language the reader is referred to [17].

CSP processes are defined in terms of the events that they can perform. The set of all possible events is denoted Σ. Events may be atomic in structure or may consist of a number of distinct components. For example, an event *write*.1 consists of two parts: a channel name *write*, and a data value 1. If N is a set of

values that can be communicated down the channel *write*, then *write*.N will be the set of events $\{write.n \mid n \in N\}$. Given a channel c, we can define the set of all events that can arise on the channel c, by $\{\mid c \mid\} = \{c.w \in \Sigma\}$.

We use the following collection of process operators:

$$P ::= p \mid STOP \mid SKIP \mid RUN_A \mid ?x : A \rightarrow P \mid \mu p.P \mid P_1 \Box P_2 \\ \mid P_1 \nleftarrow b \nrightarrow P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid P \setminus A \mid P\,[a/b] \mid P_1 \fatsemi P_2$$

where A represents a set of events, P a process expression and p is a process name (or identifier).

The process $STOP$ performs no actions and never communicates. It is useful for providing a simple model of a deadlocked system. $SKIP$ is a process that successfully terminates causing the special event ✓ (✓ is not in Σ). RUN_A can always communicate any event of set A desired by the environment. A choice process, $?x : A \rightarrow P$, can perform any event from set A and then behaves as P. For example, $RUN_A = ?x : A \rightarrow RUN_A$. Process $\mu p.P$, where P is any process involving p, represents recursion. It can return to its initial state and in that way communicate forever. The process $P_1 \Box P_2$ can behave either as P_1 or as P_2, its possible communications are those of P_1 and those of P_2. Conditional choice process $P_1 \nleftarrow b \nrightarrow P_2$ means the same as *if b then P else Q*, and has the obvious meaning as in all programming languages. Process $P_1 \underset{A}{\parallel} P_2$ runs P_1 and P_2 in parallel, making them synchronise on events in A and allowing all others events freely. The parallel combination terminates successfully when both component processes do it. This is known as distributed termination. Process $P \setminus A$, behaves as P except that events from A become invisible events τ (τ is not in Σ). The renaming operator [] is used to rename some events of a given process. We will only use injective renaming and notation $P[a/b]$ to mean that the event or channel b in P is replaced by a, and all others remain the same. Sequential composition $P_1 \fatsemi P_2$ runs P_1 until it terminates successfully producing the special event ✓, and then runs P_2.

CSP processes can be given semantics by sets of their traces. A trace is a finite sequence of events. A sequence tr is a trace of a process P if there is some execution of P in which exactly that sequence of events is performed. Examples of traces include $\langle\rangle$ (the empty trace, which is possible for any process) and $\langle a_1, a_2\rangle$ which is a possible trace of RUN_A, if $a_1, a_2 \in A$. The set $traces(P)$ is the set of all possible traces of process P.

Using traces sets, we can define traces refinement. A process P_2 is a traces refinement of another, P_1, if all the possible sequences of communications which P_2 can do are also possible for P_1. Or more formally:

$$P_1 \sqsubseteq_T P_2 \;\Leftrightarrow\; traces(P_2) \subseteq traces(P_1) \tag{1}$$

CSP processes can also be described by transition systems or state machines. The transition system of a process is a directed graph showing the states which the process can go through and the events from $\Sigma \cup \{✓, \tau\}$ that it can perform to get from one to another state. The successful termination ✓ is always the last event and leads to an end state Ω.

The FDR tool [7] is a refinement checker for CSP processes. It contains several procedures for compositional state-space reduction. Namely, before generating a transition system for a composite process, transition systems of its component processes can be reduced, while preserving semantics of the composite process [16]. FDR is also optimised for checking refinements by processes which consist of a number of component processes composed by operators such as renaming, parallel composition and hiding.

4 CSP representation of game semantics

With each type θ, we associate a set of possible events: an alphabet $\mathcal{A}_\theta$. The alphabet of a type contains events $\mathbf{q} \in \mathbb{Q}_\theta$ called questions, which are appended to channel with name Q, and for each question $\mathbf{q}$, there is a set of events $\mathbf{a} \in \mathbb{A}_\theta^{\mathbf{q}}$ called answers, which are appended to channel with name A.

$\mathcal{A}_{int} = \{0, \cdots, N_{max} - 1\}, \quad \mathcal{A}_{bool} = \{true, false\}$
$\mathbb{Q}_{exp[\tau]} = \{q\}, \quad \mathbb{A}^q_{exp[\tau]} = \mathcal{A}_\tau$
$\mathbb{Q}_{comm} = \{run\}, \quad \mathbb{A}^{run}_{comm} = \{done\}$
$\mathbb{Q}_{var[\tau]} = \{read, write.x \mid x \in \mathcal{A}_\tau\}, \quad \mathbb{A}^{read}_{var[\tau]} = \mathcal{A}_\tau, \quad \mathbb{A}^{write.x}_{var[\tau]} = \{ok\}$
$\mathbb{Q}_{\sigma_1 \times \cdots \times \sigma_k \to \sigma_0} = \{j.\mathbf{q} \mid \mathbf{q} \in \mathbb{Q}_{\sigma_j}, 0 \le j \le k\},$
$\mathbb{A}^{j.\mathbf{q}}_{\sigma_1 \times \cdots \times \sigma_k \to \sigma_0} = \{j.\mathbf{a} \mid \mathbf{a} \in \mathbb{A}^{\mathbf{q}}_{\sigma_j}\}$, for $0 \le j \le k$
$\mathcal{A}_\theta = Q.\mathbb{Q}_\theta \cup A.\bigcup_{\mathbf{q} \in \mathbb{Q}_\theta} \mathbb{A}^{\mathbf{q}}_\theta$

We now define, for any typed term-in-context $\Gamma \vdash M : \sigma$, a CSP process which represents its game semantics. This process is denoted $[\![\Gamma \vdash M : \sigma]\!]^{CSP}$, and it is over the following alphabet $\mathcal{A}_{\Gamma \vdash \sigma}$:

$\mathcal{A}_{\iota:\theta} = \iota.\mathcal{A}_\theta = \{\iota.\alpha \mid \alpha \in \mathcal{A}_\theta\}$
$\mathcal{A}_\Gamma = \bigcup_{\iota:\theta \in \Gamma} \mathcal{A}_{\iota:\theta}$
$\mathcal{A}_{\Gamma \vdash \sigma} = \mathcal{A}_\Gamma \cup \mathcal{A}_\sigma$

4.1 Expression constructs

$[\![\Gamma \vdash v : exp[\tau]]\!]^{CSP} = Q.q \to A.v \to SKIP$, $v \in \mathcal{A}_\tau$ is a constant

$$[\![\Gamma \vdash not\ B : exp[bool]]\!]^{CSP} = \\ [\![\Gamma \vdash B : exp[bool]]\!]^{CSP}[Q_1/Q, A_1/A] \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\ (Q.q \to Q_1.q \to A_1?v : \mathcal{A}_{bool} \to A.(not\ v) \to SKIP) \setminus \{\mid Q_1, A_1 \mid\}$$

$$[\![\Gamma \vdash E_1 \bullet E_2 : exp[\tau]]\!]^{CSP} = \\ [\![\Gamma \vdash E_1 : exp[\tau]]\!]^{CSP}[Q_1/Q, A_1/A] \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\ \big([\![\Gamma \vdash E_2 : exp[\tau]]\!]^{CSP}[Q_2/Q, A_2/A] \underset{\{\mid Q_2, A_2 \mid\}}{\parallel} \\ (Q.q \to Q_1.q \to A_1?v1 : \mathcal{A}_\tau \to Q_2.q \to A_2?v2 : \mathcal{A}_\tau \to \\ A.(v1 \bullet v2) \to SKIP) \setminus \{\mid Q_2, A_2 \mid\}\big) \setminus \{\mid Q_1, A_1 \mid\}$$

$[\![\Gamma \vdash \iota : exp[\tau]]\!]^{CSP} = Q.q \to \iota.Q.q \to \iota.A?v : \mathcal{A}_\tau \to A.v \to SKIP$

Any constant $v : exp[\tau]$ is represented by a process which communicates the events: a question q (what is the value of this expression), an answer v — the value of that constant, and then terminates successfully. The process that represents any arithmetic-logic operator $\bullet$ is defined in a compositional way, by parallel combination of the processes that represents both operands and a process that gets the operands values by synchronisation on the corresponding channels and returns the expected answer $(v1 \bullet v2)$. All events that participate in synchronisation are hidden. Arithmetic operators over a finite set of integers are interpreted as modulo some maximum value. The last process interprets a free identifier ι of type $exp[\tau]$ by a copycat strategy.

4.2 Command constructs

$$\llbracket \Gamma \vdash skip : comm \rrbracket^{CSP} = Q.run \rightarrow A.done \rightarrow SKIP$$

$$\llbracket \Gamma \vdash diverge : comm \rrbracket^{CSP} = STOP$$

$$\begin{aligned}
&\llbracket \Gamma \vdash C \mathbin{;} M : \sigma \rrbracket^{CSP} = \\
&\quad \llbracket \Gamma \vdash C : comm \rrbracket^{CSP}[Q_1/Q, A_1/A] \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\
&\quad (\llbracket \Gamma \vdash M : \sigma \rrbracket^{CSP}[Q_2/Q, A_2/A] \underset{\{\mid Q_2, A_2 \mid\}}{\parallel} \\
&\quad (Q?\mathbf{q} : \mathbb{Q}_\sigma \rightarrow Q_1.run \rightarrow A_1.done \rightarrow Q_2.\mathbf{q} \rightarrow A_2?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_\sigma \rightarrow A.\mathbf{a} \rightarrow SKIP) \\
&\qquad\qquad\qquad\qquad \setminus \{\mid Q_2, A_2 \mid\}) \setminus \{\mid Q_1, A_1 \mid\}
\end{aligned}$$

$$\begin{aligned}
&\llbracket \Gamma \vdash if\ B\ then\ M_1\ else\ M_2 : \sigma \rrbracket^{CSP} = \\
&\quad \llbracket \Gamma \vdash B : exp[bool] \rrbracket^{CSP}[Q_0/Q, A_0/A] \underset{\{\mid Q_0, A_0 \mid\}}{\parallel} \\
&\quad ((\llbracket \Gamma \vdash M_1 : \sigma \rrbracket^{CSP}[Q_1/Q, A_1/A] \,\square\, (A?\mathbf{a} : \textstyle\bigcup_{\mathbf{q} \in \mathbb{Q}_\sigma} \mathbb{A}^{\mathbf{q}}_\sigma \rightarrow SKIP)) \underset{\{\mid Q_1, A_1, A \mid\}}{\parallel} \\
&\quad \Big((\llbracket \Gamma \vdash M_2 : \sigma \rrbracket^{CSP}[Q_2/Q, A_2/A] \,\square\, (A?\mathbf{a} : \textstyle\bigcup_{\mathbf{q} \in \mathbb{Q}_\sigma} \mathbb{A}^{\mathbf{q}}_\sigma \rightarrow SKIP)) \underset{\{\mid Q_2, A_2, A \mid\}}{\parallel} \\
&\quad (Q?\mathbf{q} : \mathbb{Q}_\sigma \rightarrow Q_0.q \rightarrow A_0?v : \mathcal{A}_{bool} \rightarrow (Q_1.\mathbf{q} \rightarrow A_1?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_\sigma \rightarrow A.\mathbf{a} \rightarrow SKIP \\
&\qquad \not\prec v \not\succ Q_2.\mathbf{q} \rightarrow A_2?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_\sigma \rightarrow A.\mathbf{a} \rightarrow SKIP)) \\
&\qquad\qquad\qquad\qquad \setminus \{\mid Q_2, A_2 \mid\}\Big) \setminus \{\mid Q_1, A_1 \mid\}) \setminus \{\mid Q_0, A_0 \mid\}
\end{aligned}$$

$$\begin{aligned}
&\llbracket \Gamma \vdash while\ B\ do\ C : comm \rrbracket^{CSP} = \\
&\quad (\mu\, p'.(\llbracket B : comm \rrbracket^{CSP}[Q_1/Q, A_1/A] \mathbin{;} p') \,\square\, (A.done \rightarrow SKIP)) \underset{\{\mid Q_1, A_1, A \mid\}}{\parallel} \\
&\quad \Big((\mu\, p''.(\llbracket C : comm \rrbracket^{CSP}[Q_2/Q, A_2/A] \mathbin{;} p'') \,\square\, (A.done \rightarrow SKIP)) \underset{\{\mid Q_2, A_2, A \mid\}}{\parallel} \\
&\quad (Q.run \rightarrow \mu\, p.Q_1.q \rightarrow A_1?v : \mathcal{A}_{bool} \rightarrow (Q_2.run \rightarrow A_2.done \rightarrow p \not\prec v \not\succ \\
&\qquad A.done \rightarrow SKIP)) \setminus \{\mid Q_2, A_2 \mid\}\Big) \setminus \{\mid Q_1, A_1 \mid\}
\end{aligned}$$

$$\llbracket \Gamma \vdash \iota : comm \rrbracket^{CSP} = Q.run \rightarrow \iota.Q.run \rightarrow \iota.A.done \rightarrow A.done \rightarrow SKIP$$

The command *skip* is represented by the question and answer events for commands followed by the ✓ event. The command *diverge* is represented by the deadlocked process $STOP$, which matches with its game semantics, namely there is not any complete play for *diverge*. The process for a sequential composition

$\Gamma \vdash C \fatsemi M : \sigma$ is a parallel composition of the processes for the command C and term M, and a "control" process which accepts a question for type σ, runs C, then passes the original question to M, and finally copies any answer from M as the overall answer. The synchronisations between the latter process and the processes for C and M are hidden. The interpretations of branching and iteration are similar. In the case of branching, choice is used to enable the parallel composition to terminate without executing one of the processes for M_1 and M_2. For iteration, the processes for B and C are in general executed zero or more times. A free identifier is interpreted by a copycat strategy.

4.3 Variable constructs

For a free identifier of type $var[\tau]$, the copycat strategy is a choice between two processes: the first for reading from the variable, and the second for writing.

$$\llbracket \Gamma \vdash \iota : var[\tau] \rrbracket^{CSP} = \\ (Q.read \to \iota.Q.read \to \iota.A?v : \mathcal{A}_\tau \to A.v \to SKIP) \,\Box \\ (Q.write?v : \mathcal{A}_\tau \to \iota.Q.write.v \to \iota.A.ok \to A.ok \to SKIP)$$

The process for an assignment $V := M$ is a parallel composition of the processes for V and M, and a process which starts with a *run* (as the assignment is a command), evaluates M, then writes the obtained value to V, and finishes with *done*. The synchronisations with the former two processes are hidden. Note that V can be an arbitrary term of type $var[\tau]$, perhaps containing side-effects.

$$\llbracket \Gamma \vdash V := M : comm \rrbracket^{CSP} = \\ \llbracket \Gamma \vdash M : exp[\tau] \rrbracket^{CSP}[Q_1/Q, A_1/A] \underset{\{| Q_1, A_1 |\}}{\parallel} \\ (\llbracket \Gamma \vdash V : var[\tau] \rrbracket^{CSP}[Q_2/Q, A_2/A] \underset{\{| Q_2, A_2 |\}}{\parallel} \\ (Q.run \to Q_1.q \to A_1?v : \mathcal{A}_\tau \to Q_2.write.v \to A_2.ok \\ \to A.done \to SKIP) \setminus \{| Q_2, A_2 |\}) \setminus \{| Q_1, A_1 |\}$$

A dereference expression $!V$ is interpreted by reading from V:

$$\llbracket \Gamma \vdash !V : exp[\tau] \rrbracket^{CSP} = \\ \llbracket \Gamma \vdash V : var[\tau] \rrbracket^{CSP}[Q_1/Q, A_1/A] \underset{\{| Q_1, A_1 |\}}{\parallel} \\ (Q.q \to Q_1.read \to A_1?v : \mathcal{A}_\tau \to A.v \to SKIP) \setminus \{| Q_1, A_1 |\}$$

The semantics of a local variable block consists of two operations: imposing the "good variable" behaviour on the local variable, and removing all references to the variable to make it invisible outside its binding scope. The first condition is accomplished by synchronising the process for the scope with a "cell" process $U_{\iota:var[\tau],a_\tau}$ which keeps the current value of ι. The initial value a_τ is defined by $a_{int} = 0$ and $a_{bool} = false$.

$$U_{\iota:var[\tau],v} = (\ \iota.Q.read \to \iota.A!v \to U_{\iota:var[\tau],v}\ ) \,\Box \\ (\ \iota.Q.write?v' : \mathcal{A}_\tau \to \iota.A.ok \to U_{\iota:var[\tau],v'}\ ) \,\Box \\ (\ A.done \to SKIP\ )$$

The second condition is realised by hiding all interactions between the process for the scope and the cell process:

$$[\![\Gamma \vdash new[\tau]\ \iota\ in\ M : comm]\!]^{CSP} = \\ \left([\![\Gamma \vdash M : comm]\!]^{CSP} \underset{\{|\iota, A|\}}{\parallel} U_{\iota:var[\tau],a_\tau}\right) \setminus \{|\ \iota\ |\}$$

4.4 Application and functions

Application $\iota(M_1 \dots M_k)$, where ι is a free identifier, is represented by a process that communicates the usual question and answer events for the return type of the function on the channel $\iota.0$, and runs zero or more times any processes of the function arguments. Arguments events are appended on the channels comprised of the function name followed by the index of that argument.

$$[\![\Gamma \vdash \iota(M_1 \dots M_k) : \sigma]\!]^{CSP} = Q?\mathbf{q} : \mathbb{Q}_\sigma \rightarrow \iota.0.Q.\mathbf{q} \rightarrow \\ \mu\, L.\Big(\Big(\square_{j=1}^{k} \big([\![\Gamma \vdash M_j : \sigma_j]\!]^{CSP} \underset{\{|Q,A|\}}{\parallel} (\iota.j.Q?\mathbf{q} : \mathbb{Q}_{\sigma_j} \rightarrow Q.\mathbf{q} \rightarrow A?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_{\sigma_j} \\ \rightarrow \iota.j.A.a \rightarrow L)\big) \setminus \{|\ Q, A\ |\}\Big) \square\, SKIP\Big) \,;\, \iota.0.A?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_{\sigma} \rightarrow A.\mathbf{a} \rightarrow SKIP$$

Finally, we give a CSP representation for the *let* construct. The process for the scope M is put in parallel with the process for the definition term N, where the latter is enabled to execute zero or more times. Synchronisation is on interactions of M with the defined identifier ι, which are then hidden.

$$[\![\Gamma \vdash let\ \iota(\iota_1 : \sigma_1, \dots, \iota_k : \sigma_k) = N\ in\ M : \sigma]\!]^{CSP} = \\ \Big([\![\Gamma \vdash M : \sigma]\!]^{CSP} \underset{\{|\iota|\}}{\parallel} \\ \big(\mu\, p.([\![\Gamma \vdash N : \sigma']\!]^{CSP}[\iota.0.Q/Q, \iota.1/\iota_1, \dots, \iota.k/\iota_k, \iota.0.A/A] \,;\, p) \square SKIP\big)\Big) \\ \setminus \{|\ \iota\ |\}$$

5 Correctness and decidability

Our first result is that, for any term, the set of all terminated traces of its CSP interpretation is isomorphic to the language of its regular language interpretation, as defined in [9].

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

$$\mathcal{L}_R(\Gamma \vdash M : \sigma) \overset{\phi}{\cong} \mathcal{L}_{CSP}(\Gamma \vdash M : \sigma) \tag{2}$$

where

$$\mathcal{L}_R(\Gamma \vdash M : \sigma) = \mathcal{L}([\![\Gamma \vdash M : \sigma]\!]^R) \qquad (\textit{defined in [9]}) \tag{3}$$

$$\mathcal{L}_{CSP}(\Gamma \vdash M : \sigma) = \{tr \mid tr^\frown\langle\checkmark\rangle \in traces([\![\Gamma \vdash M : \sigma]\!]^{CSP})\} \tag{4}$$

and ϕ is defined by:

$$\begin{array}{ll} \phi(\langle a_1, \ldots, a_k \rangle) = \phi(a_1) \cdot \ldots \cdot \phi(a_k) & \\ \phi(Q.m) = m & \phi(A.n) = n \\ \phi(\iota.Q.m) = m^\iota & \phi(\iota.A.n) = n^\iota \end{array}$$

Proof. The proof of this Theorem is by induction on the typing rules defined in Table 1. □

Two terms M and N in type context Γ and of type σ are observationally equivalent, written $\Gamma \vdash M \equiv_\sigma N$, iff for any term-with-hole $C[-]$ such that both $C[M]$ and $C[N]$ are closed terms of type *comm*, $C[M]$ converges iff $C[N]$ converges. It was proved in [1] that this coincides to equality of sets of complete plays of the strategies for M and N, i.e. that the games model is fully abstract. (Operational semantics of IA, and a definition of convergence for terms of type *comm* in particular, can be found in the same paper.)

For the IA fragment treated in this paper, it was shown in [9] that observational equivalence coincides with equality of regular language interpretations. By Theorem 1, we have that observational equivalence corresponds to two traces refinements:

Corollary 1 (Observational equivalence).

$$\begin{aligned} \Gamma \vdash M \equiv_\sigma N \;\Leftrightarrow\; & [\![\Gamma \vdash M : \sigma]\!]^{CSP} \;\Box\; RUN_\Sigma \;\sqsubseteq_T\; [\![\Gamma \vdash N : \sigma]\!]^{CSP} \;\wedge \\ & [\![\Gamma \vdash N : \sigma]\!]^{CSP} \;\Box\; RUN_\Sigma \;\sqsubseteq_T\; [\![\Gamma \vdash M : \sigma]\!]^{CSP} \end{aligned} \tag{5}$$

Proof.

$$\begin{aligned} \Gamma \vdash M \equiv_\sigma N \;&\overset{(i)}{\Leftrightarrow}\; \mathcal{L}([\![\Gamma \vdash M : \sigma]\!]^R) \;=\; \mathcal{L}([\![\Gamma \vdash N : \sigma]\!]^R) \\ &\overset{(ii)}{\Leftrightarrow}\; \mathcal{L}_{CSP}(\Gamma \vdash M : \sigma) \;=\; \mathcal{L}_{CSP}(\Gamma \vdash N : \sigma) \\ &\overset{(iii)}{\Leftrightarrow}\; [\![\Gamma \vdash M : \sigma]\!]^{CSP} \;\Box\; RUN_\Sigma \;\sqsubseteq_T\; [\![\Gamma \vdash N : \sigma]\!]^{CSP} \;\wedge \\ &\qquad [\![\Gamma \vdash N : \sigma]\!]^{CSP} \;\Box\; RUN_\Sigma \;\sqsubseteq_T\; [\![\Gamma \vdash M : \sigma]\!]^{CSP} \end{aligned}$$

where (i) is shown in [9], (ii) holds by Theorem 1, and (iii) is straightforward. □

Refinement checking in FDR terminates for finite-state processes, i.e. those whose transition systems are finite. Our next result confirms that this is the case for the processes interpreting the IA terms. As a corollary, we have that observational equivalence is decidable using FDR.

Theorem 2. *For any term $\Gamma \vdash M : \sigma$, the CSP process $[\![\Gamma \vdash M : \sigma]\!]^{CSP}$ is finite state.*

Proof. This follows by induction on typing rules, using the fact that a process can have infinitely many states only if it uses either a choice operator $?x : A \to P$, where A is an infinite set and P varying with x, or certain kinds of recursion. □

Corollary 2 (Decidability). *Observational equivalence between terms of second-order recursion-free IA with iteration and finite data types is decidable by two traces refinements between finite-state CSP processes.* □

We now consider an example equivalence and prove it using the CSP model.

Example 1. $\Gamma \vdash \mathit{while\ true\ do\ C} \ \equiv_{comm} \ \mathit{diverge}$.

We prove the first traces refinement of the equivalence (5), i.e.

$$[\![\Gamma \vdash \mathit{while\ true\ do\ C} : \mathit{comm}]\!]^{CSP} \,\Box\, RUN_{\Sigma} \sqsubseteq_T [\![\Gamma \vdash \mathit{diverge} : \mathit{comm}]\!]^{CSP} \quad (*)$$

Since $[\![\Gamma \vdash \mathit{diverge} : \mathit{comm}]\!]^{CSP} = STOP$, and the process $STOP$ traces refines any other process, it implies that $(*)$ holds.

The second traces refinement is:

$$[\![\Gamma \vdash \mathit{diverge} : \mathit{comm}]\!]^{CSP} \,\Box\, RUN_{\Sigma} \sqsubseteq_T [\![\Gamma \vdash \mathit{while\ true\ do\ C} : \mathit{comm}]\!]^{CSP} \quad (**)$$

We have:

$$\begin{aligned} &\mathit{traces}([\![\Gamma \vdash \mathit{while\ true\ do\ C} : \mathit{comm}]\!]^{CSP}) = \\ &\quad = \{\langle\rangle, \langle Q.run\rangle, \langle Q.run\rangle^\frown tr', \langle Q.run\rangle^\frown tr, \langle Q.run\rangle^\frown tr^\frown tr', \ldots \mid \\ &\qquad \langle Q.run\rangle^\frown tr^\frown \langle A.done\rangle^\frown \langle \checkmark\rangle \in \mathit{traces}([\![\Gamma \vdash C]\!]^{CSP}), tr' \leqslant tr\} \end{aligned}$$

Since there is not any sequence with successful termination in the traces set of $[\![\Gamma \vdash \mathit{while\ true\ do\ C} : \mathit{comm}]\!]^{CSP}$, all sequences from this traces set will be also in the traces set of RUN_{Σ} process, so $(**)$ holds too. □

5.1 Property verification

Suppose ϕ is any property of terms with type context Γ and type σ such that the set of all behaviours which satisfy ϕ is a regular language $\mathcal{L}(\phi)$ over $\mathcal{A}_{\Gamma \vdash \sigma}$. A finite-state CSP process whose set of terminated traces equals $\mathcal{L}(\phi)$ can then be constructed. Thus, we can verify whether a term $\Gamma \vdash M : \sigma$ satisfies ϕ by checking the traces refinement:

$$P_{\phi} \,\Box\, RUN_{\Sigma} \sqsubseteq_T [\![\Gamma \vdash M : \sigma]\!]^{CSP} \tag{6}$$

Example 2. Consider a term $x : \mathit{var}[\tau], c : \mathit{comm} \vdash M : \mathit{comm}$. We want to check the property "a value is written into x before c is called". A CSP process which interprets this property is:

$$\begin{aligned} P_{\phi} = \mu\, p.\Big(&\big(?e : \mathcal{A}_{comm} \cup \mathcal{A}_{x:var[\tau] \setminus \{|x.Q.write|\}} \to p\big) \,\Box\, \Big(x.Q.write?v : \mathcal{A}_{\tau} \to \\ &\mu\, p'.\big(?e : \mathcal{A}_{comm} \cup \mathcal{A}_{x:var[\tau]} \to p'\big) \,\Box\, (c.Q.run \to \\ &\quad \mu\, p''.\big(?e : \mathcal{A}_{\Gamma \vdash comm} \to p''\big) \,\Box\, SKIP)\Big)\Big) \end{aligned}$$ □

6 Experimental results

We have implemented a compiler from IA terms-in-context into CSP processes which represent their game semantics. The input to the compiler is code, with some simple type annotations to indicate what finite sets of integers will be used to model integer variables.

Here, we discuss the modelling of a sorting program, report the results from our tool and compare them with the tool based on regular expressions [8]. We will analyse the bubble-sort algorithm, whose implementation is given in Figure 1.

The code includes a meta variable n, representing array size, which will be replaced by several different values. The integers stored in the array are of type $int\%3$, i.e. 3 distinct values 0, 1 and 2, and the type of index i is $int\%n{+}1$, i.e. one more than the size of the array. The program first copies the input array $x[\,]$ into a local array $a[\,]$, which is then sorted and copied back into $x[\,]$. The array being effectively sorted, $a[\,]$, is not visible from the outside of the program because it is locally defined, and only reads and writes of the non-local array $x[\,]$ are seen in the model. The transition system of final (compressed) model process for $n = 2$ is shown in Figure 2. It illustrates the dynamic behaviour of the program, where the left-side half of the model reads all possible combinations of values from $x[\,]$, while the right-side half writes out the same values, but in sorted order.

```
var int%3 x[n];  ⊢
new int%3 a[n] in
new int%n+1 i in
while (i < n)  {a[i] := x[i];   i := i + 1; }
new boolean flag := true in
while(flag){
      i := 0;
      flag := false;
      while (i < n − 1)  {
            if (a[i] > a[i + 1])  {
                  flag := true;
                  new int%3 temp in
                  temp := a[i];
                  a[i] := a[i + 1];
                  a[i + 1] := temp;   }
            i := i + 1;   } }
i := 0;
while (i < n)  {x[i] := a[i];   i := i + 1; }
: comm
```

Fig. 1. Implementation of bubble-sort algorithm

Table 2 contains the experimental results for model generation. We ran FDR on a Research Machines Xeon with 2GB RAM. The results from the tool based on regular expressions were obtained on a SunBlade 100 with 2GB RAM [8]. We list the execution time, the size of the largest generated state machine during model generation, and the size of the final compressed model. In the CSP approach, the process output by our compiler was input into FDR, which was instructed to generate a transition system for it by applying a number of compositional state-space reduction algorithms. The results confirm that both approaches give isomorphic models, where the CSP models have an extra state due to representing termination by a ✓ event.

We expect FDR to perform even better in property verification. For checking refinement by a composite process, FDR does not need to generate an explicit

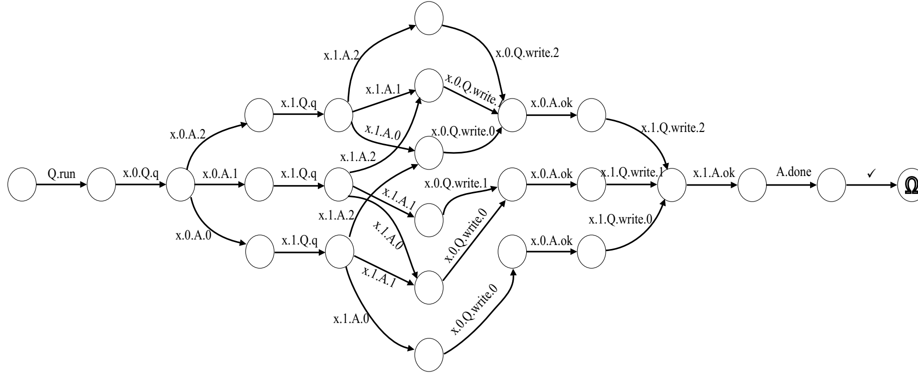

Fig. 2. A transition system of the model for $n = 2$

model of it, but only models of its component processes. A model of the composite process is then generated on-the-fly, and its size is not limited by available RAM, but by disk size.

Table 2. Experimental results for minimal model generation

n	CSP			Regular expressions		
	Time (min)	Max. states	Model states	Time (min)	Max. states	Model states
5	6	1 775	164	5	3 376	163
10	20	18 752	949	10	64 776	948
15	50	115 125	2 859	120	352 448	2 858
20	110	378 099	6 394	240	1 153 240	6 393
30	750	5 204 232	20 339	failed		

7 Conclusion

We presented a compositional representation of game semantics of an interesting fragment of Idealised Algol by CSP processes. This enables observational equivalence and a range of properties of terms-in-context (i.e. open program fragments) to be checked using the FDR tool.

We also reported initial experimental results using our prototype compiler and FDR. They show that, for minimal model generation, the CSP approach outperforms the approach based on regular expressions.

As future work, we plan to compare how the two approaches perform on a range of equivalence and property checking problems.

We also intend to extend the compiler so that parameterised IA terms (such as parametrically polymorphic programs) are translated to single parameterised CSP processes. Such processes could then be analysed by techniques which combine CSP and data specification formalisms (e.g. [6, 15]) or by algorithms based on data independence [12].

Another important direction is dealing with concurrency (e.g. [10]) and further programming language 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. S.Abramsky. Algorithmic game semantics: A tutorial introduction. Lecture notes, Marktoberdorf International Summer School 2001, 2001.
3. S.Abramsky, D.Ghica, A.Murawski and C.-H.L.Ong. Applying Game Semantics to Compositional Software Modeling and Verifications. In Proceedings of TACAS, LNCS 2988, March 2004.
4. T.Ball and S.K.Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In Proceedings of POPL, ACM SIGPLAN Notices 37(1), January 2002.
5. A.Dimovski and R.Lazić. CSP Representation of Game Semantics for Second-order Idealized Algol. Research Report, Department of Computer Science, University of Warwick, May 2004.
6. A.Farias, A.Mota and A.Sampaio. Efficient CSP-Z Data Abstraction. In Proceedings of IFM, LNCS 2999, April 2004.
7. Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 Manual. 2000.
8. D.Ghica. Game-based Software Model Checking: Case Studies and Methodological Considerations. Oxford University Computing Lab, Technical Report PRG-RR-03-11, May 2003.
9. D.Ghica and G.McCusker. The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science 309(1–3): 469–502, 2003.
10. D.Ghica, A.Murawski and C.-H.L.Ong. Syntactic Control of Concurrency. In Proceedings of ICALP, LNCS 3142, July 2004.
11. J.M.E.Hyland and C.-H.L.Ong. On full abstraction for PCF: I, II and III. Information and Computation 163: 285–408, 2000.
12. R.Lazić. A Semantic Study of Data Independence with Applications to Model Checking. DPhil thesis, Computing Laboratory, Oxford University, 1999.
13. C.-H.L.Ong. Observational equivalence of 3rd-order Idealised Algol is Decidable. In Proceedings of LICS, IEEE, July 2002.
14. J.C.Reynolds. The essence of Algol. In Proceedings of ISAL, 345–372, Amsterdam, Holland, 1981.
15. M.Roggenbach. CSP-CASL — A new Integration of Process Algebra and Algebraic Specification. In Proceedings of AMiLP, TWLT 21, Universiteit Twente, 2003.
16. A.W.Roscoe, P.H.B. Gardiner, M.H.Goldsmith, J.R.Hulance, D.M.Jackson and J.B.Scattergod. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In Proceedings of TACAS, LNCS 1019, May 1995.
17. A.W.Roscoe. The Theory and Practice of Concurrency. Prentice Hall, 1998.