Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Compositional Software Verification Based On Game Semantics and CSP*

Aleksandar Dimovski and Ranko Lazić

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

Received: date / Revised version: date

Abstract. We present an approach to software model checking based on game semantics and CSP. Open program fragments (i.e. terms-in-context) are compositionally modelled as CSP processes which represent their game semantics. This translation is performed by a prototype compiler. Observational equivalence and regular properties are checked by traces refinement using the FDR tool. We also present theorems for parameterised verification of polymorphic terms and properties. The effectiveness of the approach is evaluated on several examples.

1 Introduction

One of the main recent breakthroughs in theoretical computer science has been the development of game semantics (e.g. [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 [2,14,15].

It has recently been shown that, for several interesting programming language fragments, their game semantics yield algorithms for software model checking. The focus has been on Idealized Algol (IA) [19] 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 function mechanism. For example, simple forms of classes and objects may be encoded in IA.

* We acknowledge support by the EPSRC (GR/S52759/01). The second author was also supported by the Intel Corporation, and is also affiliated to the Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade.

For second-order recursion-free IA with iteration and finite data types, [11] shows that game semantics can be represented by regular expressions, so that observational equivalence between any two terms can be decided by equality checks of regular languages. For third order, it was established in [17] that observational equivalence and approximation are decidable using visibly pushdown automata. Other verification problems can also be solved algorithmically, such as Hoare triples (e.g. [3]).

Model checking [7,18,8] is a system verification technique based on semantics: the verifier checks whether the semantics of a given system satisfies some property. It has gained industrial acceptance because, in contrast to the approaches of simulation, testing and theorem proving, model checking offers automatic and exhaustive verification, and it also reports counter-examples.

The initial success of model checking has been mainly in the verification of hardware and communication protocols. Recently, model checking of software has become an active and important area of research and application (e.g. [6]). Unfortunately, applying model checking to software is complicated by several factors, ranging from the difficulty to model programs, due to the complexity of general purpose programming languages as compared to hardware description languages, to difficulties in specifying meaningful properties of software using the usual temporal logical formalisms. Another reason is the state explosion problem: industrial programs are large and model checking is computationally demanding.

Many of the problems above are due to difficulties in obtaining sound and complete semantic models of software and expressing such models in an algorithmic fashion suitable for automatic analysis. Game semantics has potential to overcome these problems. Compared with other approaches to software model checking, the approach based on game semantics has a number of advantages [4]:

- 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 with a maximum level of abstraction.

In this work, 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, and containment in a regular language, can then be decided by checking traces refinements between CSP processes.

Compared with the representation by regular expressions (or automata) [11], the CSP representation brings several benefits:

- CSP operators preserve traces refinement (e.g. [21]), 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 [10] can be used to explore CSP processes visually, to check refinements automatically, and to debug interactively when a refinement does not hold;
- compositional state-space reduction algorithms in FDR [20] 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.

We present two theorems for parameterised verification of terms and properties which are polymorphic, i.e. contain a data-type variable which can be instantiated by any finite data type. They are proved by observing that the resulting CSP processes are data independent, and applying the results in [16].

We have implemented a prototype compiler which, given any IA term-in-context, outputs a CSP process representing its game semantics. The effectiveness of our approach is evaluated on several variants of two examples: a sorting algorithm, and an abstract data type implementation. The experimental results show that, for minimal model generation, this approach can outperform the approach which uses regular expressions [4].

Section 2 presents the fragment of IA we are addressing. Section 3 contains brief introductions to game semantics and CSP. In Section 4, we define the CSP representation of game semantics for the language fragment. Correctness of the CSP model, and decidability of observational equivalence by traces refinement, are shown in Section 5. Section 6 shows how properties given as finite automata, or in a linear temporal logic on finite traces, can be verified. In Section 7, two theorems for parameterised verification of polymorphic terms and properties are given. The compiler and two case studies are presented in Sections 8 and 9. In Section 10, we point out some possibilities for future work.

2 The programming language

Idealised Algol [19] is a functional-imperative language. Imperative features such as assignment, sequential composition, branching and iteration are combined with a function mechanism based on a typed call-by-name λ-calculus. We consider the recursion-free second-order fragment of this language, with finite data sets. A fragment is said to be ith-order if in its type judgements, the types of free identifiers and of the term are of order less than i.

The language has data types of the integers (from 0 to $n-1$, where $n > 0$) and the Booleans. The phrase types consist of base types (expression, command, variable) and first-order function types.

$$\tau ::= int_n \mid bool$$
$$\sigma ::= exp[\tau] \mid comm \mid var[\tau]$$
$$\theta ::= \sigma \mid \sigma_1 \times \cdots \times \sigma_k \to \sigma'$$

Terms are introduced using type judgments of the form $\Gamma \vdash M : \theta$. Γ is a type context consisting of a finite number of typed free identifiers, i.e. of the form $\iota_1 : \theta_1, \cdots, \iota_k : \theta_k$.

Without loss of generality, using β-reduction, we do not consider λ-abstraction, and we only consider function application of free identifiers. The well-typed terms of the language are given by the typing rules in Table 1.

Firstly, the language contains integer and Boolean constants and operators. Next, there are the usual imperative constructs: skipping, sequential composition, conditional, iteration, assignment and dereferencing. We work with IA with *active expressions*, so a command may be sequentially composed with a phrase of expression or variable type, and if-then-else is available on all base types. The *new* construct initialises a variable, and makes it local to a given command. Finally, there are typing rules for free identifiers, function application, and function declaration.

Table 1. Typing rules

$$\frac{}{\Gamma \vdash m : exp[int_n]} \quad m \in \{0, \ldots, n-1\}$$

$$\frac{}{\Gamma \vdash b : exp[bool]} \quad b \in \{true, false\}$$

$$\frac{\Gamma \vdash E_1 : exp[\tau_1] \qquad \Gamma \vdash E_2 : exp[\tau_2]}{\Gamma \vdash E_1 * E_2 : exp[\tau']} * \in \{+, <, and, \ldots\}$$

$$\frac{}{\Gamma \vdash skip : comm} \qquad \frac{\Gamma \vdash C : comm \qquad \Gamma \vdash M : \sigma}{\Gamma \vdash C;\ M : \sigma}$$

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

$$\frac{\Gamma \vdash B : exp[bool] \qquad \Gamma \vdash C : comm}{\Gamma \vdash while\ B\ do\ C : comm}$$

$$\frac{\Gamma \vdash V : var[\tau] \qquad \Gamma \vdash E : exp[\tau]}{\Gamma \vdash V := E : comm} \qquad \frac{\Gamma \vdash V : var[\tau]}{\Gamma \vdash !V : exp[\tau]}$$

$$\frac{\Gamma, \iota : var[\tau] \vdash C : comm \qquad \Gamma \vdash E : exp[\tau]}{\Gamma \vdash new[\tau]\ \iota := E\ in\ C : comm}$$

$$\frac{}{\Gamma, \iota : \theta \vdash \iota : \theta}$$

$$\frac{\Gamma \vdash \iota : \sigma_1 \times \cdots \times \sigma_k \to \sigma' \qquad \Gamma \vdash M_i : \sigma_i}{\Gamma \vdash \iota(M_1, \ldots, M_k) : \sigma}$$

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

2.1 Syntactic sugar

A *diverge* command which stands for the simplest non-terminating computation can be defined as

$$while\ true\ do\ skip$$

As in [11], arrays of length $k > 0$ can be introduced by the following abbreviations:

$$new[\tau]\ \iota[k] := E\ in\ C \equiv$$
$$new[\tau]\ \iota.0 := E\ in\ \cdots\ new[\tau]\ \iota.(k-1) := E\ in\ C$$
$$\iota[E] \equiv$$
$$if\ E = 0\ then\ \iota.0\ else\ \cdots$$
$$if\ E = k-1\ then\ \iota.(k-1)\ else\ oub;\ \iota.0$$

where *oub* is a free identifier of type *comm*, which is called for any out-of-bounds error.

3 Background

3.1 Game semantics

We give an informal overview of game semantics of IA and we illustrate it with some examples. A more complete introduction can be found in [3].

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.

Every type in the language is modelled as a game — the set of all possible plays (sequences of moves). A term of a given type 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 v is:

O:	q	(opponent asks for value)
P:	v	(player answers to the question)

Now, consider the term $\iota : exp[int_n] \to exp[int_n] \vdash \iota(2) : exp[int_n]$, where the identifier ι is some non-locally defined function, and $n > 2$. 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 m^ι — the value returned from ι, P will copy this value and answer to the first question with m.

A sample complete play for this term, when the function ι evaluates its argument only once, is:

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

Thus, the strategy for the above term is the regular language

$$\sum_{m=0}^{n-1} q \cdot q^\iota \cdot (q^{\iota 1} \cdot 2^{\iota 1})^* \cdot m^\iota \cdot m$$

Free identifiers are interpreted by copy-cat strategies. A sample complete play of the strategy for $\iota : exp[int_n] \to exp[int_n] \vdash \iota : exp[int_n] \to exp[int_n]$ is:

O:	q	(asks for result value)
P:	q^{ι}	(P asks for value returned from function ι)
O:	$q^{\iota 1}$	(what is the first argument to ι)
P:	q^{1}	(what is the first argument to the term)
O:	k^{1}	(O answers: k)
P:	$k^{\iota 1}$	(P copies the answer)
O:	m^{ι}	(O answers: m)
P:	m	(P copies the answer)

Function application is modelled by *composition* of strategies. Strategies are composed by making them interact on moves for the argument, and then hiding those moves. For example, the play of $\iota(2)$ above is obtained from the play of ι above with $k = 2$ and the unique complete play of the constant 2.

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 of $\vdash skip : comm$ is $run \cdot done$.

Variables are represented as objects with two methods: a 'read method' for dereferencing, represented by an initial move $read$, with response a data value; and a 'write method' for assignment, represented by an initial move $write.c$ for any data value c, to which the only possible response is ok. For example, a complete play for $x : var[int_n] \vdash x := !x + 1;\ !x : exp[int_n]$, where $n > 5$, is:

O:	run	
P:	$read^{x}$	(what is the value of x)
O:	2^{x}	(O supplies the value 2)
P:	$write.3^{x}$	(write 3 into x)
O:	ok^{x}	(the assignment is complete)
P:	$read^{x}$	(ask for the value of x)
O:	5^{x}	(the value 5 is returned)
P:	$done$	

When P asks to read from x in a play as above, O can return an arbitrary value, i.e. not necessarily the last value P wrote into x. This is because, in general, the value of x can also be modified by the context into which the term will be placed.

'Good variable' behaviour is achieved by making a variable local. In game semantics of $new[\tau]\ \iota := E\ in\ C$, any read of ι within C returns the most recently written value, and all moves involving ι are hidden.

3.2 CSP

Communicating Sequential Processes [13] 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 [21].

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 $\{write.n \mid n \in N\}$. We can define the set of all events that can arise on a channel c, by $\{|\ c\ |\} = \{c.w \in \Sigma\}$.

We use the following collection of process operators:

$$P ::= p \mid STOP \mid SKIP \mid RUN_A \mid ?x : A \to P_x \mid \mu\, p.P \mid P_1 \Box P_2 \mid P_1 \lhd b \rhd P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid P\backslash A \mid P[a/b] \mid P_1 \fatsemi P_2$$

where A represents a set of events, and p a process identifier.

The process $STOP$ performs no actions. $SKIP$ is a process that successfully terminates, causing the special event $\checkmark$ ($\checkmark \notin \Sigma$). RUN_A can always communicate any event from A. $?x : A \to P$ can perform any event from set A and then behave as P. To define a process recursively by $p = P$, we write $\mu\, p.P$. For example, RUN_A is equivalent to $\mu\, p.?x : A \to p$. $P_1 \Box P_2$ can behave either as P_1 or as P_2. $P_1 \lhd b \rhd P_2$ behaves as P_1 if b is true, and as P_2 otherwise. $P_1 \underset{A}{\parallel} P_2$ runs P_1 and P_2 in parallel, making them synchronise on events in A and allowing interleaving of other events. A parallel composition terminates successfully if and only if both component processes do so. To hide all events in A from a process P (i.e. make them invisible or internal events called τ), we write $P\backslash A$. $P[a/b]$ renames an event or channel b to a. A sequential composition $P_1 \fatsemi P_2$ runs P_1, and if it terminates successfully, runs P_2.

CSP processes can be given denotational semantics by their sets of traces. A trace is a finite sequence of events. A sequence t is a trace of a process P if there is some execution of P in which exactly that sequence of events is performed. (Invisible events τ are not recorded in traces.) $traces(P)$ is the set of all traces of P.

$s^\frown t$ denotes the concatenation of traces s and t. $t \restriction A$ is obtained by restricting t to events in A.

Let $traces^{\checkmark}(P)$ be the set of all terminated traces of P:

$$traces^{\checkmark}(P) = \{t \mid t^\frown\langle\checkmark\rangle \in traces(P)\}$$

P_2 is a traces refinement of P_1 if and only if any trace of P_2 is also a trace of P_1:

$$P_1 \sqsubseteq_T P_2 \ \Leftrightarrow\ traces(P_2) \subseteq traces(P_1)$$

CSP processes can also be given operational semantics, using labelled transition systems, which are directed graphs whose nodes represent process states, and whose edges are labelled by events. Any edge whose label is $\checkmark$ leads to a special terminated state Ω.

The FDR tool [10] 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 [20]. 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 Game semantics of IA in CSP

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

$$\begin{array}{ll}
\mathcal{A}_{int_n} = \{0, \ldots, n-1\} & \mathcal{A}_{bool} = \{tt, ff\} \\
\mathbf{Q}_{exp[\tau]} = \{q\} & \mathbf{A}^{q}_{exp[\tau]} = \mathcal{A}_\tau \\
\mathbf{Q}_{comm} = \{run\} & \mathbf{A}^{run}_{comm} = \{done\} \\
\mathbf{Q}_{var[\tau]} = \{read, write.v \mid v \in \mathcal{A}_\tau\} & \\
\mathbf{A}^{read}_{var[\tau]} = \mathcal{A}_\tau & \mathbf{A}^{write.v}_{var[\tau]} = \{ok\} \\
\mathbf{Q}_{\sigma_1 \times \cdots \times \sigma_k \to \sigma'} = \{j.\mathbf{q} \mid \mathbf{q} \in \mathbf{Q}_{\sigma_j}\} \cup \mathbf{Q}_{\sigma'} & \\
\mathbf{A}^{j.\mathbf{q}}_{\sigma_1 \times \cdots \times \sigma_k \to \sigma'} = \{j.\mathbf{a} \mid \mathbf{a} \in \mathbf{A}^{\mathbf{q}}_{\sigma_j}\} & \\
\mathbf{A}^{\mathbf{q}}_{\sigma_1 \times \cdots \times \sigma_k \to \sigma'} = \mathbf{A}^{\mathbf{q}}_{\sigma'} & \\
\mathcal{A}_\theta = Q.\mathbf{Q}_\theta \cup A.\bigcup_{\mathbf{q} \in \mathbf{Q}_\theta} \mathbf{A}^{\mathbf{q}}_\theta &
\end{array}$$

Consider a term-in-context $\Gamma \vdash M : \theta$. Its game semantics is a strategy over the alphabet $\mathcal{A}_{\Gamma \vdash \theta}$ which is defined as follows:

$$\begin{array}{rcl}
\mathcal{A}_{\iota:\theta} & = & \iota.\mathcal{A}_\theta \\
\mathcal{A}_\Gamma & = & \bigcup_{\iota:\theta \in \Gamma} \mathcal{A}_{\iota:\theta} \\
\mathcal{A}_{\Gamma \vdash \theta} & = & \mathcal{A}_\Gamma \cup \mathcal{A}_\theta
\end{array}$$

The standard approach in game semantics is to define this strategy compositionally. As in [11], for simplicity of giving semantics to the *let* construct, we generalise by introducing environments. An environment u for a type context Γ is a mapping such that, for any $\iota : \theta \in \Gamma$, $u(\iota)$ is a CSP process over the alphabet $\mathcal{A}_{\iota:\theta \vdash \theta}$. To update u by mapping ι to P, we write $u \mid \iota \to P$.

We shall define, for any term-in-context $\Gamma \vdash M : \theta$ and environment u for Γ, a CSP process which represents the game semantics of $\Gamma \vdash M : \theta$ with respect to u. This process is denoted $[\![\Gamma \vdash M : \theta]\!]^{CSP} u$, and it is over the alphabet $\mathcal{A}_{\Gamma \vdash \theta}$.

The standard game semantics of $\Gamma \vdash M : \theta$ is obtained by using the environment u_0 such that, for any $\iota : \theta \in \Gamma$, $u_0(\iota)$ is the copy-cat process K^ι_θ, defined in Table 2.

The CSP process for an integer or Boolean constant replies to a question by the value of the constant, and then terminates. For an operator application $E_1 * E_2$, we compose the processes for E_1 and E_2, and a process for $*$. As with all processes which represent strategies in this paper, the composition is performed by the CSP operators of renaming, parallel and hiding. The process for $*$ asks for values of the arguments, and after obtaining them responds by performing the operation. The details are in Table 3.

Table 4 shows processes for the command constructs. For sequential composition, conditional and iteration, processes for the components are composed with a process for the construct itself, similarly to how the process for $E_1 * E_2$ was defined above. However, in case of the conditional, one of the processes for M_1 and M_2 will not be run, so $SKIP$ is used to enable such empty termination. For iteration, the processes for B and C may be run arbitrarily many times, which is achieved by placing them inside appropriate recursions.

The processes for assignment and dereferencing are straightforward. In the definition for local-variable declarations, a 'cell' process $U_{\iota,v}$ is used for remembering the initial or the most-recently written value into the variable ι. It is composed with the process for the scope of the declaration, ensuring 'good variable' behaviour. Table 5 contains the details.

Table 6 contains the remaining process definitions: for free identifier, function application and function declaration terms. In each case, environments are used to access or record processes associated with free identifiers.

Table 2. Free identifiers

$$\begin{array}{l}
K^\iota_{exp[\tau]} = Q.q \to \iota.Q.q \to \iota.A?v : \mathcal{A}_\tau \to A.v \to SKIP \\
K^\iota_{comm} = Q.run \to \iota.Q.run \to \iota.A.done \to A.done \to SKIP \\
K^\iota_{var[\tau]} = \\
\quad (Q.read \to \iota.Q.read \to \iota.A?v : \mathcal{A}_\tau \to A.v \to SKIP) \Box \\
\quad (Q.write?v : \mathcal{A}_\tau \to \iota.Q.write.v \to \iota.A.ok \to A.ok \to SKIP) \\
K^\iota_{\sigma_1 \times \cdots \times \sigma_k \to \sigma'} = \\
\quad Q?\mathbf{q} : \mathbf{Q}_\sigma \to \iota.Q.\mathbf{q} \to \\
\quad (\mu L.(\Box_{j=1}^{k} \iota.j.Q?\mathbf{q}_j : \mathbf{Q}_{\sigma_j} \to Q.j.\mathbf{q}_j \to \\
\qquad\qquad A.j?\mathbf{a}_j : \mathbf{A}^{\mathbf{q}_j}_{\sigma_j} \to \iota.j.A.\mathbf{a}_j \to L) \Box SKIP) \setminus \{\!|\, q \,|\!\} \\
\quad \iota.A?\mathbf{a} : \mathbf{A}^{\mathbf{q}}_\sigma \to A.\mathbf{a} \to SKIP
\end{array}$$

Table 3. Expression constructs

$$\begin{array}{l}
[\![\Gamma \vdash v : exp[\tau]]\!]^{CSP} u = Q.q \to A.v \to SKIP, v \in \mathcal{A}_\tau \\
[\![\Gamma \vdash E_1 * E_2 : exp[\tau']]\!]^{CSP} u = \\
\quad [\![\Gamma \vdash E_1 : exp[\tau_1]]\!]^{CSP} u[Q_1/Q, A_1/A] \;\underset{\{\!|\, Q_1, A_1 \,|\!\}}{\|}\; \\
\quad ([\![\Gamma \vdash E_2 : exp[\tau_2]]\!]^{CSP} u[Q_2/Q, A_2/A] \;\underset{\{\!|\, Q_2, A_2 \,|\!\}}{\|}\; \\
\qquad (Q.q \to Q_1.q \to A_1?v1 : \mathcal{A}_{\tau_1} \to Q_2.q \to A_2?v2 : \mathcal{A}_{\tau_2} \to \\
\qquad\quad A.(v1 * v2) \to SKIP) \setminus \{\!|\, Q_2, A_2 \,|\!\}) \setminus \{\!|\, Q_1, A_1 \,|\!\}
\end{array}$$

Table 4. Command constructs

$$[\![\Gamma \vdash skip : comm]\!]^{CSP} u = Q.run \to A.done \to SKIP$$

$$[\![\Gamma \vdash C;\ M : \sigma]\!]^{CSP} u =$$
$$[\![\Gamma \vdash C : comm]\!]^{CSP} u[Q_1/Q, A_1/A] \underset{\{|\, Q_1, A_1 \,|\}}{\|}$$
$$([\![\Gamma \vdash M : \sigma]\!]^{CSP} u[Q_2/Q, A_2/A] \underset{\{|\, Q_2, A_2 \,|\}}{\|}$$
$$(Q?\mathbf{q} : \mathbf{Q}_\sigma \to Q_1.run \to A_1.done \to Q_2.\mathbf{q} \to A_2?\mathbf{a} : \mathbf{A}^{\mathbf{q}}_\sigma$$
$$\to A.\mathbf{a} \to SKIP) \backslash \{|\, Q_2, A_2 \,|\}) \backslash \{|\, Q_1, A_1 \,|\}$$

$$[\![\Gamma \vdash if\ B\ then\ M_1\ else\ M_2 : \sigma]\!]^{CSP} u =$$
$$[\![\Gamma \vdash B : exp[bool]]\!]^{CSP} u[Q_0/Q, A_0/A] \underset{\{|\, Q_0, A_0 \,|\}}{\|}$$
$$(([\![\Gamma \vdash M_1 : \sigma]\!]^{CSP} u[Q_1/Q, A_1/A] \Box SKIP) \underset{\{|\, Q_1, A_1 \,|\}}{\|}$$
$$(([\![\Gamma \vdash M_2 : \sigma]\!]^{CSP} u[Q_2/Q, A_2/A] \Box SKIP) \underset{\{|\, Q_2, A_2 \,|\}}{\|}$$
$$(Q?\mathbf{q} : \mathbf{Q}_\sigma \to Q_0.q \to A_0?v : \mathcal{A}_{bool} \to$$
$$(Q_1.\mathbf{q} \to A_1?\mathbf{a} : \mathbf{A}^{\mathbf{q}}_\sigma \to A.\mathbf{a} \to SKIP$$
$$\not< v \not> Q_2.\mathbf{q} \to A_2?\mathbf{a} : \mathbf{A}^{\mathbf{q}}_\sigma \to A.\mathbf{a} \to SKIP))$$
$$\backslash \{|\, Q_2, A_2 \,|\}) \backslash \{|\, Q_1, A_1 \,|\}) \backslash \{|\, Q_0, A_0 \,|\}$$

$$[\![\Gamma \vdash while\ B\ do\ C : comm]\!]^{CSP} u =$$
$$(\mu\, p.([\![B : exp[bool]]\!]^{CSP} u[Q_1/Q, A_1/A] \fatsemi p) \Box SKIP)$$
$$\underset{\{|\, Q_1, A_1 \,|\}}{\|} ((\mu\, p'.([\![C : comm]\!]^{CSP} u[Q_2/Q, A_2/A] \fatsemi p')$$
$$\Box SKIP) \underset{\{|\, Q_2, A_2 \,|\}}{\|} (Q.run \to \mu\, p''.Q_1.q \to A_1?v : \mathcal{A}_{bool}$$
$$\to (Q_2.run \to A_2.done \to p'' \not< v \not> A.done \to SKIP))$$
$$\backslash \{|\, Q_2, A_2 \,|\}) \backslash \{|\, Q_1, A_1 \,|\}$$

For function application, the processes for the arguments may be run arbitrarily many times, so they are enclosed in recursions.

5 Correctness and decidability

For any term from second-order IA, the set of all terminated traces of its CSP interpretation is isomorphic to its regular language interpretation $[\![-]\!]^R$, as defined in [11]:

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

$$traces^{\checkmark}([\![\Gamma \vdash M : \theta]\!]^{CSP} u_0) \overset{\phi}{\cong} [\![\Gamma \vdash M : \theta]\!]^R u_0'$$

where u_0' is the environment that maps free identifiers of the term to copy-cat regular languages as defined in [11], and ϕ is defined by:

$$\phi(Q.a) = a \qquad \phi(A.a) = a$$
$$\phi(\iota.Q.a) = a^\iota \qquad \phi(\iota.A.a) = a^\iota$$
$$\phi(\langle a_1, \ldots, a_k \rangle) = \phi(a_1) \cdot \ldots \cdot \phi(a_k)$$

Table 5. Variable constructs

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

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

$$[\![\Gamma \vdash new[\tau]\ \iota := E\ in\ C : comm]\!]^{CSP} u =$$
$$[\![\Gamma \vdash E : exp[\tau]]\!]^{CSP} u[Q_1/Q, A_1/A] \underset{\{|\, Q_1, A_1 \,|\}}{\|}$$
$$([\![\Gamma, \iota : var[\tau] \vdash C : comm]\!]^{CSP} (u \mid \iota \to K^\iota_{var[\tau]})[Q_2/Q, A_2/A]$$
$$\underset{\{|\, Q_2, A_2, \iota \,|\}}{\|} (Q.run \to Q_1.q \to A_1?v : \mathcal{A}_\tau \to Q_2.run \to U_{\iota,v})$$
$$\backslash \{|\, Q_2, A_2, \iota \,|\}) \backslash \{|\, Q_1, A_1 \,|\}$$
$$U_{\iota,v} = (\iota.Q.read \to \iota.A!v \to U_{\iota,v}) \Box$$
$$(\iota.Q.write?v' \to \iota.A.ok \to U_{\iota,v'}) \Box$$
$$(A_2.done \to A.done \to SKIP)$$

Table 6. Functional constructs

$$[\![\Gamma \vdash \iota : \theta]\!]^{CSP} u = u(\iota)$$

$$[\![\Gamma \vdash \iota(M_1, \ldots, M_k) : \sigma']\!]^{CSP} u =$$
$$(\mu\, p_1.([\![\Gamma \vdash M_1 : \sigma_1]\!]^{CSP} u[Q.1/Q, A.1/A] \fatsemi p_1) \Box SKIP)$$
$$\underset{\{|\, Q.1, A.1 \,|\}}{\|} (\cdots$$
$$(\mu\, p_k.([\![\Gamma \vdash M_k : \sigma_k]\!]^{CSP} u[Q.k/Q, A.k/A] \fatsemi p_k) \Box SKIP)$$
$$\underset{\{|\, Q.k, A.k \,|\}}{\|} u(\iota) \backslash \{|\, Q.k, A.k \,|\} \cdots) \backslash \{|\, Q.1, A.1 \,|\}$$

$$[\![\Gamma \vdash let\ \iota(\iota_1 : \sigma_1, \ldots, \iota_k : \sigma_k) = N\ in\ M : \theta]\!]^{CSP} u =$$
$$[\![\Gamma, \iota : \sigma_1 \times \cdots \times \sigma_k \to \sigma' \vdash M : \theta]\!]^{CSP}$$
$$(u \mid \iota \to [\![\Gamma, \iota_1 : \sigma_1, \ldots, \iota_k : \sigma_k \vdash N : \sigma']\!]^{CSP}$$
$$(u \mid \iota_1 \to K^{\iota_1}_{\sigma_1}, \ldots, \iota_k \to K^{\iota_k}_{\sigma_k}))$$

Proof. Since free identifiers are interpreted by copy-cat strategies for their types in both representations, the theorem follows from the following implication:

$$\forall\, \iota_i : \theta_i \in \Gamma\ .\ traces^{\checkmark}(u(\iota_i)) \overset{\phi}{\cong} u'(\iota_i) \ \Rightarrow$$
$$traces^{\checkmark}([\![\Gamma \vdash M : \theta]\!]^{CSP} u) \overset{\phi}{\cong} [\![\Gamma \vdash M : \theta]\!]^R u' \qquad (1)$$

The claim (1) is proved by a routine induction on the typing rules in Table 1, by showing that the definitions of CSP processes in Section 4 correspond to the definitions of regular expressions in [11]. □

Two terms M and N in type context Γ and of type θ are observationally equivalent, written $\Gamma \vdash M \equiv_\theta N$, if and only if, for any term-with-hole $C[-]$ such that both $C[M]$ and $C[N]$ are closed terms of type *comm*, $C[M]$ converges (i.e. evaluates to *skip*) if and only if $C[N]$ converges. It was proved in [2] 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. (The 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 [11] that observational equivalence coincides with equality of regular language interpretations. By Theorem 1, we have that observational equivalence is captured by two traces refinements:

Corollary 1 (Observational equivalence).

$$\Gamma \vdash M \equiv_\theta N \;\Leftrightarrow$$
$$[\![\Gamma \vdash M : \theta]\!]^{CSP} u_0 \Box RUN_{\mathcal{A}_{\Gamma\vdash\theta}} \sqsubseteq_T [\![\Gamma \vdash N : \theta]\!]^{CSP} u_0 \wedge$$
$$[\![\Gamma \vdash N : \theta]\!]^{CSP} u_0 \Box RUN_{\mathcal{A}_{\Gamma\vdash\theta}} \sqsubseteq_T [\![\Gamma \vdash M : \theta]\!]^{CSP} u_0$$

Proof. From [11], we have that $\Gamma \vdash M \equiv_\theta N$ if and only if $[\![\Gamma \vdash M]\!]^R u_0' = [\![\Gamma \vdash N]\!]^R u_0'$. By Theorem 1, the latter is equivalent to

$$traces^{\checkmark}([\![\Gamma \vdash M]\!]^{CSP} u_0) = traces^{\checkmark}([\![\Gamma \vdash N]\!]^{CSP} u_0)$$

The corollary follows by the traces semantics of the $\Box$ operator and the $RUN_{\mathcal{A}_{\Gamma\vdash\theta}}$ process. $\Box$

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 $\Gamma \vdash M : \theta$, the CSP process $[\![\Gamma \vdash M : \theta]\!]^{CSP} u_0$ is finite state.*

Proof. Since the copy-cat processes K^ι_θ are finite state, the theorem is implied by the following claim: for any term $\Gamma \vdash M : \theta$ and any environment u which maps each identifier in Γ to a finite-state process, $[\![\Gamma \vdash M : \theta]\!]^{CSP} u$ is finite state.

In the fragment of CSP we are using, the only operators which can result in infinite transition systems are the infinite choice operator $?x : A \to P_x$ with an infinite set A, and recursion. The claim therefore follows by induction on the typing rules in Table 1, and these observations:

- each alphabet $\mathcal{A}_{\Gamma\vdash\theta}$ is finite;
- each use of the choice operator is over a finite set;
- the recursive process in the definition of $K^\iota_{\sigma_1\times\cdots\times\sigma_k\to\sigma'}$ is finite state
- the recursive processes $U_{\iota,v}$ with $v \in \mathcal{A}_\tau$ are finite state because $\mathcal{A}_\tau$ is a finite set;
- the recursive processes in the definitions for iteration and function application are finite state by the inductive hypothesis. $\Box$

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.* $\Box$

Example 1. Consider the process for the term

$$c : comm \vdash new[bool]\, x := true\, in\, c : comm$$

It has the same traces as

$$\begin{array}{l}(Q_1.q \to A_1.tt \to SKIP) \underset{\{|\,Q_1,A_1\,|\}}{\|} \\ ((Q_2.run \to c.Q.run \to c.A.done \to A_2.done \to SKIP) \\ \quad \underset{\{|\,Q_2,A_2,x\,|\}}{\|} (Q.run \to Q_1.q \to A_1?v \to Q_2.run \to U_{x,v}) \\ \quad \backslash\{|\,Q_2,A_2,x\,|\})\backslash\{|\,Q_1,A_1\,|\}\end{array}$$

Simplifying further yields

$$Q.run \to c.Q.run \to c.A.done \to A.done \to SKIP$$

which is the process for the term $c : comm \vdash c : comm$.

By Corollary 1, we conclude that

$$c : comm \vdash new[bool]\, x := true\, in\, c \;\equiv_{comm}\; c$$

This observational equivalence reflects the fact that a non-locally defined command cannot modify a local variable [11]. $\Box$

6 Property verification

In addition to checking observational equivalence of two program terms, it is desirable to be able to check properties of terms. Recall that for any term $\Gamma \vdash M : \theta$, the set of terminated traces $traces^{\checkmark}([\![\Gamma \vdash M : \theta]\!]^{CSP} u_0)$ is the set of all complete plays of the strategy for $\Gamma \vdash M : \theta$. We therefore focus on properties of finite traces, and take the view that $\Gamma \vdash M : \theta$ satisfies such a property if and only if all traces in $traces^{\checkmark}([\![\Gamma \vdash M : \theta]\!]^{CSP} u_0)$ satisfy it.

6.1 Properties as finite automata

Every property of program terms such that the set of all finite traces which satisfy it is regular language, can be represented in CSP. Suppose A is an automaton with finite alphabet Σ, finite set of states Q, transition relation $T \subseteq Q \times \Sigma \times Q$, initial states $Q^0 \subseteq Q$, and accepting states $F \subseteq Q$. For any $q \in Q \setminus F$, we define

$$P_q = \Box_{(q,a,q')\in T}\, a \to P_{q'}$$

For any $q \in F$, we define

$$P_q = (\Box_{(q,a,q')\in T}\, a \to P_{q'}) \,\Box\, SKIP$$

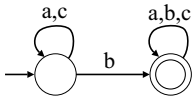

Fig. 1. A finite automaton

This is a valid system of recursive CSP process definitions, so we can let

$$P_A = \Box_{q \in Q^0} P_q$$

We then have that P_A has finitely many states and

$$t \text{ is accepted by } A \text{ iff } t \in traces^{\checkmark}(P_A)$$

Example 2. Consider the automaton A in Figure 1, whose alphabet is $\{a, b, c\}$. It accepts a finite trace t if and only if b eventually occurs in it.

The CSP process P_A is defined as:

$$P_A = P_1$$
$$P_1 = (a \to P_1) \,\Box\, (b \to P_2) \,\Box\, (c \to P_1)$$
$$P_2 = (a \to P_2) \,\Box\, (b \to P_2) \,\Box\, (c \to P_2) \,\Box\, SKIP$$

□

Therefore, given a program term $\Gamma \vdash M : \theta$ and a finite automaton A with alphabet $\mathcal{A}_{\Gamma\vdash\theta}$, we can decide whether A accepts each complete play of the strategy for $\Gamma \vdash M : \theta$ by checking (e.g. using FDR) the traces refinement

$$P_A \,\Box\, RUN_{\mathcal{A}_{\Gamma\vdash\theta}} \sqsubseteq_T [\![\Gamma \vdash M : \theta]\!]^{CSP} u_0 \qquad (2)$$

6.2 Properties in temporal logic

A standard way of writing properties of linear behaviours is by linear temporal logic. Given a finite set Σ, we consider the following formulas. In addition to propositional connectives, they contain the temporal operators 'next-time' and 'until'.

$$\phi ::= true \mid a \in \Sigma \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \bigcirc\phi \mid \phi_1 \,U\, \phi_2$$

We call this logic LTL_f^{Σ}, because we give it semantics over finite traces (i.e. sequences) of elements of Σ. For any trace t of length k, we write its elements as $t_1, \ldots, t_k$, and we write t^i for its ith suffix $\langle t_i, \ldots, t_k\rangle$.

$$\begin{array}{lll}
t \models true & & \\
t \models a & \text{iff} & t \neq \langle\rangle \text{ and } t_1 = a \\
t \models \neg\phi & \text{iff} & t \not\models \phi \\
t \models \phi_1 \vee \phi_2 & \text{iff} & t \models \phi_1 \text{ or } t \models \phi_2 \\
t \models \bigcirc\phi & \text{iff} & t \neq \langle\rangle \text{ and } t^2 \models \phi \\
t \models \phi_1 \,U\, \phi_2 & \text{iff} & \exists\, i \in \{1, \ldots, |t|+1\} \,.\, t^i \models \phi_2 \text{ and} \\
 & & \quad \forall\, j \in \{1, \ldots, i-1\} \,.\, t^j \models \phi_1
\end{array}$$

The Boolean constant *false*, and Boolean operators such as $\wedge$ and $\to$ can be defined as abbreviations in the usual ways. The same is true for the temporal operators 'eventually' and 'always':

$$\Diamond\phi = true \,U\, \phi \qquad \Box\phi = \neg\Diamond\neg\phi$$

A term $\Gamma \vdash M : \theta$ satisfies a formula ϕ of $\mathrm{LTL}_f^{\mathcal{A}_{\Gamma\vdash\theta}}$ if and only if

$$\forall\, t \in traces^{\checkmark}([\![\Gamma \vdash M : \theta]\!]^{CSP} u_0) \,.\, t \models \phi$$

Example 3. Consider the following term:

$$b : exp[bool], c : comm \vdash while\ b\ do\ c : comm$$

The terminated traces of its process consist of arbitrarily many times evaluating b, obtaining tt, and running c; followed by evaluating b and obtaining ff.

Hence, this term satisfies $\Diamond b.A.ff$, but does not satisfy $\Diamond c.Q.run$. It also satisfies $\Box(b.A.tt \to \Diamond c.Q.run)$, but does not satisfy $\Box(b.Q.q \to \Diamond c.Q.run)$. □

There is an algorithm which, given any formula ϕ of LTL_f^{Σ}, constructs a finite automaton A with alphabet Σ, which accepts a finite trace t if and only if $t \models \phi$ (see Appendix A). Using the procedure in Section 6.1, we can then obtain a CSP process P_ϕ^{Σ} with finitely many states and such that

$$t \models \phi \;\Leftrightarrow\; t \in traces^{\checkmark}(P_\phi^{\Sigma})$$

We therefore have a decision procedure which, given a term $\Gamma \vdash M : \theta$ and a formula ϕ of $\mathrm{LTL}_f^{\mathcal{A}_{\Gamma\vdash\theta}}$, checks satisfaction. It reduces the question to the following traces refinement:

$$P_\phi^{\mathcal{A}_{\Gamma\vdash\theta}} \Box RUN_{\mathcal{A}_{\Gamma\vdash\theta}} \sqsubseteq_T [\![\Gamma \vdash M : \theta]\!]^{CSP} u_0 \qquad (3)$$

7 Polymorphic terms and properties

Consider the following extension of the IA fragment in this paper:

$$\tau ::= int_n \mid bool \mid \alpha$$
$$\frac{\Gamma \vdash E_1 : exp[\alpha] \qquad \Gamma \vdash E_2 : exp[\alpha]}{\Gamma \vdash E_1 = E_2 : exp[bool]}$$

Here α ranges over an infinite set of *data-type variables*. Such a variable stands for a polymorphic data type. The only operation on values from such a data type is equality.

We also extend the syntax of LTL_f^{Σ} to allow alphabets of the form

$$\bigcup_{a \in \Sigma^\dagger} \{a\} \cup \bigcup_{a \in \Sigma^\ddagger} a.\alpha_a$$

where $\Sigma^\dagger$ and $\Sigma^\ddagger$ are finite sets, and for any $a \in \Sigma^\ddagger$, α_a is a data-type variable. (Note that, in the presence of data-type variables, the alphabets $\mathcal{A}_{\Gamma\vdash\theta}$ are of this form.) The formulas are:

$$\phi ::= true \mid a \in \Sigma^\dagger \cup \Sigma^\ddagger \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \bigcirc\phi \mid \phi_1 \,U\, \phi_2$$

Suppose α is a data-type variable.

- If τ is either some int_n or $bool$, let $-[\tau/\alpha]$ denote substitution of τ for α in types and terms.
- Suppose Σ is an alphabet as above, and S is a finite set. Let $\Sigma[S/\alpha]$ denote the alphabet

$$\bigcup_{a\in\Sigma^\dagger}\{a\}\cup\bigcup_{a\in\Sigma^\ddagger\wedge\alpha_a=\alpha}\{a.a'\mid a'\in S\}\cup\bigcup_{a\in\Sigma^\ddagger\wedge\alpha_a\neq\alpha}a.\alpha_a$$

  If ϕ is an LTL_f^Σ formula, let $\phi[S/\alpha]$ denote the formula where any $a\in\Sigma^\ddagger$ such that $\alpha_a=\alpha$ is replaced by $\bigvee_{a'\in S}a.a'$.

In what follows, we shall for simplicity work with a single data-type variable α. When we write $-[n]$ as a substitution, we mean $-[int_n/\alpha]$ (for types and terms) or $-[\{0,\ldots,n-1\}/\alpha]$ (for alphabets and formulas). Since α is the only data-type variable, these substitutions yield types, terms, alphabets and formulas which contain no data-type variables.

We shall not consider terms which contain the *let* construct for function declaration. By β-reduction, this is no loss of generality.

The following lemma states that, when applied to polymorphic terms and formulas, the definitions in Sections 4 and 6 yield CSP processes which are *data independent* with respect to α [16, Section 2.7]. It also states that performing any substitution $-[n]$ after obtaining such a CSP process is equivalent to performing it before.

Lemma 1. *(a) For any term $\Gamma\vdash M:\theta$, $[\![\Gamma\vdash M:\theta]\!]^{CSP}u_0$ is data independent with respect to α, and for any $n>0$,*

$$traces(([\![\Gamma\vdash M:\theta]\!]^{CSP}u_0)[n]) = traces([\![\Gamma[n]\vdash M[n]:\theta[n]]\!]^{CSP}u_0[n])$$

(b) For any formula ϕ of LTL_f^Σ, P_ϕ^Σ is data independent with respect to α, and for any $n>0$,

$$traces(P_\phi^\Sigma[n]) = traces(P_{\phi[n]}^{\Sigma[n]})$$

Proof. By induction on typing rules. □

The first theorem on parameterised verification of polymorphic terms and properties states that, if equality between values of type α is not used, it suffices to check satisfaction when α is substituted by a one-element data type:

Theorem 3. *Suppose $\Gamma\vdash M:\theta$ is a term which contains no equalities between values from α, and ϕ is a formula of $LTL_f^{\mathcal{A}_{\Gamma\vdash\theta}}$. If $\Gamma[1]\vdash M[1]:\theta[1]$ satisfies $\phi[1]$, then for all $n>0$, $\Gamma[n]\vdash M[n]:\theta[n]$ satisfies $\phi[n]$.*

Proof. This is a corollary of Lemma 1 and [16, Theorem 5.1.2]. □

Table 7. A measure of terms

$$w(\Gamma\vdash m:exp[int_n])=0$$
$$w(\Gamma\vdash b:exp[bool])=0$$
$$w(\Gamma\vdash E_1=E_2:exp[bool]) = \max\{w(\Gamma\vdash E_1:exp[\alpha]),w(\Gamma\vdash E_2:exp[\alpha])+1\}$$
$$w(\Gamma\vdash E_1*E_2:exp[\tau']) = \max\{w(\Gamma\vdash E_1:exp[\tau_1]),w(\Gamma\vdash E_2:exp[\tau_2])\},\ \text{otherwise}$$
$$w(\Gamma\vdash skip:comm)=0$$
$$w(\Gamma\vdash C;\ M:\sigma)=\max\{w(\Gamma\vdash C:comm),w(\Gamma\vdash M:\sigma)\}$$
$$w(\Gamma\vdash if\ B\ then\ M_1\ else\ M_2:\sigma) = \max\{w(\Gamma\vdash B:exp[bool]),w(\Gamma\vdash M_1:\sigma),w(\Gamma\vdash M_2:\sigma)\}$$
$$w(\Gamma\vdash while\ B\ do\ C:comm) = \max\{w(\Gamma\vdash B:exp[bool]),w(\Gamma\vdash C:comm)\}$$
$$w(\Gamma\vdash V:=E:comm) = \max\{w(\Gamma\vdash V:var[\alpha])+1,w(\Gamma\vdash E:exp[\alpha])\}$$
$$w(\Gamma\vdash V:=E:comm) = \max\{w(\Gamma\vdash V:var[\tau]),w(\Gamma\vdash E:exp[\tau])\},\ \text{if } \tau\neq\alpha$$
$$w(\Gamma\vdash !V:exp[\tau])=w(\Gamma\vdash V:var[\tau])$$
$$w(\Gamma\vdash new[\alpha]\,\iota:=E\ in\ C:comm) = \max\{w(\Gamma\vdash E:exp[\alpha]),w(\Gamma,\iota:var[\alpha]\vdash C:comm)+1\}$$
$$w(\Gamma\vdash new[\tau]\,\iota:=E\ in\ C:comm) = \max\{w(\Gamma\vdash E:exp[\tau]),w(\Gamma,\iota:var[\tau]\vdash C:comm)\},\tau\neq\alpha$$
$$w(\Gamma,\iota:\theta\vdash\iota:\theta)=\begin{cases}1,&\text{if }\theta\text{ contains }\alpha\\0,&\text{otherwise}\end{cases}$$
$$w(\Gamma\vdash\iota(M_1,\ldots,M_k):\sigma') = \max\{w(\Gamma\vdash M_1:\sigma_1),\ldots,w(\Gamma\vdash M_k:\sigma_k),1\},\ \text{if }\sigma'\text{ cont. }\alpha$$
$$w(\Gamma\vdash\iota(M_1,\ldots,M_k):\sigma') = \max\{w(\Gamma\vdash M_1:\sigma_1),\ldots,w(\Gamma\vdash M_k:\sigma_k)\},\ \text{otherwise}$$

For any term, we now define a natural number $w(\Gamma\vdash M:\theta)$, by recursion on the typing rules: see Table 7. Informally, this is an upper bound on the number of values of type α in any state of the process $[\![\Gamma\vdash M:\theta]\!]^{CSP}u_0$.

The following lemma states that $w(\Gamma\vdash M:\theta)$ is not less than the corresponding measure of the *symbolic transition system* of $[\![\Gamma\vdash M:\theta]\!]^{CSP}u_0$ [16, Section 3.3.3]:

Lemma 2. *For any term $\Gamma\vdash M:\theta$,*

$$w(\Gamma\vdash M:\theta)\geq W(\mathcal{S}_{[\![\Gamma\vdash M:\theta]\!]^{CSP}u_0})$$

Proof. By induction on the typing rules. □

Our second theorem applies to terms which can contain equalities between values of type α, and states that it suffices to check satisfaction when α is substituted by a data type whose size depends on the w measure:

Theorem 4. *Suppose $\Gamma \vdash M : \theta$ is a term, and ϕ is a formula of $LTL_f^{\mathcal{A}_{\Gamma\vdash\theta}}$. Let $n = w(\Gamma \vdash M : \theta) + 1$. If $\Gamma[n] \vdash M[n] : \theta[n]$ satisfies $\phi[n]$, then for all $n' > 0$, $\Gamma[n'] \vdash M[n'] : \theta[n']$ satisfies $\phi[n']$.*

Proof. This is a corollary of Lemmas 1 and 2, and [16, Theorem 5.4.7(III)]. □

Some applications of the two theorems in this section can be found in Section 9.2.

8 Compiler

We have implemented a compiler in Java [5], which automatically converts a term-in-context (i.e. an open program fragment) into a CSP process which represents its game semantics. The process is defined by a script in machine readable CSP [21].

In the input syntax, an integer constant m is implicitly of type int_{n+1}. An operation between values of types int_{n_1} and int_{n_2} produces a value of type $int_{\max\{n_1,n_2\}}$. The operation is performed modulo $\max\{n_1, n_2\}$.

The scripts output by the compiler can be loaded into the tools ProBE for interactive exploration of transition systems, and FDR for automatic analysis and interactive debugging [10]. One of the functions of FDR is to check traces refinement between two finite-state processes. As we saw above, this can be used to decide observational equivalence between two terms (Corollary 1), containment in a regular language (2), and satisfaction of a linear temporal logic formula (3).

FDR offers a number of hierarchical compression algorithms [20], which can be applied during model generation and refinement checking. The scripts which our compiler produces normally contain instructions to apply diamond elimination (which eliminates all τ events from a transition system) and strong bisimulation quotienting to subprocesses which model local variable declaration subterms. This exploits the fact that game semantics hides interactions between a local variable and its scope. The interaction events become τ events, enabling the model to be reduced.

9 Applications

We now consider applications of the approach proposed above and discuss experimental results for two kinds of example: a sorting algorithm, and an abstract data type implementation.

9.1 A sorting algorithm

In this section, we analyse the bubble-sort algorithm. The input to the compiler is in Figure 2, where the array size is a meta variable $k > 0$. For readability, the syntax is slightly different from the one in Section 2, but isomorphic. In particular, dereferencing is implicit.

```
1  x[k] : var int_3 ⊢
2  new int_3 a[k]:=0 in
3  new int_{k+1} i:=0 in
4  while (i<k) {a[i]:=x[i]; i:=i+1;}
5  new bool flag:=true in
6  while(flag) {
7    i:=0;
8    flag:=false;
9    while(i<k−1) {
10     if(a[i]>a[i+1]) {
11       flag:=true;
12       new int_3 temp:=a[i] in
13       a[i]:=a[i+1];
14       a[i+1]:=temp;}
15     i:=i+1;}}
16 i:=0;
17 while(i<k) {x[i]:=a[i]; i:=i+1;}
18 : comm
```

Fig. 2. Source code of bubble sort

The program first copies the input array x into a local array a, which is then sorted, and copied back into x. The local array a is not visible from the outside of the program, so only reads and writes of the non-local array x are seen in the model. A transition system for $k = 2$ is shown in Figure 3. The left-hand half represents reads of all possible combinations of values from x, while the right-hand half represents writes of the same values in sorted order.

Table 8 contains the experimental results for minimal model generation. The experiment consisted of running the compiler on the bubble sort implementation, and then letting FDR generate a transition system for the resulting process. The latter stage involved a number of hierarchical compressions, as outlined in Section 8. We list the execution time in minutes, the size of the largest generated transition system, and the size of the final transition system. We ran FDR on a Research Machine AMD Athlon 64(tm) Processor 3500^+ with 2GB RAM. The results from the tool based on regular expressions were obtained on a SunBlade 100 with 2GB RAM [4]. The one extra state in the CSP minimal models is the special terminated state Ω. The CSP approach yields better results in time and space. This is firstly due to composition of strategies being represented in CSP using the renaming, parallel and hiding operators, and FDR being highly optimised for verification of such networks of processes. Secondly, FDR builds the models gradually, at each stage compressing the subterm models.

Further information about minimal model generation for $k = 20$ is shown in Figure 4. FDR first produces a transition system for the subprogram which is the scope of the declaration of the local array a. Each component of a, which are indexed from 0 to 19, is represented by the process $U_{a.i,0}$ (see Table 5). FDR obtains the final

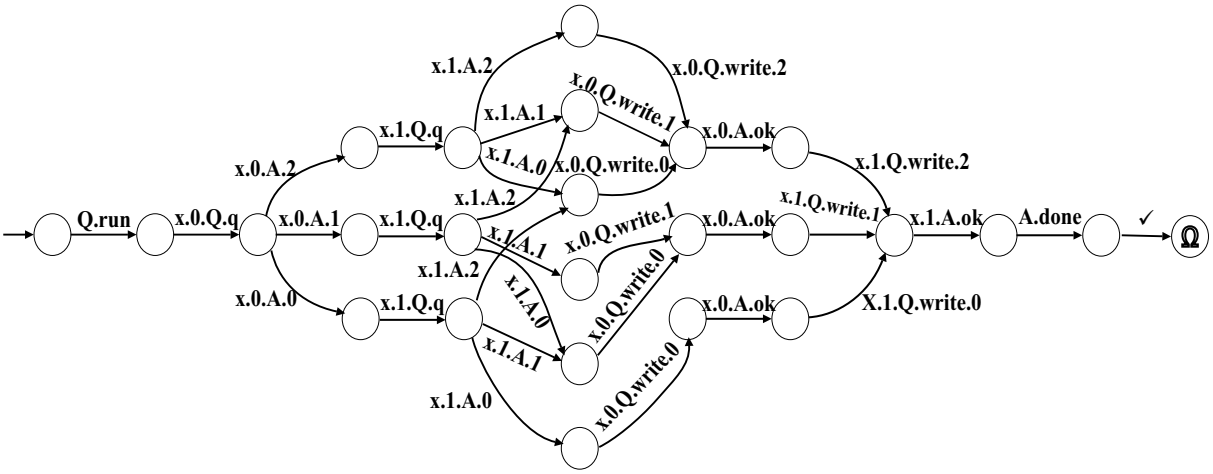

Fig. 3. A transition system for bubble sort with $k = 2$

Table 8. Bubble sort minimal model generation

k	CSP			Regular expressions		
	T	Max.	Model	T	Max.	Model
5	4	1 775	164	5	3 376	163
10	13	21 015	949	10	64 776	948
15	35	115 125	2 859	120	352 448	2 858
20	70	378 099	6 394	240	1 153 240	6 393
30	390	5 204 232	20 339	failed		

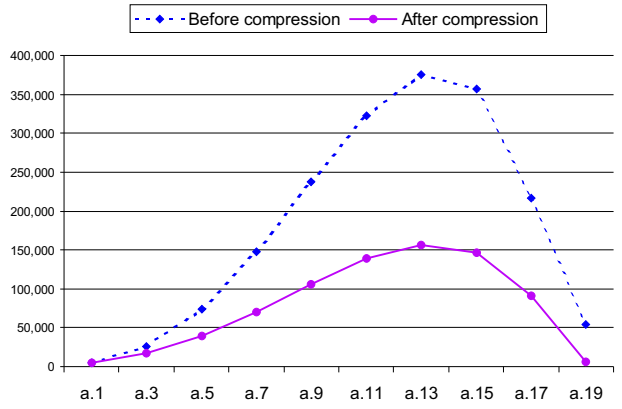

Fig. 4. Effects of compressions for bubble sort with $k = 20$

model by taking the transition system for the scope of a and composing it with transition systems for the components of a in turn. At each step, compression algorithms are applied. In the figure, we show numbers of states before and after compression, after every two steps.

We now turn to verifying absence of out-of-bounds errors, which is expressed by the formula $\neg\Diamond oub$ (see Section 2.1). Let us modify the term in Figure 2 by replacing $k-1$ in line 9 by k, which introduces an out-of-bounds error.

Table 9. Checking $\neg\Diamond oub$ for an erroneous bubble sort

k	Total	Spec.	Impl.	Ref.
29	250.5	10	240	0.5
30	317.5	12	305	0.5
31	494.2	12.5	391	0.7

Table 9 shows some experimental results for checking the corresponding traces refinement (3). We did not apply compressions after composing the last component of a with the rest of the program. Instead, a composite model is generated on-the-fly during refinement checking. This enabled us to check the property for array size 31, although the minimal model generation did not succeed for this size. The times shown in Table 9 are: total execution time, time to process the specification, time to process the implementation, and time to check refinement. They are all in minutes.

9.2 An abstract data type implementation

Figure 5 contains an implementation of a queue of maximum size k as a circular array. Values stored in the queue are of type α, which is a data-type variable. There are four free identifiers: commands *empty* and $overflow$, expression p of type α, and command $ANALYSE$ which takes two arguments. After implementing the queue by a sequence of local declarations, we export the functions *add* and *next* by calling $ANALYSE$ with arguments $add(p)$ and $next()$. Game semantics will give us a model which contains all interleavings of calls to $add(p)$ and $next()$, corresponding to all possible behaviours of the non-local function $ANALYSE$. Since the expression p is also non-local, the value of p can be different each time *add* is called. The non-local commands $empty()$ and $overflow()$ handle calls to *next* on the empty queue, respectively *add* on a full queue.

```
1  empty(): comm, overflow(): comm, p: exp α,
2  ANALYSE(comm, exp α): comm ⊢
3  new α buffer[k]:=p in
4  new int_k front:=0 in
5  new int_k tail:=0 in
6  new int_{k+1} queue_size:=0 in
7  let exp bool isempty() {return queue_size==0;} in
8  let exp bool isfull() {return queue_size==k;} in
9  let comm add(exp α x) {
10   if (isfull()) overflow(); else {
11     buffer[tail]:=x;
12     tail:=tail+1;
13     queue_size:=queue_size+1;}} in
14 let exp α next() {
15   if (isempty()) {
16     empty(); return p;}
17   else {
18     front:=front+1;
19     queue_size:=queue_size−1;
20     return buffer[front−1];}} in
21 ANALYSE(add(p),next())
22 : comm
```

Fig. 5. A queue implementation

Table 10. Checking $\neg\Diamond empty$

k	Total	Spec.	Impl.	Ref.
100	1110	935	175	0
105	1415	1200	215	0
110	1610	1340	270	0

A transition system for $k = 2$ and with int_1 substituted for α is shown in Figure 6. For clarity, ADD and $NEXT$ are written instead of $ANALYSE.1$ and $ANALYSE.2$. Also, p moves for initialising the array and within calls of *next* are not shown.

Consider the formulas $\neg\Diamond empty$ and $\neg\Diamond overflow$, which assert that the non-local error handlers are never called. By β-reduction and Theorem 3, if the queue implementation term satisfies one of these properties with int_1 substituted for α, then it satisfies the same property with any int_n substituted for α.

As expected, checking these properties with int_1 substituted for α returns that they are not satisfied. The counter-example traces which the FDR debugger gives correspond to: a single $NEXT$ call after which *empty* is called, and $k + 1$ consecutive calls of ADD after which $overflow$ is called.

Tables 10 and 11 show some experimental results for checking the two properties. For efficiency, 0 has been substituted for p in the initialisation of the array and in the declaration of *next*. As in Table 9, we list in minutes the total execution time, the times for processing the specification and the implementation, and the refinement checking time.

Properties involving local data can be checked by calling a non-local command *error* in case of violation.

Table 11. Checking $\neg\Diamond overflow$

k	Total	Spec.	Impl.	Ref.
100	880.5	705	175	0.5
105	1315.6	1100	215	0.6
110	1520.7	1250	270	0.7

For example, we can check whether, whenever a value p is enqueued and then the queue is emptied, the last value dequeued is equal to p. We replace the call to $ANALYSE$ in Figure 5 by the following, and check the formula $\neg\Diamond error$.

```
let comm validate() {
  if (isfull()) skip; else {
    new α y:=p in add(y);
    new α z:=next() in
    while (not isempty()) z:=next();
    if (z==y) then skip; else error();}} in
ANALYSE(add(p),next(),validate())
: comm
```

The modified queue implementation term contains an equality between values of type α, so Theorem 3 is not applicable. However, Theorem 4 gives us that, if $\neg\Diamond error$ is satisfied with int_{k+5} substituted for α, then it is satisfied with any int_n substituted for α.

10 Future work

In this paper, we only considered data types of finite integers and Booleans. We also obtained theorems for parameterised verification of polymorphic terms and properties, which may contain a data-type variable α. The only operator on values of type α is equality, and α may occur in types of free identifiers.

An interesting direction for extension is to consider infinite integers with all the usual operators. Counter-example guided abstraction refinement would be used to check successively more precise approximations, until either an error is found or the property is verified. A framework and some initial results have recently been obtained [9].

It is also important to address recursion, third-order functions, concurrency [12], and other programming language features.

References

1. S. Abramsky. Semantics of interaction. In *Semantics and Logics of Computation*, (Cambridge University Press, 1997), 1–32.
2. 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).

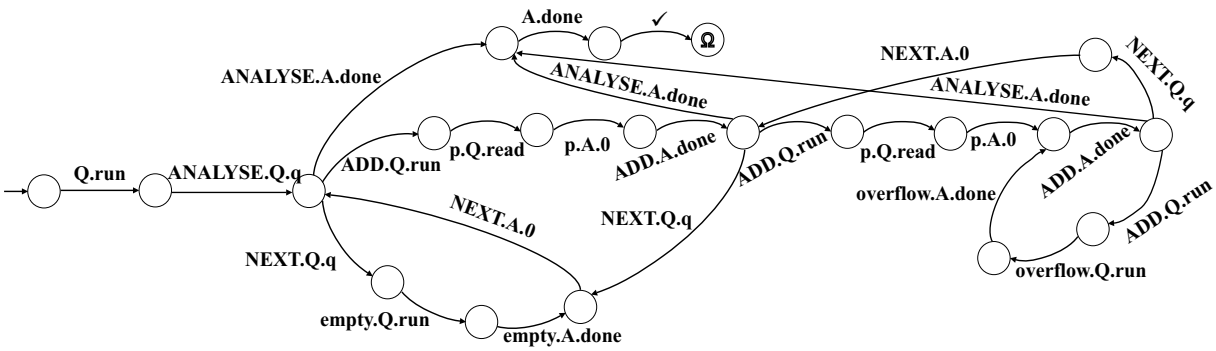

Fig. 6. A transition system for the queue implementation

3. S. Abramsky. Algorithmic game semantics: A tutorial introduction. Lecture notes, *Marktoberdorf International Summer School*. (2001).
4. 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**, (2004), 421–435.
5. A.W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*, 2nd edition. (Cambridge University Press, 2002).
6. T. Ball and S.K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In Proceedings of *POPL*, ACM SIGPLAN Notices **37**(1), (2002).
7. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings of *Logic of Programs Workshop*, LNCS **131**, (1981).
8. E.M. Clarke, O. Grumberg and D. Peled, *Model Checking*. (MIT Press, 2000).
9. A. Dimovski, D. Ghica, and R. Lazić. Data-abstraction refinement: a game semantic approach. In Proceedings of *SAS*, LNCS, (September 2005).
10. Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, (2000).
11. D. Ghica and G. McCusker. The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science **309** (1–3), (2003), 469–502.
12. D. Ghica, A.Murawski and C.-H.L.Ong. Syntactic Control of Concurrency. In Proceedings of *ICALP*, LNCS **3142**, (July 2004).
13. C.A.R. Hoare. *Communicating Sequential Processes*. (Prentice Hall, 1985).
14. J.M.E. Hyland and C.-H.L. Ong. On full abstraction for PCF: I, II and III. Information and Computation **163**, (2000), 285–400.
15. J. Laird. *A Semantic Analysis of Control*. PhD thesis, University of Edinburgh, (1998).
16. R. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. DPhil thesis, Computing Laboratory, Oxford University, (1999).
17. A. Murawski and I. Walukiewicz. Third-Order Idealized Algol with Iteration Is Decidable. In Proceedings of *FoSSaCS*, LNCS **3411**, (2005), 202–218.
18. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Proceedings of *the Fifth International Symposium on Programming*, (1981).
19. J.C. Reynolds. The essence of Algol. In Proceedings of *ISAL*, (1981), 345–372.
20. A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson and J.B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In Proceedings of *TACAS*, LNCS **1019**, (1995), 133–152.
21. A.W. Roscoe, *The Theory and Practice of Concurrency*. (Prentice Hall, 1998).
22. M.Y. Vardi and P. Wolper, An automata-theoretic approach to automatic program verification. In Proceedings of 1st LICS, (1986), 332–344.

A Finite automaton construction for LTL_f^{Σ}

Suppose ϕ is an LTL_f^{Σ} formula. Without loss of generality, we assume that ϕ contains no subformulas of the form $\neg\neg\psi$. Whenever ψ is a subformula of ϕ of the form $\neg\psi'$, if we write $\neg\psi$, we mean ψ'.

Let $Cl(\phi)$ be the set of all subformulas of ϕ and their negations, and for any subformula of the form $\psi_1 U \psi_2$, the formulas $\bigcirc(\psi_1 U \psi_2)$ and $\neg \bigcirc (\psi_1 U \psi_2)$.

We now define a finite automaton $A = (Q, T, Q^0, F)$ on finite words over Σ. The definition is a variant of the construction of automata on infinite words for state-based LTL [22].

Let Q be the set of all $q \subseteq Cl(\phi)$ such that:

(i) for each $\psi \in Cl(\phi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both;
(ii) $\neg true \notin q$;
(iii) if $a, a' \in q$, then $a = a'$;
(iv) if $\psi_1 \vee \psi_2 \in q$, then either $\psi_1 \in q$ or $\psi_2 \in q$;
(v) if $\neg(\psi_1 \vee \psi_2) \in q$, then $\neg\psi_1 \in q$ and $\neg\psi_2 \in q$;
(vi) if $\psi_1 \, U \, \psi_2 \in q$, then either $\psi_2 \in q$ or $\psi_1, \bigcirc(\psi_1 \, U \, \psi_2) \in q$;
(vii) if $\neg(\psi_1 \, U \, \psi_2) \in q$, then $\neg\psi_2 \in q$, and either $\neg\psi_1 \in q$ or $\neg \bigcirc (\psi_1 \, U \, \psi_2) \in q$.

We define $(q, a, q') \in T$ if and only if:

(I) if $a' \in q$, then $a = a'$;
(II) if $\neg a' \in q$, then $a \neq a'$;
(III) if $\bigcirc\psi \in q$, then $\psi \in q'$;

(IV) if $\neg \bigcirc \psi \in q$, then $\neg\psi \in q'$.

$Q^0 = \{q \in Q \mid \phi \in q\}$.

Let $q \in F$ if and only if no member of q is of the form a or $\bigcirc\psi$.

Proposition 1. *For any finite trace t over Σ, t is accepted by A if and only if $t \models \phi$.*

Proof. For the left-to-right implication, it suffices to show that, whenever t is accepted from some q, and $\psi \in q$, then $t \models \psi$. That claim is proved by induction on the length of t.

Consider the case $|t| = 0$, i.e. $t = \langle\rangle$. Then $q \in F$. By induction on the $\neg$-free height of ψ, i.e. the height of the syntax tree of ψ without counting the $\neg$ operator, it follows that, for any $\psi \in q$, $\langle\rangle \models \psi$.

Now, assume $|t| > 0$ and the claim holds for all t' with $|t'| = |t| - 1$. Induction on the $\neg$-free height of ψ is used again. We show only the most complex case, namely $\neg(\psi_1 \, U \, \psi_2) \in q$.

By (vii) above, we have $\neg\psi_2 \in q$, and either $\neg\psi_1 \in q$ or $\neg \bigcirc (\psi_1 \, U \, \psi_2) \in q$. By the inner inductive hypothesis, $t \models \neg\psi_2$.

If $\neg\psi_1 \in q$, the inner inductive hypothesis gives us $t \models \neg\psi_1$, so $t \models \neg(\psi_1 \, U \, \psi_2)$.

If $\neg \bigcirc (\psi_1 \, U \, \psi_2) \in q$, let q' be such that $q \xrightarrow{t_1} q'$ and t^2 is accepted from q'. By (IV) above, we have $\neg(\psi_1 \, U \, \psi_2) \in q'$. From the outer inductive hypothesis, $t^2 \models \neg(\psi_1 \, U \, \psi_2)$. Recalling that $t \models \neg\psi_2$, we have $t \models \neg(\psi_1 \, U \, \psi_2)$ as required.

For the right-to-left implication, given any finite trace t over Σ, let $q_i = \{\psi \in Cl(\phi) \mid t^i \models \psi\}$, for each $i \in \{1, \ldots, |t| + 1\}$. It is routine to show that

$$q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} \cdots q_{|t|+1}$$

is an accepting run in A. □

Example 4. Suppose $\Sigma = \{a, b\}$ and $\phi = a \, U \, b$. The construction above produces the automaton shown in Figure 7. □

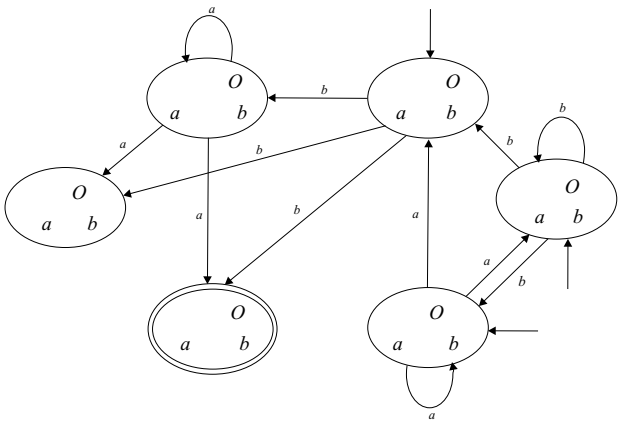

Fig. 7. A finite automaton for $a \, U \, b$