# Software Model Checking Based on Game Semantics and CSP [1]

Aleksandar Dimovski and Ranko Lazić

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

**Abstract**

We present an approach for software model checking based on game semantics and CSP. Open program fragments are compositionally modelled as CSP processes which represent their game semantics. This translation is performed by a prototype compiler. Observational equivalence and verification of properties are checked by traces refinement using the FDR tool. The effectiveness of our approach is evaluated on several examples.

*Key words:* Software model checking, Game semantics, CSP process algebra, FDR refinement checker

## 1 Introduction

Model checking [6] 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 success of model checking has been mainly for hardware and communication protocols. Recently, model checking of software has become an active and important area of research and application (e.g. [5]). Unfortunately, applying model checking to software is complicated by several factors, ranging from the difficulty to model programs, due to the complexity of 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.

[1] We acknowledge support by the EPSRC (GR/S52759/01). The second author was also supported by the Intel Corporation.

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, since it has produced the first accurate, i.e. fully abstract and fully complete, models for a variety of programming languages and logical systems (e.g. [1]). Recently, it was shown that the fully abstract game-based model of second-order finitary Idealised Algol (i.e. with finite data types and without recursion) with iteration can be represented using only regular expressions [10], and that this can be used for efficient program analysis and verification [3].

Game semantics has several very suitable features for software model checking. It produces observationally fully abstract models with a maximum level of abstraction, since internal state changes during computation are abstracted away. It can compositionally model terms-in-context, i.e. open program fragments. This ability is essential in analysing properties of software components which contain undefined (free) variables and procedures.

We present an approach for compositional software modelling and verification, based on game semantics and the CSP process algebra (e.g. [15]). We focus on second-order finitary Idealised Algol with iteration. The game semantics of any term-in-context is represented as a CSP process. Observational equivalence between terms, and verification of regular properties, are decidable by checking traces refinement between CSP processes.

Compared with the regular expressions approach [10], this CSP based approach offers several benefits. Using the wide range of operators available in CSP, program terms are represented as processes in a clear and compositional way. The process which models a term is defined as a parallel composition of processes which model its subterms, with synchronisation events being hidden. Traces refinement between CSP processes can be automatically checked by the FDR tool [9], which is highly optimised for verification of parallel networks of processes. FDR also offers a range of compositional state-space reduction algorithms [14], enabling smaller models to be generated before or during refinement checking.

We have implemented a prototype compiler which, given any term-in-context, outputs a CSP process representing its game semantics. The output can be loaded into the ProBe tool for interactive exploration, or the FDR tool for automatic analysis. FDR also contains an interactive debugger, when the answer to a refinement query is negative.

Effectiveness of our approach is evaluated on several variants of two examples: a sorting algorithm, and an abstract data type implementation. Some of the experimental results show that, for minimal model generation, we outperform the approach based on regular expressions [3].

The paper is organised as follows. Sections 2 and 3 briefly recall the programming language, the CSP representation of its game semantics, and verification of observational equivalence. Further details can be found in the companion paper [7] which concentrates on theory, whereas this paper is devoted to applications and property verification. Section 4 shows how properties given in a linear temporal logic on finite traces, or as finite automata, can be verified. In Sections 5 and 6, we present the compiler and two case studies.

## 2 The programming language

The input to our compiler is a term-in-context of second-order finitary Idealised Algol with iteration. Idealised Algol [12] is a compact programming language, similar to Core ML, which combines the fundamental features of imperative languages with a full higher-order procedure mechanism.

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

$$\tau ::= int \mid bool$$

$$\sigma ::= exp[\tau] \mid comm \mid var[\tau]$$

$$\theta ::= \sigma \mid \sigma \times \sigma \times \cdots \sigma \to \sigma$$

Terms are introduced using type judgements of the form $\Gamma \vdash M : \theta$, where $\Gamma = \{\iota_1 : \theta_1, \cdots, \iota_k : \theta_k\}$.

Expression-type terms are built from constants and arithmetic/logic operators ($E_1 * E_2$). Command-type terms are built by the standard imperative operators: assignment ($V := E$), conditional ($\mathit{if}\ B\ \mathit{then}\ M_1\ \mathit{else}\ M_2$), while loop ($\mathit{while}\, B\, \mathit{do}\, C$), sequencing ($C \mathbin{;} M$, note that commands can be sequenced not only with commands but also with expressions or variables), no-op (*skip*), and a non-terminating command (*diverge*). There are also term formers for dereferencing variables ($!\,V$), application of first-order free identifiers to arguments ($\iota M_1 \cdots M_k$), local variable declaration ($new[\tau]\ \iota\ in\ C$), and function definition constructor ($let\ \iota\ (\iota_1 : \sigma_1, \ldots, \iota_k : \sigma_k) = N\ in\ M$). Full typing rules can be found in [7].

## 3 The CSP model

Here we recall the main properties of the CSP representation of game semantics of program terms. The details of this representation and some examples can be found in [7]. An introduction to game semantics is available in [2]. A comprehensive text on CSP is [15].

### *3.1 Interpretations of types and terms*

With each type $\theta$, we associate a set of possible events, the 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 $A$.

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

We interpret any typed term-in-context $\Gamma \vdash M : \sigma$ by a CSP process $[\![\Gamma \vdash M : \sigma]\!]^{CSP}$ whose set of terminated traces

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

is the set of all complete plays of the game strategy for the term. Events of this process are from the alphabet $\mathcal{A}_{\Gamma\vdash\sigma}$, defined as:

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

The compositional definition of the processes $[\![\Gamma \vdash M : \sigma]\!]^{CSP}$ is given in Appendix A.

### *3.2 Observational equivalence*

Two terms $M$ and $N$ are observationally equivalent, written $M \equiv_\sigma N$, if and only if for any context $C[-]$ such that both $C[M]$ and $C[N]$ are closed terms of type *comm*, $C[M]$ converges if and only if $C[N]$ converges. It was proved in [1] that this coincides to equality of sets of complete plays of strategies for $M$ and $N$, i.e. that the games model is fully abstract.

For the programming language fragment treated in this paper, it was shown in [7] that observational equivalence is captured by two traces refinements:

**Theorem 3.1 (Observational equivalence)** *We have:*

$$\begin{array}{rl}
\Gamma \vdash M \equiv_\sigma N \Leftrightarrow & [\![\Gamma \vdash M : \sigma]\!]^{CSP} \,\Box\, RUN_{\mathcal{A}_{\Gamma\vdash\sigma}} \;\sqsubseteq_T\; [\![\Gamma \vdash N : \sigma]\!]^{CSP} \wedge \\
& [\![\Gamma \vdash N : \sigma]\!]^{CSP} \,\Box\, RUN_{\mathcal{A}_{\Gamma\vdash\sigma}} \;\sqsubseteq_T\; [\![\Gamma \vdash M : \sigma]\!]^{CSP}
\end{array} \tag{1}$$

*where* $RUN_A = ?x : A \to RUN_A$. $\Box$

It is also proved in [7] that the process representing any term is finite state, and so observational equivalence is decidable using FDR.

## 4 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

$\Gamma \vdash M : \sigma$, the CSP process $[\![\Gamma \vdash M : \sigma]\!]^{CSP}$ has the property that its set of terminated traces

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

is the set of all complete plays of the strategy for $\Gamma \vdash M : \sigma$. We therefore focus on properties of finite traces, and take the view that $\Gamma \vdash M : \sigma$ satisfies such a property if and only if all traces in $\mathcal{L}_{CSP}(\Gamma \vdash M : \sigma)$ satisfy it.

### *4.1 Linear temporal logic*

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

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

We call this logic $\mathrm{LTL}_f^\Sigma$, because we give it semantics over finite traces (i.e. sequences) of elements of $\Sigma$. In particular, formulas $a$, $\bigcirc\phi$ and $\phi_1 \, U \, \phi_2$ require the trace to be non-empty. For any trace $t$ of length $k$, we write its elements as $t_1, \ldots, t_k$, and we write $t^i$ for its $i^{\text{th}}$ suffix $\langle t_i, \ldots, t_k\rangle$.

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

The boolean constant *false*, and boolean operators such as $\wedge$ and $\rightarrow$ can be defined as abbreviations. The same is true of the temporal operators 'eventually' and 'always':

$$\Diamond\phi = \mathit{true} \, U \, \phi \qquad \Box\phi = \neg\Diamond\neg\phi$$

Duals of the 'next-time' and 'until' operators are also useful:

$$\odot\,\phi = \neg\bigcirc\neg\phi \qquad \phi_1 \, V \, \phi_2 = \neg(\neg\phi_1 \, U \, \neg\phi_2)$$

Their semantics is as follows:

$$\begin{array}{lll}
t \models \odot\,\phi & \text{iff} & t = \langle\rangle \text{ or } t^2 \models \phi \\
t \models \phi_1 \, V \, \phi_2 & \text{iff} & \text{either for all } i \in \{1, \ldots, \mid t \mid +1\},\ t^i \models \phi_2, \\
 & & \text{or there exists } i \in \{1, \ldots, \mid t \mid +1\} \text{ such that } t^i \models \phi_1 \\
 & & \quad \text{and for all } j \in \{1, \ldots, i\},\ t^j \models \phi_2
\end{array}$$

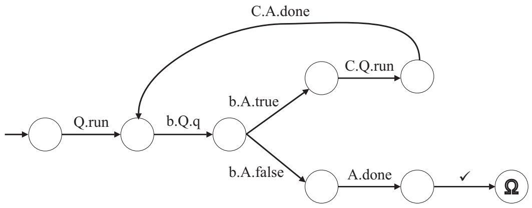


Fig. 1. Model of *while b do C*

A term $\Gamma \vdash M : \sigma$ satisfies a formula $\phi$ of $\mathrm{LTL}_f^{\mathcal{A}_{\Gamma\vdash\sigma}}$ if and only if for every trace $t \in \mathcal{L}_{CSP}(\Gamma \vdash M : \sigma)$, $t \models \phi$.

**Example 4.1** Consider the following term:

$$b : bool,\ C : comm\ \vdash\ while\ b\ do\ C : comm$$

The transition system of the CSP process for this term is shown in Figure 1. Since both $b$ and $C$ are free identifiers, the successfully terminated traces of this process are all those corresponding to executing $C$ zero or more times while the value of $b$ is true, and finishing when the value of $b$ is false.

Therefore, this term satisfies $\Diamond b.A.false$, but does not satisfy $\Diamond C.Q.run$. It also satisfies $\Box(b.A.true \rightarrow \Diamond C.Q.run)$, but does not satisfy $\Box(b.Q.q \rightarrow \Diamond C.Q.run)$.

There is an algorithm which, given any formula $\phi$ of $\mathrm{LTL}_f^{\Sigma}$, constructs:

- a CSP process $P_\phi^\Sigma$ such that $t \models \phi$ if and only if $t^\frown\langle\checkmark\rangle \in traces(P_\phi^\Sigma)$;
- a finite transition system which has the same finite traces as $P_\phi^\Sigma$.

This is similar to constructing a finite automaton which accepts a finite trace $t$ if and only if $t \models \phi$. The details can be found in Appendix B.

We therefore have a decision procedure which, given a program term $\Gamma \vdash M : \sigma$ and a formula $\phi$ of $\mathrm{LTL}_f^{\mathcal{A}_{\Gamma\vdash\sigma}}$, checks satisfaction. It works by constructing the CSP processes $[\![\Gamma \vdash M : \sigma]\!]^{CSP}$ and $P_\phi^{\mathcal{A}_{\Gamma\vdash\sigma}}$, and checking the traces refinement [2]

$$P_\phi^{\mathcal{A}_{\Gamma\vdash\sigma}} \Box\, RUN_{\mathcal{A}_{\Gamma\vdash\sigma}} \ \sqsubseteq_T\ [\![\Gamma \vdash M : \sigma]\!]^{CSP} \tag{2}$$

### *4.2 Finite automata*

More generally, any property such that the set of all finite traces which satisfy it is regular, can be represented in CSP. Suppose $A$ is an automaton with finite alphabet $\Sigma$, 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

[2] If $\phi$ contains the $U$ operator, the standard FDR procedure for constructing a transition system for $P_\phi^{\mathcal{A}_{\Gamma\vdash\sigma}}$ may not terminate. In this case, the traces refinement above can be checked by FDR either by representing as a CSP process the finite transition system for $P_\phi^{\mathcal{A}_{\Gamma\vdash\sigma}}$ which the algorithm in Appendix B produces, or by the approach in Section 4.2.

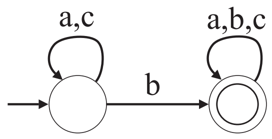


Fig. 2. A finite automaton

define

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

For any $q \in F$, we define

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

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

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

**Example 4.2** Consider the automaton $A$ in Figure 2, whose alphabet is $\{a, b, c\}$. It accepts a finite trace $t$ if and only if $t \models \Diamond b$.

The CSP process $P_A$ is defined as:

$$\begin{aligned}
&P_A = P_1\\
&P_1 = (a \to P_1) \Box (b \to P_2) \Box (c \to P_1)\\
&P_2 = (a \to P_2) \Box (b \to P_2) \Box (c \to P_2) \Box\ \mathit{SKIP}
\end{aligned}$$

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

$$P_A \Box\ \mathit{RUN}_{\mathcal{A}_{\Gamma\vdash\sigma}} \ \sqsubseteq_T\ [\![\Gamma \vdash M : \sigma]\!]^{\mathit{CSP}} \tag{3}$$

This also provides another way of deciding satisfaction of a formula $\phi$ of $\mathrm{LTL}_f^{\mathcal{A}_{\Gamma\vdash\sigma}}$: construct a finite automaton $A$ which accepts a finite trace $t$ if and only if $t \models \phi$, and then check the traces refinement above.

## 5 Compiler

We have implemented a compiler in Java [4], which automatically converts a term-in-context (i.e. an open program fragment) into a CSP process which represents its game semantics. The input is code, with simple type annotations to indicate sizes of finite integer data types. The resulting CSP process is defined by a script in machine-readable CSP [15] which the compiler outputs.

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 [9]. 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 (1), satisfaction of a linear temporal logic formula (2), and containment in a regular language (3).

FDR offers a number of hierarchical compression algorithms, which can be applied during either model generation or refinement checking. The scripts which our compiler produces normally contain instructions to apply diamond elimination 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 the subterm which is its scope. These interactions become silent (i.e. $\tau$) transitions, enabling the model to be reduced.

Instead of the infinite data type of all integers, the programming language fragment considered in this paper has finite data types. We work with sets of integers modulo $k$. Each integer variable is declared to be of type $int\%_k$ for some $k$, and different values of $k$ can be used for different variables. Operations between variables with distinct integer types $int\%_{k_1}$ and $int\%_{k_2}$ are interpreted as modulo the minimum of $k_1$ and $k_2$.

## 6 Applications

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

### *6.1 A sorting algorithm*

In this section we analyse the bubble-sort algorithm, whose implementation is given in Figure 3. 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 $int\%_3$, which consists of values 0, 1 and 2. The type of index $i$ is $int\%_{n+1}$, i.e. one more than the size of the array.

The implementation of bubble-sort is standard. 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. A transition system of the model CSP process for $n = 2$ is shown in Figure 4. 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 1 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 5. We list the execution time in minutes, the size of the

```
x[n] : var int%3 ⊢
new int%3 a[n] in
new int%n+1 i in
while (i < n)  {a[i] := x[i];   i := i + 1; }
new bool 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. 3. Implementation of bubble sort

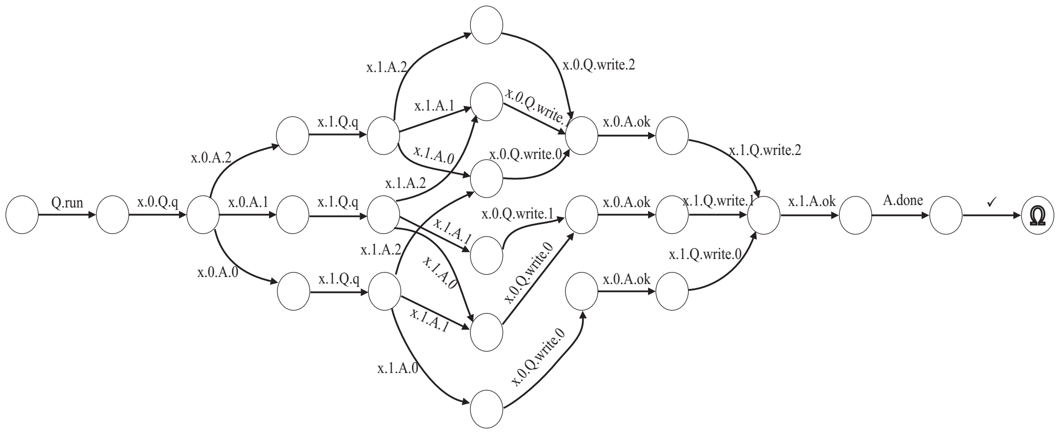


Fig. 4. Transition system of the model for $n = 2$

largest generated transition system, and the size of the final transition system. We ran FDR on a Research Machines 2.5 GHz Xeon with 2GB RAM. The results from the tool based on regular expressions were obtained on a Sun-Blade 100 with 2GB RAM [3]. The results confirm that the two approaches give isomorphic models, where the CSP models have an extra state due to representing termination by a ✓ event.

Further information about minimal model generation for $n = 20$ is shown in Figure 5. FDR first produces a transition system for the subprogram which is the scope of the declaration of the local array $a[\,]$. Each component of $a[\,]$ has an index from 0 to 19 and is represented by a process which keeps its value, and performs reads and writes to that component. FDR obtains the final model by

Table 1
Experimental results for bubble-sort minimal model generation

| $n$ | CSP | | | Regular expressions | | |
|---|---|---|---|---|---|---|
| | Time (m) | Max. st. | Model st. | Time (m) | Max. st. | Model st. |
| 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 | | |

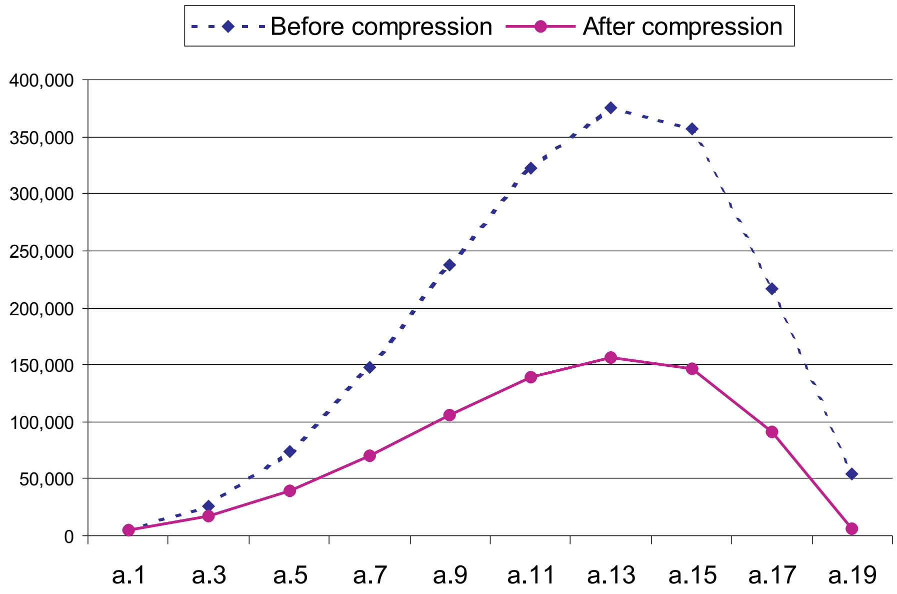


Fig. 5. Effects of compressions for bubble sort with $n = 20$

taking the transition system for the scope of $a[\,]$ and composing it (by parallel composition and hiding) 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 expect FDR to perform even better in property verification. For checking refinement by a composite process, FDR does not need to generate an explicit 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.

### *6.2 An abstract data type implementation*

Figure 6 contains an implementation of a queue of maximum size $n$ as a circular array. There are four free identifiers: commands *empty*() and *overflow*(), expression $p$ of integers type modulo $m$, and command *ANALYSE* which

$empty() : comm,$
$overflow() : comm,$
$p : exp\ int\%m,$
$ANALYSE(comm, exp\ int\%m)\ :\ comm\ \vdash$
$new\ int\%m\ buffer[n]\ in$
$new\ int\%n\ front\ in$
$new\ int\%n\ tail\ in$
$new\ int\%n{+}1\ queue\_size\ in$
$let\ exp\ bool\ isempty()\ \{return\ queue\_size == 0;\ \}\ in$
$let\ exp\ bool\ isfull()\ \{return\ queue\_size == n;\ \}\ in$
$let\ comm\ add(exp\ int\%m\ x)\ \{$
$\quad if\ (isfull())\ overflow();$
$\quad else\ \{$
$\quad\quad buffer[tail] := x;$
$\quad\quad tail := tail + 1;$
$\quad\quad queue\_size := queue\_size + 1;\ \}$
$\}\ in$
$let\ exp\ int\%m\ next()\ \{$
$\quad if\ (isempty())\ \{$
$\quad\quad empty();$
$\quad\quad return\ 0;\ \}$
$\quad else\ \{$
$\quad\quad front := front + 1;$
$\quad\quad queue\_size := queue\_size - 1;$
$\quad\quad return\ buffer[front - 1];\ \}$
$\}\ in$
$ANALYSE(add(p), next())$
$: comm$

Fig. 6. A queue implementation

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 therefore 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 queue implementation uses the non-local commands $empty()$ and $overflow()$ for handling calls to *next* on the empty queue, respectively *add* on a full queue.

A transition system of the model CSP process for a data set of size $m = 1$ and maximum queue size $n = 2$ is shown in Figure 7. For clarity, labels *ADD* and *NEXT* are used instead of *ANALYSE*.1 and *ANALYSE*.2.

Table 2 contains results for minimal model generation for a data set of size $m = 3$ and several maximum queue sizes. For maximum size $n = 10$, further information is displayed in Figure 8. Similar to the corresponding figure for the bubble-sort example, we show numbers of states before and

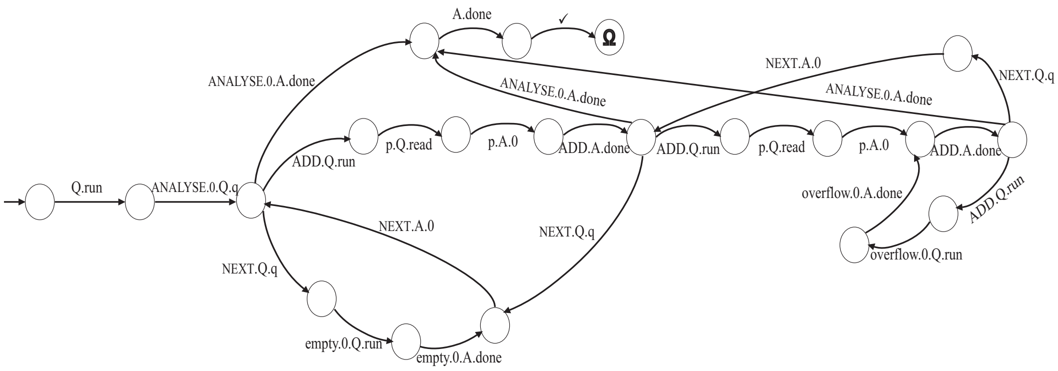


Fig. 7. Transition system of the model for $n = 2$ and $m = 1$

Table 2
Experimental results for queue minimal model generation with $m = 3$

| $n$ | Time | Model states | Max. states |
|---|---|---|---|
| 2 | 5 sec | 70 | 237 |
| 5 | 18 sec | 1 825 | 12 351 |
| 10 | 80 min | 442 870 | 5 225 757 |

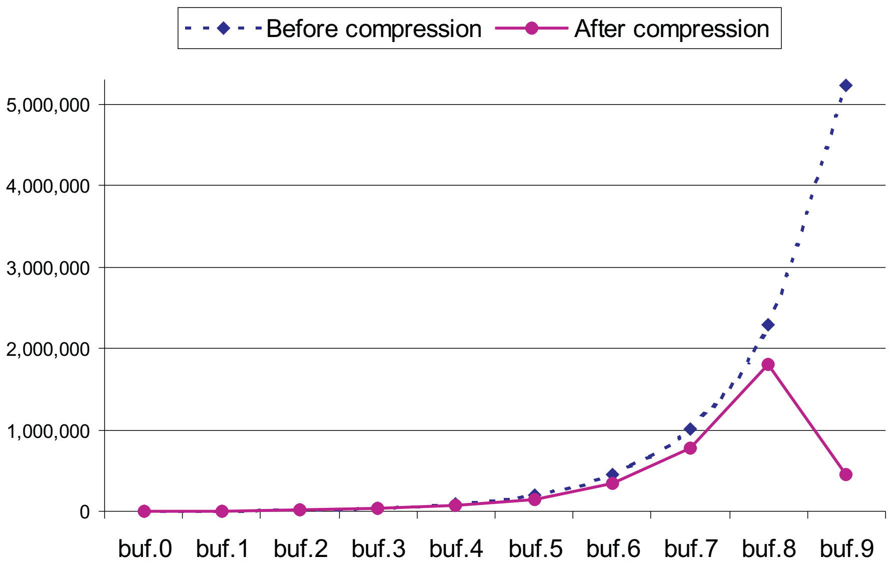


Fig. 8. Effects of compressions for queue implementation with $n = 10$

after compression, when transition systems for components of the local array *buffer*[ ] are composed in turn with the transition system for the rest of the program.

Using the techniques in Section 4, we can check a range of properties of the queue implementation. For example, linear temporal logic formulas $\neg\Diamond\mathit{overflow}$ and $\neg\Diamond\mathit{empty}$ assert that the non-local commands handling

Table 3
Experimental results for verification of properties (s=sec, m=min)

| $n$ | $\neg\Diamond \mathit{empty}$ | | $\neg\Diamond \mathit{overflow}$ | |
|---|---|---|---|---|
| | not min. model | min. model | not min. model | min. model |
| 10 | 67s+46s+0s | 67s+80m+0s | 50s+46s+97s | 50s+80m+0s |
| 12 | 103s+61s+0s | failed | 77s+61s+18m | failed |
| 13 | 127s+70s+0s | failed | 93s+70s+58m | failed |

writes to full queues and reads from empty queues are never called. As expected, checking them returns that they are not satisfied. Counter-example traces which the FDR debugger gives correspond to $n+1$ consecutive calls of *ADD* after which *overflow* is called, and a single *NEXT* call after which *empty* is called.

Specifically, the CSP process $P^{\Sigma}_{\neg\Diamond \mathit{empty}}$, where $\Sigma$ is the alphabet of events for the queue implementation term, is equivalent to

$$\mu\, p \cdot (?w : \Sigma \backslash \{\!|\ \mathit{empty}\ |\!\} \to p) \;\Box\; \mathit{SKIP}$$

Checking the traces refinement (2) on FDR produces the following counter-example trace:

| | | | |
|---|---|---|---|
| 1 | $Q.run$ | 6 | $\mathit{NEXT}.A.0$ |
| 2 | $\mathit{ANALYSE}.0.Q.run$ | 7 | $\mathit{ANALYSE}.0.A.done$ |
| 3 | $\mathit{NEXT}.Q.q$ | 8 | $A.done$ |
| 4 | $\mathit{empty}.0.run$ | 9 | $\checkmark$ |
| 5 | $\mathit{empty}.0.done$ | | |

Table 3 shows some experimental results for checking the two formulas. We compared refinement checking without first generating a minimal model against refinement checking preceded by minimal model generation. In the former case, compressions are not applied when components of the array *buffer* are composed with the rest of the program. Instead, a composite model is generated on-the-fly during refinement checking. This enabled us to check the two properties up to maximum queue size $n = 13$, whereas minimal model generation did not succeed for $n > 10$. The times shown are sums of times which FDR took to process the specification, to process the implementation, and to perform the refinement check.

In addition to checking properties of external behaviours of given terms, we can check assertions which refer to internal data. Assertions can be added to a term using a local function *assert* whose argument is a boolean expression. If the argument is true, the *assert* function does nothing, but otherwise it calls a non-local function *error*. In the game semantics of the augmented term, any occurrence of events $\mathit{error}.0.run$ and $\mathit{error}.0.done$ represents an assertion violation.

For example, we can check that whenever a value $y$ is added to the queue, and then all items except one are removed from the queue, the remaining item has value $y$. We do this by replacing the call of the *ANALYSE* function in

Figure 6 by the following code:

```
let comm assert(exp bool b) {
     if b then skip;
      else error();
} in
let exp bool validate() {
     new int%n y in
     y = p;
     add(y);
     while (queue_size > 1)
          next();
     return (next() == y);
} in
ANALYSE(add(p), next(), assert(validate()))
: comm
```

where $error()$ and $ANALYSE(comm, int\%n, comm)$ are non-local commands.

We then check whether this modified queue implementation satisfies the property $\neg\Diamond error$. We performed this on FDR for a data set of size $m = 2$ and for maximum queue size $n = 2$. As expected, a counter-example trace was produced, showing that this particular assertion is violated after $n$ calls of $ADD$, i.e. when the queue is full. When the assertion is corrected so that it applies only to non-full queues, the check succeeds.

## 7 Conclusion

In this paper, we extended the approach to software model checking proposed in [7] to checking properties given as formulas of linear temporal logic or as finite automata, and we evaluated it on two kinds of examples: a sorting algorithm, and an abstract data type implementation.

The experimental results show that open program fragments with large internal state spaces can be verified, partly due to efficiency of the FDR tool for on-the-fly checking of parallel networks of processes. They also show that this approach can outperform the approach based on regular expressions [3].

As future work, we intend to extend the compiler so that parameterised program 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. [8,13]) or by algorithms based on data independence [11].

## 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". Birkhäuser, 1997.

[2] S.Abramsky, *Algorithmic game semantics: A tutorial introduction*, Lecture notes, Marktoberdorf International Summer School, 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**, 421–435, 2004.

[4] A.W.Appel and J.Palsberg, *Modern Compiler Implementation in Java*, 2nd edition. Cambridge University Press, 2002.

[5] 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.

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

[7] A.Dimovski and R.Lazić, *CSP Representation of Game Semantics for Second-order Idealized Algol.* In Proceedings of ICFEM, LNCS, November 2004.

[8] A.Farias, A.Mota and A.Sampaio, *Efficient CSP-Z Data Abstraction.* In Proceedings of IFM, LNCS **2999**, 108–127, April 2004.

[9] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 2000.

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

[11] R.Lazić, *A Semantic Study of Data Independence with Applications to Model Checking.* DPhil thesis, Computing Laboratory, Oxford University, 1999.

[12] J.C.Reynolds, *The essence of Algol.* In Proceedings of ISAL, 345–372, Amsterdam, Holland, 1981.

[13] M.Roggenbach, *CSP-CASL — A new Integration of Process Algebra and Algebraic Specification.* In Proceedings of AMiLP, TWLT **21**, 2003.

[14] 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**, 133–152, 1995.

[15] A.W.Roscoe, *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

# A Processes for terms

*Expression constructs*

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

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

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

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

*Command constructs*

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

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

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

$$
\begin{aligned}
&[\![\Gamma \vdash \mathit{if}\ B\ \mathit{then}\ M_1\ \mathit{else}\ M_2 : \sigma]\!]^{CSP} = \\
&\quad [\![\Gamma \vdash B : \mathit{exp}[\mathit{bool}]]\!]^{CSP}[Q_0/Q, A_0/A] \underset{\{\mid Q_0, A_0 \mid\}}{\parallel} \\
&\quad \Big( ([\![\Gamma \vdash M_1 : \sigma]\!]^{CSP}[Q_1/Q, A_1/A] \,\square\, \mathit{SKIP}) \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\
&\quad (([\![\Gamma \vdash M_2 : \sigma]\!]^{CSP}[Q_2/Q, A_2/A] \,\square\, \mathit{SKIP}) \underset{\{\mid Q_2, A_2 \mid\}}{\parallel} \\
&\quad (Q?\mathbf{q} : \mathbb{Q}_\sigma \to Q_0.q \to A_0?v : \mathcal{A}_{bool} \to (Q_1.\mathbf{q} \to A_1?\mathbf{a} : \mathbb{A}_\sigma^{\mathbf{q}} \to A.\mathbf{a} \to \mathit{SKIP} \\
&\qquad \not\leqslant v \not\geqslant Q_2.\mathbf{q} \to A_2?\mathbf{a} : \mathbb{A}_\sigma^{\mathbf{q}} \to A.\mathbf{a} \to \mathit{SKIP})) \\
&\qquad\qquad\qquad\qquad \backslash \{\mid Q_2, A_2 \mid\}) \backslash \{\mid Q_1, A_1 \mid\} \Big) \backslash \{\mid Q_0, A_0 \mid\}
\end{aligned}
$$

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

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

*Variable constructs*

$$
\begin{aligned}
&[\![\Gamma \vdash \iota : \mathit{var}[\tau]]\!]^{CSP} = \\
&\qquad (Q.\mathit{read} \to \iota.Q.\mathit{read} \to \iota.A?v : \mathcal{A}_\tau \to A.v \to \mathit{SKIP}) \,\square \\
&\qquad (Q.\mathit{write}?v : \mathcal{A}_\tau \to \iota.Q.\mathit{write}.v \to \iota.A.\mathit{ok} \to A.\mathit{ok} \to \mathit{SKIP})
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma \vdash V := M : \mathit{comm}]\!]^{CSP} = \\
&\qquad [\![\Gamma \vdash M : \mathit{exp}[\tau]]\!]^{CSP}[Q_1/Q, A_1/A] \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\
&\qquad ([\![\Gamma \vdash V : \mathit{var}[\tau]]\!]^{CSP}[Q_2/Q, A_2/A] \underset{\{\mid Q_2, A_2 \mid\}}{\parallel} \\
&\qquad (Q.\mathit{run} \to Q_1.q \to A_1?v : \mathcal{A}_\tau \to Q_2.\mathit{write}.v \to A_2.\mathit{ok} \\
&\qquad\quad \to A.\mathit{done} \to \mathit{SKIP}) \ \backslash\ \{\mid Q_2, A_2 \mid\}) \ \backslash\ \{\mid Q_1, A_1 \mid\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma \vdash !V : \mathit{exp}[\tau]]\!]^{CSP} = \\
&\qquad [\![\Gamma \vdash V : \mathit{var}[\tau]]\!]^{CSP}[Q_1/Q, A_1/A] \underset{\{\mid Q_1, A_1 \mid\}}{\parallel} \\
&\qquad (Q.q \to Q_1.\mathit{read} \to A_1?v : \mathcal{A}_\tau \to A.v \to \mathit{SKIP}) \ \backslash\ \{\mid Q_1, A_1 \mid\}
\end{aligned}
$$

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

where $a_{int} = 0$, $a_{bool} = \mathit{false}$, and:

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

*Application and functions*

$$\begin{aligned} &[\![\Gamma \vdash \iota(M_1 \ldots M_k) : \sigma]\!]^{CSP} = Q?\mathbf{q} : \mathbb{Q}_\sigma \to \iota.0.Q.\mathbf{q} \to \\ &\quad \Big( \mu \, L.( \; \Box_{j=1}^{k} ([\![\Gamma \vdash M_j : \sigma_j]\!]^{CSP} \underset{\{| \; Q, A |\}}{\parallel} \iota.j.Q?\mathbf{q} : \mathbb{Q}_{\sigma_j} \to Q.\mathbf{q} \to A?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_{\sigma_j} \\ &\quad \to \iota.j.A.a \to SKIP) \setminus \{| \; Q, A \; |\} \;;\; L) \, \Box \, SKIP \Big) \;;\; \iota.0.A?\mathbf{a} : \mathbb{A}^{\mathbf{q}}_{\sigma} \to A.\mathbf{a} \to SKIP \end{aligned}$$

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

## B Processes for formulas

We show how, given any formula $\phi$ of $\mathrm{LTL}^{\Sigma}_{f}$, to construct:

- a CSP process $P^{\Sigma}_{\phi}$ such that $t \models \phi$ if and only if $t \frown \langle \checkmark \rangle \in traces(P^{\Sigma}_{\phi})$;
- a finite transition system which has the same finite traces as $P^{\Sigma}_{\phi}$.

Consider the following variant of $\mathrm{LTL}^{\Sigma}_{f}$, where only atoms can be negated, and operators $\wedge$, $\odot$ and $V$ are basic rather than derived. We call it $\mathrm{LTL}+^{\Sigma}_{f}$.

$$\phi \;::=\; true \mid \mathit{false} \mid a \mid \neg a \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \odot \, \phi \mid \phi_1 \, U \, \phi_2 \mid \phi_1 \, V \, \phi_2$$

Any formula $\phi$ of $\mathrm{LTL}^{\Sigma}_{f}$ can be transformed into an equivalent formula $\phi'$ of $\mathrm{LTL}+^{\Sigma}_{f}$, since $\wedge$, $\odot$ and $V$ are duals of $\vee$, $\bigcirc$ and $U$. The size of $\phi'$ is linear in the size of $\phi$.

For formulas $\phi$ of $\mathrm{LTL}+^{\Sigma}_{f}$, we define CSP processes $P^{\Sigma}_{\phi}$ as follows, by structural recursion on $\phi$:

$$\begin{aligned} P^{\Sigma}_{true} &= RUN^{\checkmark}_{\Sigma} \\ P^{\Sigma}_{\mathit{false}} &= STOP \\ P^{\Sigma}_{a} &= a \to RUN^{\checkmark}_{\Sigma} \\ P^{\Sigma}_{\neg a} &= (?w : \Sigma \setminus \{a\} \to RUN^{\checkmark}_{\Sigma} \,) \, \Box \, SKIP \end{aligned}$$

$$
\begin{aligned}
P^{\Sigma}_{\phi_1 \vee \phi_2} &= P^{\Sigma}_{\phi_1} \,\Box\, P^{\Sigma}_{\phi_2} \\
P^{\Sigma}_{\phi_1 \wedge \phi_2} &= P^{\Sigma}_{\phi_1} \mathop{\|}_{\Sigma} P^{\Sigma}_{\phi_2} \\
P^{\Sigma}_{\bigcirc \phi} &= ?w : \Sigma \rightarrow P^{\Sigma}_{\phi} \\
P^{\Sigma}_{\odot \phi} &= (?w : \Sigma \rightarrow P^{\Sigma}_{\phi}) \,\Box\, \mathit{SKIP} \\
P^{\Sigma}_{\phi_1 \, U \, \phi_2} &= \mu\, p \cdot P^{\Sigma}_{\phi_2} \,\Box\, (P^{\Sigma}_{\phi_1} \mathop{\|}_{\Sigma} (?w : \Sigma \rightarrow p)) \\
P^{\Sigma}_{\phi_1 \, V \, \phi_2} &= \mu\, p \cdot (P^{\Sigma}_{\phi_2} \mathop{\|}_{\Sigma} P^{\Sigma}_{\phi_1}) \,\Box\, (P^{\Sigma}_{\phi_2} \mathop{\|}_{\Sigma} ((?w : \Sigma \rightarrow p) \,\Box\, \mathit{SKIP}))
\end{aligned}
$$

where $\mathit{RUN}^{\checkmark}_{\Sigma} = (?w : \Sigma \rightarrow \mathit{RUN}^{\checkmark}_{\Sigma}) \,\Box\, \mathit{SKIP}$.

Now, for any formula $\phi$ of $\mathrm{LTL}^{\Sigma}_{f}$, we define $P^{\Sigma}_{\phi} = P^{\Sigma}_{\phi'}$, where $\phi'$ is the equivalent formula of $\mathrm{LTL}+^{\Sigma}_{f}$.

**Proposition B.1** *For any formula $\phi$ of $LTL^{\Sigma}_{f}$, we have that $t \models \phi$ if and only if $t^\frown\langle\checkmark\rangle \in traces(P^{\Sigma}_{\phi})$. A finite transition system which has the same finite traces as $P^{\Sigma}_{\phi}$ can be constructed.*

**Proof.** It suffices to prove the proposition for formulas $\phi$ of $\mathrm{LTL}+^{\Sigma}_{f}$. The proof is by structural induction on $\phi$.

□