Compositional Model Extraction for Higher-Order Concurrent Programs

D.R. Ghica[1] and A.S. Murawski[2,⋆]

[1] School of Computer Science, Univ. of Birmingham, Birmingham B15 2TT, UK
[2] Oxford University Computing Laboratory, Oxford OX1 3QD, UK

Abstract. The extraction of accurate finite-state models of higher-order or open programs is a difficult problem. We show how it can be addressed using newly developed game-semantic techniques and illustrate the solution with a model-checking tool based on such techniques. The approach has several important advantages over more traditional ones: precise account of inter-procedural behaviour, concise procedure summaries and economical extracted models.

1 Introduction and Background

Automated verification of software systems is one of the most urgent problems in computer science. This hardly needs to be argued for, as we are exposed to a world increasingly dominated by software. The theoretical and practical difficulty of the problem is well known. In general, the problem is undecidable but, even subject to simplifying assumptions and approximation techniques which make it decidable, the complexity poses a substantial challenge. Nevertheless, theoretical developments combined with an increase in available computational power give grounds for optimism, and automated verification of software systems is becoming increasingly feasible, to the point that it is about to become a meaningful part of industrial software development [1].

The most effective methods of automated software verification turn out to be based on *model checking* (MC) [2], in particular on *finite-state* model checking. A software system is represented as (or approximated by) a finite-state machine (FSM) and its interesting properties are expressed as *temporal* properties of the FSM. The challenges that need to be tackled include efficient extraction of models and automatic derivation of smaller but safe approximations. Some of the most advanced MC frameworks available centre around these issues [3, 4, 5, 6].

For programming languages with procedures, especially higher-order procedures, the extraction of an FSM representation or approximation is especially difficult because one needs to account for the often subtle interaction between procedures and other computational features such as state, concurrency or control. We can illustrate this point with a very simple example. Consider the following (second-order) procedure `p` taking as argument procedure `c`:

```
int p(void c(int d)) { int x=1; c(2); return x }.
```

⋆ Supported by the UK EPSRC (GR/R88861/01) and St John's College, Oxford.

H. Hermanns and J. Palsberg (Eds.): TACAS 2006, LNCS 3920, pp. 303–317, 2006.

In virtually any programming language if `p` returns a value then that value will be 1. There should be no way that the non-local procedure `c`, taken as an argument, can modify the value of the locally-scoped variable `x`. However, producing a FSM representation of this procedure, which makes it obvious that `x` and `c` cannot interact, turns out quite difficult. The same issues arise in modeling *open* programs, i.e. programs with procedures which are not defined locally. In both cases the obstacle is that operational techniques, which are ordinarily employed for model extraction, only apply for closed, ground-type terms.

Dealing with issues pertaining to *inter-procedural* interactions such as the one illustrated above is the subject of numerous lines of research: data-flow analysis, control-flow analysis, effect analysis, locality analysis and so on. Many of these analyses are syntactic or operational, and it is quite awkward to integrate them into a MC framework. However, they are essential in modeling and verifying higher-order programs. The problem of model extraction is particularly difficult in the presence of concurrency, because the naive model based on interleaved execution is very computationally expensive.

The research programme we are pursuing proposes a new kind of analysis, called *algorithmic game semantics*, which subsumes inter-procedural analysis and is compatible with FSM representation and model-checking. This analysis focuses on finding concrete representations for game-semantic models (or *game models*, for brevity) of programming languages. Having a *semantics-directed* approach to model construction has several important advantages:

Consistency. A semantics-directed approach provides a unified framework which encompasses and supersedes the techniques mentioned earlier in a uniform, substantially simplified fashion.

Correctness. The model extraction is correct (and in fact complete) by construction, relative to a specified notion of *observation*. In principle, any inter-procedural analysis compatible with the specified notion of observation can be derived from the semantics. For example, we will see that game models immediately validate the earlier observation about the interaction between local state and non-local procedures.

Concreteness. We can construct a concrete FSM representation of the behaviour of a higher-order program, which is independent of the syntax. Once the model is constructed we can apply standard model-checking methods to verify its properties efficiently.

Compositionality. Models are constructed *inductively on the structure of the program*, i.e. the model of a program P is constructed out of the models of its subprograms P_i. Most importantly, in constructing the models for subprograms P_i's we need not know the larger context in which they will be used. The beneficial consequences of a compositional method are:

- an ability to model and verify *open* programs, i.e. programs which must function in an unknown environment (for example *libraries*);
- the possibility to break up a larger system in smaller systems which can be modeled and verified independently (*scalability*);
- modeling procedures independently and incorporating their models efficiently into the model of a larger program (*procedure summaries*) [7].

Code-level specification. Program properties are described at *code-level* using assertions, rather than at *model-level* using temporal logics.

Note that a semantics-directed approach to model extraction was not feasible using the traditional styles of semantics extant before the introduction of game semantics, i.e. *operational* and *denotational*. Operational semantics is concrete enough, but has virtually no meta-theory, is not compositional and cannot model open programs. Denotational semantics, on the other hand, meets these requirements but is abstract and essentially non-finitary. Game semantics seems to combine the advantages of the two in a way that is particularly promising for automated verification.

Game semantics was introduced in order to tackle the long-standing *full abstraction* problem for the functional language PCF [8, 9]. The framework proved to be very useful for constructing semantics for a variety of programming languages involving diverse computational features such as state [10], control [11], concurrency [12] and more. The first steps in the direction of application of game semantics for program analysis were taken by Hankin and Malacaria [13]. The first application to model checking was proposed by Ghica and McCusker [14], and further developed by Ghica [15]. A model-checker based on these ideas was implemented in [16] with very positive results: it illustrates the ability to model open second-order programs by verifying invariants of abstract data type implementations (ADT) and it shows how the compositionality of the model construction allows the modeling of data-intensive programs such as sorting programs.

Contribution

The model checking technique described in [16] is for a *second-order sequential procedural language*. In this paper we substantially expand the expressivity of the programming language we model, by adding *higher-order procedures, shared-variable concurrency and semaphores*. The immediately relevant theoretical developments which led to this new model checking technique are a game model for shared-variable concurrency [12] and a type system used to identify decidable terms in the language [19].

Note that in this paper we focus almost exclusively on the problem of model extraction and representation. In order to tackle the other standard problems of MC (specification and efficient verification) we rely on the commercially-available model-checker FDR [17].

2 The Language SCC

We consider a higher-order call-by-name procedural language with parallel composition and binary semaphores. Its types are generated by the grammar given below

$$\beta ::= \mathsf{com} \mid \mathsf{int} \mid \mathsf{var} \mid \mathsf{sem} \qquad \theta ::= \beta \mid \theta \to \theta,$$

where com is the type of commands, int is a *finite* data-type of expressions which can take values from the set $\{0, \cdots, max\}$ ($max > 0$), var is the type of variables holding values from $\{0, \ldots, max\}$ and sem is the type of binary semaphores. The syntax of the language is defined by the standard λ-calculus rules ($\lambda x.M, MN$) augmented with rules for arithmetic, branching, iteration ($\mathsf{while}\, M\, \mathsf{do}\, N$), variable manipulation (assignment $M{:=}N$, dereferencing $!M$, variable definition with initialisation $\mathsf{newvar}\, X{:=}i\, \mathsf{in}\, M$), parallel composition ($M_1 \,||\, M_2$) and binary semaphore manipulation ($\mathsf{grb}(S)$, $\mathsf{release}(S)$, semaphore definition with initialisation $\mathsf{newsem}\, S{:=}i\, \mathsf{in}\, M$, where $S{:=}0$ means that the semaphore is released initially).

The semantics of the language is defined using a (small-step) transition relation $\Sigma \vdash M,\, s \longrightarrow M',\, s'$. Σ is a set of names of variables denoting *memory cells* and names of semaphores denoting *locks*; s, s' are states, i.e. functions $s, s' : \Sigma \to \mathbb{N}$, and M, M' are terms.

We say that a term M *may* terminate from state s, written $M, s \Downarrow$, if there exists a terminating evaluation at start state s: $\exists s',\ M, s \longrightarrow^* c, s'$, with $c \in \{0, \cdots, max, \mathsf{skip}\}$. If M is closed and $M, \emptyset \Downarrow$ we write $M \Downarrow$. We consider the program *approximation* and *equivalence* relations induced by this angelic notion of termination. They are defined contextually as follows. Two terms $\Gamma \vdash M_1$ and $\Gamma \vdash M_2$ are deemed *may-equivalent* (written $\Gamma \vdash M_1 \cong M_2$) iff $\forall \mathcal{C}[-] : \mathsf{com}, \mathcal{C}[M_1] \Downarrow$ if and only if $\mathcal{C}[M_2] \Downarrow$, where $\mathcal{C}[M_i]$ are closed programs of type com. The corresponding notion of program approximation is defined by: $\Gamma \vdash M_1 \lesssim M_2$ iff $\forall \mathcal{C}[-] : \mathsf{com}, \mathcal{C}[M_1] \Downarrow$ implies $\mathcal{C}[M_2] \Downarrow$ (where as before $\mathcal{C}[M_i]$ are closed programs of com type). Note that the two notions apply to terms with free identifiers (open terms) and are defined with respect to all possible uses (instantiations of the free identifiers).

Although we consider finite data-types and iteration rather than general recursion, it turns out that both $\cong$ and $\lesssim$ are undecidable even for terms with free identifiers of first order. Indeed, in [19] we show that, unlike in the sequential case, it is impossible to decide the equivalence or approximation of terms of the shape $p : \mathsf{com} \to \mathsf{com} \vdash M : \mathsf{com}$. The reason is that functions of type $\mathsf{com} \to \mathsf{com}$ can use their argument in any number of concurrently running threads, which is powerful enough for encoding the halting problem for counter machines as an equivalence query. In order to recover decidability one needs to weaken the meaning of free identifiers and impose bounds on the number of concurrent threads of execution.

To formalise this sort of constraint specification we introduced a new type system, called *Syntactic Control of Concurrency* (SCC) [19]. Types of that system are the same as before except that they are annotated with numeric bounds. Thus an SCC typing judgment has the shape $x_1{:}\theta_1^{n_1}, \ldots, x_k{:}\theta_k^{n_k} \vdash_r M : \theta$ where θ is generated by the grammar $\theta ::= \beta \,|\, \theta^n \to \theta$, $n \in \mathbb{N}$. The numeric bounds concern the number of concurrent threads of execution that can arise during various stages of computation.

The key rules are the four rules below. Parallel composition and application increase the degree of concurrency, whereas sequential composition (and its

iterated form, the **while** loop) does not affect the bounds in any way. Technically this is achieved by using disjoint contexts for $||$ and application (unlike in the rule for sequential composition). The bounds for shared variables can then be added up using a special *contraction* rule.

$$\frac{\Gamma \vdash_r M_1 : \mathsf{com} \quad \Gamma \vdash_r M_2 : \mathsf{com}}{\Gamma \vdash_r M_1; M_2 : \mathsf{com}} \qquad \frac{\Gamma \vdash_r M_1 : \mathsf{com} \quad \Delta \vdash_r M_2 : \mathsf{com}}{\Gamma, \Delta \vdash_r M_1 \,||\, M_2 : \mathsf{com}}$$

$$\frac{\Gamma \vdash_r M : \theta^n \to \theta' \quad \Delta \vdash_r N : \theta}{\Gamma, n\Delta \vdash_r MN : \theta'} \qquad \frac{\Gamma, x_1 : \theta^m, x_2 : \theta^n \vdash_r M : \theta'}{\Gamma, x : \theta^{m+n} \vdash_r M[x/x_1, x/x_2] : \theta'}$$

$n\Delta$ is the environment Δ in which all the outermost bounds have been multiplied by n.

Bounds have an intuitive *assume-guarantee* interpretation. A bound n is an assume (resp. guarantee) if it occurs in the left-hand scope of an even (resp. odd) number of $\to$ (the turnstile $\vdash$ is also considered an arrow for this purpose). Assumes concern the behaviour of the program context and guarantees that of the program. Intuitively, if the environment behaves according to the assumes, the program's behaviour satisfies the guarantees. For example, SCC can derive:

$$f : (\mathsf{com}^n \to \mathsf{com})^2, x : \mathsf{com}^{2n} \vdash_r f(x) \,||\, f(x) : \mathsf{com},$$

where n (occurring in the type of f) is the only assume. SCC is made flexible by the use of subsumption: assumes can be decreased and guarantees increased.

Given an SCC typing derivation of M and a context $\mathcal{C}[-]$ such that $\mathcal{C}[M]$ is closed, we can verify whether $\mathcal{C}[-]$ is consistent with the assumes of M simply by checking if $\vdash_r \mathcal{C}[M]$ can be derived from the typing derivation of M. Given $\Gamma \vdash_r M_1$ and $\Gamma \vdash_r M_2$ sharing the same assumes, we now define new approximation and equivalence relations, denoted by $\precsim_r$ and $\cong_r$. The definitions are analogous to those of $\precsim$ and $\cong$ with the exception that the quantification ranges over all contexts $\mathcal{C}[-]$ that respect the assumes of M_1 and M_2.

Unlike $\precsim$ and $\cong$, $\precsim_r$ and $\cong_r$ are decidable, which can be proved using game semantics. $\precsim_r$ and $\cong_r$ can then be shown to correspond to containment and equality of the sets of the complete plays generated by the two terms in question. These in turn can be represented by regular languages. Thus the game model for SCC seems an ideal foundation for a model-checking tool: it is sound, complete (for $\precsim_r$ and $\cong_r$) and decidable [19].

The primary interest is, of course, to verify programs written in the original type system, without bounds on concurrency. Imposing the numerical bounds brings about two limitations. First, only terms with redexes of order less than two are guaranteed to admit an SCC typing. There exist known programs, albeit contrived, that do not admit any SCC typing and thus cannot be analysed using the technique proposed here, e.g.: $(\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f\mathsf{skip}))$. Second, bounds on concurrency in the environment (i.e. the assumes) must be imposed somewhat arbitrarily, and the resulting analysis is sound only within the assumed bounds. Fortunately, the type system SCC will (automatically) certify whether in given execution contexts free identifiers are bound to terms that satisfy the bounds.

3 CSP Representation of Strategies

Game semantics interprets programs as strategies in two-player games between O (Opponent) and P (Proponent), who represent the context and the program respectively. Strategies can be viewed simply as sequences of moves (actions) of the two players, which makes it possible to employ automata-theoretic techniques to their analysis. Strategies corresponding to SCC terms can be represented by regular languages. In this section we show how to do that using CSP [17].

CSP is a particularly convenient formalism for expressing strategies, because it features primitives for (selectively synchronised) parallel composition and hiding, the two operations on which *composition* of strategies is based. Additionally, we will take advantage of CSP channels (to indicate the source of a move), the flexibility to define new alphabets for multiple tagging (to indicate and compare membership in threads) and substitution (for re-tagging). CSP has been used before to represent strategies, but only for sequential programs [18].

First we briefly review the game-theoretic notions involved in the interpretation of terms. Due to space restrictions we omit many technical details, and only try to give the flavour of the approach. Formally, games can be regarded as triples $G = \langle M_G, \lambda_G, P_G^{\max} \rangle$ where M_G is the set of available moves, $\lambda_G : M_G \to \{ O, P \}$ indicates the ownership of moves and $P_G^{\max} \subseteq M_G^*$ is the set of *complete* (maximal) positions. All other positions are simply prefixes of complete positions, so we can define the set of positions on G as $P_G = \{ s \mid \exists t \in P_G^{\max}.s \leq t \}$. Below we list the complete positions in the games corresponding to base types (the initial moves are O-moves, the final ones are P-moves; $0 \leq i \leq max$):

$[\![\mathsf{com}]\!]$	$[\![\mathsf{int}]\!]$	$[\![\mathsf{var}]\!]$	$[\![\mathsf{sem}]\!]$
$run \cdot done$	$q \cdot i$	$read \cdot i$	$grb \cdot okg$
		$write(i) \cdot ok$	$rls \cdot okr.$

SCC higher-order types have shape $\theta^n \to \theta'$, but for technical reasons it is useful to decompose such types using more elementary type constructors $\otimes$ (interleaved product), $!$ (iteration), and $\multimap$ (linear function space): $\theta^n \to \theta' = (\bigotimes_n !\theta) \multimap \theta'$. The definitions of the three game constructions are:

(M_G) $M_{!G} = M_G, M_{G_1 \otimes G_2} = M_{G_1 \multimap G_2} = M_{G_1} + M_{G_2}$

(λ_G) The constructions of $!G$ and $G_1 \otimes G_2$ preserve ownership of moves. In $G_1 \multimap G_2$ moves originating from G_2 have the same owners as in G_2, whereas O-moves (resp. P-moves) from G_1 are P-moves (resp. O-moves) in $G_1 \multimap G_2$.

($P_G^{\max}$) $P_{!G}^{\max}$ consists of sequences of complete positions from $P_G^{\max}$ ($P_{!G}^{\max} = \{ s_1 \cdots s_n \mid n > 0 \text{ and } \forall_{1 \leq i \leq n} s_i \in P_G^{\max} \}$). $P_{G_1 \otimes G_2}^{\max}$ contains interleavings of a position from $P_{G_1}^{\max}$ with a position from $P_{G_2}^{\max}$. $P_{G_1 \multimap G_2}^{\max}$ is similar except that the interleavings have to start and end with moves from G_2.

Game semantics interprets programs as strategies over games defined by the associated types (strategies are simply prefix-closed sets of positions). A term-in-context $x_1{:}\theta_1^{n_1}, \ldots, x_k{:}\theta_k^{n_k} \vdash_r M : \theta$ is then interpreted by a strategy for the game $[\![\theta_1^{n_1}]\!] \otimes \cdots \otimes [\![\theta_k^{n_k}]\!] \multimap [\![\theta]\!]$. Suppose $\theta = \alpha_l^{m_l} \to \cdots \to \alpha_1^{m_1} \to \beta$. Because

the sets of moves $M_{!_\circ G} = M_G$ and $M_{G_1 \otimes G_2} = M_{G_1 \multimap G_2} = M_{G_1} + M_{G_2}$, the game corresponding to a type consists of disjoint copies of games for base types. Hence, $M_{[\![\theta_1^{n_1}]\!] \otimes \cdots \otimes [\![\theta_k^{n_k}]\!] \multimap [\![\theta]\!]} = \sum_{i=1,k} M_{[\![\theta_i^{n_i}]\!]} + \sum_{j=1,l} M_{[\![\alpha_j^{m_j}]\!]} + M_{[\![\beta]\!]}$.

A major design decision in employing CSP to represent strategies concerns the way all the disjoint sums + are interpreted. For the instances of + distinguished above we are going to use $k + l + 1$ different channels (one for each of the components). The disjoint sums involved in the construction of $\theta_i^{n_i}$ or $\alpha_j^{m_j}$ will be tackled differently by using subscripts for $\multimap$ and numeric tags for $\otimes$ (to enumerate the threads in the game $!_\circ G \otimes \cdots \otimes !_\circ G$). In general the moves will have the shape $m_{c_1,\ldots,c_v}.d_1.\ldots.d_w$ (abbreviated as $m_{\boldsymbol{c}}.\boldsymbol{d}$), where $c_i, d_j \in \mathbb{N}$ are indices identifying the type-component of a higher-order type (the c_is) and the thread-component of a nested set of threads (the d_is). To be precise, in order to represent moves of $M_{[\![\theta^n]\!]}$ we will use the alphabet $\mathcal{A}(\theta^n)$ which is defined as follows. For base types we take $\mathcal{A}(\beta) = M_{[\![\beta]\!]}$ and further:

$$\mathcal{A}(\theta^n) = \{\, m_{\boldsymbol{c}}.i.\boldsymbol{d} \mid m_{\boldsymbol{c}}.\boldsymbol{d} \in \mathcal{A}(\theta), 1 \le i \le n \,\}$$

$$\mathcal{A}(\gamma_n \to \cdots \to \gamma_1 \to \beta) = \bigcup_{i=1}^{n} \{\, m_{i,\boldsymbol{c}}.\boldsymbol{d} \mid m_{\boldsymbol{c}}.\boldsymbol{d} \in \mathcal{A}(\gamma_i) \,\} \cup \mathcal{A}(\beta).$$

Concretely, the structure of an action used to represent a move is *identifier*.$move_{rank}$.*thread*. The channel *identifier* represents the free identifier associated with the move, or special identifier "*main*" if the move is associated with the term type. The *rank* is a tag representing the type component (from right to left) associated with the move. Finally, the list of thread indices identify the threads and the (nested) sub-threads containing the move.

We are going to define CSP processes whose traces will coincide with strategies denoting terms in such a way that complete positions $\mathsf{comp}([\![\Gamma \vdash_r M]\!])$ will be followed by special action $\surd$. This will enable us to compare complete positions defined by terms and, by the theorem below, verify program equivalence and approximation. Because we use tags for identifying threads, in order to compare strategies we will have to introduce a canonical way of tag usage, e.g. lowest unused. The convention can be enforced by putting the processes corresponding to terms in parallel with a separate CSP process that acts as a "name server".

Theorem 1 ([19]). *Given* $\Gamma \vdash_r M_i : \theta$ *(*$i = 1, 2$*) let us write* $\mathsf{comp}([\![\Gamma \vdash_r M_i]\!])$ *for the set of complete positions in* $[\![\Gamma \vdash_r M_i]\!]$*. Then* $\Gamma \vdash M_1 \sqsubseteq_{\approx r} M_2$ *iff* $\mathsf{comp}([\![\Gamma \vdash_r M_1]\!]) \subseteq \mathsf{comp}([\![\Gamma \vdash_r M_2]\!])$ *and* $\Gamma \vdash M_1 \cong_r M_2$ *iff* $\mathsf{comp}([\![\Gamma \vdash_r M_1]\!]) = \mathsf{comp}([\![\Gamma \vdash_r M_2]\!])$.

CSP processes corresponding to terms can be defined by induction on their structure. Free identifiers $x : \theta^1 \vdash_r x : \theta$ are interpreted by the *copy-cat* strategy in which O-moves are simply copied by P between the two copies of $[\![\theta]\!]$ (possibly with a delay) subject to the exchange of moves being a position in the relevant game. The behaviour of this strategy resembles that of an unbounded buffer. Its CSP process can be defined inductively on the structure of types.

Suppose $\theta = \theta_k^{n_k} \to \cdots \to \theta_1^{n_1} \to \beta$. $ID(L, R_k, \cdots, R_0, \theta)$ returns a process representing $[\![x : \theta^1 \vdash_r x : \theta]\!]$ in such a way that the moves from $[\![\theta^1]\!]$ are

```
PLUS(A1,A2,A,b) = A.q ->A1.q ->A1?x ->A2.q ->A2?y ->A.((x+y)%b) ->SKIP
EQ(A1,A2,A,b) = A.q -> A1.q -> A1?x -> A2.q -> A2?y
                -> A.(if x==y then 1 else 0) -> SKIP
ASSIGN(A2,A1,A) = A.run -> A2.q -> A2?y -> A1.write.y -> A1.wok
                -> A.done -> SKIP
PAR(A1,A2,A) = A.run -> ((A1.run -> A1.done -> SKIP)
               ||| (A2.run -> A2.done -> SKIP));(A.done -> SKIP)
SEQCOM(A1,A2,A) = A.run -> A1.run -> A1.done -> A2.run -> A2.done
                  -> A.done -> SKIP
IFCOM(A0,A1,A2,A) = A.run -> A0.q ->  A0?y -> if (y==0) then
                    (A2.run -> A2.done -> A.done -> SKIP)
                    else (A1.run -> A1.done -> A.done -> SKIP)
WHILE(A1,A2,A) = A.run -> WHILE_AUX(A1,A2,A)
WHILE_AUX(A1,A2,A) = A1.q -> A1?y -> if (y==0) then (A.done -> SKIP)
                     else (A2.run -> A2.done -> WHILE_AUX(A1,A2,A))
GRAB(A1,A) = A.run -> A1.grb -> A1.gok -> A.done -> SKIP
RELEASE(A1,A) = A.run -> A1.rls -> A1.rok -> A.done -> SKIP
CELL(A,m) = (A.read?b -> A.m.b -> CELL(A,m))
            [] (A.write?v?b -> A.wok.b -> CELL(A,v)) [] SKIP
SEM(A,m) = if (m==0) then (A.grb?b -> A.gok.b -> SEM(A,1) [] SKIP)
           else (A.rls?b -> A.rok.b -> SEM(A,0) [] SKIP)
```

Fig. 1. CSP representation of some strategies

transmitted on channel L, those from $[\![\theta_i^{n_i}]\!]$ on channel R_i and those from $[\![\beta]\!]$ on R_0. Let $P_i = ID_{aux}(LL, RR, \theta_i)$ $(1 \leq i \leq k)$ for some fresh channel names LL, RR, where $ID_{aux}(LL, RR, \theta_i)$ is defined below. For $1 \leq i \leq k, 1 \leq j \leq n_i$ define $P_{i,j} = P_i[[\, RR.m_{\boldsymbol{c}}.\boldsymbol{d} \leftarrow L.m_{i,\boldsymbol{c}}.j.\boldsymbol{d},\ LL.m_{\boldsymbol{c}}.1.\boldsymbol{d} \leftarrow R_i.m_{i,\boldsymbol{c}}.j.\boldsymbol{d}\,]]$. Let $P' = |||_{i=1}^{k}|||_{j=1}^{n_i} STAR(P_{i,j})$, where $STAR(P) = SKIP\,[\,]\,(P; STAR(P))$. Then return $[\,]_{m_1 m_2 \in P_{[\![\beta]\!]}}(R.m_1 \rightarrow L.m_1.1 \rightarrow P'); (L.m_2.1 \rightarrow R.m_2 \rightarrow SKIP)$.

$ID_{aux}(L, R, \theta)$ returns a process representing $[\![x : \theta^1 \vdash_r x : \theta]\!]$ in such a way that the moves from $[\![\theta^1]\!]$ are transmitted on channel L and those from $[\![\theta]\!]$ on R. It can be defined recursively as follows. Suppose $\theta = \theta_k^{n_k} \rightarrow \cdots \rightarrow \theta_1^{n_1} \rightarrow \beta$. Let $P_i = ID_{aux}(LL, RR, \theta_i)$ $(1 \leq i \leq k)$ for some fresh channel names LL, RR. For $1 \leq i \leq k, 1 \leq j \leq n_i$ define $P_{i,j} = P_i[[\, RR.m_{\boldsymbol{c}}.\boldsymbol{d} \leftarrow L.m_{i,\boldsymbol{c}}.j.\boldsymbol{d},\ LL.m_{\boldsymbol{c}}.1.\boldsymbol{d} \leftarrow R.m_{i,\boldsymbol{c}}.j.\boldsymbol{d}\,]]$. Let $P' = |||_{i=1}^{k}|||_{j=1}^{n_i} STAR(P_{i,j})$, where $STAR(P) = SKIP\,[\,]\,(P; STAR(P))$. Then return $[\,]_{m_1 m_2 \in P_{[\![\beta]\!]}}(R.m_1 \rightarrow L.m_1.1 \rightarrow P'); (L.m_2.1 \rightarrow R.m_2 \rightarrow SKIP)$.

The CSP representation of some of the key constants of the language is given in Fig. 1. Using different channels for moves of $[\![\alpha_j^{m_j}]\!]$ makes interpreting application relatively easy, because it suffices to use the channel corresponding $\alpha_l^{m_l}$ to synchronise the process corresponding to the function term with m_l interleaved copies of that corresponding to the argument. Suppose P_1, P_2 are the CSP processes representing $[\![\Gamma \vdash_r M_1 : \theta_1]\!]$ and $[\![\Gamma \vdash_r M_2 : \theta_2]\!]$ respectively and R_0^i are the channels on which moves from the right copies of respectively $[\![\theta_i]\!]$, $i = 1, 2$ are transmitted. Then the process P representing $\Gamma \vdash_r M_1 \Box M_2$ is:

$$P = ((P_1 \,|||\, P_2)\ [|R_0^1, R_0^2|]\ PROC_{\square}(R_0^1, R_0^1, R_0)) \setminus \{|R_0^1,\ R_0^2|\},$$

where $PROC_{\square}(\cdots)$ is the CSP representation of the $\square$ binary operator (+, =, ;, :=, etc), as given in Fig. 1. Operators of different arity (if-then-else, grab, release, etc.) are treated analogously.

Application is parallel composition synchronised on the actions corresponding to the type of the argument, followed by the hiding of those actions. Contraction amounts to renumbering threads: m threads (indexed by $1, \ldots, m$) on one channel and n threads on another (with indices from $1, \ldots, n$) have to be renumbered as threads indexed $1, \cdots, m+n$ on a new channel, done by CSP substitution.

For example, the main processes generated in the representation of the strategies for $f : \mathsf{com}^2 \to \mathsf{com}, x : \mathsf{com} \vdash fx : \mathsf{com}$ are:

```
P8 = ||| j:{0..1} @ STAR(ADD(ADD(P7,j,C7,C3),j,C5,C6))
P9 = (P8[|{|C3|}|]P3)\{|C3|}
ADD(P,j,IN,OUT)=P[[IN.done.x<-OUT.done.((x+j)%(3))|x<-{0..2}]]
                 [[IN.run.x<-OUT.run.((x+j)%(3))|x<-{0..2}]]
                 [[IN.done_1.x<-OUT.done_1.((x+j)%(2))|x<-{0..1}]]
                 [[IN.run_1.x<-OUT.run_1.((x+j)%(2))|x<-{0..1}]]
```

In the above, process P8 generates 2 interleavings of the argument (represented by P7, not shown) using auxiliary processes STAR (which iterates its argument) and ADD (which serves as the renaming server). Process P9 is the actual application (in which P3 represents the free variable f, not shown) consisting on synchronisation on channel C3 followed by the hiding of C3.

Variable and semaphore introduction can be represented by application of special (higher-order) constants newvar_m and newsem_m: $\mathsf{newvar}\, x{:=}m\, \mathsf{in}\, M \equiv \mathsf{newvar}_m(\lambda x.M)$, and $\mathsf{newsem}\, x{:=}m\, \mathsf{in}\, M \equiv \mathsf{newsem}_m(\lambda x.M)$. The applications are modeled by parallel composition with hiding using the CSP processes `CELL(···,m)` and `SEM(···,m)` respectively.

4 Tool Support and Case Studies

Using translation to CSP we can employ FDR to verify several classes of properties: program equivalences ($\cong_r$) and inequivalences, approximation ($\sqsubseteq_r$), assertions, invariants and other safety properties.

In our examples, the channel names associated with free identifiers will always have a name related to the identifier, `int$i` will stand for the type $\{0, \ldots, i-1\}$. We also use n-ary semaphores ($n > 1$), which can be easily added to SCC, writing `sem$n` for the corresponding type (`sem$1` is identical to the type `sem` of binary semaphores). In the programs below, the assumed bounds on the behaviour of the environment, e.g. $p : \mathsf{com}^2 \to \mathsf{int}$, are represented as `p:com{2}->exp`.

We implemented a tool which takes as input SCC terms (only the assumes are actually required), infers the missing guarantee bounds then compiles the term in the CSP process algebra. The FDR model-checker is invoked to verify safety properties or to check (may) equivalence of terms.

4.1 Warm-Up Example

Let us illustrate the model with a classic example from the literature [20]:

$$p : \mathsf{com} \to \mathsf{int} \to \mathsf{com} \vdash \mathsf{newvar}\, x{:=}0\, \mathsf{in}\, p(x{:=}x+1; x{:=}x-1)(x) \not\equiv p\, \mathsf{skip}\, 0.$$

The non-local procedure p can increment then decrement the local variable x or dereference it, but has no other access to it. Therefore, in a *sequential* programming language the equivalence stands. However, in a *concurrent* language the equivalence may fail because the arguments can generate race conditions. Indeed, if we give p the SCC typing $\mathsf{com}^m \to \mathsf{int}^n \to \mathsf{com}$ for some $m, n > 0$, FDR identifies a trace which can occur in the LHS but not in the RHS (we present it along with a move-by-move interpretation):

main.run	start execution, first main thread
p.run.1	start executing p's main thread
$p.q_1$.1.1	start executing p's right argument, first thread
$p.run_2$.1.1	start executing p's left argument, first thread
$p.1_1$.1.1	p's right argument in first thread produces 1

We can see that the reason for the equivalence failing was a race condition. SCC is call-by-name, i.e. the arguments are *thunks*, so the right argument may begin to be evaluated *before* the evaluation of the left argument has completed ($p.ok_2$.1.1). In fact, the diagrammatic representation of the processes produced by FDR shows quite clearly that the two processes are not similar (Fig. 2).

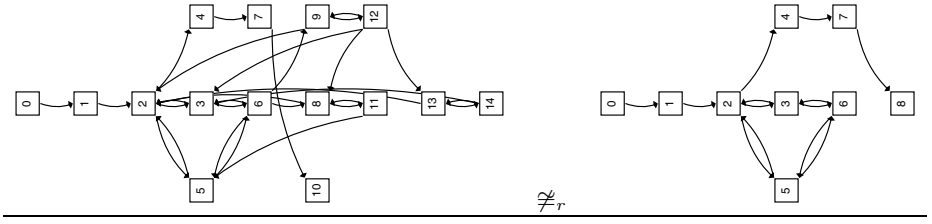

Fig. 2. Two inequivalent processes

4.2 Verifying Algorithm Implementations

Consider the code in Fig. 3, implementing the tie-breaker algorithm [21] as a procedure which takes as arguments two critical regions, two non-critical regions and two termination conditions (LHS). We can verify the algorithm by comparing it against a simpler implementation which assumes the existence of semaphores in the language and serves as a specification (RHS). By compiling the two implementations into CSP, we can use FDR to verify that, indeed, they are equivalent.

```
mutex1(crtc1:com, crtc2:com, nonc1:com, nonc2:com, b1:int$2, b2:int$2) =
 int in1, in2, last;
 while b1 do {
    in1:=1; last:=1; while (in2 & last=1) do skip; crtc1; in1:=0; nonc1 }
 || while b2 do {
    in2:=1; last:=2; while (in1 & last=2) do skip; crtc2; in2:=0; nonc2 }
≅ mutex2(crtc1:com, crtc2:com, nonc1:com, nonc2:com, b1:int$2, b2:int$2)=
  sem s; while b1 do {grab(s); crtc1; release(s); nonc1}
  || while b2 do {grab(s); crtc2; release(s); nonc2}
```

Fig. 3. The tie-breaker algorithm vs semaphores

4.3 Verifying ADT Implementations

One of the principal advantages of our approach is that we can model *open* programs, such as ADTs. For example, let us consider the stack implementation given in Fig. 4, where n is the size of the stack and *empty*, *overflow* are (unspecified) user-defined procedures to handle usage errors. The implementation stores the stack elements in an array and uses a semaphore to protect the changes to the array as well as the variable `crt` that indicates the top of the stack. However, it is not actually *thread-safe* and contains a non-trivial (but common) error which we will "discover" using our model-checker. In order to model and verify the ADT we consider the program *VERIFY push pop top*, where *VERIFY* : $(\mathsf{int} \to \mathsf{com})^1 \to \mathsf{com}^1 \to \mathsf{int}^1 \to \mathsf{com}$ plays the role of *the most general environment*. After generating the game model with FDR we can check the stack ADT for safety properties such as buffer over-runs or assertion failures. For instance, if we introduce an additional free identifier

```
empty:com,  overflow:com, VERIFY:(int->com){1}->com{1}->int{1}->com |-
  int buf[n], crt; sem s;
  let size:int = n
      isempty:int = (crt = 0)
      isfull:int = (crt = size)
      push:(int->com) = fun x : int.
        if isfull then overflow
        else (grab(s); buf[crt]:=x; crt:=crt+1; release(s))
      top:int =
        int tmp;
        if isempty then (empty; 0)
        else (grab(s); tmp:=buf[crt-1]; release(s); tmp)
      pop:com =
         if isempty then empty else (grab(s); crt:=crt-1; release(s))
  in VERIFY push pop top : com.
```

Fig. 4. A bounded-stack implementation

segf : com (segmentation fault) and arrange for *segf* to be invoked for buffer overrun errors, the FDR will identify the following safety violation:

main.run	start execution
$VERIFY.run.1$	start running *VERIFY*
$VERIFY.run_3.1.1$	call *push*
$VERIFY.q_{3,1}.1.1.1$	*push* requests an argument
$VERIFY.q_1.1.1$	call *top*
$VERIFY.0_{3,1}.1.1.1$	provide an argument to *push*
$VERIFY.run_2.1.1$	call *pop*
segf.run	a violation has occurred.

The reason for the violation is the fact that only the changes to the buffer and the top of the stack have been protected by a semaphore. As seen in the trace, a violation can still occur if *top* starts executing on a one-element stack, then *pop* is executed concurrently between the empty-stack check and the dereferencing of the buffer. A thread-safe implementation must protect with semaphores the entire scope of the stack methods, including the testing for empty and full buffer.

5 Higher-Order Procedures: Producer-Consumer

Our final example will examine a producer-consumer algorithm [21]: the procedure accepts as arguments a *producer* and a *consumer* function along with a parameter indicating when termination should occur. The value returned by the producer function is stored in a circular buffer. The consumer function takes a value from the circular buffer and performs some (unknown) action. The main procedure executes p copies of the producer process in parallel with c copies of the consumer process, each in a loop controlled by the argument $b1$ or $b2$. Information in the form of numbers from 0 to $i-1$ is shared using an n-place buffer. The values of n, p, c, i are constants.

In the implementation shown in Fig. 5, semaphores s and t are used to prevent race conditions between the producers and, respectively, the consumers. Note that a producer and a consumer may access the buffer concurrently. N-ary semaphores *full* and *empty* make the producers and the consumers wait if the buffer is full, respectively empty.

This procedure is interesting because it is not possible to reduce it meaningfully to a first-order program. The SCC typing of the *prodcon* procedure means that the analysis requires that the `consume` procedure uses its argument in at most one thread of execution, which is not an unreasonable restriction. We can perform the same safety analyses as described before, and the implementation in Fig. 5 does not produce violations. We can also perform various safety tests using the FDR-specific idiom, *refinement* [17].

However, in the case of a complex program such as this, the real challenge lies in constructing the model, so we will use this example primarily to illustrate how the state space of the model is affected by the various constants occurring in

```
1  prodcon(produce:int$i, consume:int$i{1}->com, b1:int$2, b2:int$2) =
2  int$i buf[n], front, rear;
3  sem$n full=n, empty;
4  sem s, t;
5     dopar j := 1,p while b1 do {
6       int$i tmp := produce;
7       grab(empty); grab(s);
8       buf[rear] := tmp;
9       rear := (rear + 1) mod n;
10      release(s); release(full) }
11 || dopar j := 1,c while b2 do {
12      grab(full); grab(t);
13      int$i tmp := buf[front];
14      front := (front + 1) mod n;
15      release(t); release(empty);
16      consume(tmp) }
```

Fig. 5. A higher-order producer-consumer procedure

the program. We will also compare the size of the resulting model, as produced by FDR, with the size of a *naive* model generated by state exploration and interleaving of basic operational steps. (According to the operational semantics of the language each thread needs around 30 such steps.)

The results of the comparison are given in Tbl. 1. The *workspace* column indicates the largest *intermediate* model generated in the course of creating the final model. It is clear from the data above that the savings achieved using an *observational model*, which hides state changes that are not externally observable, are substantial. This is consistent with our earlier analysis of data-intensive algorithms. We can also see the importance of *compositional* model construction and *concise procedure summaries*, because a client of the `prodcon` procedure can now be modeled using the very compact observational model. Inlining the procedure even a couple of times would generate models of unmanageable size.

Table 1. Benchmark results

n	p	c	i	naive model	game model	FDR workspace	time (s)
3	1	1	1	40,000	114	2,554	112
4	1	1	1	62,500	143	5,168	142
3	2	1	1	1,000,000	1,684	39,758	247
3	1	2	1	1,000,000	1,735	43,206	351
3	1	1	2	5,120,000	464	4,632	223
2	2	2	1	14,062,500	6,478	495,621	1,733
3	2	2	1	25,000,000	13,813	760,389	4,889
2	2	2	2	3,600,000,000	24,489	1,763,637	54,617

Our experiments also confirm that increasing the amount of *observable* concurrency in the system has a far worse effect over model size than increasing the amount of data available to the system. The last case is perhaps the most interesting. The very large naive model state space is due to increasing the size of variable `tmp` which occurs in 4 threads. But the variable is local, hence invisible from outside its scope, so it does not contribute directly to the final model.

6 Conclusions

Game semantics provides a new technique for software model extraction which, as we have seen, has several advantages. The semantics-directed nature of the approach ensures correctness and completeness by construction and a compositional, incremental way of generating the model. What makes game models substantially different from more traditional models is a focus on *observational behaviour*, i.e. on the interaction between a program and its context, and hiding the non-observable details such as internal state changes.

Our experiments show that game semantics leads to much more compact models than those obtained by naive interleaving. We believe that further gains in efficiency are possible with the help of partial-order reduction techniques [23]. However, their incorporation into game semantics has not been investigated yet, especially the subtle relation between partial-order reduction and composition, and we leave it for future work.

Software verification using game models is still in its infancy but the initial developments are promising. However, we are some distance away from providing true competition to industrial-level tools. The following developments, which are within reach, will however bring us closer to realistic applications:

Real languages. The language we study here is realistic and expressive, but it is ultimately an academic exercise. We believe game-semantic techniques are now mature enough so that we can soon tackle a real programming language, such as a substantial subset of Java or C.

Liveness. The game model for SCC is derived from an *angelic* notion of termination which corresponds to trace equivalence. This does not account for deadlock or live-lock. Upgrading the semantic model to handle these phenomena is the subject of on-going research.

Algorithmics. So far we have used off-the-shelf model checkers which do not exploit the features of our semantics perfectly. FDR, for example lacks features which are common in modern model checkers, such as BDD representation. SPIN [6] is a powerful model-checker, but (unlike FDR) is essentially stateful. Moreover, neither of the two supports *composition*, i.e. creating a model from two independently generated models (although FDR uses compositional reductions internally).

Refinement. Our use of data abstraction in this model checker is relatively informal. The problem of automatically abstracting and refining the model is critical for software verification, and is dealt with separately [22].

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: technology transfer of formal methods inside Microsoft. In: IFM 2004, LNCS **2999** 1–20
2. Clarke, E.M., Grumberg, O., Peled, P.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
3. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC / SIGSOFT FSE (2003) 267–276
4. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV 2001, LNCS **2102** 260–264
5. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: CAV 2004, LNCS **3114** 484–487
6. Holzmann, G.J.: The Spin model checker. IEEE Trans. on Soft. Eng. **23** (1997) 279–295
7. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: POPL (2004) 245–255
8. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Information and Computation **163** (2000)
9. Hyland, J. M. E., Ong, C.-H. L.: On full abstraction for PCF: I, II and III. Information and Computation **163** (2000)
10. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. ENTCS **3** (1996)
11. Laird, J.: Full abstraction for functional languages with control. In: LICS (1997) 58–67
12. Ghica, D.R., Murawski, A.S.: Angelic semantics of fine-grained concurrency. In: FOSSACS 2004, LNCS **2987** 211–225
13. Hankin, C., Malacaria, P.: Generalised flowcharts and games. In: ICALP (1998) 363–374
14. Ghica, D.R., McCusker, G.: Reasoning about Idealized ALGOL using regular languages. In: ICALP 2000, LNCS **1853** 103–116
15. Ghica, D.R.: A Games-based Foundation for Compositional Software Model Checking. PhD thesis, Queen's University, Kingston, Canada (2002)
16. Abramsky, S., Ghica, D. R., Murawski, A. S., Ong, C.-H. L.: Applying game semantics to compositional software modeling and verification. In: TACAS'04, LNCS **2988** 421–435
17. Roscoe, W.A.: Theory and Practice of Concurrency. Prentice-Hall (1998)
18. Dimovski, A., Lazic, R.: CSP Representation of Game Semantics for Second-Order Idealized Algol. In: ICFEM 2004, LNCS **3308** 146–191.
19. Ghica, D.R., Murawski, A. S., Ong, C.-H. L.: Syntactic control of concurrency. In: ICALP'04, LNCS **3142** 683–694
20. Brookes, S.: The essence of Parallel Algol. In: LICS (1996) 164–173
21. Andrews, G.: Concurrent Programming: principles and practice. Addison-Wesley Publishing Company (1991)
22. Dimovski, A., Ghica, D.R., Lazic, R.: Data-Abstraction Refinement: A Game Semantic Approach In: SAS'05, LNCS **3672** 102-117
23. Rajeev, A., et. al.: Partial-Order Reduction in Symbolic State-Space Exploration. Formal Methods in System Design **18**(2): 97-116 (2001)