Applying Game Semantics to Compositional Software Modeling and Verification*

Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, C.-H. Luke Ong

Oxford University Computing Laboratory
Parks Road, Oxford, OX1 3QD, U. K.

Abstract. We describe a software model checking tool founded on game semantics, highlight the underpinning theoretical results and discuss several case studies. The tool is based on an interpretation algorithm defined compositionally on syntax and thus can also handle open programs. Moreover, the models it produces are equationally fully abstract. These features are essential in the modeling and verification of software components such as modules and turn out to lead to very compact models of programs.

1 Introduction and Background

Game Semantics has emerged as a powerful paradigm for giving semantics to a variety of programming languages and logical systems. It has been used to construct the first syntax-independent fully abstract models for a spectrum of programming languages ranging from purely functional languages to languages with non-functional features such as control operators and locally-scoped references [1–6].

We are currently developing Game Semantics in a new, algorithmic direction, with a view to applications in computer-assisted verification and program analysis. Some promising steps have already been taken in this direction. Hankin and Malacaria have applied Game Semantics to program analysis, e.g. to certifying secure information flows in programs [7, 8]. A particularly striking development was the work by Ghica and McCusker [9] which captures the game semantics of a procedural language in a remarkably simple form, as regular expressions. This leads to a decision procedure for observational equivalence on this fragment. Ghica has subsequently extended the approach to a call-by-value language with arrays [10], to model checking Hoare-style program correctness assertions [11] and to a more general model-checking friendly specification framework [12].

Game Semantics has several features which make it very promising from this point of view. It provides a very *concrete* way of building *fully abstract* models. It has a clear operational content, while admitting *compositional methods* in the style of denotational semantics. The basic objects studied in Game

* The authors gratefully acknowledge the support of UK EPSRC, Canadian NSERC and St. John's College, Oxford University.

Semantics are games, and strategies on games. Strategies can be seen as certain kinds of highly-constrained processes, hence they admit the same kind of automata-theoretic representations central to model checking and allied methods in computer-assisted verification. Moreover, games and strategies naturally form themselves into rich mathematical structures which yield very accurate models of advanced high-level programming languages, as the various full abstraction results show. Thus the promise of this approach is to carry over the methods of model checking (see e.g. [13]), which has been so effective in the analysis of circuit designs and communications protocols, to much more *structured* programming situations, in which data-types as well as control flow are important.

A further benefit of the algorithmic approach is that by embodying game semantics in tools, and making it concrete and algorithmic, it should become more accessible and meaningful to practitioners. We see Game Semantics as having the potential to fill the role of a "Popular Formal Semantics," called for in an eloquent paper by Schmidt [14], which can help to bridge the gap between the semantics and programming language communities. Game Semantics has been successful in its own terms as a semantic theory; we aim to make it useful to and usable by a wider community.

Model checking for state machines is a well-studied problem (e.g. Murϕ [15], Spin [16] and Mocha [17] to name a few systems). Software model checking is a relatively new direction (see e.g. [18]); the leading projects (e.g. SLAM [19], and *Bandera* [20]) excel in tool constructions. The closest to ours in terms of target applications is the SLAM project, which is able to check safety properties of C programs. This task is reduced in stages to the problem of checking if a given statement in an instrumented version of the program in question is reachable, using ideas from data-flow and inter-procedural analysis and abstract interpretation.

In relation to the extensive current activity in model checking and computer assisted verification, our approach is distinctive, being founded on a highly-structured *compositional* semantic model. This means that we can directly apply our methods to *open program phrases* (i.e. terms-in-context with free variables) in a high-level language with procedures, local variables and data types. This ability is essential in analyzing properties of software components. The soundness of our methods is guaranteed by the properties of the semantic models on which they are based. By contrast, most current model checking applies to relatively "flat" unstructured situations.

Our semantics-driven approach has some other additional benefits: it is generic and fully automated. The prototype tool we have implemented has the level of automation of a compiler. The input is a program fragment, with very little instrumentation required, and the output is a finite-state (FS) model. The resulting model itself can be analyzed using third-party model-checking tools, or our tool can automatically extract traces with certain properties, e.g. error traces.

Software model checking is a fast-developing area of study, driven by needs of the industry as much as, if not more than, theoretical results. Often, tool development runs well ahead of rigorous considerations of soundness of the methods being developed. Our aim is to build on the tools and methods which have been developed in the verification community, while exploring the advantages offered by our semantics-directed approach.

2 A Procedural Programming Language

Our prototypical procedural language is a simply-typed call-by-name lambda calculus with basic types of booleans (**bool**), integers (**exp**), assignable variables (**var**) and commands (**comm**). We denote the basic types by σ and the function types by θ. *Assignable* variables, storing integers, form the state while commands change the state. In addition to abstraction ($\lambda x : \sigma.M$) and application (FA), other terms of the language are conditionals, uniformly applied to any type, ($\mathbf{if}\, B\, \mathbf{then}\, M\, \mathbf{else}\, N$), recursion ($\mathbf{fix}\, x : \sigma.M$), constants (integers, booleans) and arithmetic-logic operators ($M * N$); we also have command-type terms which are the standard imperative operators: dereferencing (explicit in the syntax, $!V$), assignment ($V := N$), sequencing ($C; M$, note that we allow, by sequencing, expressions with side-effects), no-op (**skip**) and local variable block ($\mathbf{new}\, x\, \mathbf{in}\, M$). We write $M : \sigma$ to indicate that term M has type σ.

This language, which elegantly combines state-based procedural and higher-order functional programming, is due to Reynolds [21] and its semantic properties have been the object of extensive research [22].

If the programming language is restricted to first-order procedures, (more precisely, we restrict types to $\theta ::= \sigma \mid \sigma \to \theta$) tail recursion (iteration) and finite data-types then the Abramsky-McCusker fully abstract game model for this language [3] has a very simple regular-language representation [9]. The formulation of the regular-language model in loc. cit. is very well suited for proving equivalences "by hand," but we will prefer a slightly different but equivalent presentation [23] because it is more uniform and more compact. The referenced work gives motivation and numerous examples for the model presented below.

2.1 Extended Regular Expressions

This section describes the representation of the game model using a language of extended regular expressions. Due to space constraints, a basic understanding of game semantics must be assumed as background. Otherwise, the reader is encouraged to refer to the literature mentioned in the Introduction.

Terms are interpreted by languages over alphabets of moves $\mathcal{A}$. The languages, denoted by $\mathcal{L}(R)$, are specified using extended regular expressions R. They include the standard regular expressions consisting of the empty language $\emptyset$, the empty sequence ϵ, concatenation $R \cdot S$, union $R + S$, Kleene star R^*, and the elements of the alphabet taken as sequences of unit length. We also use the additional constructs of intersection $R \cap S$, direct image under homomorphism ϕR and inverse image $\phi^{-1} R$. The languages defined by these extensions are the obvious ones.

It is a standard result that any extended regular expression constructed from the operations described above denotes a regular language, which can be recognized by a finite automaton which can be effectively constructed from the regular expression [24].

We will often use the disjoint union of two alphabets to create a larger alphabet: $\mathcal{A}_1+\mathcal{A}_2 = \{a^{\langle 1\rangle} \mid a\in\mathcal{A}_1\} \cup \{b^{\langle 2\rangle} \mid b\in\mathcal{A}_2\} = \mathcal{A}_1^{\langle 1\rangle} \cup \mathcal{A}_2^{\langle 2\rangle}$. The tags $-^{\langle i\rangle}$ are used on a lexical level, resulting in new and distinct symbols belonging to the larger alphabet. The disjoint union gives rise to the canonical maps: $\mathcal{A}_1 \underset{\text{outl}}{\overset{\text{inl}}{\rightleftarrows}} \mathcal{A}_1 + \mathcal{A}_2 \underset{\text{outr}}{\overset{\text{inr}}{\leftrightarrows}} \mathcal{A}_2$. The definition of the maps is:

$$\begin{array}{lll} \text{inl}\, a = a^{\langle 1\rangle} & \text{outl}\, a^{\langle 1\rangle} = a & \text{outr}\, a^{\langle 1\rangle} = \epsilon \\ \text{inr}\, b = b^{\langle 2\rangle} & \text{outl}\, b^{\langle 2\rangle} = \epsilon & \text{outr}\, b^{\langle 2\rangle} = b \end{array}$$

If $\phi : \mathcal{A} \to \mathcal{B}^*$ and $\phi' : \mathcal{C} \to \mathcal{D}^*$ are homomorphisms then we define their sum $\phi+\phi' : \mathcal{A}+\mathcal{C} \to (\mathcal{B}+\mathcal{D})^*$ as $(\phi+\phi')(a^{\langle 1\rangle}) = (\phi a)^{\langle 1\rangle}$, respectively $(\phi+\phi')(c^{\langle 2\rangle}) = (\phi' c)^{\langle 2\rangle}$.

Definition 1 (Composition). *If R is a regular expression over alphabet $\mathcal{A}+\mathcal{B}$ and S a regular expression over alphabet $\mathcal{B}+\mathcal{C}$ we define the* composition $R \circ S$ *as the regular expression $R \circ S = out\big(out_1^{-1}(R) \cap out_2^{-1}(S)\big)$, over alphabet $\mathcal{A}+\mathcal{C}$, with maps $\mathcal{A}+\mathcal{B} \underset{out_1}{\overset{in_1}{\rightleftarrows}} \mathcal{A}+\mathcal{B}+\mathcal{C} \underset{out_2}{\overset{in_2}{\leftrightarrows}} \mathcal{B}+\mathcal{C}$ and $\mathcal{A}+\mathcal{C} \underset{out}{\overset{in}{\rightleftarrows}} \mathcal{A}+\mathcal{B}+\mathcal{C}$.*

Regular expression composition is very similar to composition of finite state transducers [25]. Sets $\mathcal{A}$ and $\mathcal{B}$ represent, respectively, the input and the output of the first transducer; sets $\mathcal{B}$ and $\mathcal{C}$ represent, respectively, the input and the output of the second transducer. The result is a transducer of inputs $\mathcal{A}$ and output $\mathcal{C}$. For example, let $\mathcal{A} = \{a\}$, $\mathcal{B} = \{b\}$, $\mathcal{C} = \{c\}$; then $(ab)^* \circ (bcc)^* = (acc)^*$.

2.2 Alphabets

We interpret each type θ by a language over an alphabet $\mathcal{A}[\![\theta]\!]$, containing the *moves* from the game model. For basic types σ it is helpful to define alphabets of questions $\mathcal{Q}[\![\sigma]\!]$ and answers $\mathcal{A}_q[\![\sigma]\!]$ for each $q \in \mathcal{Q}[\![\sigma]\!]$. The alphabet of type σ is then defined as $\mathcal{A}[\![\sigma]\!] = \mathcal{Q}[\![\sigma]\!] \cup \bigcup_{q\in\mathcal{Q}[\![\sigma]\!]} \mathcal{A}_q[\![\sigma]\!]$. The basic type alphabets are:

$$\begin{array}{ll} \mathcal{Q}[\![\mathbf{exp}]\!] = \{\mathsf{q}\},\ \mathcal{A}_\mathsf{q}[\![\mathbf{exp}]\!] = \mathsf{N} & \mathcal{Q}[\![\mathbf{bool}]\!] = \{\mathsf{q}\},\ \mathcal{A}_\mathsf{q}[\![\mathbf{bool}]\!] = \{\mathsf{t},\mathsf{f}\} \\ \mathcal{Q}[\![\mathbf{var}]\!] = \{\mathsf{q}\} \cup \{\mathsf{w(n)} \mid \mathsf{n} \in \mathsf{N}\}, & \mathcal{A}_\mathsf{q}[\![\mathbf{var}]\!] = \mathsf{N}, \mathcal{A}_{\mathsf{w(n)}} = \{\star\} \\ \mathcal{Q}[\![\mathbf{comm}]\!] = \{\mathsf{q}\},\ \mathcal{A}_\mathsf{q}[\![\mathbf{comm}]\!] = \{\star\}. & \end{array}$$

where $\mathsf{N} = \{-n, \cdots, -1, 0, 1, \cdots, n\}$.

Alphabets of function types are defined by $\mathcal{A}[\![\sigma \to \theta]\!] = \mathcal{A}[\![\sigma]\!] + \mathcal{A}[\![\theta]\!]$.

A typing judgement $\Gamma \vdash M : \theta$ is interpreted by a regular expression $R = [\![\Gamma \vdash M : \theta]\!]$ over alphabet $\sum_{x_i : \theta_i \in \Gamma} \mathcal{A}[\![\theta_i]\!] + \mathcal{A}[\![\theta]\!]$.

For any type $\theta = \sigma_1 \to \cdots \to \sigma_k \to \sigma$, it is convenient to define the regular language K_θ over alphabet $\mathcal{A}[\![\theta]\!] + \mathcal{A}[\![\theta]\!]$, called the *copy-cat* language:

$$K_\theta = \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 2 \rangle} \cdot q^{\langle 1 \rangle} \cdot \Big(\sum_{i=1,k} R_i \Big)^* \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 1 \rangle} \cdot a^{\langle 2 \rangle},$$

where $R_i = \sum_{q \in \mathcal{Q}[\![\sigma_i]\!]} q^{\langle 2 \rangle} \cdot q^{\langle 1 \rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma_i]\!]} a^{\langle 1 \rangle} \cdot a^{\langle 2 \rangle}$. This regular expression represents the so-called copy-cat strategy of game semantics, and it describes the generic behaviour of a sequential procedure. At second-order [26] and above [2] this behaviour is far more complicated.

2.3 Regular-Language Semantics

We interpret terms using an evaluation function $[\![-]\!]$ mapping a term $\Gamma \vdash M : \theta$ and an environment u into a regular language R. The environment is a function, with the same domain as Γ, mapping identifiers of type θ to regular languages over $\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![\theta]\!]$. The evaluation function is defined by recursion on the syntax.

Identifiers. Identifiers are read from the environment: $[\![\Gamma, x : \theta \vdash x : \theta]\!]u = u(x)$.

Let. $[\![\Gamma \vdash \mathbf{let}\, x\, \mathbf{be}\, M\, \mathbf{in}\, N]\!]u = [\![\Gamma, x : \theta \vdash N]\!](u \mid x \mapsto [\![\Gamma \vdash M]\!]u)$.

Abstraction. $[\![\Gamma \vdash \lambda x : \sigma . M : \sigma \to \theta]\!]u = \phi\big([\![\Gamma, x : \sigma \vdash M : \theta]\!](u \mid x \mapsto K_\sigma)\big)$, where ϕ is the (trivial) associativity isomorphism.

Linear application and contraction. $[\![\Gamma, \Delta \vdash MN]\!]u = [\![\Gamma \vdash M]\!]u \circ \big([\![\Delta \vdash N]\!]u\big)^*$, with composition $- \circ -$ defined as before. Contraction is

$$[\![\Gamma, z : \theta \vdash M[z/x, z/x'] : \theta]\!]u = (\mathrm{id}_1 + \delta + \mathrm{id}_2)\big([\![\Gamma, x : \theta, x' : \theta \vdash M : \theta]\!]u\big),$$

where id_1 and id_2 are identities on $\mathcal{A}[\![\Gamma]\!]$ and, respectively, $\mathcal{A}[\![\theta]\!]$. It is well known that application can be harmlessly decomposed in linear application and contraction. The homomorphism $\delta : \mathcal{A}[\![\theta]\!] + \mathcal{A}[\![\theta]\!] \to \mathcal{A}[\![\theta]\!]$ only removes tags from moves. Note that this interpretation is specific to first-order types. In higher-order types this interpretation of contraction by un-tagging can result in ambiguities.

Block Variables. Consider the following regular expression over alphabet $\mathcal{A}[\![\mathbf{var}]\!]$: $\mathrm{cell} = \Big(\sum_{\mathsf{n} \in \mathsf{N}} \mathsf{w(n)} \cdot \star \cdot (\mathsf{q} \cdot \mathsf{n})^*\Big)^*$. Intuitively, this regular expression describes the sequential behaviour of a memory cell: if a value n is written, then the same value is read back until the next write, and so on. We define block variables as $[\![\Gamma \vdash \mathbf{new}\, x\, \mathbf{in}\, M : \sigma]\!]u = [\![\Gamma, x : \mathbf{var} \vdash M : \sigma]\!]u \circ \mathrm{cell}$,

Constants. Finally, the interpretation of constants is:

$$[\![n : \mathbf{exp}]\!] = \mathsf{q} \cdot \mathsf{n}, [\![\mathbf{true} : \mathbf{bool}]\!] = \mathsf{q} \cdot \mathsf{t}, [\![\mathbf{false} : \mathbf{bool}]\!] = \mathsf{q} \cdot \mathsf{f}$$

$$[\![-\mathbf{op}- : \sigma \to \sigma \to \sigma']\!] = \sum_{\mathsf{p} \in \mathsf{N}} \sum_{\substack{\mathsf{m,n} \in \mathsf{N} \\ \mathsf{p} = \mathsf{m}\, op\, \mathsf{n}}} \mathsf{q}^{\langle 3 \rangle} \cdot \mathsf{q}^{\langle 1 \rangle} \cdot \mathsf{m}^{\langle 1 \rangle} \cdot \mathsf{q}^{\langle 2 \rangle} \cdot \mathsf{n}^{\langle 2 \rangle} \cdot \mathsf{p}^{\langle 3 \rangle}$$

$$[\![- := - : \mathbf{var} \to \mathbf{exp} \to \mathbf{comm}]\!] = \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{q}^{\langle 3 \rangle} \cdot \mathsf{q}^{\langle 2 \rangle} \cdot \mathsf{n}^{\langle 2 \rangle} \cdot \mathsf{w(n)}^{\langle 1 \rangle} \cdot \star^{\langle 1 \rangle} \cdot \star^{\langle 3 \rangle}$$

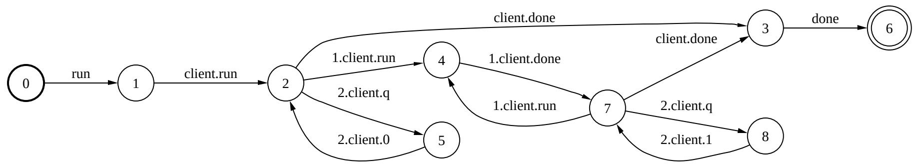

Fig. 1. A simple switch

$$
\begin{aligned}
&[\![\mathbf{if} - \mathbf{then} - \mathbf{else} - : \mathbf{bool} \to \sigma \to \sigma \to \sigma]\!] \\
&\qquad = \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 4\rangle} \cdot \mathsf{q}^{\langle 1\rangle} \cdot \mathsf{t}^{\langle 1\rangle} \cdot q^{\langle 2\rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 2\rangle} \cdot a^{\langle 4\rangle} \\
&\qquad + \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 4\rangle} \cdot \mathsf{q}^{\langle 1\rangle} \cdot \mathsf{f}^{\langle 1\rangle} \cdot q^{\langle 3\rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 3\rangle} \cdot a^{\langle 4\rangle} \\
&[\![-;-: \mathbf{comm} \to \sigma \to \sigma]\!] = \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 3\rangle} \cdot \mathsf{q}^{\langle 1\rangle} \cdot \star^{\langle 1\rangle} \cdot q^{\langle 2\rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 2\rangle} \cdot a^{\langle 3\rangle} \\
&[\![\mathbf{while} - \mathbf{do} - : \mathbf{bool} \to \mathbf{comm} \to \mathbf{comm}]\!] \\
&\qquad = \mathsf{q}^{\langle 3\rangle} \cdot \left(\mathsf{q}^{\langle 1\rangle} \cdot \mathsf{t}^{\langle 1\rangle} \cdot \mathsf{q}^{\langle 2\rangle} \cdot \star^{\langle 2\rangle}\right)^* \cdot \mathsf{q}^{\langle 1\rangle} \cdot \mathsf{f}^{\langle 1\rangle} \cdot \star^{\langle 3\rangle} \\
&[\![\mathbf{div} : \mathbf{comm}]\!] = \emptyset, \qquad [\![\ \mathbf{skip} : \mathbf{comm}]\!] = \mathsf{q} \cdot \star.
\end{aligned}
$$

The operator **op** ranges over the usual arithmetic-logic operators, and *op* is its obvious interpretation.

2.4 A Warm-up Example

This simple example illustrates quite well the way the game-based model works. It is a toy abstract data type (ADT): a switch that can be flicked on, with implementation:

```
client : com -> exp -> com |-
    new var v:= 0 in
    let set be v := 1 in
    let get be !v in
    client (set, get) : com.
```

The code consists of local integer variable `v`, storing the state of the switch, together with functions `set`, to flick the switch on, and `get`, to get the state of the switch. The initial state of the switch is *off.* The non-local, undefined, identifier `client` is declared at the left of the turnstile `|-`. It takes a command

and an expression-returning functions as arguments. It represents, intuitively, "the most general context" in which this ADT can be used.

A key observation about the model is that the *internal state* of the program is abstracted away, and only the observable actions, of the *nonlocal* entity `client`, are represented, insofar as they contribute to terminating computations. The output of the modeling tool is given in Fig. 1.

Notice that no references to `v`, `set`, or `get` appear in the model! The model is only that of the possible behaviours of the `client`: whenever the client is executed, if it evaluates its second argument (get the state of the switch) it will receive the value 0 as a result; if it evaluates the first argument (set the switch on), one or more times, then the second argument (get the state of the switch) will always evaluate to 1. The model does not, however, assume that `client` uses its arguments, or how many times or in what order.

2.5 Full Abstraction

Full abstraction results are crucial in semantics, as they are a strong qualitative measure of the semantic model. Full abstraction is defined with respect to observational equivalence: two terms are equivalent if and only if they can be substituted in all program contexts without any observable difference in the outcome of computation. This choice of observables is therefore canonical, and arises naturally from the programming language itself. In practice, fully abstract models are important because they identify all and only those programs which are observationally equivalent.

Formally, terms M and N are defined to be observationally equivalent, written $M \equiv N$, if and only if for any context $\mathcal{C}[-]$ such that both $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are closed terms of type **comm**, $\mathcal{C}[M]$ converges if and only if $\mathcal{C}[N]$ converges. The theory of observational equivalence, which is very rich (see e.g. [9] for a discussion), has been the subject of much research [22].

Theorem 1 (Full abstraction [3, 9]). *$\Gamma \vdash M \equiv N$ iff $\mathcal{L}\big(\llbracket \Gamma \vdash M : \theta \rrbracket u_0\big) = \mathcal{L}\big(\llbracket \Gamma \vdash N : \theta \rrbracket u_0\big)$, where $u_0(x) = K_\theta$ for all $x : \theta$ in Γ.*

As an immediate consequence, observational equivalence for the finitary fragment discussed here is decidable. It can be shown that the full abstraction result holds relative to contexts drawn from either the restricted fragment or the full programming language [27].

3 Applications to Analysis and Verification

The game model is *algorithmic, fully abstract* and *compositional*, therefore it provides excellent support for compositional program analysis and verification.

The initial decidability result of the previous section was extended to higher-order (recursion and iteration-free) call-by-name procedural programming by Ong [26] and, for call-by-value, by Murawski [28]. This required the use of deterministic pushdown automata [29, 30], since the associated sets of complete

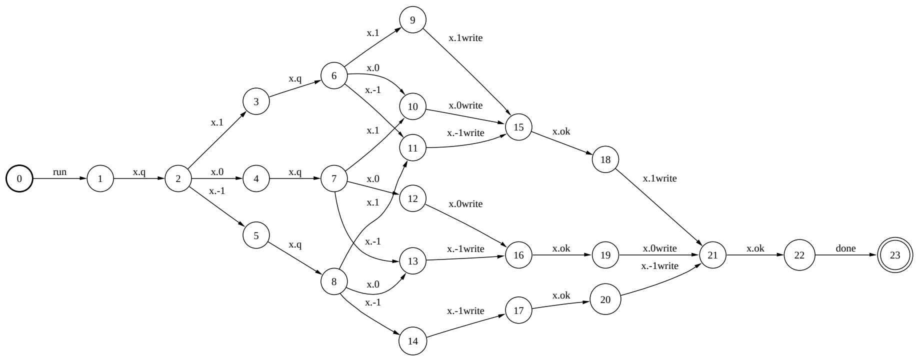

Fig. 2. A model of sorting

plays in the game semantics are no longer regular. Various other extensions of the programming fragment, e.g. by introducing unrestricted recursion [26] or further increasing the order of the fragment [31], lead to undecidability. The game-theoretic approach seems to offer a useful and powerful tool for investigating the algorithmic properties of programming language fragments, e.g. the complexity of program equivalence [32].

A different direction of research is the development of game-based, model-checking friendly specification languages. Such specification languages are necessary in order to fully exploit the compositionality of the game-based approach. It is of little use to reason about program fragments if properties of the whole program cannot be then compositionally inferred, without requiring further model-checking. The first steps in this direction are taken in [12].

3.1 Tool Support and Case Studies

The theoretical applications of game semantics have been very successful. However, since the complexity of the regular-language algorithms involved in the generation of the finite-state machines representing the game models is exponential (both in time and in space), it was unclear whether the technique was practicable. This is in fact a common situation in software model checking: the asymptotic complexity of the algorithms involved is high, but it turns out that the worst-case scenario only happens in pathological cases. Many programs can be in fact verified. But the only way to make such pragmatic assessments is to implement and experiment. We have implemented a prototype tool, and the results are very positive.

Our tool converts an open procedural program into the finite-state machine representation of the regular-language game model. Very little user instrumentation of the source code is required. The data-abstraction schemes (i.e. what finite sets of integers will be used to model integer variables) for integer-typed variables need to be supplied, using simple code annotations. The tool is implemented in CAML; most of the back-end heavy duty finite-state machine processing is done using the AT&T FSM library [33]. A more complete description of the tool is available in [34].

In the following we will present two case studies which best illustrate the distinctive features of our model: a sorting program and an abstract data type implementation.

3.2 Sorting

In this section we will discuss the modeling of a sorting program, a pathological problem for model checking because of the connection between data and control flow. We will focus on *bubble-sort*, not for its algorithmic virtues but because it is one of the most straightforward non-recursive sorting algorithms. The implementation we will analyze is the one in Fig. 3. Meta-variable n, representing the size of the array, will be instantiated to several different values. Observe that the program communicates with its environment using non-local **var**-typed identifier `x:var` only. Therefore, the model will only represent the actions of `x`. Since we are in a call-by-name setting, `x` can represent any **var**-typed procedure, for example interfacing with an input/output channel. Notice that the array being effectively sorted, `a[]`, is not visible from the outside of the program because it is locally defined.

We first generate the model for $n = 2$, i.e. an array of only 2 elements, in order to obtain a model which is small enough to display and discuss. The type of stored data is integers in the interval $[-1, 1]$, i.e. 3 distinct values. The resulting model is as in Fig. 2. It reflects the dynamic behaviour of the program in the following way: every trace in the model is formed from the actions of reading all $3 \times 3 = 9$ possible combinations of values from `x`, followed by writing out the same values, but in sorted order. Fig. 4 gives a snapshot of the model for $n = 20$.

The output is a FS machine, which can be analyzed using standard FS-based model checking tools. Moreover, this model is an *extensional* model of sorting: all sorting programs on an array of size n will have isomorphic models. Therefore, a straightforward method of verification is to compare the model of a sorting program with the model of another implementation which is known to be correct. In the case of our finite-state models, this is a decidable operation.

Increases in the array lead to (asymptotically exponential) increases in the time and space of the verification algorithm. In the table below we give several benchmark results for running the tool on our development machine (SunBlade 100, 2GB RAM). We give the execution time, the size of the largest automaton generated in the process, and the size of the final automaton. For reference, we also include the size of the state space of the program, i.e. the number of states representable by the array and the other variables in the program.

```
x:var |-
  array a[n] in
  new var i:=0 in
  while !i < n do a[!i]:=!x; i:=!i+1 od;
  new var flag:=1 in
  while !flag do
    new var i:=0 in
    flag:=0;
    while !i < n - 1 do
      if !a[!i] > !a[!i+1] then
        flag:=1;
        new var temp:=!a[!i] in
          a[!i]:=!a[!i+1];
          a[!i+1]:=!temp
        else skip fi;
      i:=!i+1
    od
  od;
  new var i:=0 in
  while !i < n do x:=!a[!i]; i:=!i+1 od : com.
```

Fig. 3. An implementation of sorting

n	Time (mins.)	State space (Max.)	State space (Model)	State space (Program)
5	5	3,376	163	60,750
10	10	64,776	948	3,542,940
15	120	352,448	2,858	96,855,122,250
20	240	1,153,240	6,393	55,788,550,416,000

For arrays larger than 30 the memory requirements could not be handled.

We can see that models have very compact sizes. Moreover, the maximum size of the work space is significantly less than that used by a (naive) state exploration algorithm. The key observation is the following: the fact that the state of the array is *internalized* and only a purely behavioural, observationally fully abstract model is presented leads to significant savings in required memory space. Moreover, the compositional nature of our construction ensures that all intermediate models are observationally fully abstract, and allows us to perform local minimizations at every step.

This kind of "observational" abstraction, which comes for free with our fully abstract model, is fundamentally different than other, syntactic and "stateful," abstraction techniques such as slicing [35].

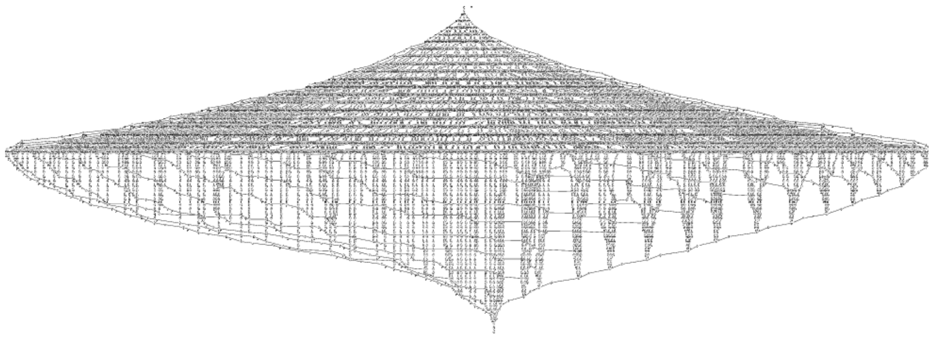

Fig. 4. A model of sorting: 20 element-array

3.3 ADT Invariants

We define an assertion as a function which takes as argument a boolean, the condition to be asserted. It does nothing if the condition is true and calls an (nonlocal) `error` procedure if the condition is false. In the resulting model, any trace containing the actions `error.run, error.done` will represent a usage of the ADT which violates the invariant, i.e. an *error trace*.

The encoding of safety properties using code-level assertions is quite standard in SMC, e.g. [19], and it is also known that every safety property can be encoded in a regular language [36]. Using the assertion mechanism in conjunction with modeling open programs, such as modules, offers an elegant solution to the problem of checking equational properties or invariants of ADTs.

For example, consider an implementation of a finite-size stack, using a fixed-size array. The interface of the stack is through functions `push(n)` and `pop`. Their implementation is the obvious one (see Fig. 5). In addition, the stack component assumes the existence of functions `overflow` and `empty` to call if a `push` is attempted on a full stack, respectively a `pop` is attempted on an empty stack. These functions need not be implemented.

Suppose that we want to check, for a size 2 stack, whether it is the case that the last value pushed onto the stack is the value at the top of the stack. We do this by using the assertion `invariant` on lines 21–24 of Fig. 5. Notice the undefined component `VERIFY` of this program: it stands for *all* possible uses of the stack module and the assertion to be checked. The idea of providing such a generic closure of an open program can be traced back to [37], and several game-like solutions have been already proposed [38, 39]. The game model which we use provides this closure, correct and complete, directly at the level of the concrete programming language.

The tool automatically builds the model for the above and extracts its shortest failure trace (see Fig. 6).

```
empty:com, overflow:com, m:exp, error:com,
VERIFY : com -> exp -> com -> com |-
      let assert be fun a : exp.
          if a then skip else error fi in
      array buffer[n] in
      let size be n in
      new var crt:=0 in
      let isempty be !crt = 0 in
      let isfull be !crt = size in
      let push be fun x : exp.
          new var temp:=x in
          if isfull then overflow
          else buffer[!crt]:=!temp;
          crt:=!crt+1 fi
          in
      let pop be
          if isempty then empty; 0
          else crt:=!crt - 1;
               !buffer[!crt] fi
          in
      let invariant be
          new var x:=m in
          push(!x); pop = !x
          in
      VERIFY(push(m), pop, assert(invariant))
: com.
```

Fig. 5. A stack module

Action `1.VERIFY` represents a `push` action. So the simplest possible error is caused by pushing 3 times the value 1 onto the 2-element stack. Indeed, if the stack is already full, pushing a new element will cause an overflow error. The failure of the assertion in this case should have been expected since the stack is finite size.

4 Limitations and Further Research

The initial results of our effort to model and verify programs using Game Semantics are very encouraging: this approach proves to give compact, practicable representations of many common programs, while the ability to model open programs allows us to verify software components, such as ADT implementations.

We are considering several further directions:

language extensions: the procedural language fragment we are currently handling only includes basic imperative and functional features. We are consid-

```
0       1       run
1       2       VERIFY.run
2       3       1.VERIFY.run
3       4       m.q
4       5       m.1
5       6       1.VERIFY.done
6       7       1.VERIFY.run
7       8       m.q
8       9       m.1
9       10      1.VERIFY.done
10      11      3.VERIFY.run
11      12      m.q
12      13      m.0
13      14      overflow.run
14      15      overflow.done
15      16      error.run
16      17      error.done
17      18      3.VERIFY.done
18      19      VERIFY.done
19      20      done
20
```

Fig. 6. Shortest failure trace of stack component

ering several ways to extend it, and the principal emphasis is on adding concurrency features. A game semantic model for shared-variable parallelism has been recently developed by our group [40]. We are also considering a version of this tool which would handle call-by-value languages.

specifications: in order to truly support compositional verification we intend to expand the tool to model *specifications* of open programs, rather than just open programs. A theoretical basis for that is already provided in [12], which is in turn inspired by the game-like ideas of *interface automata* [38].

tools and methodology: enriching the features of the tool and making it more robust and user friendly. For example, the definability result in [3] guarantees that any trace in the model can be mapped back into a program. Using this, we can give the user *code* rather than *trace* counterexamples to failed assertions. We would also like to investigate applying the tool to the modeling and verification of a larger, more realistic case study.

scalable model checking: our methods so far apply only to *finite* data and store. Verifying a program operating on finite data and store is an excellent method for bug detection and provides a fairly high measure of confidence in the correctness of the code, but it does not represent a *proof.* There is, in general, no guarantee that the properties of a program of given size generalize. But we hope that recent results in *data independence* [41, 42] can help overcome such limitations.

We are actively engaged in investigating the above topics, and we are grateful to the Engineering and Physical Sciences Research Council of the United Kingdom for financial support in the form of the research grant *Algorithmic Game Semantics and its Applications*; there is also a related project on *Scalable Software Model Checking based on Game Semantics* by Ranko Lazic of the University of Warwick.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Information and Computation **163** (2000)
2. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II and III. Information and Computation **163** (2000)
3. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In: Proc. of 1996 Workshop on Linear Logic. ENTCS **3**, Elsevier (1996) Also [22, Chap. 20].
4. Abramsky, S., McCusker, G.: Full abstraction for Idealized Algol with passive expressions. Theoretical Computer Science **227** (1999) 3–42
5. Abramsky, S., Honda, K., McCusker, G.: A fully abstract game semantics for general references. In: Proc. of LICS'98 (1998)
6. Laird, J.: Full abstraction for functional languages with control. In: Proc. of LICS'97 (1997) 58–67
7. Hankin, C., Malacaria, P.: Generalised flowcharts and games. LNCS **1443** (1998)
8. Hankin, C., Malacaria, P.: Non-deterministic games and program analysis: an application to security. In: Proc. of LICS'99 (1999) 443–452
9. Ghica, D.R., McCusker, G.: Reasoning about Idealized ALGOL using regular languages. In: Proc. of 27th ICALP. LNCS **1853**, Springer-Verlag (2000)
10. Ghica, D.R.: Regular language semantics for a call-by-value programming language. In: 17th MFPS. ENTCS **45**, Aarhus, Denmark, Elsevier (2001) 85–98
11. Ghica, D.R.: A regular-language model for Hoare-style correctness statements. In: Proc. of the Verification and Computational Logic 2001, Florence, Italy (2001)
12. Ghica, D.R.: A Games-based Foundation for Compositional Software Model Checking. PhD thesis, Queen's University School of Computing, Kingston, Ontario, Canada (2002)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
14. Schmidt, D.A.: On the need for a popular formal semantics. ACM SIGPLAN Notices **32** (1997) 115–116
15. Dill, D.L.: The Murϕ verfication system. In: Proc. of CAV'96. LNCS **1102**, Springer-Verlag (1996) 390–393
16. Holzmann, G.J., Peled, D.A.: The state of SPIN. In: Proc. of CAV'96. LNCS **1102**, Springer-Verlag (1996) 385–389
17. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S.: MOCHA: Modularity in model checking. In: Proc. of CAV'98, Springer-Verlag (1998) 521–525
18. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of 29th POPL, ACM Press (2002) pp. 58–70
19. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Proc. of CAV'01. (2001) 260–275
20. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Zheng, H.: Bandera. In: Proc. of the 22nd International Conference on Software Engineering, ACM Press (2000) 439–448

21. Reynolds, J.C.: The essence of ALGOL. In de Bakker, J.W., van Vliet, J.C., eds.: Algorithmic Languages, Proc. of the International Symposium on Algorithmic Languages, Amsterdam, North-Holland, Amsterdam (1981) 345–372 Also [22, Chap. 3].
22. O'Hearn, P.W., Tennent, R.D., eds.: ALGOL-like Languages. Progress in Theoretical Computer Science. Birkhäuser, Boston (1997) Two volumes.
23. Abramsky, S.: Algorithmic game semantics: A tutorial introduction. Lecture notes, Marktoberdorf International Summer School 2001. (2001)
24. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addidon Wesley (1979)
25. Reape, M., Thompson, H.S.: Parallel intersection and serial composition of finite state transducers. COLING-88 (1988) 535–539
26. Ong, C.H.L.: Observational equivalence of third-order Idealized Algol is decidable. In: Proc. of LICS'02. (2002) 245–256
27. Ghica, D.R., McCukser, G.: The regular-language semantics of first-order Idealized ALGOL. Theoretical Computer Science (to appear)
28. Murawski, A.S.: Variable scope and call-by-value program equivalence. in preparation (2003)
29. Senizergues: L(A) = L(B)? decidability results from complete formal systems. TCS: Theoretical Computer Science **251** (2001)
30. Stirling, C.: Deciding DPDA equivalence is primitive recursive. LNCS **2380** (2002) 821–865
31. Murawski, A.S.: On program equivalence in languages with ground-type references. In: Proc. of LICS'03, IEEE Computer Society Press (2003)
32. Murawski, A.S.: Complexity of first-order call-by-name program equivalence. submitted for publication (2003)
33. — AT&T FSM Library$^{\text{tm}}$ – general-purpose finite-state machine software tools `http://www.research.att.com/sw/tools/fsm/`.
34. Ghica, D.R.: Game-based software model checking: Case studies and methodological considerations. Technical Report PRG-RR-03-11, Oxford University Computing Laboratory (2003)
35. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher-Order and Symbolic Computation **13** (2000) 315–353
36. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In Dwork, C., ed.: Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing, Québec City, Québec, Canada, ACM Press (1990) 377–408
37. Colby, C., Godefroid, P., Jagadeesan, L.: Automatically closing open reactive programs. In: Proc. of PLDI'98, Montreal, Canada (1998) 345–357
38. de Alfaro, L., Henzinger, T.A.: Interface automata. In Gruhn, V., ed.: Proc. of the 9th ESEC/FSE-01. Software Engineering Notes **26, 5**, New York, ACM Press (2001) 109–120
39. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. of the ACM **49** (2002) 672–713
40. Ghica, D.R., Murawski, A.S.: Angelic semantics of fine-grained concurrency. In Proc. of FOSSACS'04 (2004) To appear.
41. Lazic, R.S.: A Semantic Study of Data Independence with Applications to Model Checking. PhD thesis, University of Oxford (1999)
42. Lazic, R., Nowak, D.: A unifying approach to data-independence. LNCS **1877** (2000)