Compositional Predicate Abstraction from Game Semantics[⋆]

Adam Bakewell and Dan R. Ghica

University of Birmingham, U.K.

Abstract. We introduce a technique for using conventional predicate abstraction methods to reduce the state-space of models produced using game semantics. We focus on an expressive procedural language that has both local store and local control, a language which enjoys a simple game-semantic model yet is expressive enough to allow non-trivial examples. Our compositional approach allows the verification of incomplete programs (e.g. libraries) and offers the opportunity for new heuristics for improved efficiency. Game-semantic predicate abstraction can be embedded in an abstraction-refinement cycle in a standard way, resulting in an improved version of our experimental model-checking tool Mage, and we illustrate it with several toy examples.

1 Introduction

The most important technical challenge for automatic software verification is the so-called *state-explosion problem*, the fact that the state-complexity of the model checking problem is exponential in the size of the program. As a direct consequence of this, automatic verification is said not to *scale*, i.e. only rather small programs can be handled.

A variety of techniques are used to handle the systems with very large state spaces that occur in automatic verification. Taken together, they can lead to surprisingly effective tools, which can handle fully automatically an impressive range of programs [1, 2]. But in a series of papers [3–6] we have argued that while such techniques are very effective on small to medium sized programs, in order for automatic verification to scale up to large and very large programs it is necessary to be based on *compositional* methods, i.e. have the ability to verify *fragments* of programs, then make correctness judgements about the whole based on correctness judgements about the parts. We believe game semantics [7, 8] provides a solid theoretical foundation on which such methods can be developed.

In this paper we develop a *predicate abstraction* [9] from game-based models. The technical challenge is combining the compositional and semantic-directed model construction of game semantics with the syntactic constructs of predicate abstraction and its essential use of global state. In the paper we formulate predicate abstraction for games, prove relevant technical results (decidability, soundness of approximation), discuss new heuristics stemming from this style of

⋆ Supported by EPSRC grants EP/D070880/1 and EP/D034906/1

predicate abstraction and illustrate it with some examples. The implementation is based on our existing experimental tool MAGE.[1]

2 The Language

The technique that we present here can be used to abstract any programs written in a language that has a game-semantic model. To have a focused presentation we will select a fragment of the language that is expressive enough to allow interesting examples, yet simple enough to allow a concise presentation. We call this language IAL. The starting point is IA [10], a well studied language which combines lambda calculus with the simple imperative language. We will use an enhanced variant of a language that in addition to local variables also uses block-structured control, a generalisation of C's `break` and `continue` operations. A similar language, IAX, was studied by Laird [11]. IA-like languages are supposed to use lambda-abstraction uniformly over all types, but this, in conjunction with the call-by-name procedural mechanism, leads to confusing phenomena such as *interference* or *bad variables* [12]. To avoid such issues, which raise the complexity of our presentation but are ultimately irrelevant to the matter of predicate abstraction, we impose some restrictions on the way variables and labels can be used in the language by disallowing variable and label-typed terms in the language. Variables and labels are "named constants" rather than programming language identifiers [13]. We disallow recursion and higher-order functions because they introduce infinite-state models in a way that is not related to the store. Finally to further focus the presentation on store abstraction rather than functional aspects, we only allow a very simple function-definition mechanism, similar to that of C, where all functions are defined in global scope. This language is in general quite close to a large subset of C and we are building up towards real code in the near future.

2.1 Syntax and Operational Semantics

IAL has a discrete set of labels $\mathcal{L}$ and a discrete set of locations $\mathcal{C}$. The base types T of the language are commands `com`, booleans `bool` and integers `int`. Function types are defined by the grammar $U ::= T_1 \times \cdots \times T_k \to T$. For each type T there is a discrete set of identifiers of that type $\mathcal{F}_T$. We use a distinct type `prog` for programs. The type rules of the language are given in Fig. 1, where by $\mathcal{L}(M)$ and $\mathcal{C}(M)$ we mean the set of labels and locations, respectively, used in M and by $\mathcal{V}(M)$ the set of (free) variables of a term.

The rules for new, break, continue perform the introduction of a fresh location or label name x. This is apparently syntactically restrictive, e.g. the term $\texttt{new}\ x.\texttt{new}\ x.!x$ does not type-check, but any such term can be alpha-converted to a legal term, i.e. $\texttt{new}\ x.\texttt{new}\ y.!y$.

The "big-step" operational semantics are standard for an IA-like language. Let $\mathbf{V}$ be the set of values, including natural numbers, booleans and skip, $\mathbf{V} =$

[1] `http://www.cs.bham.ac.uk/research/projects/mage`

$$\frac{x \in \mathcal{F}_T}{x : T} \qquad \frac{}{n : \mathtt{int}} \qquad \frac{}{\mathtt{true} : \mathtt{bool}} \qquad \frac{}{\mathtt{false} : \mathtt{bool}} \qquad \frac{}{\mathtt{skip} : \mathtt{com}}$$

$$\frac{M : \mathtt{com} \qquad N : T}{M ; N : T} \qquad \frac{M : \mathtt{int} \qquad N : \mathtt{int}}{M \oplus N : \mathtt{int}} \qquad \frac{B : \mathtt{bool} \qquad M_i : T}{\mathtt{if}\ B\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 : T}$$

$$\frac{x \in \mathcal{C} \qquad M : \mathtt{int}}{x\mathtt{:=}M : \mathtt{com}} \qquad \frac{x \in \mathcal{C}}{!x : \mathtt{int}} \qquad \frac{x \in \mathcal{L}}{\mathtt{goto}\ x : \mathtt{com}} \qquad \frac{f \in \mathcal{F}_{T_1 \times \cdots \times T_k \to T} \qquad M_i : T_i}{f(M_1, \ldots, M_k) : T}$$

$$\frac{M : T \qquad x \notin \mathcal{C} \setminus \mathcal{C}(M)}{\mathtt{new}\ x.M : T} \qquad \frac{M : \mathtt{com} \qquad x \notin \mathcal{L} \setminus \mathcal{L}(M)}{\mathtt{break}\ x.M : \mathtt{com}} \qquad \frac{M : \mathtt{com} \qquad x \notin \mathcal{L} \setminus \mathcal{L}(\mathcal{M})}{\mathtt{cont}\ x.M : \mathtt{com}}$$

$$\frac{M : T}{M : \mathtt{prog}} \qquad \frac{\mathcal{V}(M) = \{x_1, \ldots, x_k\} \qquad f \in \mathcal{F}_{T_1 \times \cdots \times T_k \to T} \qquad M : T \qquad N : \mathtt{prog}}{\mathtt{let}\ f(x_1, \ldots, x_k) = M\ \mathtt{in}\ N : \mathtt{prog}}$$

Fig. 1. Typing rules for IAL

$$\frac{M, \Sigma \Downarrow \mathtt{skip}, \Sigma' \qquad N, \Sigma' \Downarrow E, \Sigma''}{M; N, \Sigma \Downarrow E, \Sigma''} \qquad \frac{M, \Sigma \Downarrow G, \Sigma'}{M; N, \Sigma \Downarrow G, \Sigma''}$$

$$\frac{M, \Sigma \Downarrow m, \Sigma' \qquad N, \Sigma' \Downarrow G, \Sigma''}{M \oplus N, \Sigma \Downarrow G, \Sigma''} \qquad \frac{M, \Sigma \Downarrow G, \Sigma'}{M \oplus N, \Sigma \Downarrow G, \Sigma''}$$

$$\frac{M, \Sigma \Downarrow m, \Sigma' \qquad N, \Sigma' \Downarrow n, \Sigma'' \qquad p = m \oplus n}{M \oplus N, \Sigma \Downarrow p, \Sigma''}$$

$$\frac{B, \Sigma \Downarrow b, \Sigma' \qquad M_b, \Sigma' \Downarrow E, \Sigma''}{\mathtt{if}\ B\ \mathtt{then}\ M_{true}\ \mathtt{else}\ M_{false}, \Sigma \Downarrow E, \Sigma''} \qquad \frac{B, \Sigma \Downarrow G, \Sigma'}{\mathtt{if}\ B\ \mathtt{then}\ M_{true}\ \mathtt{else}\ M_{false}, \Sigma \Downarrow G, \Sigma'}$$

$$\frac{M, \Sigma \Downarrow m, \Sigma'}{x\mathtt{:=}M, \Sigma \Downarrow \mathtt{skip}, \Sigma'[x \mapsto m]} \qquad \frac{M, \Sigma \Downarrow G, \Sigma'}{x\mathtt{:=}M, \Sigma \Downarrow G, \Sigma'} \qquad \frac{}{!x, \Sigma \Downarrow \Sigma(x), \Sigma}$$

$$\frac{M, \Sigma \otimes (x \mapsto 0) \Downarrow E, \Sigma' \otimes (x \mapsto n)}{\mathtt{new}\ x.M, \Sigma \Downarrow E, \Sigma'} \qquad \frac{M, \Sigma \Downarrow \mathtt{goto}\ x, \Sigma'}{\mathtt{break}\ x.M, \Sigma \Downarrow \mathtt{skip}, \Sigma'} \qquad \frac{M, \Sigma \Downarrow \mathtt{skip}, \Sigma'}{\mathtt{break}\ x.M, \Sigma \Downarrow \mathtt{skip}, \Sigma'}$$

$$\frac{M, \Sigma \Downarrow \mathtt{goto}\ x, \Sigma' \qquad \mathtt{cont}\ x.M, \Sigma' \Downarrow E, \Sigma''}{\mathtt{cont}\ x.M, \Sigma \Downarrow E, \Sigma''} \qquad \frac{M, \Sigma \Downarrow \mathtt{skip}, \Sigma'}{\mathtt{cont}\ x.M, \Sigma \Downarrow E, \Sigma'}$$

$$\frac{P, \mathcal{U} \otimes (f \mapsto F), \Sigma \Downarrow E, \Sigma'}{\mathtt{let}\ f(x_1, \ldots, x_k) = F\ \mathtt{in}\ P, \mathcal{U}, \Sigma \Downarrow E, \Sigma'} \qquad \frac{\mathcal{U}(f) = F \qquad F[M_i/x_i], \mathcal{U}, \Sigma \Downarrow E, \Sigma'}{f(M_1, \ldots, M_k), \mathcal{U}, \Sigma \Downarrow E, \Sigma'}$$

Fig. 2. Operational semantics for IAL

$\mathbf{N} + \mathbf{B} + \mathbf{1}$. Let $\mathbf{G} = \{\mathtt{goto}\ x \mid x \in \mathcal{L}\}$ be the set of non-local jumps. Let the set of *final forms* be $\mathbf{E} = \mathbf{V} + \mathbf{G}$. We assume $V \in \mathbf{V}, G \in \mathbf{G}$, etc. We also use an *environment* $\mathcal{U}$ which is a map from function-identifiers to terms. Let $\Sigma : \mathcal{C} \to \mathbb{Z}$ be a *store*, let $\Sigma \otimes (x \mapsto n)$ represent the extension of Σ to domain $\mathcal{C} + \{x\}$ such that $\Sigma \otimes (x \mapsto n)(x) = n$, and let $\Sigma[x \mapsto n]$ be a store equal to Σ except that $\Sigma[x \mapsto n](x) = n$. The operational semantics of the language are relations of the form $M, \mathcal{U}, \Sigma \Downarrow E, \Sigma'$, meaning term M in environment $\mathcal{U}$ and state Σ evaluates to final form $E \in \mathbf{E}$ and final state Σ'. If the environment is not used in the rule it will be omitted, for simplicity. The operational semantics is given in Fig. 2. Note that continue is expressive enough to encode iteration: $\mathtt{while}\ M\ \mathtt{do}\ N \equiv \mathtt{cont}\ y.\mathtt{if}\ M\ \mathtt{then}\ N; \mathtt{goto}\ y\ \mathtt{else}\ \mathtt{skip}$. Also note that a notion of abnormal termination can be encoded with `goto abort`, where `abort` is a reserved label. With `abort`, assertions can be encoded as $\mathtt{assert}(M) \equiv \mathtt{if}\ M\ \mathtt{then}\ \mathtt{skip}\ \mathtt{else}\ \mathtt{goto}\ \mathtt{abort}$.

2.2 Game Semantics

In this section we will present a game-like model along the lines of [14], but with the important distinction that state will be modelled explicitly in a way rather similar to [15] and [16]. We can do this because of the greatly simplified role that locations can play in the language. The absence of `var`-type terms makes interference and bad variables impossible and supports a *global store* model.

A state $\Sigma : A \to \mathbb{Z}$ maps a set of names A to integer values. Given an alphabet $\mathcal{A}$ and a set of names A, a stateful sequence $s^{\Sigma\Sigma'}$ consists of a sequence $s \in \mathcal{A}^*$ and two states $\Sigma, \Sigma' : A \to \mathbb{Z}$. If $s = \epsilon$, the empty sequence, we require $\Sigma = \Sigma'$. We define the following operations on sets of stateful sequences, i.e. stateful languages: $S \cdot T = \left\{(s \cdot t)^{\Sigma\Sigma''} \mid s^{\Sigma\Sigma'} \in S, t^{\Sigma'\Sigma''} \in T\right\}$. Also,

$$S^{(0)} = \{\epsilon^{\Sigma,\Sigma} \mid \Sigma : A \to \mathbb{Z}\}, \quad S^{(k)} = S \cdot S^{(k-1)}, \quad S^* = \bigcup_{k\in\mathbb{N}} S^{(k)}.$$

If t, u are stateless sequences then we define $t \cdot s^{\Sigma\Sigma'} \cdot u = (t \cdot s \cdot u)^{\Sigma\Sigma'}$.

With every type U of the language we associate an alphabet $[\![U]\!]$:

$$[\![\mathtt{int}]\!] = \{q\} \cup \mathbb{Z}, \quad [\![\mathtt{bool}]\!] = \{q, t, f\}, \quad [\![\mathtt{com}]\!] = \{q, a\}.$$

For function types we have

$$[\![T_1 \times \cdots \times T_n \to T]\!] = \sum_{i=1,n} [\![T_i]\!] + [\![T]\!]. \tag{1}$$

Terms $M : T$ are modelled by languages over alphabet

$$\mathcal{A}_M = [\![T]\!] + \sum_{\substack{U \ s.t. \\ \mathcal{V}(M)\cap\mathcal{F}_U \neq \emptyset}} [\![U]\!] + \sum_{y\in\mathcal{L}(M)} \mathsf{go}^y. \tag{2}$$

To make the disjoint sum more explicit, we syntactically tag elements of $[\![U]\!]$ with the identifier x. The symbols in the alphabet are the so-called game-semantic “moves”. They represent the *observable* actions that a term can perform. Every language that denotes a meaning of a term has a certain form, given by all its possible initial and final moves, called *bracketing moves.* If the final action belongs to the normal alphabet associated with the type, the trace is a complete computation leading to value a, and we denote it by $(\!|M|\!)_a$. Another possible final action is go^x for some label x and it denotes an attempt to jump out of the scope of the term; we denote such traces $\{\!|M|\!\}$. The meaning of terms at ground type can be decomposed as:

$$\begin{aligned}
[\![M : \mathtt{com}]\!] &= q \cdot (\!|M|\!)_a \cdot a + q \cdot \{\!|M|\!\} \\
[\![M : \mathtt{bool}]\!] &= q \cdot (\!|M|\!)_t \cdot t + q \cdot (\!|M|\!)_f \cdot f + q \cdot \{\!|M|\!\} \\
[\![M : \mathtt{int}]\!] &= \sum_{n\in\mathbb{Z}} q \cdot (\!|M|\!)_n \cdot n + q \cdot \{\!|M|\!\}.
\end{aligned}$$

$$
\begin{aligned}
&\llparenthesis \texttt{skip} \rrparenthesis = \epsilon^{\Sigma\Sigma}, \quad \{\!|\texttt{skip}|\!\} = \emptyset\\
&\llparenthesis n \rrparenthesis_n = \epsilon^{\Sigma\Sigma}, \quad \llparenthesis m \rrparenthesis_n = \emptyset \text{ if } m \neq n, \quad \{\!|u|\!\} = \emptyset\\
&\llparenthesis M_1; M_2 \rrparenthesis_p = \llparenthesis M_1 \rrparenthesis \cdot \llparenthesis M_2 \rrparenthesis_p, \quad \{\!|M_1; M_2|\!\} = \{\!|M_1|\!\} + \llparenthesis M_1 \rrparenthesis \cdot \{\!|M_2|\!\}\\
&\llparenthesis M_1 \oplus M_2 \rrparenthesis_p = \sum_{\substack{m,n,p\in\mathbb{Z}\\ m\oplus n = p}} \llparenthesis M_1 \rrparenthesis_m \cdot \llparenthesis M_2 \rrparenthesis_n, \quad \{\!|M_1 \oplus M_2|\!\} = \{\!|M_1|\!\} + \sum_{m\in\mathbb{Z}} \llparenthesis M_1 \rrparenthesis_m \cdot \{\!|M_2|\!\}\\
&\llparenthesis x \texttt{:=} M \rrparenthesis = \sum_{n\in\mathbb{Z}} \left(\llparenthesis M \rrparenthesis_n^{\Sigma\Sigma'}\right)^{\Sigma\Sigma'[x\mapsto n]}, \quad \{\!|x\texttt{:=}M|\!\} = \{\!|M|\!\}\\
&\llparenthesis !x \rrparenthesis_n = \epsilon^{\Sigma\Sigma} \text{ if } \Sigma(x) = n, \quad \llparenthesis !x \rrparenthesis_n = \emptyset \text{ if } \Sigma(x) \neq n, \quad \{\!|!x|\!\} = \emptyset\\
&\llparenthesis \texttt{new}\ x.M \rrparenthesis = \left(\llparenthesis M \rrparenthesis^{\Sigma\otimes(x\mapsto 0),\Sigma'\otimes(x\mapsto n)}\right)^{\Sigma,\Sigma'}, \quad \{\!|\texttt{new}\ x.M|\!\} = \left(\{\!|M|\!\}^{\Sigma\otimes(x\mapsto 0),\Sigma'\otimes(x\mapsto n)}\right)^{\Sigma,\Sigma'}\\
&\llparenthesis \texttt{if}\ M\ \texttt{then}\ M_1\ \texttt{else}\ M_2 \rrparenthesis_a = \llparenthesis M \rrparenthesis_t \cdot \llparenthesis M_1 \rrparenthesis_a + \llparenthesis M \rrparenthesis_f \cdot \llparenthesis M_2 \rrparenthesis_a,\\
&\qquad\qquad \{\!|\texttt{if}\ M\ \texttt{then}\ M_1\ \texttt{else}\ M_2|\!\} = \{\!|M|\!\} + \llparenthesis M \rrparenthesis_t \cdot \{\!|M_1|\!\} + \llparenthesis M \rrparenthesis_f \cdot \{\!|M_2|\!\}\\
&\llparenthesis \texttt{goto}\ x \rrparenthesis_a = \emptyset, \quad \{\!|\texttt{goto}\ x|\!\} = (\mathsf{go}^x)^{\Sigma\Sigma}\\
&\llparenthesis \texttt{break}\ x.M \rrparenthesis_a = \llparenthesis M \rrparenthesis_a + \{\!|M|\!\}_x, \quad \{\!|\texttt{break}\ x.M|\!\} = \{\!|M|\!\}_y \cdot \mathsf{go}^y, x \neq y\\
&\llparenthesis \texttt{cont}\ x.M \rrparenthesis_a = \{\!|M|\!\}_x^* \cdot \llparenthesis M \rrparenthesis_a, \quad \{\!|\texttt{cont}\ x.M|\!\} = \{\!|M|\!\}_x^* \cdot \{\!|M|\!\}_y \cdot \mathsf{go}^y, x \neq y.
\end{aligned}
$$

Fig. 3. Game-semantic evaluations

Intuitively, $\llparenthesis M \rrparenthesis_a, \{\!|M|\!\}$ are the observable effects of the actual computation that M carries out in order to produce a or jump, respectively.

For a term M we define a pattern-matching operator that extracts traces with a given initial and final states $\llparenthesis M \rrparenthesis^{\Sigma\Sigma'} \triangleq \{s \mid s^{\Sigma\Sigma'} \in \llparenthesis M \rrparenthesis\}$ and similarly for $\{\!|M|\!\}$. We also use the notation $\{\!|M|\!\} \triangleq \{\!|M|\!\}_x \cdot \mathsf{go}^x$ and we implicitly sum over all states Σ. Most of the semantic valuations are given in Fig. 3.

Note that constants have no observable side-effects and cannot jump. For `break`, normal termination is either the normal termination of M or a jump to the breaking label x; any other termination can only be a jump. For `continue`, any jump to x causes a restart of M, until it terminates normally or until it jumps to a different location than x.

As in [14] we only give a game-semantic definition for function application of a free function identifier, i.e. a function where the definition is not known. We choose not to present function application in general because it is too complex for this presentation, unrelated to the issue of predicate abstraction. In the absence of recursion β-redexes (i.e. function calls with known definitions) can be reduced operationally. It is fair to say that the entire apparatus of game semantics and the entire development to this point is necessary only insofar as it allows the formulation of this rule:

$$
\llparenthesis f(M_1, \ldots, M_n) \rrparenthesis_k = q^f \cdot \left(\sum_{i=1}^{n} \sum_{a \in [\![T_i]\!]} q^{fi} \cdot \llparenthesis M_i \rrparenthesis_a \cdot a^{fi} \right)^* \cdot k^f
$$

$$\{\!|f(M_1,\ldots,M_n)|\!\} = q^f \cdot \left(\sum_{i=1}^{n}\sum_{a\in[\![T_i]\!]} q^{fi}\cdot (\!|M_i|\!)_a \cdot a^{fi}\right)^* \cdot \left(\sum_{i=1}^{n} q^{fi}\cdot \{\!|M_i|\!\}\right).$$

Moves q^f, k^f are markers delineating the overall beginning and end of computation. Moves q^{fi}, a^{fi} are markers delineating the beginning and execution of each argument. A normal execution of a function is an arbitrary sequence of executions of its arguments. If one of the arguments causes a non-local jump then the function call terminates with that non-local jump. The locality of the jumps ensures that all jumps from M_is are either local or outside of the scope. It is not possible for arguments to cause jumps to each other.

Finally, for completeness, if $x : T$ is a base-type free variable then $(\!|x|\!)_a = x$ and $\{\!|x|\!\} = \emptyset$: its meaning is an unspecified action labelled with the variable.

We are mainly interested in proving safety properties. Suppose that there is a special label called `abort`. A term is abort-free if it has no occurrence of `goto abort`. We say that a term M is *safe* if for any abort-free context with a hole $\mathcal{C}[-]$ and for any state Σ we have $\mathcal{C}[M], \Sigma \Downarrow E, \Sigma'$, $E \neq$ `goto abort`. The connection between the operational and game semantics is given by:

Theorem 1. *A term of* IAL M *is safe if and only if* $\{\!|M|\!\}_{\texttt{abort}} = \emptyset$.

The proof of this result is routine, similar to that in [4].

Example 1. Show `new x.f(c; x := !x + 2, assert (!x % 2 <> 0))` is safe.

This example illustrates the uniqueness of the game-semantic approach, because it requires reasoning about a non-trivial interaction between non-local function `f`, non-local procedure `c` and the store. The set of locations is $\mathcal{L} = \{x\}$ and the state is $\Sigma : \{x\} \to \mathbb{Z}$. For simplicity we denote the function $(x \mapsto n)$ simply as n. Following simple calculations we have

$$(\!|\texttt{c;x:=!x+2}|\!) = c\cdot \epsilon^{n,n+2} = c^{n,n+2} \qquad (\!|\texttt{c;x:=!x+2}|\!) = \emptyset$$
$$(\!|\texttt{assert(!x\%2!=0)}|\!) = \epsilon^{2k,2k} \qquad \{\!|\texttt{assert(!x\%2!=0)}|\!\} = (\mathsf{go}^{\mathsf{abort}})^{2k+1,2k+1}.$$

Applying `f` gives:

$$(\!|\texttt{f(c;x:=!x+2,assert(!x\%2<>0))}|\!)$$
$$= q^f \cdot \left(\sum_n q^{f1}\cdot c^{n,n+2}\cdot a^{f1} + \sum_k q^{f2}\cdot \epsilon^{2k,2k}\cdot a^{f2}\right)^* \cdot a^f$$
$$\{\!|\texttt{f(c;x:=!x+2,assert(!x\%2<>0))}|\!\}$$
$$= q^f\cdot \left(\sum_n q^{f1}\cdot c^{n,n+2}\cdot a^{f1} + \sum_k q^{f2}\cdot\epsilon^{2k,2k}\cdot a^{f2}\right)^* \cdot \left(\sum_k q^{f2}\cdot(\mathsf{go}^{\mathsf{abort}})^{2k+1,2k+1}\right).$$

By a simple inductive argument, $\left(\sum_n q^{f1}\cdot c^{n,n+2}\cdot a^{f1} + \sum_k q^{f2}\cdot \epsilon^{2k,2k}\cdot a^{f2}\right)^*$ always produces traces of the form $s^{n,n+2k}$, therefore

$$(\!|\texttt{new x.f(c;x:=!x+2,assert(!x\%2<>0))}|\!) = q^f\cdot\left(q^{f1}\cdot c\cdot a^{f1} + q^{f2}\cdot a^{f2}\right)^*\cdot a^f$$
$$\{\!|\texttt{new x.f(c;x:=!x+2,assert(!x\%2<>0))}|\!\} = \emptyset,$$

since the rule for `new` forces the initial state to be 0 and removes the (only) location x from the state. According to Thm. 1 this means the term is safe. Note that if we take the set of integers to be finite the set of state annotations is also finite and the formula can be mechanically verified.

3 Predicate Abstraction

The key problem of automatic software verification is that the set of all possible states $\varSigma$ is very large. If we restrict IAL to finite k-bit integers, then a set of states over n variables has, obviously, 2^{nk} elements. Predicate abstraction in game semantics is about reducing the size of this set, in a way that is compatible with the compositional (denotational) structure of the semantics and which can still model the subtle interplay between store and procedural behaviour. To further simplify the presentation we will only consider predicate abstraction for assignments that only use *pure expressions* on the RHS, i.e. expressions that do not change the state while returning a value. This is not a substantial restriction, as all programs can be converted to that form using assignment to intermediate values.

We abstract a state $\varSigma$ in the standard way (e.g. [9]) by representing it as a set of predicates over $\mathrm{dom}(\varSigma)$. If σ is approximated by p predicates then the number of possible values is 2^p, which can be far smaller than 2^{nk}. We denote an abstracted state by $\varPsi$.

We introduce the following notations. Given a set of states $\mathcal{S} = \{\varSigma \mid \varSigma : \mathcal{L} \to \mathbb{Z}\}$ over locations $\mathcal{L}$, let $\mathbb{P}_{\mathcal{L}}$ be the set of all predicates definable using its locations as variables, and let $\mathcal{P}_{\mathcal{L}} \in \mathbb{P}^*_{\mathcal{L}}$ a (finite) list of its elements, constituting the predicate abstraction of $\mathcal{S}$. The predicates $\varPsi \in \mathcal{P}_{\mathcal{L}}$ are called *abstract states.* A set of abstract states is *satisfiable* written $\mathrm{sat}(\varPsi_0, \ldots, \varPsi_k)$ if there is an assignment of their variables that makes each $\varPsi_i$ true; we call such predicates that are simultaneously satisfiable *compatible.*

We define pa-traces similar to stateful traces, $s^{\varPsi\varPsi'}$, with concatenation of pa-languages defined as $S \cdot T = \{(st)^{\varPsi\varPsi'} \mid s^{\varPsi\varPsi_0} \in S, t^{\varPsi'_0\varPsi'} \in T, \mathrm{sat}(\varPsi_0, \varPsi'_0)\}$. Note that concatenation of pa-traces is non-deterministic, due to the possible choices for $\varPsi_0, \varPsi'_0$, unlike stateful trace concatenation which is deterministic. Exponentiation and iterated closure are defined similarly to stateful traces.

Let $\mathcal{E}_N$ and $\mathcal{E}_B$ be the languages of integer and boolean expressions constructed from constants, arithmetic and logic operators and uninterpreted variables. The predicate-abstracted semantics is defined in terms of pa-traces and is structurally similar to that of the original game semantics. $\overline{[\![\texttt{int}]\!]} = \{q\} \cup \mathcal{E}_N$, $\overline{[\![\texttt{bool}]\!]} = \{q\} \cup \mathcal{E}_B$, $\overline{[\![\texttt{com}]\!]} = \{q, a\}$. Function-type and term alphabets are analogously to Eqns. 1 and 2. Note that the sub-alphabet of result moves is expanded from the set of all *values* of a given type to the set of all *syntactic expressions* over $\mathcal{L}$ of a given type. Trace decompositions are analogous to game semantics: $\overline{[\![M : \texttt{com}]\!]} = q \cdot \overline{(\!|M|\!)}_a \cdot a + q \cdot \overline{\{\!|M|\!\}}$, and so on for the other types. The semantic rules for constants, sequential composition, control and function application are also analogous to those of the original game semantics. We only present the rules that are substantially different: branching, arithmetic and logic, assignment and dereferencing, local variable.

We introduce the notation $\overline{\llparenthesis M \rrparenthesis}_B^{\langle \Psi_0 \Psi_1 \rangle} \triangleq \left(\overline{\llparenthesis M \rrparenthesis}_B^{\Psi_0 \Psi_1}\right)^{\Psi_0 \Psi_1}$ to identify particular traces. Note that $\overline{\llparenthesis - \rrparenthesis}^{\Psi_0, \Psi_1}$ is a trace-selection operator whereas $(-)^{\Psi_0,\Psi_1}$ is an annotation. The pa-semantics of branching is:

$$\begin{aligned}
&\overline{\llparenthesis \texttt{if } M \texttt{ then } M_1 \texttt{ else } M_2 \rrparenthesis}_a \\
&\qquad = \sum_{\substack{B \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1, B)}} \overline{\llparenthesis M \rrparenthesis}_B^{\langle \Psi_0 \Psi_1 \rangle} \cdot \overline{\llparenthesis M_1 \rrparenthesis}_a + \sum_{\substack{B' \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1', \neg B')}} \overline{\llparenthesis M \rrparenthesis}_{B'}^{\langle \Psi_0' \Psi_1' \rangle} \cdot \overline{\llparenthesis M_2 \rrparenthesis}_a \\
&\overline{\{\!|\texttt{if } M \texttt{ then } M_1 \texttt{ else } M_2|\!\}} \\
&\qquad = \overline{\{\!|M|\!\}} + \sum_{\substack{B \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1, B)}} \overline{\llparenthesis M \rrparenthesis}_B^{\langle \Psi_0 \Psi_1 \rangle} \cdot \overline{\{\!|M_1|\!\}} + \sum_{\substack{B' \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1', \neg B')}} \overline{\llparenthesis M \rrparenthesis}_{B'}^{\langle \Psi_0' \Psi_1' \rangle} \cdot \overline{\{\!|M_2|\!\}},
\end{aligned}$$

Note that the guard M evaluates to a *syntactic* expression B, rather than a value. The branch to be executed is chosen depending on whether the expression is compatible with the state or whether its negation is. Note that it is possible that both conditions are satisfied, case in which the branching becomes non-deterministic. Arithmetic and logic operators evaluate to a syntactic expression rather than a value:

$$\begin{aligned}
\overline{\llparenthesis M_1 \oplus M_2 \rrparenthesis}_{E_1 \oplus E_2} &= \overline{\llparenthesis M_1 \rrparenthesis}_{E_1} \cdot \overline{\llparenthesis M_2 \rrparenthesis}_{E_2}, \\
\overline{\{\!|M_1 \oplus M_2|\!\}} &= \overline{\{\!|M_1|\!\}} + \overline{\llparenthesis M_1 \rrparenthesis} \cdot \overline{\{\!|M_2|\!\}}.
\end{aligned}$$

The rule for assignment is:

$$\overline{\llparenthesis x \texttt{:=} M \rrparenthesis} = \sum_{E \in \mathcal{E}_N} \left(\overline{\llparenthesis M \rrparenthesis}_E^{\Psi \Psi'}\right)^{\Psi \Psi''}, \quad \overline{\{\!|x \texttt{:=} M|\!\}} = \overline{\{\!|M|\!\}},$$

where $\mathrm{sat}(\Psi'', \Psi'[E/x])$.

Note that after assignment a new pa-state Ψ'' must be chosen, which is compatible to the old state in which x has become E. Note that the choice of Ψ'' can introduce non-determinism in the interpretation. The pa-semantics of assignment is non-deterministic, unlike the game-semantic interpretation.

Dereferencing returns x, seen as a syntactic expression:

$$\overline{\llparenthesis !x \rrparenthesis}_x = \epsilon^{\Psi \Psi}, \quad \overline{\llparenthesis !x \rrparenthesis}_E = \emptyset \text{ if } E \neq x, \quad \overline{\{\!|!x|\!\}} = \emptyset.$$

Local variable introduction must cope with the fact that the bound variable cannot appear outside of its context, therefore the pa-states inside the block must come from a different set than that used outside the block, which uses a smaller set of locations.

$$\begin{aligned}
\overline{\llparenthesis \texttt{new } x.M \rrparenthesis} &= \left(\overline{\llparenthesis M \rrparenthesis}^{\Psi_0 \Psi_1}\right)^{\Psi_0', \Psi_1'}, \\
\overline{\{\!|\texttt{new } x.M|\!\}} &= \left(\overline{\{\!|M|\!\}}^{\Psi_0 \Psi_1}\right)^{\Psi_0', \Psi_1'}.
\end{aligned}$$

where $\Psi_0', \Psi_1' \in \mathcal{P}_{\mathcal{L}}$, $\Psi_0, \Psi_1 \in \mathcal{P}_{\mathcal{L}+\{x\}}$, $\mathrm{sat}(\Psi_0, \Psi_0', x = 0), \mathrm{sat}(\Psi_1, \Psi_1')$. As in the case of assignment, the choice of updated state is not necessarily deterministic. It is assumed that local variables are initialized to zero.

Example 2. We reconsider the Example 1 program, using pa-semantics to show that `new x.f(c; x := !x + 2, assert (!x % 2 <> 0))` is safe.

Assume the singleton predicate set $\mathcal{P}_{\{x\}} = \{\mathrm{even(x)}\}$, so the only possible abstracted states are $e = \mathrm{even(x)}$ and $o = \neg\mathrm{even(x)}$. Following simple calculations we have

$$\overline{\llparenthesis \texttt{!x+2} \rrparenthesis}_{x+2} = \epsilon^{o,o} + \epsilon^{e,e}, \quad \overline{\{\!| \texttt{!x+2} |\!\}}_{x+2} = \emptyset, \quad \overline{\llparenthesis \texttt{!x+2} \rrparenthesis}^{o,o}_{x+2} = \epsilon, \quad \overline{\llparenthesis \texttt{!x+2} \rrparenthesis}^{e,e}_{x+2} = \epsilon$$

$$\overline{\llparenthesis \texttt{x:=!x+2} \rrparenthesis} = \left(\overline{\llparenthesis \texttt{!x+2} \rrparenthesis}^{o,o}_{x+2}\right)^{o,\Psi} + \left(\overline{\llparenthesis \texttt{!x+2} \rrparenthesis}^{e,e}_{x+2}\right)^{e,\Psi'},$$

where $\mathrm{sat}(\neg\mathrm{even}(x+2), \Psi)$, $\mathrm{sat}(\mathrm{even}(x+2), \Psi')$, so $\Psi = e, \Psi' = o$. Therefore $\overline{\llparenthesis \texttt{x:=!x+2} \rrparenthesis} = \epsilon^{o,o} + \epsilon^{e,e}$. Also, $\overline{\{\!| \texttt{x:=!x+2} |\!\}} = \emptyset$. The two arguments are

$$\overline{\llbracket \texttt{c;x:=!x+2} \rrbracket} = q \cdot \overline{\llparenthesis \texttt{c;x:=!x+2} \rrparenthesis} \cdot a + q \cdot \overline{\{\!| \texttt{c;x:=!x+2} |\!\}} = qca^{o,o} + qca^{e,e}$$
$$\overline{\llbracket \texttt{assert(!x\%2<>0)} \rrbracket} = q \cdot a^{e,e} + (q \cdot \mathsf{go}^{\mathsf{abort}})^{o,o}.$$

Applying f gives:

$$\overline{\llbracket \texttt{f(c;x:=!x+2,assert(!x\%2<>0))} \rrbracket}$$
$$= q^f \cdot \left(q^{f1} \cdot c^{e,e} \cdot a^{f1} + q^{f1} \cdot c^{o,o} \cdot a^{f1} + q^{f2} \cdot \epsilon^{e,e} \cdot a^{f2}\right)^* \cdot a^f$$
$$\overline{\{\!| \texttt{f(c;x:=!x+2,assert(!x\%2<>0))} |\!\}}$$
$$= q^f \cdot \left(q^{f1} \cdot c^{e,e} \cdot a^{f1} + q^{f1} \cdot c^{o,o} \cdot a^{f1} + q^{f2} \cdot \epsilon^{e,e} \cdot a^{f2}\right)^* \cdot \left(q^{f2} \cdot (\mathsf{go}^{\mathsf{abort}})^{o,o}\right).$$

Obviously the iteration $\left(q^{f1} \cdot c^{e,e} \cdot a^{f1} + q^{f1} \cdot c^{o,o} \cdot a^{f1} + q^{f2} \cdot \epsilon^{e,e} \cdot a^{f2}\right)^*$ always produces either traces of the form $s^{e,e}$ or $s^{o,o}$, therefore

$$\overline{\llbracket \texttt{new x.f(c;x:=!x+2,assert(!x\%2<>0))} \rrbracket} = q^f \cdot \left(q^{f1} \cdot c \cdot a^{f1} + q^{f2} \cdot a^{f2}\right)^* \cdot a^f$$
$$\overline{\{\!| \texttt{new x.f(c;x:=!x+2,assert(!x\%2<>0))} |\!\}} = \emptyset,$$

since the rule for `new` forces the initial state to be compatible with $x = 0$ (i.e. $\Psi = e$); the outer set of predicates is defined over the empty set of locations and is omitted. Note that in this (not typical) case the pa-semantics and the game semantics give the same interpretation.

3.1 Formal Properties

The technical hurdle is formulating the pa-semantics; with the definitions in place proving the relevant technical properties is a routine exercise.

This ancillary result is important towards proving decidability.

Proposition 1. *For any $M : T$, $\overline{T} \in \{\texttt{bool}, \texttt{int}\}$ there is only a finite set of syntactic expressions E such that $\llparenthesis M : T \rrparenthesis_E \neq \emptyset$.*

Proposition 2 (Decidability). *If $\mathcal{P}_{\mathcal{L}}$ is finite then $\overline{\{\!|M|\!\}} = \emptyset$ is decidable.*

The proof relies on the fact that, given a finite set of pa-state annotations Ψ, and considering the finitary encoding of the alphabet (from Prop. 1) the pa-semantics accepts a regular-language formulation. Let us write $\overline{\{\!|M|\!\}}_{\langle\mathcal{P}\rangle}$ when we need to emphasise that a pa-semantics is over predicate set $\mathcal{P}$.

Proposition 3 (Monot.). *If $\mathcal{P} \subseteq \mathcal{P}'$ and $\overline{\{\!|M|\!\}}_{\langle\mathcal{P}\rangle} = \emptyset$ then $\overline{\{\!|M|\!\}}_{\langle\mathcal{P}'\rangle} = \emptyset$*

The contra-positive has an immediate proof; if $\overline{\{\!|M|\!\}}_{\langle\mathcal{P}'\rangle}$ has a trace then removing a predicate does not invalidate any of the satisfiablity conditions, therefore $\overline{\{\!|M|\!\}}_{\langle\mathcal{P}\rangle}$ will have a trace. The monotonicity property states that "improving" the abstraction does not remove any possible failure traces.

Proposition 4 (Correctness). *If $\mathcal{P} = \{x = n \mid x \in \mathcal{L}, n \in \mathbb{Z}\}$ then $\overline{\{\!|M|\!\}} = \emptyset$ iff $\{\!|M|\!\} = \emptyset$.*

The proof is immediate, as the Ψ's are a precise predicate representations of the state Σ in the stateful formulation of the game semantic model.

From Correctness and Monotonicity, along with Thm. 1 it follows that

Theorem 2 (Soundness). *If $\overline{\{\!|M|\!\}} = \emptyset$ then M is safe.*

Thm. 2 and Prop. 2 state that any finite PA semantics is a sound and effective approximation for the concrete semantics, and it can be used for automatic proving of safety properties of IAL terms.

4 Heuristics

Our experimental model checker MAGE implements automatic verification algorithms within the framework described in previous sections. We use a finite set of predicates of form $\Psi = \bigwedge_{P\in\mathcal{P}} \delta_i(P)$ where each δ_i is either the identity or the negation operation and each P is a proposition over the set of locations. Such predicates can be efficiently represented as bit-vectors.

4.1 Internal and External Compositionality

Games-based verification tools are compositional in the sense that they can handle open terms (see earlier examples); call this kind of compositionality *external*. Additionally, games-based tools are *internally compositional*, i.e. the model of a term is built inductively from the models of its sub-terms. This approach is helpful because it allows the modification of the abstraction scheme within the term being constructed; some branches of the syntax tree can be heavily abstracted while others can be much more precise. This feature has been discussed in the context of games-based data-abstraction and refinement [4, 6], and it can be used again to great effect with predicate abstraction by changing the predicate set within the term.

4.2 Flexibility and Efficiency

The pa-semantics allows the predicate set to change at every composition point in the program — i.e. between every sub-term and its successor in the parse tree. This flexibility can be exploited by removing from the pa-state Ψ predicates P deemed irrelevant and reintroducing them whenever precision needs to be improved. It is well known that minimizing the predicate set size is essential to avoid a state explosion; moreover, n predicate-bits are typically much more expensive to maintain than an n-bit state in conventional model checking, because each bit represents an arbitrarily complex predicate.

Predicate annotations. Running a satisfiability check over a current state and all possible next-states, and allowing the entire predicate set to change at every composition is too expensive in general. To make our treatment of variable abstraction schemes more perspicuous, we shall use syntactic annotations "`newp`" and "`endp`" to delimit predicate scopes at the level of the source code. This is important because for any assignment we can track the state-change precisely by only using two predicates (i.e. a two-bit vector), representing the state before and after the assignment. For example, in the code fragment below no more than two predicates are needed at any one time to track the following execution accurately and validate the assertion (`c:com` is a free procedure identifier).

```
newp (x = 0); x := 0; c;
newp (x = 1); x := !x + 1; c; endp (x = 0);
newp (x = 11); x := !x + 10; c; endp (x = 1);
newp (x = 111); x := !x + 100; c; endp (x = 11);
assert(x = 111); endp (x = 111)
```

Another simplifying restriction we impose is that predicate scopes are well nested and use instead "`letp` p `in` M" annotations. Next-state calculation (which predicates can change valuation at a given program point), and identification of relevant predicates for current-state tests become much simpler because the predicates form a stack that can be mapped into the standard program stack. On the other hand, a disadvantage of nested scoping is that a series of overlapping scopes cannot always be kept tight. In the same example the maximum size of the bit vector used to represent the pa-state is now four:

```
letp (x = 0) in x := 0; c;
letp (x = 1) in x := !x + 1; c;
letp (x = 11) in x := !x + 10; c;
letp (x = 111) in x := !x + 100; c;
assert(x = 111)
```

4.3 Predicate Scope

Our experimental tool, Mage resolves satisfiability tests with the external SMT engine Yices[2]. Two fully automatic predicate annotation schemes are used, both in the compositional "`letp`" style:

1. All (pure) conditional expressions in the program are predicates in the model, each given maximal scope to maximize the chance of a successful check (but expensive in situations where a scope contains many conditionals).
2. Conditionals are made predicates with minimal scope initially; checking is therefore much faster and much more likely to be inconclusive. A refinement loop is added to expand the scope of some predicates (see Section 4.4).

It is this issue of *predicate scope* that the internal compositionality of the game-based formulation exposes and allows to be manipulated to the advantage of the verification process: for success the selected predicates must be both adequate and have sufficient scope; but too many predicates with excessive scope will make checking infeasible.

Example 3 (Number magic). The magician asks the stooge to think of an n, double it, add 50, divide by 2, subtract n and add 100. The `distract:com` procedure is just that, an irrelevant diversion.

```
new m.new n.n := stooge(); distract;
m := !n + !n;  distract; assert(m = 2 * n);
m := !m + 50;  distract; assert(m = 50 + 2 * n);
m := !m / 2;   distract; assert(m = 25 + n);
m := !m - !n;  distract; assert(m = 25);
m := !m + 100; distract; assert(m = 125)
```

The checks on magic answer m provide the intermediate invariants needed to prove the trick by predicate abstraction and the distractions prevent Mage treating the assignment sequence as a single basic block (which makes the proof trivial)!

This is easily verified by Mage using maximally scoped predicates. But notice that with five conditionals and hence five predicates (and no clever optimizations) the satisfiability of 2^5 possible next-states must be tested at each composition point in the pa-model where the state can change. This takes around 250 seconds.

Mage can also verify the term much faster, in about 2 seconds, using minimally scoped predicates, widened just enough to include the assignment that makes it true and the assertion that declares it. This is achieved by preserving the valuations of predicates from the end of their `letp` scope until the next assignment. For example, in `letp (m = 25) in (m := !m - !n; distract(); assert(m = 25));` at the assignment-composition, the valuation `m = 25 + n` from the previous scope is compatible only with next-state `m = 25`, as per the definition of pa-trace concatenation.

[2] `http://yices.csl.sri.com/`

Hence the problem of having to nest overlapping scopes is eliminated as the model of `letp` p `in` M_1`;` `letp` p `in` M_2 can reach exactly the same states as that of `letp` p `in (`M_1`;`M_2`)` because pa-trace concatenation always kicks in at the boundary between `letp`s.

There is considerable potential for further speed up the implementation by developing tighter solver integration; the incremental-SMT approach to predicate abstraction is fully compatible with our approach and would shrink run-times by a substantial factor although it cannot in itself avoid the state explosions.

4.4 Counter-Example Guided Scope Refinement

Counterexample certification checks trace *feasibility* of a pa-trace. Pa-trace concatenation offers a *local* check of compatibility between pa-states, but each such concatenation point is a source of non-determinism. After several such concatenations the global trace from start to end may be actually not possible.

Consider for example the following code, with abstracting predicates written explicitly as program annotation in `letp` style:

```
letp (x > y) in letp (x < y) in letp (z <> 2) in
      x := n; y := m;
      if !x > !y then z := 1 else z := 0;
      if !x < !y then z := !z + 1 else skip;
      assert(!z <> 2)
```

The second assignment to `z` may go from a pa-state in which `z <> 2` to a pa-state in which `z <> 2` or to one in which `z = 2`. Because these distinctions cannot be made locally, the feasibility check must involve a *global* satisfiability test of the entire concatenation. In this case, we must check whether all predicates `x1 = n, y1 = m, x1 <= y1, z1 = 0, x1 < y1, z2 = z1 + 1, z2 = 2` are compatible.

Refinement is realised by modifying the predicate abstraction with the aim of eliminating some infeasible counterexamples and the guarantee of not introducing new infeasible counterexamples. In our framework it can be achieved by adding predicates or extending the scope of existing predicates. Mage in predicate-scope refinement mode begins by using the guard of each if statement (implicit in assert) as a tightly-scoped predicate, represented by the annotation:

$$\texttt{if } B \texttt{ then } M \texttt{ else } N \Rightarrow \texttt{if (letp } B \texttt{ in } B\texttt{) then } M \texttt{ else } N$$

Note that the unannotated term and the tight annotation have identical models as the predicates never live long enough to be absorbed into the global state. So for the simple algorithm in Mage (with no interpolation or other methods for adding predicates that are not conditionals), refinement now simply means extending the scope of some `letp` that appears in an infeasible counterexample!

Example 4. Consider the safe term

```
new x.new y.new z.
      x := !i; y := !i;
      if letp (x > y) in !x > !y
      then z := !x - !y; assert(letp (z > 0) in !z > 0)
      else skip
```

Checking generates a counterexample that needs to satisfy `x1 = i1, y1 = i2, x1 > y1, z1 = x1 - y1, z1 > 0` which is infeasible. Maximizing the two predicate scopes would eliminate the counterexample in one step, but by expanding them more gradually we arrive at the following provably safe scopes:

```
new x.new y.new z.x := !i; y := !i;
      letp (x > y) in if !x > !y
      then letp (z > 0) in z := !x - !y; assert(!z > 0)
      else skip
```

The tight scopes require a running time of 0.75 sec. while the loose scope requires 7.5 sec. of execution.

Note that this scheme is guaranteed to terminate and the program will be verified if it is verifiable when annotated with maximally-scoped conditionals. The gradual scope expansion creates two problems: more refinement iterations and the need for heuristics regarding which `letp` to expand. By a depth-first expansion scheme the tightest safe annotation will be found. This makes ours an alternative approach to the idea of *predicate minimization* used in other tools [17] so the burden of the expanded predicates on verification in other parts of the model should be minimized. The problem of more iterations is mitigated if model checking on the intermediate refinements is restricted to testing for the presence of the infeasible counterexample.

5 Further Work

We believe we are only beginning to exploit the new possibilities of attacking state-space explosion through internal compositionality. The next instance of the tool will incorporate at least a form of refinement by delayed SMT: we will extend the semantics and tool to allow the precision of a fixed pa to be gradually improved by delaying SMT tests for up to a specified number of concatenations. This achieves the same effect as expanding the scope of each predicate without actually increasing the state, but at the cost of more false counterexamples.

There is now a real potential to combine game-semantic models with more complex state-representation assertion languages, such as those arising from separation logic [18] and target heap-oriented programs compositionally. We are currently examining this, as well as game-semantic models of more realistic programming languages.

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) 1–20
2. Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: SPIN. (2005) 25–26
3. Abramsky, S., Ghica, D.R., Murawski, A.S., Ong, C.H.L.: Applying game semantics to compositional software modeling and verification. In: TACAS. (2004) 421–435
4. Dimovski, A., Ghica, D.R., Lazic, R.: Data-abstraction refinement: A game semantic approach. In: SAS. (2005) 102–117
5. Ghica, D.R., Murawski, A.S.: Compositional model extraction for higher-order concurrent programs. In: TACAS. (2006) 303–317
6. Bakewell, A., Ghica, D.R.: On-the-fly techniques for game-based software model checking. In: TACAS. (2008) 78–92
7. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. **163**(2) (2000) 409–470
8. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. **163**(2) (2000) 285–408
9. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In Halbwachs, N., Peled, D., eds.: CAV. Volume 1633 of Lecture Notes in Computer Science., Springer (1999) 160–171
10. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. Electr. Notes Theor. Comput. Sci. **3** (1996)
11. Laird, J.: A fully abstract game semantics of local exceptions. In: LICS. (2001) 105–114
12. Reynolds, J.: The craft of programming. Prentice-Hall Intl. (1981)
13. Pitts, A.M.: Reasoning about local variables with operationally-based logical relations. In O'Hearn, P.W., Tennent, R.D., eds.: Algol-Like Languages. Volume 2. Birkhauser (1997) 173–193 Reprinted from *Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.
14. Ghica, D.R., McCusker, G.: The regular-language semantics of second-order Idealized Algol. Theor. Comput. Sci. **309**(1-3) (2003) 469–502
15. Ong, C.H.L.: Observational equivalence of 3rd-order Idealized Algol is decidable. In: LICS. (2002) 245–256
16. Laird, J.: A game semantics of names and pointers. Annals of Pure and Applied Logic **151**(2-3) (February 2008) 151–169 First Games for Logic and Programming Languages Workshop.
17. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: In Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME, Springer (2003)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. (2002) 55–74