Assume-Guarantee Software Verification Based on Game Semantics*

Aleksandar Dimovski and Ranko Lazić

Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
{aleks, lazic}@dcs.warwick.ac.uk

Abstract. We show how game semantics, counterexample-guided abstraction refinement, assume-guarantee reasoning and the L^* algorithm for learning regular languages can be combined to yield a procedure for compositional verification of safety properties of open programs. Game semantics is used to construct accurate models of subprograms compositionally. Overall model construction is avoided using assume-guarantee reasoning and the L^* algorithm, by learning assumptions for arbitrary subprograms. The procedure has been implemented, and initial experimental results show significant space savings.

1 Introduction

One of the most effective methods for automated software verification is model checking [8]. A software system to be verified is modelled as a finite-state transition system and a property to be established is expressed as a temporal logic formula. Given that the state explosion problem is particulary acute in software model checking, the most desirable feature of this approach is *scalability*. *Compositional modelling and verification* achieve scalability by breaking up a larger software system in smaller systems which can be modelled and verified independently. Hence, the properties of a program can be established from the properties of its individually checked components without requiring to check the whole program as an atomic "flat" entity.

Game semantics meets the first requirement for achieving scalability: *compositional modelling*. Game semantics is denotational, i.e. defined recursively on the syntax, therefore the model of a larger program is constructed from the models of its subprograms, using a notion of strategy composition. The other benefits that game semantics brings to software model checking, compared with classical state-based approaches [6,17], are:

Modularity. There is a model for any open program, which enables verification of program fragments which contain free variable and procedure names.

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

Z. Liu and J. He (Eds.): ICFEM 2006, LNCS 4260, pp. 529–548, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Correctness. The generated model is fully abstract (sound and complete), i.e., two programs have the same models if and only if they cannot be distinguished with respect to operational tests (such as abnormal termination) in any program context. This means that the model can be used to deduce properties of programs (soundness), and moreover every observable property of programs is captured by the model (completeness).

Efficiency. Programs are modelled by how they interact with their environments. Details of their internal state during computations are not recorded, which results in small models with a maximum level of abstraction.

Assume-guarantee reasoning addresses the second challenge: *compositional verification*. To check that a property P is satisfied by a model M composed of two components M_1 and M_2, it suffices to find an assumption A such that

1. the composition of M_1 and A satisfies P, and
2. M_2 is a refinement of A

If such an assumption A can be found and it is significantly smaller than M_2, then we can verify whether M satisfies P (by checking 1 and 2) without having to build the whole M.

In this paper, we describe an automatic procedure which generates assumptions as above using the L^* algorithm for learning a game strategy. L^* iteratively learns a minimal deterministic finite automaton, which represents the unknown strategy, from membership and equivalence queries. In each iteration, L^* produces a candidate assumption A which is used to check 1 and 2. Depending on results of the checks, we may conclude that the required property is satisfied, violated in which case a witness counterexample is reported, or the current A needs to be revised. This procedure is set within an abstraction refinement loop which automatically extracts a game-semantic model from a data-abstracted program and refines the program if a spurious counterexample is found.

Programs are abstracted through approximating infinite integer data types by partitionings. Any partitioning contains a finite number of partitions, i.e. sets of integers, which are called abstracted integers. Abstracted programs operationally behave like their concrete counterparts, but an abstracted integer argument in any operation is nondeterministically replaced by some concrete integer from its set of integers (partition), and the concrete integer result is replaced by the abstracted integer (partition) to which it belongs. As shown in [11], this is a conservative abstraction. By quotienting over abstracted integers, the models become finite and can be model-checked. Whenever a spurious counterexample is found, it is used to refine the partitionings of the program, by splitting some of their partitions.

We have implemented this approach in the GameChecker tool [12]. We report some initial experimental results, which indicate significant memory savings compared to a non assume-guarantee approach.

The paper is organized as follows. After discussing related work, Section 2 introduces the programming language we are considering. Game semantics of the language is presented in Section 3, followed by a description of the L^* algorithm

in Section 4. Details of the verification framework are given in Section 5. Finally, we present the implementation in Section 6, and conclude in Section 7.

1.1 Related Work

Game semantics emerged in the last decade as a potent framework for modelling programming languages [2,3,16,18]. The first applications of game-semantic models to model checking were proposed in [14,1,10]. They were then extended by adapting the counterexample-guided abstraction refinement technique to this setting [11]. A tool (GameChecker) based on these ideas was presented in [12].

The assume-guarantee paradigm is the best studied approach to compositional reasoning [20]. The primary difficulty in applying this approach to realistic systems is that, in general, the appropriate assumptions have to be constructed manually.

The work presented in this paper is motivated by a recently proposed approach [9], which uses learning algorithms to automate assume-guarantee reasoning. In [9], a variant of Angluin's L^* algorithm [5,21] for learning a regular language is used to generate appropriate assumptions. Compared to this approach, which is applied at the design level of a software system, our work makes the following contributions. Firstly, we apply the method at the implementation level, and verify safety properties of open program fragments. Secondly, while in [9] the method is used for verifying multi-threaded programs by building models and checking their constituting threads independently, here we apply compositional verification on sequential programs where individually checked components can be arbitrary subprograms of the given input program. Then, the L^* algorithm is adapted to the specific game semantics setting for learning a game strategy. Finally, the method is integrated with a counterexample-guided abstraction refinement style loop. We thus obtain a procedure which embodies both compositional modelling and compositional verification.

The L^* learning algorithm has found a number applications to automatic verification. For example, adaptive model checking [15] uses learning to compute an accurate finite state model of an unknown system starting from an approximate model; substitutability analysis of evolving software systems [7] verifies an upgraded software system by learning; [4] uses a symbolic implementation of the L^* algorithm for compositional reasoning about symbolic modules, etc.

2 The Programming Language

The language which will be considered, Abstracted Idealized Algol (AIA), is an expressive programming language combining usual imperative features, locally-scoped variables and call-by-name procedures. It also incorporates data abstraction annotations, which enable the writing of abstracted programs in a syntax similar to that of concrete programs.

The data types of AIA are booleans and *abstracted integers* ($D ::= \mathsf{bool} \mid \mathsf{int}_\pi$). The phrase types are expressions, variables and commands ($B ::= \mathsf{exp}D \mid \mathsf{var}D \mid \mathsf{com}$) plus functions ($T ::= B \mid T \to T$).

The *abstractions* π range over computable finite partitionings of the integers $\mathbb{Z}$. Any such partitioning consists of a finite number of partitions (i.e. sets of integers). To say that $m, n \in \mathbb{Z}$ are in the same partition of π, we write $m \approx_\pi n$. In particular, we use the following abstractions:

$$[\,] = \{\mathbb{Z}\} \qquad [n, m] = \{<n, \{n\}, \{n+1\}, \ldots, \{0\}, \ldots, \{m-1\}, \{m\}, >m\}$$

where $<n = \{n' \mid n' < n\}$, $>n = \{n' \mid n' > n\}$. Instead of $\{n\}$, we may write just n. Abstractions are refined by splitting abstract values: $[\,]$ to $[0, 0]$ by splitting $\mathbb{Z}$, $[n, m]$ to $[n-1, m]$ by splitting $<n$, or to $[n, m+1]$ by splitting $>m$.

We write $\Gamma \vdash M : T$ to indicate that term M with free identifiers in Γ has type T. The syntax of the language is defined by the standard typing rules for forming and applying functions ($\lambda\, x.M$, MN), augmented with rules for logic and arithmetic ($M \,\mathsf{op}\, N$), branching ($\mathsf{if}\, B \,\mathsf{then}\, M \,\mathsf{else}\, N$), iteration ($\mathsf{while}\, B \,\mathsf{do}\, M$), sequencing (M ; N, expressions with side effects are also allowed), assignment ($M := N$), de-referencing ($!M$), local variable declaration ($\mathsf{new}D\; x := M \; in \, N$), then "do nothing" command (skip), and a command which causes abnormal termination (abort). The typing rules can be found in [11].

The operational semantics is defined as a big-step reduction relation $M, s \Longrightarrow \mathcal{K}$, where M is a term whose free identifiers are assignable variables (i.e. of type var), s is a state which assigns data values to the free variables, and $\mathcal{K}$ is a final configuration. The final configuration can be either a pair V, s' with V a value (i.e. a language constant or an abstraction $\lambda\, x : T.M$) and s' a state, or a special error configuration $\mathcal{E}$.

The reduction rules are similar to those for IA, with two differences. First, whenever an integer value n with data type int_π participates in an operation, any other integer n' can be used nondeterministically so long as $n' \approx_\pi n$.

$$\frac{N_1, s_1 \Longrightarrow n_1, s_2 \qquad N_2, s_2 \Longrightarrow n_2, s}{N_1 \,\mathsf{op}\, N_2, s_1 \Longrightarrow n', s}\; n'_i \approx_{\pi_i} n_i,\; n' \approx_\pi n'_1 \,\mathsf{op}\, n'_2$$

Assignment and de-referencing have similar non-deterministic rules.

Second, the abort program with any state reduces to $\mathcal{E}$, and a composite program reduces to $\mathcal{E}$ if a subprogram is reduced to $\mathcal{E}$.

$$\mathsf{abort}, s \Longrightarrow \mathcal{E} \qquad \frac{M, s \Longrightarrow \mathcal{E}}{M \,\mathsf{op}\, N, s \Longrightarrow \mathcal{E}}$$

A term M of type com is said to *terminate* in state s if there exists configuration $\mathcal{K}$ such that $\mathcal{K} = \mathcal{E}$, or $\mathcal{K} = \mathsf{skip}, s'$ for some state s' such that $M, s \Longrightarrow \mathcal{K}$. M is *safe* iff it cannot be reduced (from any state) to $\mathcal{E}$.

Term $\Gamma \vdash M : T$ *approximates* term $\Gamma \vdash N : T$, denoted by $\Gamma \vdash M \precsim N$ if and only if for all contexts $\mathcal{C}[-] : \mathsf{com}$, i.e. terms with a hole such that $\vdash \mathcal{C}[M] : \mathsf{com}$ and $\vdash \mathcal{C}[N] : \mathsf{com}$ are well formed closed terms of type com, if $\mathcal{C}[M]$ may terminate abnormally (resp. successfully) then $\mathcal{C}[N]$ also may terminate abnormally (resp. successfully). If two terms approximate each other they are considered safe-equivalent, denoted by $\Gamma \vdash M \cong N$.

A context is safe if it does not include occurrences of the abort command. A term is *safe* if for any safe context $\mathcal{C}_{\mathrm{safe}}[-]$ program $\mathcal{C}_{\mathrm{safe}}[M]$ is safe; otherwise the term is *unsafe*.

3 Game Semantics of AIA

In this section we review the fundamental concepts of game semantics for call-by-name programming languages [3].

Game semantics is denotational semantics which models types as *games*, computation as *plays* of a game, and programs as *strategies* for a game. In this approach, a kind of game is played by two participants. The first, Player, represents the program under consideration, while the second, Opponent, represents the environment (context) in which the program is used. The two take turns to make moves, each of which is either a question (a demand for information) or an answer (the supply of information).

We now proceed by presenting game semantics formally. A game is played in an *arena* which can be thought of as a playing area setting out basic rules and conventions for the game.

Definition 1. *An* arena *A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where:*

- *M_A is a countable set of* moves
- *$\lambda_A : M_A \to \{\mathsf{O},\mathsf{P}\} \times \{\mathsf{Q},\mathsf{A}\}$ is a labelling function which indicates whether a move is by* Opponent(O) *or* Player(P)*, and whether it is a* question(Q) *or an* answer(A)*. We write λ_A^{OP} for the composite of λ_A with the left projection, so that $\lambda_A^{\mathsf{OP}}(m) = \mathsf{O}$ if $\lambda_A(m) = \mathsf{OQ}$ or $\lambda_A(m) = \mathsf{OA}$. λ_A^{QA} is defined as λ_A followed by the right projection in a similar way. We denote by $\overline{\lambda}_A$ the labelling with O/P part reversed, i.e. $\overline{\lambda}_A^{\mathsf{OP}}(m) = \mathsf{O}$ iff $\lambda_A^{\mathsf{OP}}(m) = \mathsf{P}$.*
- *$\vdash_A$ is a binary relation between $M_A + \{*\}$ ($* \notin M_A$) and M_A, called* enabling *(if $m \vdash_A n$ we say that m enables move n), which satisfies the following conditions:*
  - *Initial moves (a move enabled by $*$ is called* initial*) are Opponent questions, and they are not enabled by any other moves besides $*$;*
  - *Answer moves can only be enabled by question moves;*
  - *Two participants always enable each others moves, never their own (i.e. an Opponent move can only enable a Player move and vice versa).*

A *justified sequence* in arena A is a finite sequence of moves of A together with a pointer from each non-initial move n to an earlier move m such that $m \vdash_A n$. We say that n is (explicitly) justified by m. A *legal play* is a justified sequence with some additional constraints: *alternation* (Opponent and Player moves strictly alternate), *well-bracketed* condition (when an answer is given, it is always to the most recent question which has not been answered), *visibility* condition (a move to be played depends upon a certain subsequence of the play so far, rather than on all of it), and *haltness* (no moves can follow an *abort* move). The set of all legal plays in arena A is denoted by L_A.

We say that n is *hereditarily justified* by a move m in a legal play s if there is a subsequence of s starting with m and ending in n such that every move is justified by the preceding move in it. We write $s\lceil m$ for the subsequence of s containing all moves hereditarily justified by m. We similarly define $s\lceil I$ for

a set I of initial moves in s to be the subsequence of s consisting of all moves hereditarily justified by a move of I.

Definition 2. *A game is a structure* $A = \langle M_A, \lambda_A, \vdash_A, P_A \rangle$ *where* $\langle M_A, \lambda_A, \vdash_A \rangle$ *is an arena, and* P_A *is a non-empty, prefix-closed subset of* L_A*, called the* valid plays*, such that for* $s \in P_A$ *and* I *a set of initial moves of* s *we have* $s \lceil I \in P_A$*.*

Example 1. The simplest game is the empty game $I = \langle \emptyset, \emptyset, \emptyset, \epsilon \rangle$, where ϵ is the empty sequence. The base types are interpreted by the following games:

$M_{[\![\mathsf{exp}D]\!]} = \{q, abort, n \mid n \in D\}$ $\quad$ $M_{[\![\mathsf{com}]\!]} = \{run, done, abort\}$

$\lambda(q) = \mathsf{OQ},\ \lambda(n, abort) = \mathsf{PA}$ $\quad$ $\lambda(run) = \mathsf{OQ},\ \lambda(done, abort) = \mathsf{PA}$

$* \vdash_{[\![\mathsf{exp}D]\!]} q;\quad q \vdash_{[\![\mathsf{exp}D]\!]} n, abort$ $\quad$ $* \vdash_{[\![\mathsf{com}]\!]} run;\quad run \vdash_{[\![\mathsf{com}]\!]} done, abort$

$P_{[\![\mathsf{exp}D]\!]} = \{\epsilon, q, q \cdot abort, q \cdot n \mid n \in D\}$ $\quad$ $P_{[\![\mathsf{com}]\!]} = \{\epsilon, run, run \cdot done, run \cdot abort\}$

$$M_{[\![\mathsf{var}D]\!]} = \{read, n, write(n), ok, abort \mid n \in D\}$$
$$\lambda_{[\![\mathsf{var}D]\!]}(read, write(n)) = \mathsf{OQ},\ \lambda_{[\![\mathsf{varD}]\!]}(n, ok, abort) = \mathsf{PA}$$
$$* \vdash_{[\![\mathsf{var}D]\!]} read, write(n);\quad read \vdash_{[\![\mathsf{var}D]\!]} n, abort;\quad write(n) \vdash_{[\![\mathsf{var}D]\!]} ok, abort$$
$$P_{[\![\mathsf{var}D]\!]} = \big\{\epsilon, read, write(n), read \cdot \{n, abort\}, write(n) \cdot \{ok, abort\} \mid n \in D\big\}$$

Thus in the game $[\![\mathsf{exp}D]\!]$, there is an initial move q (a question: "What is the value of the expression?") and corresponding to it a value from D or *abort* [1] (an answer to the question). In the game $[\![\mathsf{com}]\!]$, there is an initial move *run* to initiate a command, an answer move *done* to signal successful termination of a command, and *abort* to signal abnormal termination. In the game $[\![\mathsf{var}D]\!]$, for each $n \in D$ there is an initial move $write(n)$, representing an assignment. There are two possible responses to this move: *ok*, which signals successful completion of the assignment, and *abort*. For dereferencing, there is an initial move *read*, to which Player may respond with any element of D or *abort*. □

Given games A and B, we define new games $A \times B$ and $A \multimap B$ as follows:

$M_{A \times B} = M_A + M_B$ (disjoint union)

$\lambda_{A \times B} = [\lambda_A, \lambda_B]$

$* \vdash_{A \times B} n \Leftrightarrow * \vdash_A n \vee * \vdash_B n$

$m \vdash_{A \times B} n \Leftrightarrow m \vdash_A n \vee m \vdash_B n$

$P_{A \times B} = P_A + P_B$

$M_{A \multimap B} = M_A + M_B$

$\lambda_{A \multimap B} = [\overline{\lambda}_A, \lambda_B]$

$* \vdash_{A \multimap B} n \Leftrightarrow * \vdash_B n$

$m \vdash_{A \multimap B} n \Leftrightarrow m \vdash_A n \vee m \vdash_B n \vee [* \vdash_B m \wedge * \vdash_A n]$

$P_{A \multimap B} = \{s \mid s \upharpoonright A \in P_A, s \upharpoonright B \in P_B\}$

where $s \upharpoonright A$ is the subsequence of s consisting of moves from M_A. A valid play of $A \times B$ is either a play from A or a play from B. Valid plays of $A \multimap B$ are interleavings of single plays from A and B, and each such play has to begin in B and only Player can switch between the interleaved plays.

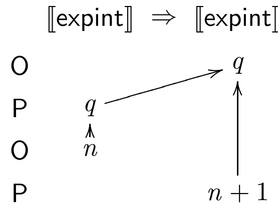
[1] Since expressions with side effect are allowed in AIA, evaluating an expression may indeed abort.

Given a game A, we define the game $!A$ as follows: $M_{!A} = M_A$, $\lambda_{!A} = \lambda_A$, $\vdash_{!A} = \vdash_A$ and $P_{!A} = \{s \in L_{!A} \mid \text{for each initial move m, } s\lceil \mathrm{m} \in \mathrm{P_A}\}$. Hence, legal plays of $!A$ are interleavings of a finite number of plays from P_A. Finally, the arena $A \Rightarrow B$ is defined as $!A \multimap B$. From now on, we work with (*well-opened*) games where initial moves can only happen at the first move.

Definition 3. *A* strategy σ *for a game* A *(written as* $\sigma : A$*) is a prefix-closed non-empty set of even-length plays in* P_A*.*

A strategy specifies what options Player has at any given point of a play, and it does not restrict the Opponent moves. We say that a play in σ is *complete* if either the opening question is answered, or the special move *abort* has been played.

Example 2. The only strategy for the empty game I is the empty strategy $\bot = \{\epsilon\}$. For the game $[\![\mathsf{expint}]\!]$, there is the empty strategy, and one strategy interpreting each natural number n, namely $\{\epsilon, q \cdot n\}$. A strategy which interprets the successor function $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$ is as follows:



Here Opponent begins a play by asking for output of succ, and Player replies asking for input. When Opponent provides input n (which can be any number since a strategy does not restrict O moves), Player will give output $(n + 1)$. The above strategy can be represented as a regular language (pointers are disregarded) $\sum_{n \in \mathsf{int}} q \cdot q \cdot n \cdot (n + 1)$, where suitable closure operator is applied. □

The notion of composition of strategies is central to game semantics: just as small programs can be put together to form large ones, so strategies can be composed to form new strategies. Strategies compose in a way which is reminiscent of the two stage process of "parallel composition plus hiding" in CSP [22].

Given a strategy $\sigma : A \Rightarrow B$, we define its *promotion* $\sigma^\dagger : !A \multimap !B$, which can play several interleaved copies of σ, by:

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial m, } s\lceil \mathrm{m} \in \sigma\}$$

Let $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ are two strategies. Then the composition $\sigma \fatsemi \tau : A \Rightarrow C$ is defined as $\sigma^\dagger; \tau$, where ; is linear composition of strategies.

Given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, the linear composition $\sigma; \tau : A \multimap C$ is defined in the following way. For a sequence u of moves from games A, B, C with justification pointers, we define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C (if a pointer from one of these points

to a move of A, delete that pointer). Similarly define $u \upharpoonright A, B$. We say that u is an *interaction sequence* of A, B, C if $u \upharpoonright A, B \in P_{A \multimap B}$ and $u \upharpoonright B, C \in P_{B \multimap C}$. The set of all such sequences is written as $int(A, B, C)$.

The parallel composition is defined by

$$\sigma \parallel \tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B \in \sigma,\ u \upharpoonright B, C \in \tau\}$$

So $\sigma \parallel \tau$ consists of sequences generated by playing σ and τ in parallel, making them synchronize on moves in B.

Suppose $u \in int(A, B, C)$. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C, but where there was a pointer from a move $m_A \in M_A$ to an initial move $m \in I_B$ extend the pointer to the initial move in C which was pointed to from m. Thus, we complete the definition of composition by hiding the interaction between σ and τ in B.

$$\sigma;\ \tau = \{u \upharpoonright A, C \mid u \in \sigma \parallel \tau\}$$

The *identity strategy* $id_A : A \Rightarrow A$ for a game A is defined by

$$\{s \in P_{A \Rightarrow A} \mid \forall\, s' \sqsubseteq^{even} s.\ s' \upharpoonright A_l = s' \upharpoonright A_r\}$$

where we use the l and r tags to distinguish between the two occurrences of A and $s' \sqsubseteq^{even} s$ means that s' is an even-length prefix of s. So, in any identity strategy id_A, a move by Opponent in either occurrence of A is immediately copied by Player to the other occurrence.

A term $\Gamma \vdash M : T$, where $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, is interpreted by a strategy $[\![\Gamma \vdash M : T]\!]$ for the game:

$$[\![\Gamma \vdash T]\!] = [\![T_1]\!] \times \ldots \times [\![T_n]\!] \Rightarrow [\![T]\!]$$

Language constants and constructs are interpreted by strategies and compound terms are modelled by composition of the strategies that interpret their constituents. For example, some of the strategies [16] are: $[\![\mathsf{n} : \mathsf{expint}]\!] = \{\epsilon, q \cdot n\}$, $[\![\mathsf{skip} : \mathsf{com}]\!] = \{\epsilon, run \cdot done\}$, $[\![\mathsf{abort} : \mathsf{com}]\!] = \{\epsilon, run \cdot abort\}$, free identifiers are interpreted by identity strategies, etc.

Using standard game-semantic techniques, it has been shown in [11] that this model is fully abstract for AIA.

Theorem 1 (Full abstraction). *For any terms $\Gamma \vdash M, N : T$, we have $\Gamma \vdash M \cong N$ iff $[\![\Gamma \vdash M : T]\!] = [\![\Gamma \vdash N : T]\!]$.*

We say that a play is *safe* if it does not terminate in *abort*, and a strategy if it consists only of safe plays; otherwise, we will call plays and strategies *unsafe.* From the full abstraction result, it follows that:

Corollary 1 (Safety). *$\Gamma \vdash M : T$ is safe iff $[\![\Gamma \vdash M : T]\!]$ is safe.*

This result ensures that, for any term, model-checking its strategy for safety (i.e. for unreachability of the *abort* move) is equivalent to proving the safety of a term.

In the rest of the paper, we work with the 2nd-order recursion-free fragment of AIA (i.e., AIA_2). The 2nd-order restriction means that the function types are restricted to $T ::= B \mid B \to T$. Also, without loss of generality, we only consider terms in β-normal form. For this language fragment, terms define strategies for which justification pointers are uniquely determined by plays, and they can be disregarded. Thus, it has been shown in [11] that:

Proposition 1. *For any finitely abstracted AIA_2 term $\Gamma \vdash M : T$, the strategy $[\![\Gamma \vdash M : T]\!]$ is a regular language.*

Example 3. Consider the term [2]

$$f : \mathsf{com} \to \mathsf{com} \;\vdash\; \mathsf{newint}\, x := 0\, \mathsf{in} \\ f(x := x+1)\,; \\ \mathsf{if}\,(x == 0)\,\mathsf{then\ abort};$$

in which x is a local block-allocated variable and f is a non-local (safe) procedure. The procedure-call mechanism is by-name, so every call to the first argument of f increments x.

The strategy interpreting this term is shown in Fig 1, where dashed edges indicate moves of the Opponent and solid edges moves of the Player. Accepting states are designated by an interior circle. The states whose interior circles are filled in, correspond to complete plays in the strategy. We use subscripts to indicate the component of the context ($[\![\Gamma]\!]$), i.e. the free identifier, to which a move belongs to. For example, the subscript '$f, 1$' denotes that a move corresponds to the first argument of the procedure f. The model illustrates only the possible behaviors of this term: if the non-local procedure f does not evaluate its argument at all then the term terminates abnormally; otherwise if f calls its argument, one or more times, then the term terminates successfully. Notice that no references to the variable x appear in the model because it is locally defined and so not visible from the outside of the term. □

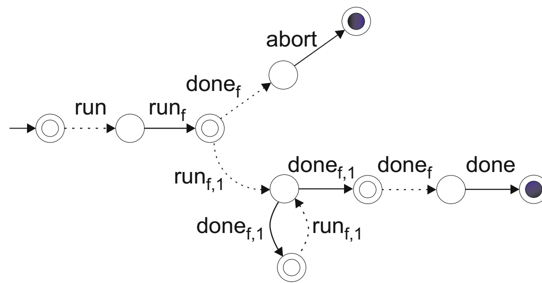


Fig. 1. A strategy as a finite automaton

Definition 4. *If σ and τ are strategies for a game A, we define a binary relation,* refinement, *$\leq$ as: $\sigma \leq \tau \Leftrightarrow \sigma \subseteq \tau$*

[2] This is actually an IA_2 term since no data-abstractions are applied to x.

4 The Learning Algorithm

Central to our compositional verification procedure is an algorithm for learning strategies, which can be represented as regular languages (see Proposition 1). The algorithm is an adaptation of the L^* algorithm introduced by Angluin [5] which learns an unknown regular language. Since L^* needs to learn strategies, the adaptation will consider only non-empty prefix-closed sets of even-length sequences (words) in which Opponent and Player moves alternate, thus achieving greater efficiency.

Let $A = \langle M_A, \lambda_A, \vdash_A, P_A \rangle$ be a game. Let $\mathsf{O}^A = \{m \in M_A \mid \lambda_A^{\mathsf{OP}}(m) = \mathsf{O}\}$ and $\mathsf{P}^A = \{m \in M_A \mid \lambda_A^{\mathsf{OP}}(m) = \mathsf{P}\}$ denote the sets of *Opponent* and *Player* moves in A, respectively. Since λ_A is a total function, $\{\mathsf{O}^A, \mathsf{P}^A\}$ is a partition of M_A. Given that the sequences from a strategy for a game A are valid and satisfy the alternation condition, it follows that they are sequences from $(\mathsf{O}^A \mathsf{P}^A)^*$.

Let σ be an unknown strategy for a game A. L^* iteratively learns the structure of σ using assistance from a *Teacher* who can answer two kinds of questions about σ:

Membership query. Given a sequence s from $(\mathsf{O}^A \mathsf{P}^A)^*$, the Teacher answers *true* if $s \in \sigma$, and *false* otherwise.

Equivalence query. Given a DFA (Deterministic Finite Automaton) D, the Teacher replies that D is either correct, when $\mathcal{L}(D) = \sigma$, or incorrect, and in the latter case gives a counterexample which is a sequence in the symmetric difference of $\mathcal{L}(D)$ and σ.

The basic data structure of the L^* algorithm is a two-dimensional table, called observation table (S, E, T), which keeps information about a finite collection of sequences over $(\mathsf{O}^A \mathsf{P}^A)^*$, classified as members or non-members of σ. S is a *prefix-closed* set of even-length sequences, $E \subseteq (\mathsf{O}^A \mathsf{P}^A)^*$ is a *suffix-closed* set of even-length sequences, and T is a function mapping $(S \cup S \cdot \mathsf{O}^A \mathsf{P}^A) \cdot E \to \{\mathit{true}, \mathit{false}\}$, such that:

$$\forall\, s \in S \cup S \cdot \mathsf{O}^A \mathsf{P}^A . \forall\, e \in E : T(s, e) = \mathit{true} \Leftrightarrow s \cdot e \in \sigma$$

The rows of the table are the elements of $(S \cup S \cdot \mathsf{O}^A \mathsf{P}^A)$, while the columns are the elements of E. Finally T denotes the table entries.

Let us define a function $\mathit{row}(s)$ for any $s \in S \cup S \cdot \mathsf{O}^A \mathsf{P}^A$ as follows:

$$\forall\, e \in E : \mathit{row}(s)(e) = T(s, e)$$

A table is *closed* if for each $s \cdot m_O m_P \in S \cdot \mathsf{O}^A \mathsf{P}^A$ such that $T(s, \epsilon) = \mathit{true}$, there is some $s' \in S$ such that $\mathit{row}(s') = \mathit{row}(s \cdot m_O m_P)$. A table is *consistent* if for each $s, s' \in S$ such that $\mathit{row}(s) = \mathit{row}(s')$, either $T(s, \epsilon) = T(s', \epsilon) = \mathit{false}$, or for each $m_O m_P \in \mathsf{O}^A \mathsf{P}^A$, we have that $\mathit{row}(s \cdot m_O m_P) = \mathit{row}(s' \cdot m_O m_P)$.

We define an equivalence relation $\equiv$ over sequences in $S \cup S \cdot \mathsf{O}^A \mathsf{P}^A$ such that $s \equiv s'$ iff $\mathit{row}(s) = \mathit{row}(s')$. Denote by $[s]$ the equivalence class which includes s. Given a closed and consistent table (S, E, T), L^* constructs a candidate DFA

$D = (Q, q_0, \mathsf{O}^A\mathsf{P}^A, \delta)$ as follows: $Q = \{[s] \mid s \in S, T(s, \epsilon) = true\}$, $q_0 = [\epsilon]$, and for every $s \in S$ and $m_O m_P \in \mathsf{O}^A\mathsf{P}^A$, the transition from $[s]$ on input $m_O m_P$ is enabled iff $T(s \cdot m_O m_P, \epsilon) = true$ and then $\delta([s], m_O m_P) = [s \cdot m_O m_P]$. The facts that the table is closed and consistent guarantee that the transition relation is well-defined. All states in the automaton are accepting, since the language we learn is prefix closed. Note that every transition in this automaton is labelled by two-letters sequence: an Opponent and a Player move.

```
let L*(S, E) be
  repeat :
    Update T using queries
    while (S, E, T) is not consistent or not closed do
        if (S, E, T) is not consistant then
            find s ∈ S, m_O m_P ∈ O^A P^A, e ∈ E :
                row(s) = row(s') and T(s · m_O m_P, e) ≠ T(s' · m_O m_P, e)
            E = E ∪ {m_O m_P · e}
            Update T using queries
        if (S, E, T) is not closed then
            find s ∈ S, m_O m_P ∈ O^A P^A
                s · m_O m_P ∉ [t], for all t ∈ S
            S = S ∪ {s · m_O m_P}
            Update T using queries
    D = MakeAutomaton(S, E, T)
    if D is correct then
        return D
    else
        let c be reported counterexample
        foreach (s ∈ even_prefix(c) and s ∉ S) S = S ∪ {s}
```

Fig. 2. L^* algorithm

Fig. 2 contains the L^* algorithm. Each iteration of this algorithm starts with either a table with $S = E = \{\epsilon\}$, or a table which was prepared in the previous step. Then T is updated using membership queries until the table is consistent and closed. Next a candidate automaton D is proposed and an equivalence query with D is made. If the answer for the equivalence query is *true*, L^* terminates and returns the automaton D. Otherwise, L^* analyzes the counterexample c reported by the Teacher and adds all even-length prefixes of c to S. Then, a new iteration is started.

L^* is guaranteed to construct a minimal DFA equivalent to the unknown strategy using at most $n - 1$ equivalence queries where n is the number of states in the minimal DFA, and in time polynomial in n and the length of the longest counterexample provided by the Teacher.

Each new call to L^* starts normally with $S = E = \{\epsilon\}$. But in cases where a previously learned candidate exists, we want to start the algorithm by reusing the information proposed in the previous table. Thus with this *dynamic version*

of L^*, we try to speed up the learning by reusing the previously inferred sets S and E for strategy σ, to learn a new modified strategy σ' which differs slightly from σ. We apply this optimisation using the fact that if L^* starts with any non-empty valid table (i.e. valid function T) then it will terminate with a correct result [7]. A table is said to be valid if the answers to the membership queries for all sequences in the table are correct with respect to the unknown language which is learned by L^*.

We can apply some further optimizations to the L^* algorithm specific for the languages we learn. Since the sequences from an strategy are valid plays, we test for membership only valid plays. All other sequences are certainly not in the strategy, and they are marked as *false* without any checks. Then, a prefix closed language has the property that extensions of rejected sequences are rejected, i.e., if $s \notin \sigma$, then no extension of s is in σ. Therefore, since the language we learn is prefix closed, before any membership query $s \in \sigma$, we first test whether it is an extension of a sequence already observed to be rejected. If so, we add the result immediately to the table.

5 Compositional Verification

In this section we describe in detail the compositional verification procedure which combines assume-guarantee reasoning and abstraction refinement.

5.1 Overview

We first examine how the game semantics of β-normal AIA_2 terms $\Gamma \vdash M : B$ is obtained. Since terms are interpreted recursively over the typing rules, consider a derivation tree of such a term $\Gamma \vdash M : B$. At the leaves, we have base subterms, which are language constants and free identifiers, and are interpreted by appropriate constant and identity strategies. At each node, there is a subterm obtained by a language construct c from some children subterms $M_1, \ldots, M_n$. Then, $\mathsf{c}(M_1, \ldots, M_n)$ is interpreted by composing the interpretations of the subterms and of the construct σ_{c}:

$$[\![\mathsf{c}(M_1, \ldots, M_n)]\!] = ([\![M_1]\!], \ldots, [\![M_n]\!]) \fatsemi \sigma_{\mathsf{c}} = ([\![M_1]\!]^\dagger, \ldots, [\![M_n]\!]^\dagger) \, ; \, \sigma_{\mathsf{c}}$$

We also note that $\dagger$ is applied only to strategies σ for games of the form $[\![\Gamma]\!] \Rightarrow [\![B']\!]$, where B' are base types. The games $[\![B']\!]$ are flat, i.e. all their questions are initial and Player moves can only be answers. So $\sigma^\dagger$ consists of iterated plays of σ, such that a new play of σ can be started only when the previous one is completed. Basically, $\sigma^\dagger$ contains plays of the form $s_1 \ldots s_k s_{k+1}$ where each s_i is a play of σ and $s_1, \ldots, s_k$ are complete. That is a regular language operation.

Now, for any strategies $\sigma_1, \ldots, \sigma_n$ and τ, we have $\big((\sigma_1^\dagger, \ldots, \sigma_n^\dagger) \, ; \, \tau\big)^\dagger = (\sigma_1^\dagger, \ldots, \sigma_n^\dagger) \, ; \, \tau^\dagger$ [3]. By thus distributing $\dagger$ over ; , we conclude that the game semantics of $\Gamma \vdash M : B$ can be obtained by repeatedly applying ; to promoted

strategies for base subterms and language constructs. In other words, † does not need to be applied to any composite strategy.

By the same argument, if $\Gamma' \vdash N : B'$ is a subterm of $\Gamma \vdash M : B$, the game semantics of $\Gamma \vdash M : B$ is given by:

$$[\![\Gamma \vdash M[N] : B]\!] = [\![\Gamma \vdash M[-] : B]\!]([\![\Gamma' \vdash N : B']\!]^\dagger)$$

where $[\![\Gamma \vdash M[-] : B]\!](\sigma)$ is an operator on regular languages, which is obtained from the game semantic definitions for $\Gamma \vdash M : B$ by replacing the promoted interpretation of the subterm $\Gamma' \vdash N : B'$ by σ, and in which only ; is applied to languages obtained from σ.

To check safety of $[\![\Gamma \vdash M[N] : B]\!]$, we use the concept of assume-guarantee (AG) reasoning. We define an *assumption* for a game A as a prefix-closed non-empty set of even-length sequences from $(\mathsf{O}^A\mathsf{P}^A)^*$.

Let σ be an assumption for $[\![\Gamma']\!] \Rightarrow ![\![B']\!]$. We use the following AG rule:

$$\frac{\begin{array}{l}[\![\Gamma \vdash M[-] : B]\!](\sigma) \text{ is SAFE} \\ [\![\Gamma' \vdash N : B']\!]^\dagger \le \sigma\end{array}}{[\![\Gamma \vdash M[N] : B]\!] \text{ is SAFE}}$$

The rule states that if there is an assumption σ for $[\![\Gamma']\!] \Rightarrow ![\![B']\!]$, such that $[\![\Gamma \vdash M[-] : B]\!](\sigma)$ is safe and σ is an abstraction of $[\![\Gamma' \vdash N : B']\!]^\dagger$, then $[\![\Gamma \vdash M[N] : B]\!]$ is safe. Our goal is to construct such an assumption σ.

Theorem 2. *The AG rule is sound and complete.*

Proof. By monotonicity of composition of strategies with respect to the $\le$ ordering, we have that if $\sigma \le \sigma'$ then $[\![\Gamma \vdash M[-] : B]\!](\sigma) \le [\![\Gamma \vdash M[-] : B]\!](\sigma')$. To establish soundness, we use the fact that if σ' is safe and $\sigma \le \sigma'$ then σ is also safe. Completeness follows by taking $\sigma = [\![\Gamma' \vdash N : B']\!]^\dagger$. □

For any operator $[\![\Gamma \vdash M[-] : B]\!]$, where the hole $-$ is in the place of a subterm of type $\Gamma' \vdash B'$, we define the *weakest safe strategy* $\sigma_W : [\![\Gamma']\!] \Rightarrow ![\![B']\!]$ as follows. Given an even-length play s of $[\![\Gamma']\!] \Rightarrow ![\![B']\!]$, let τ_s be the strategy consisting of s and all its even-length prefixes. Let σ_W consist of all s such that $[\![\Gamma \vdash M[-] : B]\!](\tau_s)$ is safe.

Proposition 2. *For any AIA_2 term with a hole $\Gamma \vdash M[-] : B$, σ_W is a regular language.*

By the definitions of $[\![\Gamma \vdash M[-] : B]\!]$ and ; , we have that, for any strategy $\sigma : [\![\Gamma']\!] \Rightarrow ![\![B']\!]$,

$$[\![\Gamma \vdash M[-] : B]\!](\sigma) = \bigcup\{[\![\Gamma \vdash M[-] : B]\!](\tau_s) \mid s \in \sigma\}$$

Hence, $[\![\Gamma \vdash M[-] : B]\!](\sigma)$ is safe if and only $\sigma \le \sigma_W$. For this strategy σ_W, the AG rule is guaranteed to return conclusive results: either the resulting term is safe or unsafe, and in the latter case a counterexample is reported. We use the L^* algorithm to learn σ_W.

The verification procedure **CompVer** which uses the AG rule is presented in Fig. 3. Given two terms $\Gamma \vdash M[-] : B$ and $\Gamma' \vdash N : B'$, it checks safety of $\Gamma \vdash M[N] : B$. The procedure uses an **AGCheck** algorithm, and iteratively performs the following steps:

1. Let $[\![\Gamma_1 \vdash M_1[-] : B_1]\!]$ and $[\![\Gamma'_1 \vdash N_1 : B'_1]\!]$ be obtained by data abstraction, and $S_1^1 = E_1^1 = \{\epsilon\}$.
2. Apply **AGCheck** on $[\![\Gamma_i \vdash M_i[-] : B_i]\!]$ and $[\![\Gamma'_i \vdash N_i : B'_i]\!]$, using S_i^1 and E_i^1. If the result is *true*, then terminate with answer SAFE. Otherwise, a counterexample c is returned as well as updated values of S_i^k and E_i^k.
3. If c is a nondeterministic (i.e. spurious) play, obtain $[\![\Gamma_{i+1} \vdash M_{i+1}[-] : B_{i+1}]\!]$ and $[\![\Gamma'_{i+1} \vdash N_{i+1} : B'_{i+1}]\!]$ by refining the abstractions in the current terms which were involved in causing the nondeterminism in c. Set $S_{i+1}^1 = S_i^k$ and $E_{i+1}^1 = E_i^k$ [3], and repeat from 2.
4. Otherwise, c is deterministic (i.e. genuine) and the procedure terminates with answer UNSAFE.

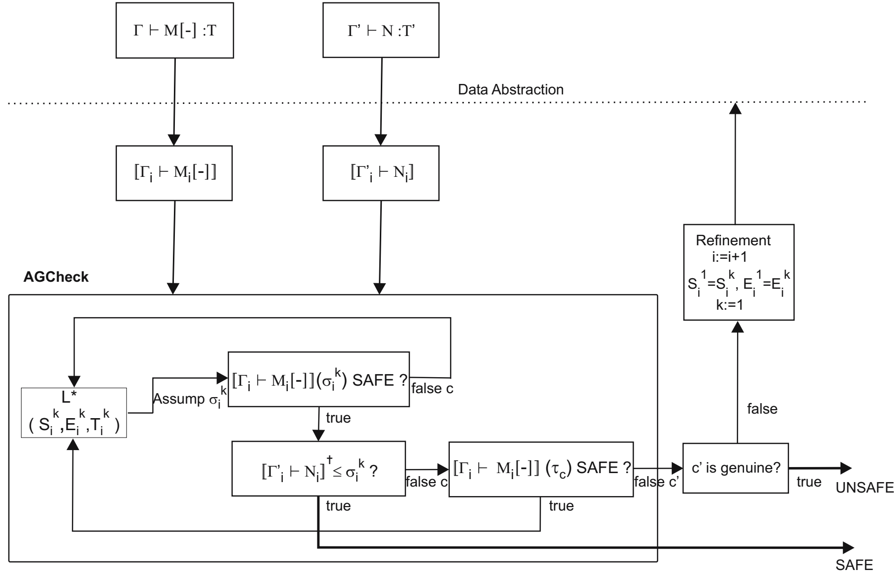


Fig. 3. The compositional verification procedure **CompVer**

We say that a play is *nondeterministic* if it contains a special marker move nd, which identifies points in plays at which abstraction gives rise to nondeterminism. This happens when an arithmetic/logic operation produces more than one result.

[3] If some sequences in S_i^k (E_i^k) contain abstract values whose abstractions are refined, we replace them with sequences which are compatible with newly refined abstractions.

We continue by describing the **AGCheck** algorithm. Details of the data abstraction procedure and the abstraction refinement process are beyond the scope of this paper and can be found in [11].

5.2 Assume-Guarantee Algorithm

The **AGCheck** algorithm takes as inputs $[\![\Gamma_i \vdash M_i[-] : B_i]\!]$ and $[\![\Gamma_i' \vdash N_i : B_i']\!]$ as well as S_i^1 and E_i^1, and returns as answer *true* or a counterexample. **AGCheck** is actually the L^* algorithm given in Fig. 2, where the membership and equivalence queries are answered using model checking. **AGCheck** proceeds as follows:

1. Generate a candidate assumption σ_i^k using L^*.
2. If $[\![\Gamma_i \vdash M_i[-] : B_i]\!](\sigma_i^k)$ is not safe, then return a counterexample to the L^* algorithm, set $k := k+1$ and repeat from 1.
3. If $[\![\Gamma_i' \vdash N_i : B_i']\!]^\dagger \leq \sigma_i^k$ is true, terminate with answer *true*.
4. Otherwise, among the even-length counterexamples from $[\![\Gamma_i' \vdash N_i : B_i']\!]^\dagger$, report a deterministic one, c. If such one does not exist, then report a nondeterministic one, c.
5. Generate a strategy τ_c from the sequence c which contains c and all its even-length prefixes. If $[\![\Gamma_i \vdash M_i[-] : B_i]\!](\tau_c)$ is safe, then report c to L^*, set $k := k+1$ and repeat from 1.
6 Otherwise, terminate reporting a deterministic counterexample c'. If such one does not exist, report a nondeterministic play c'.

If in Step 2 a counterexample c is returned to L^*, then $c \in \sigma_i^k \backslash \sigma_W$, i.e. the current assumption σ_i^k is too weak and it has to be strengthened by removing some sequences from it. Similarly, if in Step 5 a counterexample c is reported to L^*, then $c \in \sigma_W \backslash \sigma_i^k$, i.e. the current σ_i^k must be weakened by adding some sequences.

In the above procedure, L^* iteratively learns the strategy σ_W, but the procedure terminates as soon as conclusive results are obtained. This is often before the weakest safe strategy σ_W is computed by L^*. The Teacher which interacts with L^* is implemented using model checking. To answer a membership query for a sequence s, the Teacher first builds a strategy $\tau_s = \{s' \mid s' \sqsubseteq^{\text{even}} s\}$. The Teacher then model checks $[\![\Gamma \vdash M[-]]\!](\tau_s)$ for safety. If true is returned, then $s \in \sigma_W$ and the Teacher answers *true*, otherwise it answers *false*. An equivalence query is answered by model-checking two premises of the AG rule in Steps 2 and 3. If both checks succeed, then the answer is *true*, otherwise either a counterexample is reported to L^* or an unsafe counterexample is found.

Theorem 3. *Given* $[\![\Gamma_i \vdash M_i[-] : B_i]\!]$ *and* $[\![\Gamma_i' \vdash N_i : B_i']\!]$*, the* **AGCheck** *algorithm terminates with either true or an unsafe play from* $[\![\Gamma_i \vdash M_i[N_i] : B_i]\!]$.

Proof. The algorithm returns true when both premises of the AG rule return true, and therefore correctness is guaranteed by the AG rule. An unsafe play is returned when there is a sequence s of $([\![\Gamma_i' \vdash N_i]\!])^\dagger$ which, when applied to $[\![\Gamma_i \vdash M_i[-]]\!]$ produces an unsafe play, which implies that $[\![\Gamma_i \vdash M_i[N_i] : B_i]\!]$ is not safe.

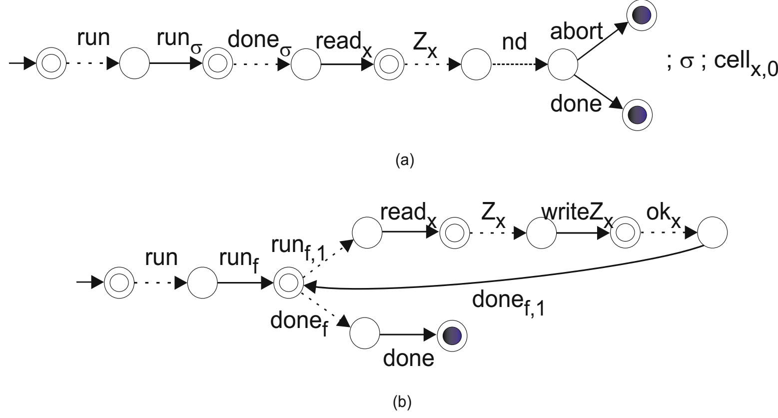


Fig. 4. Strategies at AR iteration 1: (a) $[\![f \vdash M[-]]\!](\sigma)$ (b) $[\![f, x \vdash f(x := x + 1)]\!]$

Termination of **AGCheck** algorithm is implied by the termination of the L^* algorithm. At any iteration, **AGCheck** either terminates or provides a counterexample to L^*. Thus, L^* will eventually produce σ_W at some iteration and the algorithm will return conclusive results and terminate. □

Theorem 4. *If* **CompVer** *terminates, its answer is correct.*

Proof. This follows from the correctness of the abstraction refinement procedure, which was shown in [11], and Theorem 3. □

5.3 Example

Consider the term

$$f : \mathsf{com} \rightarrow \mathsf{com} \ \vdash \ \mathsf{newint}\, x := 0\, \mathsf{in} \\ f(x := x + 1)\, ; \\ \mathsf{if}\ (x == 0)\ \mathsf{then\ abort};$$

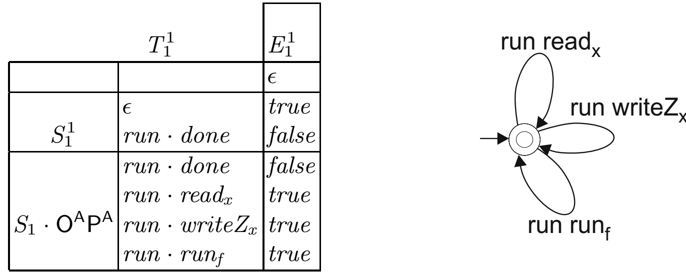
in which x is a local variable, and f is a non-local (safe) function. We want to check whether this term is safe from terminating abnormally for all safe instantiations of f. The program is not safe if function f does not use its argument at all.

We start with applying the coarsest abstraction $[\,]$ to x, which means that x can only have the value $\mathbb{Z}$ (i.e. a nondeterministic choice over all integers).

Let the arbitrary subterm N be $f(x := x + 1)$. The model of the whole term is obtained by composing the model for the scope of variable declaration with the strategy $\mathsf{cell}_{x,0}$, which is used for remembering the initial (0) or the most recently written value into the variable x. This strategy ensures "good variable" behavior of x.

In Fig. 4 are shown the models $[\![f \vdash M[-]]\!](\sigma)$ and $[\![f, x \vdash f(x := x + 1)]\!]$ at the first Abstraction Refinement iteration. The *nd* move [4] in the first strategy

[4] It is neither Opponent nor Player, but a special marker move.



T_1^1		E_1^1
		ϵ
S_1^1	ϵ	*true*
	$run \cdot done$	*false*
$S_1 \cdot \mathsf{O}^{\mathsf{A}}\mathsf{P}^{\mathsf{A}}$	$run \cdot done$	*false*
	$run \cdot read_x$	*true*
	$run \cdot writeZ_x$	*true*
	$run \cdot run_f$	*true*

Fig. 5. Observation table and assumption at AR iteration 1

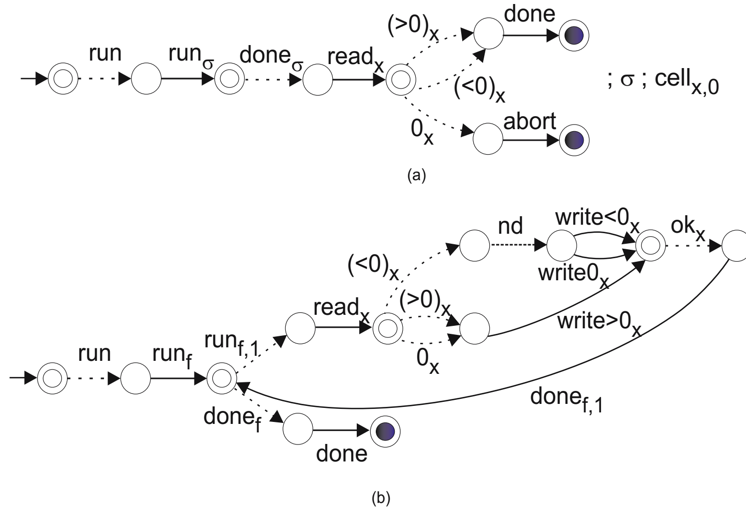


Fig. 6. Strategies at AR iteration 2: (a) $[\![f \vdash M[-]]\!](\sigma)$ (b) $[\![f, x \vdash f(x := x + 1)]\!]$

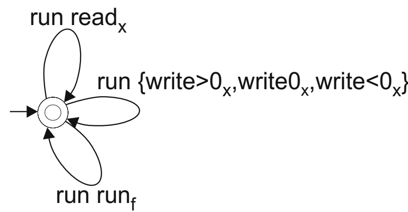


Fig. 7. Assumption at AR iteration 2

marks that nondeterminism has occurred due to abstraction. In this case, the guard of 'if' command has been evaluated nondeterministically to *true* or *false*, since the value of x might be any integer.

At each iteration, L^* updates its observation table and constructs a candidate assumption whenever the table becomes consistent and closed. The first such table produced and its associated assumption are given in Fig. 5. Note that in observation tables we list only sequences from $S \cdot \mathsf{O}^{\mathsf{A}}\mathsf{P}^{\mathsf{A}}$ which are valid plays, and all other sequences are *false* by default. The equivalence query is then asked. The second AG premise fails and the Teacher returns a negative answer with a counterexample $s = \langle run \cdot run_f \cdot done_f \cdot done \rangle$, which is not safe when applied to $[\![f \vdash M[-]]\!]$. Thus, **AGCheck** reports $s' = \langle run \cdot run_f \cdot done_f \cdot nd \cdot abort \rangle$. Since this play is nondeterministic, our procedure decides to refine abstractions that caused the nondeterminism in s' and to continue. In this case, the abstraction of x is refined to $[0, 0]$, which contains three possible values: < 0, 0 and > 0.

At the second abstraction refinement iteration, the strategies $[\![f \vdash M[-]]\!](\sigma)$ and $[\![f, x \vdash f(x := x + 1)]\!]$ are given in Fig. 6.

Since we use a dynamic version of L^*, it starts with an observation table where S_2^1 and E_2^1 are the same as in the previous table T_1^1. The next candidate assumption is shown in Fig. 7. The second AG rule premise fails giving $s = \langle run \cdot run_f \cdot done_f \cdot done \rangle$. Now, **AGCheck** reports a genuine counterexample $s' = \langle run \cdot run_f \cdot done_f \cdot abort \rangle$, and the procedure terminates informing that the input term is not safe.

6 Implementation

We implemented the compositional verification procedure in the GameChecker tool [12]. GameChecker compiles an abstracted open program into a process in the CSP process algebra (e.g. [22]), whose finite traces set represents the game-semantic model of the program. Membership and equivalence queries are answered using the FDR refinement checker [13]. If a counterexample is reported by the procedure, GameChecker is used to analyse the counterexample and do abstraction refinement.

Consider the following implementation of a stack of maximum size n (a meta variable). After implementing the stack by a sequence of local declarations, we export the functions $push(x)$ and pop by calling $ANALYSE$ with arguments $push(p)$ and pop. In effect, the model contains all interleavings of calls to $push(p)$ and pop, corresponding to all possible behaviours of the non-local expression p and non-local function $ANALYSE$.

```
empty : com, overflow : com, p : exp int,
ANALYSE(com, exp int) : com ⊢
new int buffer[n] := 0 in new int top := 0 in
let com push(int x) {
   if (top == n) then overflow else {buffer[top] := x;  top := top + 1}} in
let exp int pop {
   if (top == 0) then empty else {top := top − 1;  return buffer[top + 1]}} in
ANALYSE(push(p), pop)
```

Table 1. Experimental results for checking a stack implementation

	empty		overflow	
n	Direct	AG	Direct	AG
3	271	107	286	147
10	306	135	937	441
15	331	155	1462	651
25	381	195	2662	1071

By replacing the free identifier empty (resp. overflow) with the abort command, we can check the safety property that there are no reads from empty stacks (resp. writes to full stacks). Both errors are present for any n. For the 'empty' error, a genuine counterexample is reported after refining the abstraction of *top* to $[0, 0]$. For the 'overflow' error, the abstraction is $[0, n]$. The counterexamples correspond to a single call of the *pop* method (resp. $n + 1$ consecutive calls of the *push* method) after which abort is executed. We applied the AG procedure by learning an appropriate assumption for the *push* (resp. *pop*) method. In both cases, we obtain conclusive assumptions with 0 states, since counterexamples are reported for all valid plays of the subterms we learn.

Table 1 contains the experimental results for checking the two properties by using the AG procedure and the direct verification procedure without AG reasoning [12]. We list the size of the largest generated transition system in each case for different values of n.

7 Conclusion

This paper presents a fully compositional approach for verifying safety properties of open programs. Game semantics is used for compositional modelling of programs and an automated assume-guarantee procedure with learning is used for compositional verification.

Important topics for future work are extending data abstractions to arbitrary predicates, dealing with concurrent programs, and using assume-guarantee reasoning for verifying liveness properties.

References

1. S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying Game Semantics to Compositional Software Modeling and Verification. In Proceedings of *TACAS*, LNCS **2988**, (2004), 421–435.
2. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full Abstraction for PCF. *Information and Computation*, **163(2)**, (2000).
3. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P.W.O'Hearn and R.D.Tennent, editors, *Algol-like languages.* (Birkhaüser, 1997).
4. R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In Proceedings of *CAV*, LNCS **3576**, (2005), 548–562.

5. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, **75(2)**, (1987), 87–106.
6. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In Proceedings of *SPIN*, LNCS **2057**, (2001), 103–122.
7. S. Chaki, E. Clarke, N. Sharygina, and N. Sinha. Dynamic Component Substiutability Analysis. In Proceedings of *FM*, LNCS **3582**, (2005), 512–528.
8. E.M. Clarke, O. Grumberg and D. Peled, *Model Checking*. (MIT Press, 2000).
9. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning Assumptions for Compositional Verification. In Proceedings of *TACAS*, LNCS **2619**, (2003), 331–346.
10. A. Dimovski and R. Lazic. CSP Representation of Game Semantics for Second-Order Idealized Algol. In Proceedings of *ICFEM*, LNCS **3308**, (2004), 146–161.
11. A. Dimovski, D. R. Ghica, and R. Lazic. Data-Abstraction Refinement: A Game Semantic Approach. In Proceedings of *SAS*, LNCS **3672**, (2005), 102–117.
12. A. Dimovski, D. R. Ghica, and R. Lazic. A Counterexample-Guided Refinement Tool for Open Procedural Programs. In Proceedings of *SPIN*, LNCS **3925**, (2006).
13. Formal Systems (Europe) Ltd (http://www.fsel.com), *Failures-Divergence Refinement: FDR2 Manual*, 2000.
14. D. R. Ghica and G. McCusker. The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science **309** (1–3), (2003), 469–502.
15. A. Groce, D. Peled, and M. Yannakakis. Adaptive Model Checking. In Proceedings of *TACAS*, LNCS **2280**, (2002), 357–370.
16. R. Harmer. Games and Full Abstraction for Nondeterministic Languages. PhD thesis, Imperial College, 1999.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In Proceedings of *SPIN*, LNCS **2648**, (2003), 235–239.
18. J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I, II, and III. *Information and Computation* **163**, (2000), 285–400.
19. J. Laird. A Fully Abstract Game Semantics of Local Exceptions. In Proceedings of *LICS*, (2001), 105–114.
20. A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. *Logic and Models of Concurrent Systems* **13**, (1984), 123–144.
21. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, **103(2)**, (1993), 299–347.
22. A. W. Roscoe. *Theory and Practice of Concurrency*. (Prentice-Hall, 1998).