Data-Abstraction Refinement: A Game Semantic Approach*

Aleksandar Dimovski[1], Dan R. Ghica[2], and Ranko Lazić[1]

[1] Department of Computer Science, Univ. of Warwick, Coventry, CV4 7AL, UK
[2] School of Computer Science, Univ. of Birmingham, Birmingham, B15 2TT, UK

Abstract. This paper presents a semantic framework for data abstraction and refinement for verifying safety properties of open programs. The presentation is focused on an Algol-like programming language that incorporates data abstraction in its syntax. The fully abstract game semantics of the language is used for model-checking safety properties, and an interaction-sequence-based semantics is used for interpreting potentially spurious counterexamples and computing refined abstractions for the next iteration.

1 Introduction

Abstraction refinement has proved to be one of the most effective methods of automated verification of systems with very large state spaces, especially software systems. Current state-of-the art tools implementing abstraction refinement algorithms [5, 16] combine model checking and theorem proving: model checking is used to verify whether an *abstracted system* satisfies a property, while theorem proving is used to *refine the abstraction* using the counterexamples discovered by model checking. Since abstractions are *conservative over-approximations* the safety of any abstracted program implies the safety of the concrete program. The converse is not true, and the refinement process may not terminate if the concrete program has an infinite state space.

This paper introduces a purely semantic approach to (data) abstraction refinement, based on game semantics [2, 17]. In order to keep the presentation focussed, the main vehicle of our development is the language *Abstracted Idealized Algol* (AIA), an expressive programming language combining imperative features, locally-scoped variables and (call-by-name) procedures. The key feature of this language is the use of abstraction schemes at the level of data-types, which allows the writing of abstracted programs in a syntax similar to that of concrete programs. In fact, a concrete program is a particular abstracted program, in which all the abstractions are identities.

The following is a simple example illustrating this method. Consider the (concrete) program fragment below, which uses local variable x, *non-local* function f,

* This research is supported by the EPSRC (GR/S52759/01). The 3rd author is also supported by a grant from the Intel Corporation, and affiliated to the Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade.

and a command abort which causes abnormal termination. Is this program safe for all instantiations of f, or is it possible for its execution to terminate abnormally? [3]

$$\mathsf{newint}\ x := 0\ \mathsf{in}\ f(x := !x + 1, \mathsf{if}\ !x = 5\ \mathsf{then}\ \mathsf{abort}\ \mathsf{else}\ \mathsf{skip})$$

The procedure-call mechanism is by-name, so every call to the first argument increments x, and any call to the second uses the new value of x. So the program is not safe if function f uses its first argument precisely 5 times, then its second argument.

We approximate the set of integers by a finite set of partitioning intervals. Let the initial abstraction have only one partition. We denote this in the program by annotation (see Table 1):

$$\mathsf{newint}_{[]}\ x := 0\ \mathsf{in}\ f(x := !x +_{[]} 1, \mathsf{if}\ !x = 5\ \mathsf{then}\ \mathsf{abort}\ \mathsf{else}\ \mathsf{skip})$$

A counterexample execution trace exists, corresponding to the function evaluating its second argument. During the execution of this argument, the value of x is not 0 but, because of the abstraction, possibly any integer, chosen nondeterministically. If the chosen value is 5 then abort occurs. Of course, this counterexample is spurious because it is made possible only by the nondeterminism caused by over-abstraction. However, the counterexample informs the refinement procedure that the abstraction of x needs to be improved. Iterations like this one are performed until we obtain

$$\mathsf{newint}_{[0,5]}\ x := 0\ \mathsf{in}\ f(x := !x +_{[0,5]} 1, \mathsf{if}\ !x = 5\ \mathsf{then}\ \mathsf{abort}\ \mathsf{else}\ \mathsf{skip})$$

at which point a genuine counterexample is discovered, corresponding to the behaviour resulting in abnormal termination.

In addition to giving a precise account of data abstraction-refinement, this approach has several advantages compared to alternative approaches:

Modularity To our knowledge, examples such as the one described before, cannot be generally handled by known inter-procedural abstraction-refinement techniques. [8] has cogently advocated a need for such techniques, and we believe that we are meeting the challenge, although our approach is technically different.

Completeness and correctness A concrete representation of a fully abstract semantic model is guaranteed to be accurate and is set on a firm theoretical foundation.

Compositionality The semantic model is denotational, i.e. defined recursively on the syntax, therefore the model of a larger program is constructed from the models of its constituting sub-programs. This entails an ability to model program fragments, containing non-locally defined procedures as in the example above. This feature is the key to *scalability*, the modeling and verification of software systems that are too large to be dealt with as a whole.

[3] $!x$ denotes dereferencing.

Efficiency As already emphasised in previous work on games-based model checking [1], finite-state representations of strategies give models of programs often several orders of magnitude smaller than state-exploration based models, essentially due to the fact that the details of local-state manipulation are hidden during composition.

2 Abstracted Idealized Algol

The data types of AIA are booleans and *abstracted integers* ($\tau ::= \mathsf{bool} \mid \mathsf{int}_\pi$). We use π to denote computable binary predicates on $\mathbb{Z}$. The *abstractions* π range over computable equivalence relations (i.e. partitions) of the integers $\mathbb{Z}$. To say that $m, n \in \mathbb{Z}$ are in the same class of π, we write $m \approx_\pi n$.

The phrase types of AIA are base types of expressions, variables and commands ($\sigma ::= \mathsf{exp}\tau \mid \mathsf{var}\tau \mid \mathsf{com}$) and function types ($\theta ::= \sigma \mid \theta \to \theta$).

We say that a type is *concrete* if it contains no abstractions other than the identity abstraction $\kappa = \big\{\{i\} \mid i \in \mathbb{Z}\big\}$. For any type θ, we write $\widetilde{\theta}$ for the concrete type obtained by replacing all abstractions with κ. For simplicity, we write int_κ as simply int.

The syntax of the language consists of imperative features (local variables, assignment, dereferencing, sequencing, branching, iteration, skip and abort) and functional features (abstraction, application, recursion, arithmetic-logic constants and operators). It is convenient to present the syntax of AIA in a "functionalised" form [3], using function-constants rather than term combinators, as in:

$$\mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N \cong \mathsf{if}\ B\ M\ N$$
$$\mathsf{new}\tau\ x := E\ \mathsf{in}\ C \cong \mathsf{new}\ E\ (\lambda\, x : \mathsf{var}\tau.\, C),\ \textit{etc.}$$

Combinators can be reintroduced as syntactic sugar, to improve readability.

The base-type constants are:

$$\mathsf{true}, \mathsf{false} : \mathsf{expbool} \quad \mathsf{abort}, \mathsf{skip} : \mathsf{com} \quad n : \mathsf{expint}_\pi$$

The functional constants are:

$$\begin{array}{ll}
\mathsf{new} : \mathsf{exp}\tau_1 \to (\mathsf{var}\tau_2 \to \mathsf{com}) \to \mathsf{com}, & \mathsf{if} : \mathsf{expbool} \to \sigma_1 \to \sigma_2 \to \sigma, \\
\quad \widetilde{\tau_1} = \widetilde{\tau_2} & \quad \widetilde{\sigma_1} = \widetilde{\sigma_2} = \widetilde{\sigma} \\
\mathsf{asg} : \mathsf{var}\tau_1 \to \mathsf{exp}\tau_2 \to \mathsf{com},\ \widetilde{\tau_1} = \widetilde{\tau_2} & \mathsf{while} : \mathsf{expbool} \to \mathsf{com} \to \mathsf{com} \\
\mathsf{der} : \mathsf{var}\tau \to \mathsf{exp}\tau & \mathsf{rec} : (\theta \to \theta) \to \theta \\
\mathsf{seq} : \mathsf{com} \to \sigma \to \sigma & \mathsf{op} : \mathsf{exp}\tau_1 \to \mathsf{exp}\tau_2 \to \mathsf{exp}\tau
\end{array}$$

where op stands for any arithmetic-logic operator whose concrete type is $\mathsf{exp}\widetilde{\tau_1} \to \mathsf{exp}\widetilde{\tau_2} \to \mathsf{exp}\widetilde{\tau}$. For example, for any abstractions π_1 and π_2, AIA contains an equality operator $=$ of type $\mathsf{expint}_{\pi_1} \to \mathsf{expint}_{\pi_2} \to \mathsf{expbool}$.

For types of new, asg and if to be valid, it is required above that corresponding subterms of types have equal concretisations, but their abstractions can be

Notation for sets of integers:

$$<n = \{n' \mid n' < n\}, \quad n = \{n\}, \quad >n = \{n' \mid n' > n\}$$

Notation for abstractions:

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

Fig. 1. Some integer abstractions

different. For example, for any abstractions π_1 and π_2, we can assign expressions of type int_{π_2} to variables of type int_{π_1}. This flexibility, which is also present in the rule for functional application below, enables abstractions within a term to be changed independently of each other while preserving well-typed-ness.

$\Gamma \vdash M : \theta$, where Γ is a list of typed identifiers, indicates that term M with free identifiers in Γ has type θ. The typing rules are:

$$\Gamma \vdash \mathsf{k} : \theta \text{ (k is a language constant of type θ)} \qquad \Gamma, x : \theta \vdash x : \theta$$

$$\frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda\, x : \theta.M : \theta \to \theta'} \qquad \frac{\Gamma \vdash M : \theta_1 \to \theta \qquad \Gamma \vdash N : \theta_2}{\Gamma \vdash M\ N : \theta}\ \widetilde{\theta_1} = \widetilde{\theta_2}$$

Whenever we write $\Gamma \vdash M : \theta$, we are considering implicitly a particular derivation of that typing judgment from the rules above. Such a derivation contains typing judgments for all sub-terms of M. When we need to be explicit about which derivation was used, we shall annotate M with abstractions. For example, with the notations in Fig. 1,

$$x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,4]\to[0,1]\to[0,3]} 1_{[0,1]} : \mathsf{com}$$

means that the operator $+$ was used with type $\mathsf{expint}_{[0,4]} \to \mathsf{expint}_{[0,1]} \to \mathsf{expint}_{[0,3]}$. Here the combinators $:=$ and $!$ are syntactic sugar for applications of the functional constants asg and der.

We say that a term is *concrete* if it contains no abstractions other than the identity abstraction κ. For any term $\Gamma \vdash M : \theta$, we write $\widetilde{\Gamma} \vdash \widetilde{M} : \widetilde{\theta}$ for the concrete term obtained by replacing all abstractions with κ.

The operational semantics is defined as a big-step reduction relation $M, s \Longrightarrow \mathcal{K}$, where M is a *program* (all free identifiers are assignable variables), s is a *state* (a function assigning data values to the 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 $\lambda\, x : \theta.M$) and s' a state, or special error configuration $\mathcal{E}$.

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

$$\frac{N_1, s_1 \Longrightarrow n_1, s_2 \qquad N_2, s_2 \Longrightarrow n_2, s}{\mathsf{op}_{\mathsf{int}_{\pi_1} \to \mathsf{int}_{\pi_2} \to \mathsf{int}_{\pi}}\ N_1\ N_2, s_1 \Longrightarrow n', s}\ n'_i \approx_{\pi_i} n_i,\ i = 1, 2,\ n' \approx_{\pi} \mathsf{op}\, n'_1\ n'_2$$

Assignment and dereferencing have similar nondeterministic 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 and results in $\mathcal{E}$. For any language operator op

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

2.1 Observational safety

A program M 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 $M, s \Longrightarrow \mathcal{K}$. If $\mathcal{K} \neq \mathcal{E}$ we say M is *safe*. Term $\Gamma \vdash M : \theta$ *approximates* term $\Gamma \vdash M' : \theta$, denoted by $\Gamma \vdash M \sqsubseteq_{\approx} M'$ if for all contexts $\mathcal{C}[-]$ and states s, if $\mathcal{C}[M], s \Longrightarrow \mathcal{K}$ then $\mathcal{C}[M'], s \Longrightarrow \mathcal{K}$. If two terms approximate each other they are considered *equivalent*, denoted by $\Gamma \vdash M \cong M'$.

A context is safe if it does not include occurrences of the abort command. A term M 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

Game semantics emerged in the last decade as a potent framework for modeling programming languages [2, 17, 3, 18, 11, 15]. It is an alternative (to Scott-Strachey) denotational semantics which interprets types as *arenas* (i.e. structured sets of atomic *moves*), computation as *plays* (i.e. structured sequences of moves) and terms as *strategies* (i.e. structured sets of plays). Strategies compose, much like CSP-style processes, which makes it possible to define denotational models. For technical details, the reader is referred to loc. cit.

Except for the presence of abort, AIA is syntactic sugar on top of IA with Erratic choice (EIA). We will use the may-termination model presented in [14, Chap. 3]. For any integer abstraction π, let $\mathsf{blur}_{\mathsf{expint}_{\pi}} : \mathsf{expint} \to \mathsf{expint}$ denote an EIA term which, given an integer n, returns a nondeterministically chosen integer n' such that $n' \approx_{\pi} n$.[4] For all other AIA types θ, we define EIA terms $\mathsf{blur}_{\theta} : \widetilde{\theta} \to \widetilde{\theta}$ as follows:

$$\mathsf{blur}_{\mathsf{expbool}} = \lambda\, x : \mathsf{expbool}.x \qquad \mathsf{blur}_{\mathsf{com}} = \lambda\, x : \mathsf{com}.x$$
$$\mathsf{blur}_{\mathsf{var}\tau} = \lambda\, x : \mathsf{var}\widetilde{\tau}.\mathsf{mkvar}\,(\lambda\, y : \mathsf{exp}\widetilde{\tau}.\mathsf{asg}\, x\, (\mathsf{blur}_{\mathsf{exp}\tau}\, y))\, (\mathsf{blur}_{\mathsf{exp}\tau}(\mathsf{der}x))$$
$$\mathsf{blur}_{\theta \to \theta'} = \lambda\, f : \widetilde{\theta} \to \widetilde{\theta'}.\, \lambda\, x : \widetilde{\theta}.\mathsf{blur}_{\theta'}(f(\mathsf{blur}_{\theta} x))$$

[4] Since abstractions are assumed computable, such terms are definable in EIA by iteratively testing all integers n'. However, in addition to the possibilities to choose any n' with $n' \approx_{\pi} n$ nondeterministically, there is the possibility of divergence. Therefore, this approach works only for may-termination semantics, which is sufficient in this paper.

For any AIA type θ, its translation $\ulcorner\theta\urcorner$ into EIA is $\widetilde{\theta}$. The translation of any AIA term into EIA is defined by:

$$\ulcorner \mathsf{k} : \theta \urcorner = \mathsf{blur}_\theta \mathsf{k} : \widetilde{\theta} \qquad \ulcorner M\ N : \theta \urcorner = \ulcorner M : \theta_1 \to \theta \urcorner \ulcorner N : \theta_2 \urcorner$$
$$\ulcorner x : \theta \urcorner = \mathsf{blur}_\theta x : \widetilde{\theta} \qquad \ulcorner \lambda\, x : \theta . M : \theta \to \theta' \urcorner = \lambda\, x : \widetilde{\theta} . \ulcorner M : \theta' \urcorner$$

The semantic model of AIA is therefore essentially that of EIA, which is presented in detail in [15]. Below, we give a sketch of the model.

An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where M_A is a set of *moves*, $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ is a function determining for each $m \in M_A$ whether it is an *Opponent* or a *Proponent move*, and a *question* or an *answer*. We write $\lambda_A^{OP}, \lambda_A^{QA}$ for the composite of λ_A with respectively the first and second projections. $\vdash_A$ is a binary relation on M_A, called *enabling*, satisfying: if $m \vdash_A n$ for no m then $\lambda_A(n) = (O, Q)$, if $m \vdash_A n$ then $\lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$, and if $m \vdash_A n$ then $\lambda_A^{QA}(m) = Q$. If $m \vdash_A n$ we say that m *enables* n. We shall write I_A for the set of all moves of A which have no enabler; such moves are called *initial*. Note that an initial move must be an Opponent question.

An arena is called *flat* if its questions are all initial (consequently the P-moves can only be answers). Flat arenas interpret base types, and are determined by their enabling relations:

$$\begin{array}{ll} \mathsf{com} : & \mathit{run} \vdash \mathit{done}, \mathit{abort} \\ \mathsf{exp}\tau : & q \vdash n, \mathit{abort} \\ \mathsf{var}\tau : & \mathit{read} \vdash n, \mathit{abort}, \quad \mathit{write}(n) \vdash \mathit{ok}, \mathit{abort}. \end{array}$$

The *product* $(A \times B)$ and *arrow* $(A \Rightarrow B)$ arenas are defined by:

$$\begin{array}{ll} M_{A \times B} = M_A + M_B & M_{A \Rightarrow B} = M_A + M_B \\ \lambda_{A \times B} = [\lambda_A, \lambda_B] & \lambda_{A \Rightarrow B} = [\langle \lambda_A^{PO}, \lambda_A^{QA} \rangle, \lambda_B] \\ \vdash_{A \times B} = \vdash_A + \vdash_B & \vdash_{A \Rightarrow B} = \vdash_A \cup \vdash_B \cup (I_B \times I_A) \end{array}$$

where $\lambda_A^{PO}(m) = O$ iff $\lambda_A^{OP}(m) = P$.

A *justified sequence* in arena A is a finite sequence of moves of A equipped with pointers. The first move is initial and has no pointer, but each subsequent move n must have a unique pointer to an earlier occurrence of a move m such that $m \vdash_A n$. We say that n is (explicitly) justified by m or, when n is an answer, that n answers m. A *legal play* is a justified sequence with some additional constraints. *Alternation* and *well-threaded-ness* are standard in game semantics, to which we add the following:

Definition 1 (Halting plays). *No moves can follow* abort.

This represents the abrupt termination caused by aborting. The set of all legal plays in arena A is denoted by P_A.

A *strategy* is a prefix-closed set of even length plays. Strategies compose in a way which is reminiscent of parallel composition plus hiding in process calculi.

We call a play *complete* if either the opening question is answered or the special move *abort* has been played.

Two strategies $\sigma : A \Rightarrow B'$ and $\tau : B'' \Rightarrow C$ can be composed by considering their possible interactions in the shared arena B (the decorations are only used to distinguish the two occurrences of this type). Moves in B are subsequently hidden yielding a sequence of moves in A and C.

Let u be a sequence of moves from arenas A, B', B'' and C with justification pointers from all moves except those initial in C, such that pointers from moves in C cannot point to moves in A and vice versa. Define $u \upharpoonright B'', C$ to be the subsequence of u consisting of all moves from B'' and C (pointers between A-moves and B''-moves are ignored). $u \upharpoonright A, B'$ is defined analogously (pointers between B' and C are then ignored).

Definition 2 (Interaction sequence). *We say that justified sequence u is an* interaction sequence *of A, B', B'' and C if:*

1. *any move from $I_{B''}$ is followed by its copy in $I_{B'}$,*
2. *any answer to a move in $I_{B'}$ is followed by its copy in $I_{B''}$*
3. *$u \upharpoonright A, B' \in P_{A \Rightarrow B'}$, $u \upharpoonright A, C \in P_{A \Rightarrow C}$, $u \upharpoonright B'', C \in P_{B'' \Rightarrow C}$.*

The set of all such sequences is written as $int(A, B, C)$. Composing the two strategies σ and τ yields the following set of interaction sequences:

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

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 its copy $m_{B'}$. The strategy which is the composition of σ and τ is then defined as $\sigma;\ \tau = \{u \upharpoonright A, C \mid u \in \sigma \mathbin{\fatslash} \tau\}$.

Strategies are used to give denotations to terms. Language constants, including functional constants, are interpreted by strategies and terms are constructed using strategy composition. Lambda abstraction and currying are isomorphisms consisting only of re-tagging of move occurrences. We interpret abort using the strategy $[\![\mathsf{abort}]\!] = \{\epsilon,\ run \cdot abort\}$.

3.1 Full abstraction

Using standard game-semantic techniques we can show that the above model is fully abstract for AIA.

Theorem 1 (Full abstraction). *For any terms $\Gamma \vdash M, M' : \theta$, $\Gamma \vdash M \sqsubseteq\!\!\!\approx M'$ iff $[\![\Gamma \vdash M : \theta]\!] \subseteq [\![\Gamma \vdash M' : \theta]\!]$.*

Proof (sketch). The proof follows the pattern of [14, Sec. 3.8]. In the presence of abort it is no longer necessary to use quotienting on strategies, as they are characterised by their full set of plays. The proof of this property is similar to that of the Characterisation Theorem for IA [3, Thm. 25]. The basic idea is that we can interrupt any play (not necessarily complete) by composing it with a stateful strategy that plays *abort* at the right moment. □

Note that in the presence of abort it is no longer the case that strategies are characterised by their set of complete plays, as it is the case for EIA. This is consistent with the fact that terms such as $c : \mathsf{com} \vdash c$; diverge and diverge are no longer equivalent although they both have same set of complete plays (empty). Command c may cause abort, thus preventing divergence. This is a common property of languages with control [18], and abort is such a feature.

Let us call a play *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). *$\Gamma \vdash M : \theta$ is safe iff $[\![\Gamma \vdash M : \theta]\!]$ is safe.*

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

3.2 Quotient semantics

Given a base type expint_π or varint_π of AIA, we can quotient the arena and game for expint or varint (respectively) in a standard way, by replacing any integer n with its equivalence class $\{m \mid m \approx_\pi n\}$. This extends compositionally to any type θ of AIA: we can quotient the arena and game for $\widetilde{\theta}$ by the abstractions in θ. For any play t of the game for $\widetilde{\theta}$, let $\overline{t}$ denote the image play of the quotient game, obtained by replacing each integer in t by its equivalence class in the corresponding abstraction in θ.

It is straightforward to check that, for any term $\Gamma \vdash M : \theta$ of AIA, and plays t and t' of the game for $\widetilde{\Gamma} \vdash \widetilde{\theta}$, such that $\overline{t} = \overline{t'}$, we have

$$t \in [\![\Gamma \vdash M : \theta]\!] \;\Leftrightarrow\; t' \in [\![\Gamma \vdash M : \theta]\!]$$

Therefore, the quotient of the strategy $[\![\Gamma \vdash M : \theta]\!]$ by the abstractions in Γ and θ loses no information.

Moreover, the quotient strategies can be defined compositionally, i.e. by recursion on the typing rules of AIA. The most interesting case is functional application $\Gamma \vdash M\,N : \theta$, where $\Gamma \vdash M : \theta_1 \to \theta$, $\Gamma \vdash N : \theta_2$, and $\widetilde{\theta_1} = \widetilde{\theta_2}$. Since the abstractions in θ_1 and θ_2 may be different, we need to allow a move which contains an equivalence class c to interact with any move obtained by replacing c with some c' such that $c \cap c' \neq \emptyset$. Hence, even if the quotient strategies for M and N are deterministic, the one for $M\,N$ may be nondeterministic.

In the rest of the paper, $[\![\Gamma \vdash M : \theta]\!]$ will denote the quotient strategy.

Example 1. Consider the quotient strategy

$$[\![x : \mathsf{varint}_{[0,4]} \vdash x := !x +_{[0,4]\to[0,1]\to[0,3]} 1_{[0,1]} : \mathsf{com}]\!]$$

If the abstract value (i.e. equivalence class) 3 is read from the variable x, the result of the addition is >3, because it belongs to the abstraction $[0,3]$. When >3 is assigned to x which is abstracted by $[0,4]$, it is nondeterministically converted to either 4 or >4. Thus, the following are two possible complete plays:

$$\mathit{run}\,\mathit{read}_x\,3_x\,\mathit{write}(4)_x\,\mathit{ok}_x\,\mathit{ok}, \quad \mathit{run}\,\mathit{read}_x\,3_x\,\mathit{write}(>4)_x\,\mathit{ok}_x\,\mathit{ok}$$

3.3 Interaction semantics

In *standard* semantics, which is presented above, to obtain the strategy $[\![\Gamma \vdash M\ N : \theta]\!]$, the strategies $[\![\Gamma \vdash M : \theta_1 \to \theta]\!]$ and $[\![\Gamma \vdash N : \theta_2]\!]$ are composed, and moves which interact are hidden. (Here $\widetilde{\theta_1} = \widetilde{\theta_2}$.)

Let $\langle\!\langle - \rangle\!\rangle$ denote an alternative semantics, where moves which interact are not hidden. We call this the *interaction* semantics, and its building blocks interaction plays and interaction strategies.

For any term $\Gamma \vdash M : \theta$ of AIA, its interaction semantics can be easily reconciled with its standard semantics, by performing all the hiding at once. In the following, $- \restriction \Gamma, \theta$ indicates restriction to the arenas corresponding to base types occurring in Γ and θ.

Proposition 1. $[\![\Gamma \vdash M : \theta]\!] = \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle \restriction \Gamma, \theta$.

Standard plays are alternating sequences of Opponent and Player moves. Interaction plays in addition contain internal moves, which do not interact in subsequent compositions, but which record all intermediate steps taken during the computation.

Consider composing $\langle\!\langle \Gamma \vdash M : \theta_1 \to \theta \rangle\!\rangle$ and $\langle\!\langle \Gamma \vdash N : \theta_2 \rangle\!\rangle$ to obtain $\langle\!\langle \Gamma \vdash M\ N : \theta \rangle\!\rangle$. According to the definition of interaction sequences above, for any moves r_1 and r_2 whose types σ_1 and σ_2 (respectively) are corresponding base types in θ_1 and θ_2, and which interact, they are both recorded in $\langle\!\langle \Gamma \vdash M\ N : \theta \rangle\!\rangle$. Indeed, since we only have $\widetilde{\theta_1} = \widetilde{\theta_2}$, r_1 and r_2 may be different. However, if σ_1 and σ_2 are not types of integer expressions or integer variables, then $\sigma_1 = \sigma_2$ and $r_1 = r_2$. In such cases, when presenting interaction plays and strategies, we may record r_1 and r_2 only once, for readability.

Example 2. Consider the interaction strategy of the term in Example 1. Here is one of its complete interaction plays, corresponding to the second standard play in Example 1. Any internal move is tagged with the coordinates of the corresponding sub-term. For instance, $q_{2,1}$ is the question to the sub-term $!x$, which is the 1st immediate sub-term of $!x + 1$, which in turn is the 2nd immediate sub-term of $x := !x + 1$. Observe also the double occurrences of integer internal moves, in line with how interaction plays are composed. In this example, those pairs are equal because, in any functional application, any two corresponding abstractions are equal. An abstract value needs to be converted to another abstraction only within the strategy for asg, where a value with abstraction $[0, 3]$ is assigned to a variable with abstraction $[0, 4]$.

$$run\ q_2\ q_{2,1}\ q_{2,1,1}\ read_x\ 3_x\ 3_{2,1,1}\ 3_{2,1,1}\ 3_{2,1}\ 3_{2,1}\ q_{2,2}\ 1_{2,2}\ 1_{2,2}\ (>3)_2\ (>3)_2\ write(>4)_1\ write(>4)_x\ ok_x\ ok_1\ ok$$

The interaction semantics, rather than the standard semantics, will be used for the purpose of abstraction refinement. The reason is that, given an unsafe standard play of an abstracted term, it does not in general contain sufficient information to decide that it can be produced by the concrete version of the

term (i.e. that it is not a *spurious counterexample*), or to choose one or more abstractions to be refined for the next iteration.

In classical, stateful, abstraction-refinement an abstract counterexample to a safety property is guaranteed to be genuine if the computation was deterministic (or, at least, the nondeterminism was not caused by over-abstraction). In standard semantics, however, all internal steps within a computation are hidden. This results in standard strategies of abstracted terms in general not containing all information about sources of their nondeterminism.

Example 3. Consider the following abstracted term, with notation in Fig. 1:

$$\vdash \mathsf{newint}_{[]}\ x := 0_{[0,0]}\ \mathsf{in}\ \mathsf{if}\ (x \neq 0_{[0,0]})\ \mathsf{abort}\ \mathsf{skip} : \mathsf{com}$$

Its complete standard plays are *run abort* and *run ok*. In fact, its strategy is the same as the strategy of the EIA term abort or skip. However, the counterexample *run abort* is spurious, and the abstraction of x needs to be refined, but internal moves which point to this abstraction as the source of nondeterminism have been hidden.

4 Conservativity of abstraction

As interaction plays contain internal moves, we can distinguish those whose underlying computation did not pass through any nondeterministic branching that is due to abstraction.

Definition 3. *(a) Given integer abstractions π and π', and an abstract value (i.e. equivalence class) c of π, we say that converting c to π' is* deterministic *if there exists an abstract value c' of π' such that $c \subseteq c'$.*

(b) Given an abstracted operation $\mathsf{op} : \mathsf{exp}\tau_1 \to \mathsf{exp}\tau_2 \to \mathsf{exp}\tau$ and abstract values c_1 and c_2 of type τ_1 and τ_2 respectively, we say that the application of op to c_1 and c_2 is deterministic *if there exists an abstract value c of type τ such that $\forall\, v_1 \in c_1, v_2 \in c_2,\ \mathsf{op}\, v_1\, v_2 \in c$.*[5]

(c) An interaction play $u \in \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$ is deterministic *if each conversion of an abstract integer value in u is deterministic, and each application of an arithmetic-logic operator in u is deterministic.*

For abstractions π and π', we say that π' *refines* π if, for any equivalence class c' of π', there exists an equivalence class c of π such that $c' \subseteq c$. When π' refines π, and c is an equivalence class of π, we say that π' *splits* c if c is not an equivalence class of π'.

We say that a term $\Gamma' \vdash M' : \theta'$ *refines* a term $\Gamma \vdash M : \theta$ if $\widetilde{\Gamma'} = \widetilde{\Gamma}$, $\widetilde{M'} = \widetilde{M}$, $\widetilde{\theta'} = \widetilde{\theta}$, and each abstraction in $\Gamma' \vdash M' : \theta'$ refines the corresponding abstraction in $\Gamma \vdash M : \theta$.

Theorem 2. *Suppose $\Gamma' \vdash M' : \theta'$ refines $\Gamma \vdash M : \theta$.*

[5] Here we regard the abstract values tt and ff as singleton sets $\{tt\}$ and $\{ff\}$.

(i) *For any* $t \in [\![\Gamma' \vdash M' : \theta']\!]$, *we have* $\overline{t} \in [\![\Gamma \vdash M : \theta]\!]$. *The same is true for the* $\langle\!\langle - \rangle\!\rangle$ *semantics.*
(ii) *For any deterministic* $u \in \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$, *there exists* $t \in \langle\!\langle \Gamma' \vdash M' : \theta' \rangle\!\rangle$ *such that* $u = \overline{t}$.[6]

Proof. By induction on the typing rules of AIA. □

The following consequence of Corollary 1, Proposition 1 and Theorem 2 will justify the correctness of the abstraction refinement procedure.

Corollary 2. *Suppose* $\Gamma' \vdash M' : \theta'$ *refines* $\Gamma \vdash M : \theta$.

(i) *If* $[\![\Gamma \vdash M : \theta]\!]$ *is safe, then* $\Gamma' \vdash M' : \theta'$ *is safe.*
(ii) *If* $\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$ *has a deterministic unsafe interaction play, then* $\Gamma' \vdash M' : \theta'$ *is unsafe.*

5 Abstraction refinement

In the rest of the paper, we work with the 2nd-order recursion-free fragment of AIA. In particular, function types are restricted to $\theta ::= \sigma \mid \sigma \to \theta$. Instead of the functional constant new is more convenient to use the combinator $\mathsf{new}\tau\, x := E \,\mathsf{in}\, M$ which binds free occurrences of x in M. Without loss of generality, we consider only normal forms with respect to β-reduction.

An abstraction π is *finitary* if it has finitely many equivalence classes. A term is *finitely abstracted* if it contains only finitary abstractions.

A set of abstractions is *effective* if their equivalence classes have finite representations, and if conversions of abstract values between abstractions, and all arithmetic-logic operators over abstract values, are computable.

Proposition 2. *For any finitely abstracted term* $\Gamma \vdash M : \theta$ *with abstractions from an effective set, the set* $[\![\Gamma \vdash M : \theta]\!]$ *is a regular language. Moreover, an automaton which recognises it is effectively constructible. The same is true for the* $\langle\!\langle - \rangle\!\rangle$ *semantics.*

Proof. Since the abstractions are finitary, $\Gamma \vdash M : \theta$ can be seen as a term of 2nd-order recursion free EIA with finite data types and abort. We can extend the construction in [10] to obtain effectively an automaton which recognises $[\![\Gamma \vdash M : \theta]\!]$. Note that the construction in loc. cit. characterises strategies in terms of their *complete plays*, i.e. those plays in which the initial question is answered. However, in the presence of abort strategies are defined by their full sets of plays (Theorem 1), so to each finite state machine corresponding to the semantic definitions used in loc. cit. we apply a suitable prefix closure operator, which preserves the finite-state property.

To obtain an automaton for $\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$, interacting moves are tagged with sub-term coordinates rather than hidden. □

[6] This can be strengthened to apply to interaction plays which are deterministic with respect to the abstractions in $\Gamma' \vdash M' : \theta'$. The latter notion allows nondeterministic conversions of, and operator applications to, abstract values which are not split by the corresponding abstractions in $\Gamma' \vdash M' : \theta'$.

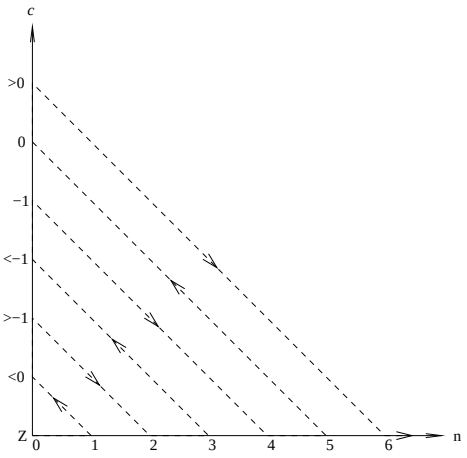

Fig. 2. A possible definition of $\sqsubseteq$.

Let $A[\![\Gamma \vdash M : \theta]\!]$ and $A\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$ denote the automata obtained as in the proof of Proposition 2. Since there is no hiding in the construction of $A\langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$, this automaton is deterministic.

Given a finite word u and a deterministic automaton A which accepts u, we call u *cycle-free* if the accepting run visits any state of A at most once.

Apart from the identity abstraction κ, for simplicity, from now on we work only with the abstractions $[]$ and $[n, m]$, where $n \leq 0 \leq m + 1$ (see Fig. 1). Observe that these abstractions are finitary and form an effective set.

Let $\prec$ denote the following computable linear ordering between abstract values:

$$\mathbb{Z} \prec (<0) \prec (>-1) \prec (<-1) \prec -1 \prec 0 \prec (>0) \prec \cdots \\ (<-(n+1)) \prec -(n+1) \prec n \prec (>n) \prec \cdots$$

For two moves (possibly tagged with sub-term coordinates) r and r' which are equal except for containing different abstract integer values c and c', let $r \prec r'$ if $c \prec c'$, and $r' \prec r$ if $c' \prec c$. Now, we extend this ordering to a computable linear ordering on all moves (in an arbitrary but fixed way), and denote it by $\prec$. Let $\prec$ also denote the linear orderings on plays obtained by lifting the linear ordering on moves lexicographically.

Let $(n, c) \sqsubseteq (n', c')$ be any computable linear ordering between pairs of non-negative integers and abstract integer values which is obtained by extending the partial ordering defined by $n \leq n'$ and $c \preceq c'$, and which admits no infinite strictly decreasing sequences, and no infinite strictly increasing sequences bounded above (see Fig. 2). For any play u, let $|u|$ denote its length, and $\max(u)$ denote the $\prec$-maximal abstract integer value in u (or $\mathbb{Z}$ if there is no such value). Let $u \sqsubseteq u'$ mean $(|u|, \max(u)) \sqsubseteq (|u'|, \max(u'))$. Now, let $\trianglelefteq$ be the linear order-

The procedure checks safety of a given concrete term $\Gamma \vdash M : \theta$.

1 Let $\Gamma_0 \vdash M_0 : \theta_0$ be a finitely abstracted anti-refinement of $\Gamma \vdash M : \theta$, i.e. be obtained from $\Gamma \vdash M : \theta$ by replacing κ by finitary abstractions. Let $i := 0$.
2 If $A[\![\Gamma_i \vdash M_i : \theta_i]\!]$ accepts only safe plays, terminate with answer SAFE.
3 Otherwise, if $A\langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$ accepts a deterministic unsafe interaction play, terminate with answer UNSAFE.
4 Otherwise, let u be the $\trianglelefteq$-minimal unsafe interaction play accepted by $A\langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$. Let $\Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1}$ be obtained by refining one or more abstractions in $\Gamma_i \vdash M_i : \theta_i$ by finitary abstractions, provided that at least one abstract value which occurs in u is split. Let $i := i + 1$, and repeat from **2**.

Fig. 3. Abstraction refinement procedure

ing between plays such that $u \trianglelefteq u'$ if and only if either $u \sqsubset u'$, or $|u| = |u'|$, $\max(u) = \max(u')$ and $u \preceq u'$.

Lemma 1. *In the linear order of all plays with respect to $\trianglelefteq$:*

(i) there is no infinite strictly decreasing sequence;
(ii) there is no infinite strictly increasing sequence which is bounded above.

Proof. This is due to the following two facts. Firstly, the $\sqsubseteq$ ordering between pairs of nonnegative integers and abstract integer values has the properties (i) and (ii). Secondly, for any such pair (n, c), there are only finitely many plays u such that $|u| = n$ and $\max(u) = c$. □

The abstraction refinement procedure (ARP) is given in Fig. 3. Note that, in step **1**, the initial abstractions can be chosen arbitrarily; and in step **4**, arbitrary abstractions can be refined in arbitrary ways, as long as that splits at least one abstract value in u. These do not affect correctness and semi-termination, but they allow experimentation with different heuristics in concrete implementations.

Theorem 3. *ARP is well-defined and effective. If it terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : \theta$ is safe (unsafe, respectively).*

Proof. For well-defined-ness, Lemma 1 (i) ensures that the $\trianglelefteq$-minimal unsafe interaction play u accepted by $A\langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$ always exists. Since the condition in step **3** was not satisfied, u is not deterministic. Therefore, u cannot contain only singleton abstract values, so there is at least one abstract value in u which can be split.

Effectiveness follows from Proposition 2, by the fact that it suffices to consider cycle-free plays in step **4**, and from computability of $\trianglelefteq$.

If ARP terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : \theta$ is safe (unsafe, respectively) by Corollary 2, since any abstraction is refined by the identity abstraction κ. □

Theorem 4. *If $\Gamma \vdash M : \theta$ is unsafe then ARP will terminate with UNSAFE.*

Proof. By Corollary 1 and Proposition 1, there exists an unsafe $t \in \langle\!\langle \Gamma \vdash M : \theta \rangle\!\rangle$.

For each i, let U_i be the set of all unsafe $u \in \langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$, and let $u_i^\dagger$ be the $\trianglelefteq$-minimal element of U_i.

It follows by Theorem 2 that, for any $u \in \langle\!\langle \Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1} \rangle\!\rangle$, $\overline{u} \in \langle\!\langle \Gamma_i \vdash M_i : \theta_i \rangle\!\rangle$. Also, we have $\overline{u} \trianglelefteq u$. Now, step **4** ensures that, for any i, $u_i^\dagger \notin \langle\!\langle \Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1} \rangle\!\rangle$.

Therefore, $u_0^\dagger \triangleleft u_1^\dagger \triangleleft \cdots u_i^\dagger \triangleleft \cdots$. But, for each i, $u_i^\dagger \trianglelefteq \overline{t}^i \trianglelefteq t$. By Lemma 1 (ii), ARP must terminate for $\Gamma \vdash M : \theta$! □

ARP may diverge for safe terms. This is generally the case with abstraction refinement methods since the underlying problem is undecidable. A simple example is the term

$$e : \mathsf{expint} \vdash \mathsf{newint}\, x := e \,\mathsf{in}\, \mathsf{if}\, (!x = !x + 1)\, \mathsf{abort}\, \mathsf{skip} : \mathsf{com}$$

This term is safe, but any finitely abstracted anti-refinement of it is unsafe.

6 Conclusions and related work

In this paper, we extended the applicability of game-based software model checking by a data-abstraction refinement procedure which applies to open program fragments which can contain infinite integer types, and which is guaranteed to discover an error if it exists. The procedure is made possible and it was justified by a firm theoretical framework. Some interesting topics for future work are dealing with terms which contain recursion, and extending to a concurrent programming language [12] or higher-order fragments [22].

The pioneering applications of game models to program analysis were by Hankin and Malacaria [13, 19–21], who also use nondeterminism as a form of abstraction. Their abstraction techniques apply to higher-order constructs rather than just data, by forgetting certain information used in constructing the game models (the *justification pointers*). It is an interesting question whether this style of abstraction can be iteratively refined. The first applications of game-semantic models to model checking were by Ghica and McCusker [10]. The latter line of research was further pursued as part of the *Algorithmic Game Semantics* research programme at the University of Oxford [1], and by Dimovski and Lazić [9].

On the topics of data abstraction [7] and abstraction refinement [6], there is a literature too vast to mention. Good entry points, which also represented essential motivation for our work, are the articles written on the SLAM model-checker [4]. It is too early to compare our approach with traditional, stateful, model-checkers. The first obstacle is the use of different target languages to express programs, but we hope to move towards more realistic target languages in the near future. The second obstacle stems from a difference of focus. Stateful techniques are already very mature and can target realistic industrial software; their overriding concern is efficiency. Our main concern, on the other hand, is *compositionality*, which we believe can be achieved in a clean and theoretically solid way by using a semantics-directed approach. In order to narrow the gap

between the efficiency of stateful tools and game-based tools, many program analysis techniques need to be re-cast using this new framework. Judging by the positive initial results, we trust the effort is worthwhile. Compositionality is a worthwhile long-term goal as compositional techniques are the best guarantee of scalability to large systems.

Aside from compositionality, one important advantage of game-based models is their small size, which is achieved by hiding all unobservable internal actions. However, in order to identify and analyse counterexample traces it is necessary, as we have pointed out in Sec. 3.3, to expose internal actions. In order to implement this abstraction refinement procedure reasonably, we must proceed by first identifying counter-example *standard* plays, and then obtaining corresponding *interaction* plays by "uncovering" the hidden moves. We are currently developing a model-checking tool based on representing strategies in the process algebra CSP [23], which can be verified using the FDR model checker. We can exploit a feature of FDR which allows identification of hidden events in counterexample traces, in order to implement the "uncovering" operation necessary to compute interaction plays efficiently.

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. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Proceedings of *TACAS*, LNCS **2280**, (2002), 158–172.
5. T. Ball and S. K. Rajamani. Debugging System Software via Static Analysis. n Proceedings of *POPL*, ACM SIGPLAN Notices **37**(1), (2002), 1–3.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Proceeding of *CAV*, LNCS **1855**, (2000), 154–169.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of *POPL*, (1977), 238–252.
8. P. Cousot and R. Cousot. Modular static program analysis. In Proceedings of *CC*, (2002).
9. A. Dimovski and R. Lazic. Csp representation of game semantics for second-order idealized algol. In Proceedings of *ICFEM*, LNCS **3308**, (2004), 146–161.
10. D. R. Ghica and G. McCusker. The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science **309** (1–3), (2003), 469–502.
11. D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency. In Proceedings of *FoSSaCS*, LNCS **2987**, (2004), 211-255.
12. D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. In Proceedings of *ICALP*, LNCS **3142**, (2004).

13. C. Hankin and P. Malacaria. Program analysis games. *ACM Comput. Surv.*, 31(3es), (1999).
14. R. Harmer Games and Full Abstraction for Nondeterministic Languages. Ph. D. Thesis Imperial College, 1999.
15. R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In Proceedings of*LICS*, (1999).
16. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In Proceedings of *SPIN*, (2003).
17. 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.
18. J. Laird. A fully abstract game semantics of local exceptions. In Proceedings of *LICS*, (2001).
19. P. Malacaria and C. Hankin. Generalised flowcharts and games. In Proceedings of *ICALP*, (1998).
20. P. Malacaria and C. Hankin. A new approach to control flow analysis. In Proceedings of *CC*, (1998).
21. P. Malacaria and C. Hankin. Non-deterministic games and program analysis: An application to security. In Proceedings of *LICS*, (1999).
22. A. Murawski and I. Walukiewicz Third-Order Idealized Algol with Iteration Is Decidable. In Proceedings of *FoSSaCS*, LNCS **3411**, (2005), 202–218.
23. W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.