On-the-Fly Techniques for Games-Based Software Model Checking

Adam Bakewell and Dan R. Ghica

University of Birmingham, U.K.
{a.bakewell,d.r.ghica}@cs.bham.ac.uk

Abstract. We introduce on-the-fly composition, symbolic modelling and lazy iterated approximation refinement for game-semantic models. We present MAGE, a model checker implementing this new technology, discuss several typical examples and compare MAGE with GAMECHECKER, the existing state-of-the-art in games-semantics based model checking.

1 Introduction and background

Automated software verification evolved rapidly in the last few years, culminating in the development of industry-strength verification toolkits such as SLAM [1] and BLAST [2]. These toolkits represent impressive feats of engineering, combining techniques from model checking [3] and theorem proving, especially satisfiability. They employ various methods intended to alleviate the so-called *state-explosion problem*, i.e. the fact that the space complexity of the software verification problem is very high. Some the most effective such methods are:

On-the-fly model checking Also known as *lazy* model checking [3, Sec. 9.5], it is used whenever a larger (finite-state) model needs to be constructed from the intersection of two (or more) models; after that, a reachability test is performed. In lazy model checking, the test is conducted while the intersection is performed, rather than after. If the test succeeds then the rest of the intersection is not computed, hence the gain in efficiency.

Symbolic model checking This terminology is overloaded. We mean representing a model by equations, rather than explicitly by concrete states and transitions [4].

Abstract interpretation The key idea [5] is to construct, in a precisely defined sense, *best safe approximations* of systems. That is, an "abstracted" system that is smaller than the system to be verified but has richer behaviour than it. Very large economies of space that can be achieved by this method, indeed, finite-state approximations can be found for infinite-state systems. The tradeoff is that additional behaviour in the "abstracted" system may lead to "false positives," i.e. it may report errors that do not exist in the original.

Iterated refinement This technique is used in conjunction with the previous one: if an approximation is too coarse and results in false positives, the false positives are used to *refine* the approximation, i.e. to make it more precise [6].

The success of the combined approaches enumerated above has been extraordinary, allowing for the successful fully-automated verification of a range of important programs such as device drivers. Even so, it is unrealistic to believe that further developments and refinements along the lines of improved algorithmic methods can scale up automated verification to large and complex software projects. Such large programs cannot be modelled and verified monolithically, but compositionally. They must be broken up into small subprograms which can be verified separately.

A promising new approach to software verification uses game semantics [7, 8]. This technique of modelling programming languages is inherently compositional, and was shown to give models both sound and complete (*fully abstract*) for many languages. Subsequent research showed that game models can be given effective algorithmic representations [9] and used as a basis for model checking.

Even a naive implementation of a model checker based on game semantics proved surprisingly effective in the verification of challenging programs such as sorting, or certain kinds of abstract data types [10]. In a step towards full automation of the verification process a counterexample-guided refinement technique was adapted to the game model [11], and a prototype tool was developed [12]. However, all these efforts focus on model extraction, and use off-the-shelf backends for the heavy-duty model checking.

Older, more established model checking techniques benefit from elaborate implementations. In order for games-based model checking to close the gap it needs to adapt the state-of-the-art methods for mitigating the state-explosion problem to the particular context of game models. We make significant steps in this paper by introducing *on-the-fly composition, symbolic modelling and lazy iterated refinement for game models*.

Game-based models are defined inductively on syntax and use *composition* of models of sub-terms to generate the model of a given term. This indicates that the scope for gains through lazy modelling is considerable. We push this method to the extreme: we do not explicitly construct any of the component models, only a *tree of automata*, then we combine a search through the tree with searches in the models which are at the leaves of the tree using an algorithm that is compatible with composition of game models.

We take a similar lazy approach to approximation and refinement. Rather than refining whole models, we only refine along those paths that yield counterexamples, refining further when the counterexample is potentially spurious and backtracking whenever refinement leads into a dead end.

Last, but not least, our model-checker, MAGE, has a simple (but not simplistic!) and elegant implementation. It uses no external tools or libraries, so it may serve as a concise, self-contained, example of the most effective state-of-the-art model checking techniques in action. Programming MAGE in Haskell allowed us to take advantage of lazy evaluation, and naturally resulted in a compact implementation.[1]

[1] Get MAGE and a test suite at `http://www.cs.bham.ac.uk/~axb/games/mage/`.

2 Syntax and semantics

We analyse IA (see Sec. 5 for examples), the procedural programming language presented in [11]. Its data types τ are booleans bool and integers int. The term types are base types σ of expressions $\mathsf{exp}\tau$, assignable variables $\mathsf{var}\tau$ and commands com, along with first-order function types $\theta ::= \sigma \mid \sigma \to \theta$.

IA has base-type constants, the usual imperative features (local variables, branching, iteration), λ-abstraction (of ground-type variables) and application.

The operational semantics of IA is standard, see [11]. It has a fully abstract game semantic model which can be expressed as an algebra of languages. We briefly present this model using notation taken from [10].

Game models of terms are languages R over alphabets of moves $\mathcal{A}$. They include the standard languages consisting of: the empty language $\emptyset$; the empty sequence ϵ; concatenation $R \cdot S$; union $R + S$; Kleene star R^* and the elements of the alphabet taken as sequences of unit length. In addition we use: intersection $R \cap S$; direct image under homomorphism ϕR and inverse image $\phi^{-1} R$. The languages defined by these extensions are the obvious ones. It is a standard result that languages constructed from regular languages using these operations are regular and can be recognized by a finite automaton effectively constructible from the language [13].

The disjoint union of two alphabets creates a larger alphabet $\mathcal{A}_1 + \mathcal{A}_2$. Disjoint union gives rise to canonical inclusion maps $\mathrm{in}_i : \mathcal{A}_i \to \mathcal{A}_1 + \mathcal{A}_2$. Concretely, these maps are *tagging* operations. We use the same notation for the homomorphism $\mathrm{in}_i : \mathcal{A}_i \to (\mathcal{A}_1 + \mathcal{A}_2)^*$ and take $\mathrm{out}_i : \mathcal{A}_1 + \mathcal{A}_2 \to \mathcal{A}_i^*$ to be the homomorphism defined by $\mathrm{out}_i a = a_i$ if a is in the image of in_i and ϵ otherwise. If $\phi_1 : \mathcal{A}_1 \to \mathcal{B}_1^*$ and $\phi_2 : \mathcal{A}_2 \to \mathcal{B}_2^*$ are homomorphisms then their sum $\phi_1 + \phi_2 : \mathcal{A}_1 + \mathcal{A}_2 \to (\mathcal{B}_1 + \mathcal{B}_2)^*$ as $(\phi_1 + \phi_2)a = \mathrm{in}_i(\phi_i a)$ if a_i is in the image of in_i.

Definition 1 (Composition). *If R is a language over alphabet $\mathcal{A} + \mathcal{B}$ and S a language over alphabet $\mathcal{B} + \mathcal{C}$ we define the* composition $S \circ R$ *as the language $S \circ R = out_3\big(out_1^{-1}(R) \cap out_2^{-1}(S)\big)$, over alphabet $\mathcal{A} + \mathcal{C}$, with maps*

$$\mathcal{A} + \mathcal{B} \underset{out_1}{\overset{in_1}{\rightleftarrows}} \mathcal{A} + \mathcal{B} + \mathcal{C} \ , \ \mathcal{B} + \mathcal{C} \underset{out_2}{\overset{in_2}{\rightleftarrows}} \mathcal{A} + \mathcal{B} + \mathcal{C} \ \text{ and } \ \mathcal{A} + \mathcal{C} \underset{out_3}{\overset{in_3}{\rightleftarrows}} \mathcal{A} + \mathcal{B} + \mathcal{C} \ .$$

Type θ is interpreted by a language over alphabet $\mathcal{A}[\![\theta]\!]$, containing the *moves* from the game model. Terms are functionalized, so $C; D$ is treated as $\mathsf{seq}\, C\, D$ and $\mathsf{int}\, x; C$ is treated as $\mathsf{newvar}(\lambda x.C)$ and so on. Term $\Gamma \vdash M : \theta$, with typed free identifiers $\Gamma = \{x_i : \theta_i\}$, is interpreted by a language $R = \mathcal{R}\,[\![\Gamma \vdash M : \theta]\!]$ over alphabet $\sum_{x_i : \theta_i \in \Gamma} \mathcal{A}[\![\theta_i]\!] + \mathcal{A}[\![\theta]\!]$. This interpretation is defined compositionally, by induction on the syntax of the functionalized language.

See [10, 11] for full details of the semantic model. Here we only emphasize the aspect that is most relevant to the model-checking algorithm: function application. The semantics of application is defined by

$$\mathcal{R}\,[\![\Gamma, \Delta \vdash MN : \theta]\!] = \mathcal{R}\,[\![\Delta \vdash N : \tau]\!]^* \circ \mathcal{R}\,[\![\Gamma \vdash M : \tau \to \theta]\!],$$

with the composition $- \circ -$ of Def. 1. This application model uses three operations: homomorphisms (tagging and de-tagging), Kleene-star and intersection.

At the level of automata manipulation, the first operation is linear time, the second is constant time and the third is $\mathcal{O}(m \cdot n)$ where m, n are the sizes of the automata to be composed. Clearly intersection dominates the problem, algorithmically. For a term with k syntactic elements, therefore, calculating the game model requires k automata intersections. Computing them explicitly will incur a huge penalty if, in the end, all that is wanted is a safety check (e.g. that some bad action never occurs). Hence on-the-fly techniques are particularly useful in this context.

3 On-the-fly composition

In this section we reformulate composition (Def. 1) twice. First, we write an automata-oriented, rather than language-oriented, definition that is equivalent for safety checking. Then, we further refine the definition in a way that is compatible with lazy evaluation.

We define an automaton $\mathbf{A} : A \to B$ as a tuple $\mathbf{A} = \langle S, A, B, X, \delta, s_0 \rangle$ where: S is a set of states, A, B are sets of symbols called *active* symbols, X is a set of symbols called *passive* symbols, $\delta : (A + B + X) \to S \to \mathcal{P}(S)$ is a *next state* function, $s_0 \in S$ is a distinguished *initial state*.

If $|S| \in \mathbb{N}$ then the automaton is *finite-state*. We say that an automaton $\mathbf{A}$ *accepts a string* $t \in (A+B+X)^*$ *from a set of states* S_0 if and only if $t = \epsilon$ and $S_0 \neq \emptyset$ or $t = m \cdot t'$ with $m \in A + B + X$ and $t' \in (A + B + X)^*$ such that $\mathbf{A}$ accepts t' from a state in $\delta m S_0$. If $S_0 = \{s_0\}$ we say just that $\mathbf{A}$ *accepts* t. We denote by $\mathcal{L}(\mathbf{A})$ the set of strings accepted by $\mathbf{A}$.

Definition 2 (Composition of automata). *Given two automata* $\mathbf{A}_1 : A \to B = \langle S, A, B, X, \delta, s_0 \rangle$ *and* $\mathbf{A}_2 : B \to C = \langle T, B, C, Y, \lambda, t_0 \rangle$ *we define their composition* $\mathbf{A}_2 \circ \mathbf{A}_1 = \big\langle S \times T, A, C, B + X + Y, \lambda \cdot \delta, \langle s_0, t_0 \rangle \big\rangle$ *where*

$$
\begin{aligned}
&A + B + X \xrightarrow{\text{in}_1} A + B + C + X + Y \\
&B + C + Y \xrightarrow{\text{in}_2} A + B + C + X + Y \\
&(\lambda \cdot \delta) m \langle s, t \rangle = (\textit{if } m \in \text{in}_1(A + B + X) \textit{ then } \delta m s \textit{ else } \{s\}) \\
&\qquad\qquad \times (\textit{if } m \in \text{in}_2(B + C + Y) \textit{ then } \lambda m s \textit{ else } \{t\})
\end{aligned}
$$

The language of composed automata is the required composition:

Proposition 1. *Given two automata* $\mathbf{A}_1 : A \to B$ *and* $\mathbf{A}_2 : B \to C$*,*

$$\text{out}_2(\mathcal{L}(\mathbf{A}_2)) \circ \text{out}_1(\mathcal{L}(\mathbf{A}_1)) = \text{out}(\mathcal{L}(\mathbf{A}_2 \circ \mathbf{A}_1)),$$

where $A+B+X \xrightarrow{\text{out}_2} A+B$*,* $B+C+Y \xrightarrow{\text{out}_1} B+C$*,* $A+B+C+X+Y \xrightarrow{\text{out}} A+C$*.*

This, and the subsequent propositions, have elementary proofs which are omitted. The Def. 2 formulation of automata composition is neither entirely lazy nor entirely eager. It is lazy in the sense that given a tuple $\langle s, t \rangle$ we can calculate the transitions from that composite state without constructing the

whole automaton, but it is eager in the sense that we must calculate *all* the transitions from the given composite state. Given the high branching degree in some game model, this can be a serious problem.

We give an (extensionally) equivalent definition of an automaton to emphasize on-the-fly composition. A *lazy automaton* $\tilde{\mathbf{A}} : A \to B$ is defined as before, except the *next state* function δ has type $\delta : (A + B + X) \to S \to \mathbb{N} \to S_\perp$, where $S_\perp = S + \{\perp\}$, such that $\delta msn = \perp$ implies $\delta ms(n+1) = \perp$.

The transition function of a lazy automaton gives the ith next-state, rather the entire set at once. The monotonicity of δ ensures that if requesting the jth next state returns "none" then requesting any $j+k$th next state will also return "none". It can easily be shown that any automaton $\mathbf{A}$ has an equivalent lazy automaton $\tilde{\mathbf{A}}$ such that $\mathcal{L}\tilde{\mathbf{A}} = \mathcal{L}\mathbf{A}$; we denote its transition function by $\tilde{\delta}$.

Definition 3 (Lazy composition of automata). *Given two automata* $\mathbf{A}_1 : A \to B = \langle S, A, B, X, \delta, s_0 \rangle$ *and* $\mathbf{A}_2 : B \to C = \langle T, B, C, Y, \lambda, t_0 \rangle$ *we define their* lazy composition $\mathbf{A}_2 \tilde{\circ} \mathbf{A}_1 = \left\langle S \times T, A, C, B + X + Y, \tilde{\lambda} \cdot \tilde{\delta}, \langle s_0, t_0 \rangle \right\rangle$ *where*

$$
\begin{array}{l}
\tilde{\lambda} \cdot \tilde{\delta} = (\tilde{\lambda} \star \tilde{\delta}) \circ \langle \mathrm{id}, \simeq \rangle \\
\mathrm{id} \textit{ is the identity function} \\
\simeq : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \textit{ is a monotonic bijection} \\
\perp = \langle s, \perp \rangle = \langle \perp, t \rangle \\
A + B + X \xrightarrow{\mathrm{in}_1} A + B + C + X + Y \\
B + C + Y \xrightarrow{\mathrm{in}_2} A + B + C + X + Y \\
(\tilde{\lambda} \star \tilde{\delta}) m \langle s, t \rangle \langle n_1, n_2 \rangle = \langle\, (\textit{if } m \in \mathrm{in}_1(A + B + X) \textit{ then } \tilde{\delta} msn_1 \textit{ else } s), \\
\qquad\qquad\qquad\qquad (\textit{if } m \in \mathrm{in}_2(B + C + Y) \textit{ then } \tilde{\lambda} mtn_2 \textit{ else } t) \rangle
\end{array}
$$

It is straightforward to show that the above is correct, i.e. $\mathcal{L}(\mathbf{A} \circ \mathbf{B}) = \mathcal{L}(\mathbf{A} \tilde{\circ} \mathbf{B})$.

In game models it is more natural to reduce safety to event reachability rather than to state reachability. Given an automaton $\mathbf{A}$ we say that event m is *reachable* if there exists string t such that $tm \in \mathcal{L}(\mathbf{A})$. We can now give an algorithm for (lazy) reachability of move m_0 in (lazy) automaton $\tilde{\mathbf{A}}$, using the composition method described above:

Definition 4 (Lazy reachability for lazy automata).

visited := $\emptyset$
frontier := $[s_0]$
iterate state *over* frontier
  visited := visited $\cup$ {state}
  iterate move *over* $(A + B + X)$
    iterate state′ *over* $\tilde{\delta}$ move state
      if move $= m_0$ *then return REACHABLE*
      if state′ $\notin$ visited *then* frontier := [state′] : frontier
return UNREACHABLE.

This algorithm is a depth-first-search (DFS) through the automata tree, generating only those branches that are necessary. The lazy implementation of $\tilde{\delta}$ ensures that the iterator over ($\tilde{\delta}$ Move State) returns one state at a time, rather than sets of states, until $\perp$ is produced and it stops.

In MAGE further improvements in performance are achieved in the implementation of the iterator over move set $A + B + X$. The set X can be very large but at any moment only some subset can lead to transitions. This subset of "plausible" moves can be identified by allowing the iterator to look at the state variable, which is a tree of "elementary" states contributed by the composed automata. This tells us which are the currently "active" automata and consequently which next-moves are possible.

3.1 Symbolic automata

In the tree of automata that models a term, the leaves are automata representing the constants of the language and the free identifiers. These can all be defined *symbolically*, further reducing memory requirements.

The strategies for free identifiers are given by the so-called *copy-cat strategies* [8], which simply copy information between a term and its environment. For instance, the strategy modelling a free identifier of natural type has plays of the shape $[\![x : \mathsf{nat} \vdash x : \mathsf{nat}]\!] = \{q \cdot x.q \cdot x.n \cdot n \mid n \in \mathbb{N}\}$. A symbolic representation of this language can be given by the automaton with states $S = \mathbb{Z}$, symbols $A = \{x.q\} \cup \{x.n \mid n \in \mathbb{N}\}$, $B = \{q\} \cup \mathbb{N}$, $X = \emptyset$, initial state $s_0 = -1$ and transitions $\delta q(-1) = \{-2\}$, $\delta x.q(-2) = \{-3\}$, $\delta x.n(-3) = \{n\}$, $\delta nn = \{-4\}$ (and $\delta ms = \emptyset$ otherwise). The corresponding lazy automaton representation is easily constructed. Similarly, the symbolic automaton of any arithmetic operator $\oplus$ has state set $S = \mathbb{N} \times \mathbb{Z} \times \mathbb{Z}$, initial state $s_0 = (0, 0, 0)$ and transitions

$$\begin{array}{ll} \delta q\,(0,0,0) = \{(1,0,0)\}, & \delta q\,(3,m,0) = \{(4,m,0)\}, \\ \delta q\,(1,0,0) = \{(2,0,0)\}, & \delta n\,(4,m,0) = \{(5,m,n)\}, \\ \delta m\,(2,0,0) = \{(3,m,0)\}, & \delta(m \oplus n)\,(5,m,n) = \{(6,0,0)\}. \end{array}$$

4 On-the-fly approximation and refinement

Because they involve large subsets of the integers, automata representing game-semantic models are defined over enormous alphabets and, consequently, have huge state sets. [11, 12] shows how to apply approximation-refinement in the context of games. We briefly describe this method in the context of automata.

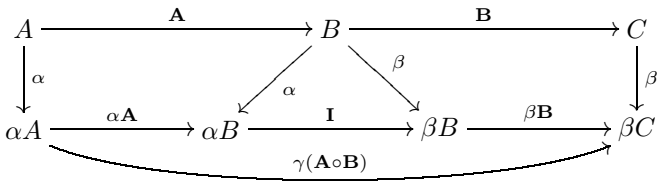
Two apparently insurmountable problems prevent us using the popular abstract interpretation framework of [15]. First, the automata-theoretic and game-theoretic formulations of the model seem to be at odds with the lattice-theoretic semantics of abstract interpretation. Second, and more seriously, abstract interpretation is compositional but not *functorial* — applying the abstract interpretation of a function to the abstract interpretation of an argument does not

necessarily yield the same as the abstract interpretation of the result of the application in the concrete domain [14]. [15] argues convincingly that the practical consequences of the requirement to preserve functoriality are too restrictive.

Therefore we use a simplified framework based only on *approximation*. An approximation of language $\mathcal{L}$ is a function $\alpha : \mathcal{L} \to \hat{\mathcal{L}}$. Interesting approximations are, obviously, non-injective. An *automaton approximation* for automaton $\mathbf{A} = \langle S, A, B, X, \delta, s_0 \rangle$ is a tuple $\alpha = \langle \alpha_S : S \to \hat{S}, \alpha_{A+B+X} : A+B+X \to \hat{A}+\hat{B}+\hat{X} \rangle$ which defines an automaton $\hat{\mathbf{A}} = \alpha(\mathbf{A}) = \langle \hat{S}, \hat{A}, \hat{B}, \hat{X}, \hat{\delta}, \hat{s}_0 \rangle$ where $\hat{s}_0 = \alpha_S(s_0)$ and $\hat{\delta}$ is any function such that $\hat{\delta}\hat{m}\hat{s} \supseteq \alpha_S(\delta m s)$ for any $m \in A + B$, $s \in S$, $\hat{m} = \alpha_{A+B+X} m$, $\hat{s} = \alpha_S s$. Approximation is sound in the following sense:

Proposition 2. *If $m \in A + B + X$ is reachable in automaton $\mathbf{A}$ then for any automata approximation α, $\alpha_{A+B+X}(m)$ is reachable in $\alpha(\mathbf{A})$.*

Given two automata $\mathbf{A} : A \to B = \langle S, A, B, X, \delta, s_0 \rangle$ and $\mathbf{B} : B \to C = \langle T, B, C, Y, \lambda, t_0 \rangle$ and two approximations α and β the resulting automata $\alpha\mathbf{A} : \alpha_A A \to \alpha_B B$ and $\beta\mathbf{B} : \beta_B B \to \beta_C C$ are not immediately composable. However, we can use a "glue" automaton $\mathbf{I} : \alpha_B B \to \beta_B B$ to perform the composition as indicated by the diagram below



A *glue automaton* $\mathbf{I} : \alpha B \to \beta B$ is an approximation of the "copy-cat" automaton on $B \to B$, i.e. an automaton that accepts strings of shape $(\Sigma_{m \in B} mm)^*$ which uses α_B to approximate the domain alphabet and β_B the codomain alphabet. Using glue automata we can show that approximation is compositional.

Proposition 3. *For any automata $\mathbf{A} : A \to B = \langle S, A, B, X, \delta, s_0 \rangle$ and $\mathbf{B} : B \to C = \langle T, B, C, Y, \lambda, t_0 \rangle$ and approximations α, β there exists an approximation γ such that $\beta\mathbf{B} \circ \alpha\mathbf{A} = \gamma(\mathbf{B} \circ \mathbf{A})$.*

This flexible approximation framework allows each automaton in an automata tree to be approximated individually, in a compositional and sound way.

Definition 5. *Given a language $\mathcal{L}$ and approximation $\alpha : \mathcal{L} \to \hat{\mathcal{L}}$, we call $\alpha' : \mathcal{L} \to \hat{\mathcal{L}}'$ a* refinement *of the approximation α if there exists a map $\alpha'' : \hat{\mathcal{L}}' \to \hat{\mathcal{L}}$ such that $\alpha = \alpha'' \circ \alpha'$.*

4.1 Approximating game automata

Approximations for game automata for the language fragment we use here are most naturally done by finitely approximating the alphabets and using an approximation of the set of states induced by the alphabet approximation.

Definition 6 (Data approximation). *An approximation α, is said to be a* data approximation *of automaton* $\mathbf{A}$ *if*

- $\hat{S} = S/{\cong}$, *and* α_S *is its representation function, where* $\cong \subseteq S \times S$ *is the least reflexive relation such that* $s_1 \cong s_2$ *if* $s_1' \cong s_2'$, $s_1 \in \delta m_1 s_1'$, $s_2 \in \delta m_2 s_2'$ *and* $\alpha_{A+B}(m_1) = \alpha_{A+B}(m_2)$.
- $\hat{\delta}\hat{m}\hat{s} = \alpha_S(\delta m s)$.

This means that the states of $\hat{S}$ are the equivalence classes of S under $\cong$. So states are identified by a data abstraction only when they are targets of transitions with identified moves from already identified states.

The definition of data approximation is not algorithmic, because it depends in a non-trivial way on the automaton itself. However, the following property along with the fact that we can rather easily find data approximations for the particular automata that represent game-semantic models ensures that we can use data approximation in our models:

Proposition 4. *If automata* $\mathbf{A} : A \to B$ *and* $\mathbf{B} : B \to C$ *are data-approximated as* $\alpha(\mathbf{A})$ *and* $\beta(\mathbf{B})$ *then there exists a data approximation* γ *for* $\mathbf{B} \circ \mathbf{A}$ *such that* $\alpha(\mathbf{B}) \circ \mathbf{I} \circ \beta(\mathbf{A}) = \gamma(\mathbf{B} \circ \mathbf{A})$, *with* $\mathbf{I} : \alpha B \to \beta B$ *a data-approximated glue automaton.*

In other words, a composition of data-approximate automata is itself a data-approximated automaton. Data-approximation can lead to finite-state automata.

Proposition 5. *For any automaton* $\mathbf{A}$ *representing a game-semantic model of IA and for any data approximation such that* $|\mathrm{range}\, \alpha_{A+B+X}| \in \mathbb{N}$, *the automaton* $\hat{\mathbf{A}}$ *is finite-state.*

We approximate game automata using data approximation. More precisely, we use partitions of the set of integers into a finite set of intervals, wherever necessary. The refinement of such an approximation using intervals is the obvious one: using smaller intervals.

Note that this approximation is compatible with a symbolic representation, as discussed in Sec. 3.1. Moreover, the approximate symbolic automata can be parameterized lazily by the approximation scheme. This is only interesting for arithmetic and logical operators. To implement their lazy and symbolic approximations we effectively need to extend the operator from acting on integers to intervals, in the obvious way. Every arithmetic operation $\oplus : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ becomes a *finite relation* $\hat{\oplus} \subseteq \alpha\mathbb{Z} \times \alpha'\mathbb{Z} \times \alpha''\mathbb{Z}$, defined as follows: $\big([m_1, m_1'], [m_2, m_2'], [m, m']\big) \in \hat{\oplus}$ if and only if $[m, m'] \in \alpha''\big([\min\{x_1 \oplus x_2 \mid x_i \in [m_i, m_i']\}, \max\{x_1 \oplus x_2 \mid x_i \in [m_i, m_i']\}]\big)$.

4.2 Fast early detection of counterexamples

As is well known, the converse of Prop. 2 is not true, since approximation can introduce new behaviour. A reachability test in an approximate automaton will

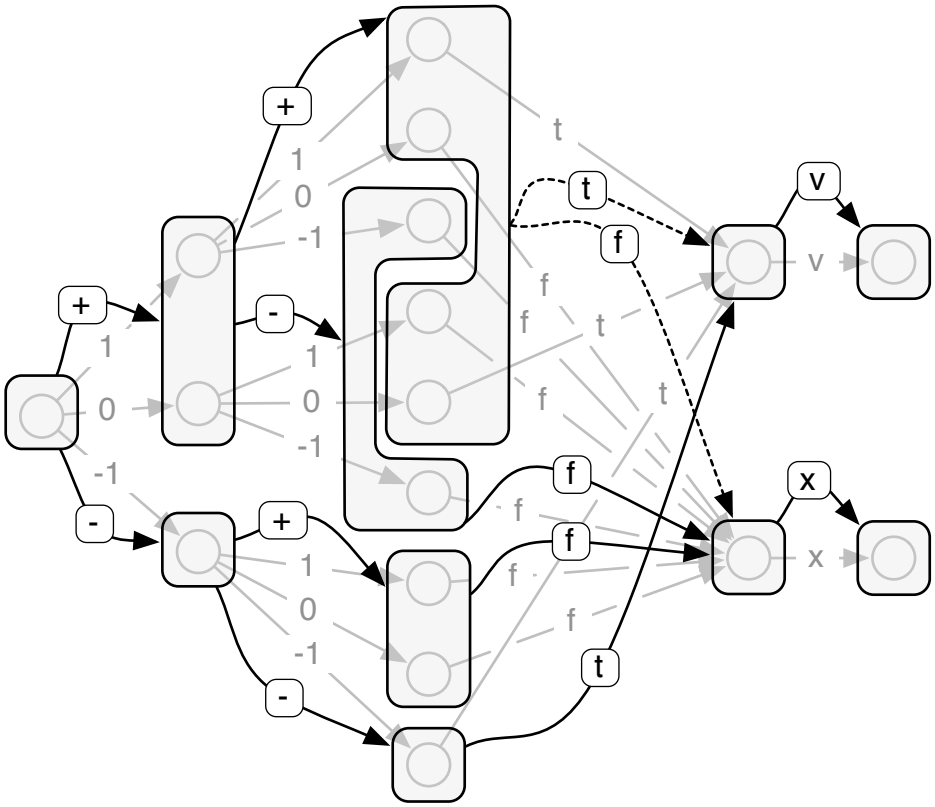


Fig. 1. Data-approximated automaton

return a string that needs to be "certified" for authenticity, i.e. that it indeed is the image, under approximation, of a string in the original automaton.

The usual approach in model checking is to analyse a counterexample trace using a SAT solver. We could follow that approach. However, by using domain-specific knowledge about the automata and the approximations we obtain a simpler and more efficient solution.

A trivial test for identifying valid counterexamples can be implemented starting from the following fact:

Proposition 6. *For any interface automaton* $\mathbf{A}$ *and data approximation* α*, if* $\hat{m}_0 \cdots \hat{m}_k \in \mathcal{L}(\hat{\mathbf{A}})$ *and* $\alpha^{-1}(\hat{m}_i) = \{m_i\}$ *then* $m_0 \cdots m_k \in \mathcal{L}(\mathbf{A})$.

In words, a trace is valid if it contains no approximated symbols. Note that this is only true of data-approximated automata. This test has the advantage that it can be carried out in linear time, but it has the major disadvantage that it requires a very "deep" refinement of a model. In MAGE we use a test that accepts traces with approximate symbols if the approximation does not cause

non-determinism in the approximated automaton for transitions that are deterministic in the precise version. Fig. 1 shows a simple data approximation of an automaton[2] that checks for equality on the set $\{-1, 0, 1\}$ by accepting two symbols then t if they are equal and f otherwise; if f, we mark "success" by symbol v and otherwise we mark "failure" by x. The data approximation is induced by $\alpha = \{-1 \mapsto -, 0 \mapsto +, 1 \mapsto +, t \mapsto t, f \mapsto f, x \mapsto x, v \mapsto v\}$. The precise automaton is grayed out and the approximated version superimposed. Approximated transitions that introduce nondeterminism are dashed (e.g. $+.+.f.x$); approximated transitions not introducing non-determinism (and which pass the test) are solid (e.g. $+.-.f.x$).

To apply this new criterion the counterexample must contain the visited states (as well as the symbols in the trace), but that only adds a constant-factor time and space algorithmic overhead.

Definition 7. *Given an automaton* $\mathbf{A}$*, a state* $s \in S$ *is said to be* forced *if for all* $(s, m', s'), (s, m'', s'') \in \delta$*,* $m' = m''$ *and* $s' = s''$*.*

Proposition 7. *For any automaton* $\mathbf{A}$*, data approximation* α *and sequence* $(\hat{s}_0, \hat{m}_0) \cdots (\hat{s}_k, \hat{m}_k)$ *such that* $(\hat{s}_j, \hat{m}_j, \hat{s}_{j+1}) \in \hat{\delta}$*, if* $\hat{s}_i$ *is forced whenever some state* $s \in \alpha_S^{-1}(\hat{s}_i)$ *is forced then* $m_0 \cdots m_k \in \mathcal{L}(\mathbf{A})$*.*

As before, this criterion is only valid for data approximation, and it spares the need for an expensive SAT test. Instead, we can use a simple, linear-time test. When automata compose, forced states in the components correspond to forced states in the composite automaton. It thus suffices to recognize when forced states become non-forced through approximation in leaf automata, and record this information whenever such states are visited, in order to indicate a trace that fails the test of Prop. 7 and requires further refinement.

5 Case studies and benchmarks

We present a small selection of simple case studies to illustrate our main techniques: laziness and symbolic models, approximation and refinement. Then we compare our tool's precise and approximation-refinement variants. Finally, we compare with a similar games-based approximation-refinement tool that does not use laziness. More examples are available on the MAGE web-page.

5.1 Lazy precise models

Consider the following simple implementation of a stack. It demonstrates well the advantages of checking open terms, which games-based methods enable. The (undefined) procedure `check` serves as a "most general environment" in which the stack can be used.

[2] For brevity, it is more concise than the corresponding IA game model.

```
uflo : com,           // exception called when empty stack popped
oflo : com,            // exception called when full stack pushed
input : nat%2,       // free var in {0,1} supplying pushed values
output : nat%2,      // free var in {0,1} receiving popped values
check : com -> com -> com                    // arbitrary context
|-
new nat%2[size] buffer in          // fixed-size stack of numbers
new nat%(size+1) top := 0 in         // first free buffer element
let push be
  if top=size then oflo                      // raise oflo if full
  else buffer[top] := input; top := top+1 fi  // push and inc top
in let pop be
  if top=0 then uflo                        // raise uflo if empty
  else top := top-1; output := buffer[top] fi  // pop and dec top
in check(pop,push)   // context can do any seq of pushes and pops
```

While considering precise models we use a stack of naturals in $\{0, 1\}$ — realistic integer types have very large precise models which can cause time and/or space problems.

To appreciate the scale of the model checking problem, note that the buffer and top have $\mathcal{O}(2^{\texttt{size}+1})$ states. MAGE can generate a representation of the full model (i.e. automaton) in the form of a set of traces that cover every path through the model without repeating or following loops; the number of such traces is $10 \times 2^{\texttt{size}-1} - 2$; it is not practical to try this for `size`s above about 13.

Tbl. 1 presents the time taken by MAGE to search the model (generated on-demand) until it encounters a call of `oflo` or `uflo` for a range of stack sizes.[3] The rapid generation of counterexamples clearly demonstrates the benefit of lazy model building. The times depend on stack size and model search order: with `pop` put first, the obvious `uflo` counterexample "pop empty" is generated immediately. With `push` put first, a longer counterexample that fills then empties then underflows the stack is found. Similarly, search order affects the `oflo` search time. MAGE can mitigate this effect by choosing transitions (when iterating over frontier in the terminology of Def. 4) in a randomized order instead; the last two columns in Tbl 1 show how this tends to average out the differences in search time due to order.

5.2 Approximation

Switching from precise to approximate model building cuts model size, introduces non-determinism and introduces possible false counterexamples. The MAGE approximation/refinement scheme begins by setting the approximation domain of each program variable to contain one value (the full range, determined by declared size).

[3] We used a 1.86GHz laptop PC; MAGE is compiled with GHC6.4.2 and run with a 250MB heap.

stack array size	fixed-order search time (sec) check(push,pop) oflo	uflo	check(pop,push) oflo	uflo	randomized search averaged time (sec) oflo	uflo
2	0.04	0.04	0.04	0.03	0.04	0.03
4	0.05	0.05	0.06	0.03	0.06	0.03
8	0.08	0.09	0.10	0.03	0.09	0.04
16	0.17	0.21	0.21	0.04	0.20	0.06
32	0.49	0.61	0.77	0.04	0.64	0.22
64	1.57	2.05	1.96	0.05	1.78	0.82
128	5.96	7.62	7.13	0.06	6.38	1.43
256	23.20	30.44	28.09	0.08	25.57	7.35
512	96.42	127.14	115.73	0.14	106.37	22.91
1024	433.43	575.56	525.27	0.30	501.02	189.26

Table 1. Lazy precise stack model searches.

In the very best case, searching the starting approximate model can reveal that a given condition never occurs. For example, searching the approximate model of the following program tells us straight away that no out-of-bounds element access can occur.

```
i:nat%4294967296, // 32bit natural number variable
a:nat%4294967296[4294967296] // array of 2^32 elements |-
a[i]:=i
```

The model has a single trace because the types exclude the possibility of `i` supplying out-of-bounds indices. The starting approximate model of the next example exhibits a typical false counterexample, for $\mathtt{x} = [0, 4294967295]$, which although approximate is identified by Prop. 7 as including a valid trace:

```
i:nat%4294967296 |-
new nat%4294967296 x := i in
assert (x<1000000000 | x>1000) abort   // abort if assertion fails
```

After three refinement iterations the domain of `x` is precise enough for MAGE to prove the program never aborts.

5.3 On-demand refinement

Refinement is implemented lazily: only paths in the automata tree that indicate potential counterexamples are refined. If a counterexample turns out spurious then the search backtracks to the next most-precise potential counterexample and continues from there. This example illustrate the lazy refinement algorithm as it backtracks searching for a solution to a constraint problem: Bill is twice as old as Ben was when Bill was as old as Ben is now. Their combined ages are 84. A program to solve this constraint problem is:

```
Bill:nat%85, Ben:nat%85 |-
new nat%85 bill := Bill in
new nat%85 ben := Ben in
new nat%85 diff := bill-ben in
(bill+ben = 84) & (bill = 2*(ben-diff))
```

A search of the precise model for a result of true with MAGE finds their correct ages in 256.53 seconds.

With approximation/refinement turned on, the ages are discovered in ten iterations taking only 6.5 seconds. The approximate values of `Bill` and `Ben` in the word found at each iteration are: [0,84] and [0,84]; [1,42] and [1,42]; [22,42] and [22,42]; [33,42] and [33,42]; [43,84] and [23,32]; [44,63] and 33; [45,53] and [34,37]; [46,49] and 35; [48,49] and [36,37]; 48 and 36.

Thus the various refinements and search backtracks mean that the solution is found when the domains of `Bill` and `Ben` are $\{0, [1, 21], 22, [23, 32], [33, 42], 43, 44, 45, 46, 47, 48, 49, [50, 53], [54, 63], [64, 84]\}$ and $\{0, [1, 21], 22, [23, 32], 33, 34, 35, 36, 37, [38, 42], [43, 84]\}$ Of course, this simple approach to refinement is most successful for continuous search spaces.

5.4 Comparison of precise vs. approximate modelling in MAGE

Quite informative is an analysis of the performance of on-the-fly model checking in precise models against that in approximated models, with refinement. The iterated-refinement times in Tbl. 2, compared with the precise searches on a stack of booleans in Tbl. 1, exhibit some pros and cons of approximation/refinement.

The `uflo` search is even quicker despite the increase in element domain size. The `oflo` searches are somewhat slower than the precise search. While the increased element size would make the precise search task impossible, this is not a major factor in the time increase seen here because, as with `uflo`, repeatedly pushing *any* precise value will cause an `oflo` — indeed, the timings are not much changed by reducing the element type right down to `nat%1`. What is going on instead, is that each iteration identifies that a chain of pushes of any value could lead to an error so the refinement "learns" to keep the approximation domain of the array elements very small, while each iteration makes the array index domain more precise; each array index must be written for an `oflo`, so over the course of log(size) iterations the array index types are refined down to be fully precise; this roughly doubles the number of distinct indices each time, so the refinement amounts to the same thing as searching for `oflo` in arrays with a tiny element domain and an index domain of size 2^i for each i from 0 to log(size).

It happens that the `oflo` counterexample is easy to find in that no backtracking from false counterexamples is needed. However, the search algorithm retains the information it needs should some backtracking be called for, so search with approximation/refinement tends to incur further slowing as the memory gradually fills with the backtrack queue. There is clearly potential to optimize the search process, perhaps with significant performance gains.

array	Mage		GameChecker	
size	`oflo` (iters)	`uflo` (iters)	`oflo` (iters)	`uflo` (iters)
2	0.05 (2)	0.03 (2)	10.14 (4)	5.31 (2)
4	0.09 (3)	0.03 (2)	27.53 (6)	8.21 (2)
8	0.17 (4)	0.03 (2)	112.56 (10)	20.29 (2)
16	0.40 (5)	0.03 (2)	780.67 (18)	78.26 (2)
32	1.15 (6)	0.03 (2)	12,268.12 (35)	494.20 (2)
64	3.86 (7)	0.03 (2)	>7 hrs (n/a)	8,982.13 (2)
128	13.94 (8)	0.03 (2)	-	>7 hrs (n/a)
256	54.78 (9)	0.03 (2)	-	-
512	215.32 (10)	0.03 (2)	-	-
1024	864.75 (11)	0.03 (2)	-	-

Table 2. Stack verification using Mage and GameChecker.

5.5 Comparison with GameChecker

GameChecker [11, 12] is a recent model checker based on game semantics that incorporates approximation and refinement. The main theoretical difference is that it does not use our on-the-fly/symbolic techniques; the main practical difference is that it is a Java front-end coupled with an industrial model checker as a back-end whereas Mage is implemented directly in Haskell. For a fair comparison we modify the stack example program so the stack elements are 32-bit integers and search for `oflo`'s and `uflo`'s using Mage with approximation/refinement on, and GameChecker using counterexample-guided refinement (on infinite integers). The results in Tbl. 2 support the intuition that the lazy techniques reap massive rewards: the GameChecker results show that building even approximate full models before analysing them incurs a severe penalty. As stack size increases, building a full model of stack behaviour before searching is almost all wasted work.

There is another reason for GameChecker's relative poor performance: it uses infinite integers and to avoid getting trapped into infinite refinement of spurious counterexamples, GameChecker uses a clever schedule of refinements at various points in the analysed program. For this technique to work, the model checker finds the smallest counterexample relative to a (rather complex) order. Termination is not an issue for us, as we operate in a finite setting. However, we believe that checking only programs with realistically large (i.e. 32 bit or more), instead of infinite, integers is a design decision vindicated by the massive improvement in performance.

6 Conclusion

Games-based software model checking offers the advantage of compositionality, which we believe is essential to scale up to larger programs. Early work in this area showed how the technique can be used in principle [10], and how the essential method of iterated refinement can be adapted to the model [11]. In this

work we take the next step in making this technique practical by incorporating lazy/on-the-fly modelling techniques with what appear to be massive gains in efficiency. To this end we implemented, and made available, the first model checker specifically targeted to take advantage of the multi-layered compositional nature of game models.

The choice of our target language was dictated by our desire to compare our work to previous work, but a switch to call-by-value can be easily accomplished. Concurrency can be also added using the work in [16]. Genuinely new developments that seem compatible with our approach are the introduction of recursion and higher-order functions, the game models of which admit finite-state over-approximations.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In: IFM. (2004) 1–20
2. Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: SPIN. (2005) 25–26
3. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
4. Ball, T., Rajamani, S.K.: BEBOP: A symbolic model checker for boolean programs. In: SPIN. (2000) 113–130
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
7. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. **163**(2) (2000) 409–470
8. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. **163**(2) (2000) 285–408
9. Ghica, D.R., McCusker, G.: The regular-language semantics of second-order idealized Algol. Theor. Comput. Sci. **309**(1-3) (2003) 469–502
10. Abramsky, S., Ghica, D.R., Murawski, A.S., Ong, C.H.L.: Applying game semantics to compositional software modeling and verification. In: TACAS. (2004) 421–435
11. Dimovski, A., Ghica, D.R., Lazic, R.: Data-abstraction refinement: A game semantic approach. In: SAS. (2005) 102–117
12. Dimovski, A., Ghica, D.R., Lazic, R.: A counterexample-guided refinement tool for open procedural programs. In: SPIN. (2006) 288–292
13. J.E. Hopcroft, J.D. Ullman: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
14. Abramsky, S.: Abstract interpretation, logical relations and Kan extensions. J. Log. Comput. **1**(1) (1990) 5–40
15. Cousot, P., Cousot, R.: Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In: Proc. Int'l Conf. on Computer Languages, Toulouse, France, IEEE Computer Society Press, Los Alamitos, CA (1994) 95–112
16. Ghica, D.R., Murawski, A.S.: Compositional model extraction for higher-order concurrent programs. In: TACAS. (2006) 303–317