On-the-Fly Techniques for Game-Based Software Model Checking[*]

Adam Bakewell and Dan R. Ghica

University of Birmingham, UK
{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, an experimental model checker implementing this new technology. We discuss several typical examples and compare Mage with Blast and GameChecker, which are the state-of-the-art tools in on-the-fly software model checking, and game-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 [6] and Blast [19]. These toolkits represent impressive feats of engineering, combining techniques from model checking [10] 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 [10, 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 [8].

Abstract interpretation. The key idea [11] 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 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.

[*] Work supported by EPSRC grants EP/D070880/1 and EP/D034906/1.

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 [9].

The success of the combination of methods enumerated above has been extraordinary, allowing tools to perform such feats as fully-automated verification of device drivers and other important programs. However, to scale up automated verification to large and complex software projects, modelling and verification cannot remain monolithic operations. Instead, they must be done compositionally, but in such a way that the above methods can be utilised.

A promising new approach to software verification uses game semantics [3,20]. This technique of modelling programming languages is inherently compositional, and known to give models both sound and complete (*fully abstract*) for many languages. Subsequent research showed that game models can be given effective algorithmic representations [15] and used as a basis for model checking.

Even a naïve implementation of game-based model checking was surprisingly effective in verifying challenging programs such as sorting, or certain abstract data types [2]. In a step towards fully automated verification a counterexample-guided refinement technique was adapted to the game model [13], and a prototype tool was developed [14]. However, all these efforts focus on model extraction, and use off-the-shelf back-ends 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] A compact presentation of our early results with MAGE is given in [4]. More detail on the material here is given in [5].

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

2 Idealized Algol: Syntax and Semantics

We analyse IA[2], the procedural programming language presented in [13]. IA has boolean and integer constants, the usual arithmetic and logical operators, sequencing, branching and iteration commands, first (base-type) and second order (function-type) variables, λ-abstraction of first-order variables, term application and block variable declaration.

The operational semantics is standard, see [13]. The game-semantic model is fully abstract and can be expressed as an algebra of languages. We briefly present this model using notation taken from [2].

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 recognised by a finite automaton effectively constructable from the language [21].

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*

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

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 [2,13] for full details of the semantic model. Here we only emphasise the aspect that is most relevant to the model-checking algorithm: function application. The semantics of application is defined by

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

[2] See the webpage for example programs.

with the composition $- \circ -$ of Def. 1. This application model uses three operations: (1) homomorphisms (tagging and de-tagging); (2) Kleene-star; (3) intersection. At the automata level: (1) is linear time; (2) the second is constant time; (3) is $\mathcal{O}(m \cdot n)$ where m, n are the sizes of the automata to be composed. Clearly intersection dominates. For a term with k syntactic elements, therefore, calculating the game model needs $k-1$ automata intersections. Computing them explicitly incurs a huge penalty if, in the end, we only want a safety check (e.g. that some bad action never occurs). Hence on-the-fly techniques are particularly useful in this context.

3 Automata Formulation: On-the-Fly Composition

We reformulate composition (Def. 1) to be explicitly automata-oriented, in a way that emphasises on-the-fly composition.

Let a *lazy automaton* $\mathbf{A} : A \to B$ be 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; $\delta : (A + B + X) \to S \to \mathbb{N} \to S_\perp$, where $S_\perp = S + \{\perp\}$, such that $\delta msn = \perp$ implies $\delta ms(n+1) = \perp$ is a next-state function that gives the ith next-state (rather than giving a set of all next states); $s_0 \in S$ is a distinguished *initial state*.

If $|S| \in \mathbb{N}$ then the lazy automaton is *finite-state*. Lazy automaton $\mathbf{A}$ *accepts a string* $t \in (A+B+X)^*$ *from a set of states* S_0 iff $t = \epsilon$ 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 $\mathbf{A}$.

The monotonicity of next-state function δ ensures that if requesting the jth next state returns "none" then requesting any $j+k$th next state returns "none".

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

$$\begin{aligned}
&(\lambda \star \delta) m \langle s, t \rangle \langle n_1, n_2 \rangle = \langle (\textit{if } m \in \mathrm{in}_1(A + B + X) \textit{ then } \delta m s n_1 \textit{ else } s), \\
&\qquad\qquad\qquad\qquad\qquad (\textit{if } m \in \mathrm{in}_2(B + C + Y) \textit{ then } \lambda m t n_2 \textit{ else } t) \rangle \\
&\textit{and } \lambda \cdot \delta = (\lambda \star \delta) \circ \langle \mathrm{id}, \simeq \rangle \\
&\textit{and } \perp = \langle s, \perp \rangle = \langle \perp, t \rangle \\
&\textit{and } A + B + X \xrightarrow{\mathrm{in}_1} A + B + C + X + Y \\
&\textit{and } B + C + Y \xrightarrow{\mathrm{in}_2} A + B + C + X + Y.
\end{aligned}$$

Above, id is the identity function and $\simeq : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ any monotonic bijection. The language of composed lazy automata is that required:[3]

Proposition 1. *Given two lazy automata* $\mathbf{A}_1 : A \to B$ *and* $\mathbf{A}_2 : B \to C$, $\mathrm{out}_2(\mathcal{L}(\mathbf{A}_2)) \circ \mathrm{out}_1(\mathcal{L}(\mathbf{A}_1)) = \mathrm{out}(\mathcal{L}(\mathbf{A}_2 \circ \mathbf{A}_1))$, *where* $A + B + X \xrightarrow{\mathrm{out}_2} A + B$, $B + C + Y \xrightarrow{\mathrm{out}_1} B + C$, $A + B + C + X + Y \xrightarrow{\mathrm{out}} A + C$.

[3] The propositions have elementary proofs, which are omitted.

Above, we need to "project" the languages of the composite automata on their active symbols, because automata compose "without hiding." This move from the "black-box" models of game semantics to "grey-box" models allows some exposure of internal actions and is needed to identify spurious counterexamples.

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

Definition 3 (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* $\delta$ 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 necessary transitions. The lazy implementation of δ ensures that iteration over δ move state returns one state at a time, rather than sets of states, until $\perp$ is produced and it stops.

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: instead of constructing the transition system corresponding to the leaf automata explicitly, as in the older games-based model checkers [2,14] we only represent the transition function of the automaton. This may sound silly, because the transition function *is* the automaton, and they have the same size (theoretically). However, many of the automata involved have particular structures (copy-cat, arithmetic, logic) and their transition functions have efficient implementations in the programming language in which the model checker is implemented (and, of course, on the underlying hardware). Addition of finite integers, for example, is implemented far more efficiently than a table of all possible pairs of operands and their results!

For example, 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, for $m, n \in \mathbb{Z}$, transitions:

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

3.2 Implementing Efficient Lazy Composition

The automata-theoretic formulation of lazy composition in Definition 2 omits a key aspect of the original game model which leads to serious inefficiency if implemented literally.

The problem occurs for active symbols common to the automata being composed: the definition suggests that both sub-automata should be queried about their transition for each such symbol. By analogy with abstract machines, this is like implementing application by taking each value v in the argument type, asking the argument term if it can produce v, and asking the function what it will do with v, and proceeding whenever both respond positively!

The key aspect that must be restored is the "proponent/opponent" (i.e. input/output) polarity of the game-semantic moves. At every composite state, one component must be asked about its next move and the other component asked only about particular moves. Mage records the necessary polarity information and acts accordingly.

Another key inefficiency in Definition 2 is the iteration over *all* moves in $A + B + X$. In practice, knowing which leaf in the automata tree will be asked about its next move dramatically reduces the set of possible next moves: there is only really a choice when a free variable reads from the environment.

4 CEGAR: 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. [13] shows how to apply approximation-refinement in the context of games. We develop the ideas there in several directions by generalising the definition of data abstraction from games to automata in general and by giving a general and efficient criterion for recognising genuine counterexamples in approximated automata. This fast detection criterion plays in important role in the efficient implementation of approximation-refinement in Mage.

Two apparently insurmountable problems prevent us using the popular abstract interpretation framework of [12]. Firstly, the automata-theoretic and game-theoretic formulations of the model seem to be at odds with the lattice-theoretic semantics of abstract interpretation. Secondly, 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 [1]. [12] 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

$$\begin{tikzcd}
A \arrow[rr, "\mathbf{A}"] \arrow[d, "\alpha"] & & B \arrow[rr, "\mathbf{B}"] \arrow[dl, "\alpha"'] \arrow[dr, "\beta"] & & C \arrow[d, "\beta"] \\
\alpha A \arrow[r, "\alpha\mathbf{A}"] \arrow[rrrr, bend right, "\gamma(\mathbf{A}\circ\mathbf{B})"'] & \alpha B \arrow[rr, "\mathbf{I}"] & & \beta B \arrow[r, "\beta\mathbf{B}"] & \beta C
\end{tikzcd}$$

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 $(\Sigma_{m \in B} m m)^*$ which uses α_B to approximate the domain alphabet and β_B the co-domain 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 4. *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

Approximation of our game automata is most naturally done by finitely approximating the alphabets and using an approximation of the set of states induced by the alphabet approximation.

Definition 5 (Data approximation). *An approximation α, is termed 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.

This approximation is compatible with the symbolic representation discussed in Sec. 3.1. Moreover, 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 extend the operators 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 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* $m_0 \cdots m_k \in \mathcal{L}(\mathbf{A})$.

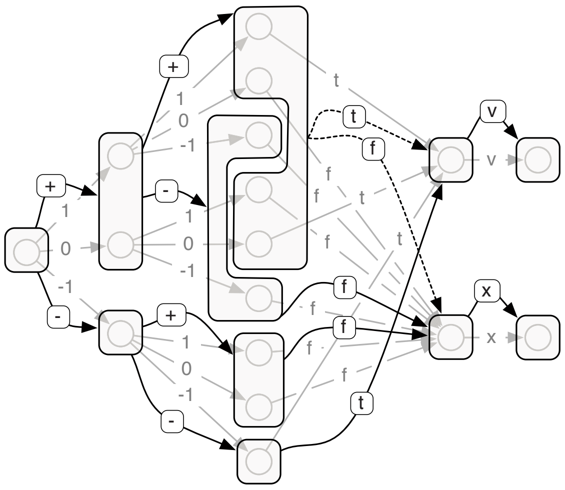

Fig. 1. Data-approximated automaton

In words, a trace is valid if it contains no approximated symbols. This is only true of data-approximated automata. This test is linear time, but it requires a very "deep" refinement of a model. MAGE uses a test that accepts traces with approximate symbols when 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[4] 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 greyed 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 6. *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

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

states in the composite automaton. It thus suffices to recognise 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 Mage: Empirical Results and Comparisons

We present a small selection of simple case studies to illustrate our main techniques; and to compare MAGE with both non-lazy game-based model checking and lazy non-game-based model checking. The example programs are given in the appendix. They and many more are also available on the webpage.

5.1 Lazy Precise Models

```
uflo : com,            // exception called when empty stack popped
oflo : com,             // exception called when full stack pushed
input : nat1,         // free var in {0,1} supplying pushed values
output : nat1,        // free var in {0,1} receiving popped values
check : com -> com -> com                           // arbitrary context
|-
new nat32[size] buffer in           // fixed-size stack of numbers
new nat(log size) 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
```

The first example searches the model of the fixed-size integer stack ADT above for sequences of calls of its push and pop methods that will result in `uflo` (i.e. ’pop empty’) or `oflo` (i.e. ’push full’), for different stack sizes. This example is a classic model checking problem with a history in the games-literature making it suitable for comparison in later sections, as well as being a good example of how we can verify open terms. It is also a good example for testing our lazy techniques because it presents a huge model in which a few specific paths exhibit the properties of interest.

While checking the unapproximated model we make the stack elements be naturals in $\{0, 1\}$ — bigger integer types makes the precise models far too big. Table 1 presents the time for MAGE to search the lazy model for various stack sizes.[5] The rapid generation of counterexamples clearly demonstrates the benefit of lazy model building. The times depend on stack size and model search order: with the pop method given priority, the obvious `uflo` counterexample (“pop”) is generated immediately. With push prioritised, a longer counterexample that

[5] MAGE is compiled with GHC6.4.2 and run with a 250MB heap on a 1.86GHz PC.

Table 1. Lazy precise stack model verification with MAGE

stack size	fixed-order search times (sec) push prioritised oflo	uflo	pop prioritised oflo	uflo	randomized search averaged times (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

fills then empties then underflows the stack is found. Similarly, search order affects the harder `oflo` search problem. MAGE can mitigate this effect by choosing transitions (when iterating over frontier in the terminology of Def. 3) in a randomised order instead; the last two columns in Table 1 show how this tends to average out differences in search time caused by order.

5.2 Approximated Models and On-Demand Refinement

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 approximated domain of each program variable to contain one value (the full range, determined by declared size).

In the very best cases, searching the starting approximated model quickly reveals safety. The starting approximated model of the term
`new nat32 x := i in assert (x<1000000000 | x>1000)`
exhibits a typical false counterexample. After three refinement iterations the domain of `x` is precise enough for MAGE to prove the assertion.

In addition to the early stopping, refinement is lazy in that only paths in the automata that indicate potential counterexamples get refined. Spurious counterexample makes the search backtrack to the next most-precise potential counterexample and forget recent refinements. Consider this constraint problem:

Bill is twice as old as Ben was when Bill was as old as Ben is now.
Their combined ages are 84.

```
bill:nat7, ben:nat7 |-                          // 7-bit natural inputs
new nat7 y := bill in new nat7 ny := ben in        // read the inputs
(y + ny = 84) & (y = 2 * (ny - (y - ny)))           // age constraint
```

The above program returns true when its inputs are a solution. Searching its precise model with MAGE finds the correct ages in 256.5 seconds. Using approximation/refinement, the ages are found in only 6.5 seconds after four backtracks and six refinement iterations.

5.3 Comparison of Precise vs. Approximate Modelling in Mage

Returning to the stack example, we make the elements realistic 32-bit integers and show in Table 2 the iterated-refinement times. This exposes some pros and cons of approximation-refinement, compared to the precise model verification in Table 1. `uflo` search is even quicker, despite the increase in element domain size. The `oflo` searches get 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* value causes an `oflo` — indeed, the times are not much changed by reducing the element type right down to `nat1`. What is going on instead, is that each iteration identifies that a chain of pushes (of any value) could lead to `oflo` so the refinement "learns" to keep the approximation domain of the array elements vague, and each iteration makes the array index domain more precise; each array index must be written for an `oflo`, so over log(size) iterations the array index types are refined 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 a backtrack queue, so search with approximation/refinement tends to incur further slowing as the memory gradually fills with the backtrack queue. There is clearly potential to optimise the search process, perhaps with significant performance gains.

5.4 Comparison with GameChecker

GameChecker [13,14] is a recent game-based model checker that incorporates approximation and refinement. The main theoretical difference is that it does not use our on-the-fly or symbolic techniques; the main practical difference is that it is a Java front-end coupled to an industrial model checker whereas Mage is implemented directly in Haskell. For a fair comparison we modify the stack ADT program so the stack elements are 32-bit integers and search for `oflo`'s and `uflo`'s using Mage with approximation/refinement on, and with GameChecker using counterexample-guided refinement (on infinite integers). The results in Table 2 support the expectation that the lazy techniques should reap massive rewards: the GameChecker times 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.

A secondary reason for GameChecker's poorer performance is that it always uses infinite integers, and to avoid infinite refinement of spurious counterexamples it uses clever refinement schedules. This involves finding the smallest counterexample relative to a rather complex order. This is not an issue with the finite types in Mage, and the resultant performance gain seems to vindicate the decision to use realistically large (i.e. 32-bit or more) integers.

Table 2. CEGAR stack verification with MAGE, GAMECHECKER and BLAST

stack	MAGE				GAMECHECKER				BLAST	
size	`oflo`	(iters)	`uflo`	(iters)	`oflo`	(iters)	`uflo`	(iters)	`oflo`	(iters)
2	0.1	(2)	0.03	(2)	10.1	(4)	5.31	(2)	1.6	(2)
4	0.1	(3)	0.03	(2)	27.6	(6)	8.21	(2)	3.3	(4)
8	0.2	(4)	0.03	(2)	112.6	(10)	20.29	(2)	4.6	(8)
16	0.4	(5)	0.03	(2)	780.7	(18)	78.26	(2)	7.8	(16)
32	1.2	(6)	0.03	(2)	12,268.1	(36)	494.20	(2)	17.3	(32)
64	3.9	(7)	0.03	(2)	>7 hrs	-	8,982.13	(2)	43.7	(64)
128	13.9	(8)	0.03	(2)	-	-	>7 hrs	-	145.3	(128)
224	19.1	(9)	0.03	(2)	-	-		-	506.4	(224)
225	19.3	(9)	0.03	(2)	-	-		-	-	-
256	54.8	(9)	0.03	(2)	-	-		-	-	-
512	215.3	(10)	0.03	(2)	-	-		-	-	-
1024	864.7	(11)	0.03	(2)	-	-		-	-	-

5.5 Comparison with Other Model Checkers

From the multitude of non-game-based approaches to model checking, in this section we focus on what we regard as the leading tool based on *predicate abstraction*, BLAST. This is a useful comparison because BLAST has achieved significant performance improvements over SLAM, by incorporating laziness into the cycle of abstraction, verification and refinement. Thus we can think of MAGE vs. GAMECHECKER as a game-based analogue of BLAST vs. SLAM.

Of course, the game-based tools are experiments in pure model checking for a simple language whereas SLAM and BLAST are quite mature tools that handle the C language. As a simple example of a defect of pure model checking compared to the predicate abstraction tools, verifying the following with MAGE requires a search of all 2^{32} possible inputs whereas BLAST can declare the corresponding C code safe in a fraction of a second.

```
input : nat32 |- new nat32 x := input in assert (x != 1+x)
```

Approaches to laziness. Laziness in BLAST consists in rearranging the perfectly eager CEGAR cycle of SLAM which: (1) constructs a predicate-abstraction of the program [7]; (2) model-checks; (3) refines the abstraction using the counterexample. Instead, BLAST updates the predicate-abstraction while constructing the model, informed by a continuous examination of the counterexamples yielded.This makes tremendous savings by zooming in on program parts that need close examination and leaving the rest suitably abstract.

The incremental model building in MAGE is very different but like BLAST it builds and refines only the parts needed. The other big separation is that MAGE does not pick up "interesting" predicates from the program as a starting point; it just partitions the integers. Then refinement requires no syntactic manipulation of source code; instead we just change the model semantics. This allows spurious counterexamples to be identified without using an external theorem prover. In BLAST the theorem prover ends up dominating the verification process [18, p. 3].

The potential disadvantage is that the initial approximation in Mage is blind to any useful features in the program being analysed.

Stacks. Like Mage, Blast can detect `uflo`'s in less than a second for stack sizes into billions of elements. Blast detects `oflo` in a stack of size n after n iterations. Table 2 shows that we were able to do this without exhausting our resources for stacks up to 224 elements. Much of the poor performance of Blast with larger stacks is because it only generates a precise analysis of the first n iterations of a loop, and only extends n by one at each refinement iteration, so failures occurring only after very large numbers of iterations can be hard for it to find. By comparison, the Mage data refinement tends to home loop counters in on the number needed to generate the failure.

6 Conclusion

Games-based software model checking offers the advantage of compositionality, which we believe essential for scaling to larger programs. Early work in this area showed how the technique can be used in principle [2], and how the essential method of iterated refinement can be adapted to the model [13]. The present paper takes the next step in making this technique practical by incorporating lazy/on-the-fly modelling techniques, with apparent massive efficiency gains. We implemented, and made available, the first model checker specifically targeted to take advantage of the multi-layered compositional nature of game models.

Our choice of target language was dictated by a desire to compare Mage to previous work; 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.

Comparison with the state-of-the art model checker Blast suggests that for unsafe programs Mage is able to zoom in on the error faster. On safe programs Mage's total ignorance of specific predicates used in the program gives Blast a substantial edge. It should be noted that Mage consists of 2,250 lines of Haskell code, whereas Blast's source distribution is 64MB, excluding the required external theorem prover. Finding common ground between the semantic-direct approach of Mage and the syntax-oriented techniques of predicate abstraction might be the best way forward in automated software verification.

References

1. Abramsky, S.: Abstract interpretation, logical relations and kan extensions. J. Log. Comput. 1(1), 5–40 (1990)
2. Abramsky, S., Ghica, D.R., Murawski, A.S., Ong, C.-H.L.: Applying game semantics to compositional software modeling and verification. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 421–435. Springer, Heidelberg (2004)
3. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)

4. Bakewell, A., Ghica, D.R.: Game-Based Safety Checking with Mage. In: SAVCBS, pp. 85–87 (2007)
5. Bakewell, A., and Ghica, D. R. On-the-Fly Techniques for Game-Based Software Model Checking (extended report). Technical Report TR-07-8, School of Computer Science, University of Birmingham (2007)
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
7. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 203–213 (2001)
8. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
10. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
12. Cousot, P., Cousot, R.: Higher-order abstract interpretation, invited paper. In: Proc. 1994 International Conference on Computer Languages, pp. 95–112. IEEE Computer Society Press, Los Alamitos (1994)
13. Dimovski, A., Ghica, D.R., Lazic, R.: Data-abstraction refinement: A game semantic approach. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 102–117. Springer, Heidelberg (2005)
14. Ghica, D.R., Lazić, R.S., Dimovski, A.: A counterexample-guided refinement tool for open procedural programs. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 288–292. Springer, Heidelberg (2006)
15. Ghica, D.R., McCusker, G.: The regular-language semantics of second-order idealized Algol. Theor. Comput. Sci. 309(1-3), 1–3 (2003)
16. Ghica, D.R., Murawski, A.S.: Compositional model extraction for higher-order concurrent programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 303–317. Springer, Heidelberg (2006)
17. Henzinger, T., Jhala, R., Majumdar, R., Sanvido, M.: Extreme model checking (2003)
18. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast (2003)
19. Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 25–26. Springer, Heidelberg (2005)
20. Hyland, J.M.E., Ong, C.-H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163(2), 285–408 (2000)
21. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
22. Laird, J.: A fully abstract game semantics of local exceptions. In: LICS, pp. 105–114 (2001)