Game-Based Safety Checking with Mage

Adam Bakewell
University of Birmingham, UK
a.bakewell@cs.bham.ac.uk

Dan R. Ghica
University of Birmingham, UK
d.r.ghica@cs.bham.ac.uk

ABSTRACT

Mage is a new experimental model checker based on game semantics. It adapts several techniques including lazy (on-the-fly) modelling, symbolic modelling, C.E.G.A.R. and approximated counterexample certification to game models. It demonstrates the potential for truly compositional verification of real software.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms

Verification

Keywords

Software model checking, game models, symbolic automata, compositional verification, data approximation, refinement

1. GAME-BASED SAFETY CHECKING...

Game Models Intuitively, the game model of a program can be generated by calling the program with every possible combination of arguments; and when the program calls on one of its free identifiers returning every possible result. The game model is then the set of sequences of values passed in and out. Game models have the following key advantages.

1. **Compositionality** The model of a composite program $f(a)$ is obtained by applying a simple 'compose' rule to the models of f and a. Thus components can be modelled and checked independently.

2. **Full abstraction** That is, both soundness (presence of an error-action in the model implies a fault in the program) and completeness (all program faults are present as error-actions in the model).

3. **Black-box** The game model only reports observable actions. This inherent abstraction provides the usual benefits: code privacy, model concision, separate analysis of components.

Safety Checking In game models, program safety reduces to event reachability (i.e. the program passing out an error value or calling an exception). Building an automaton representating the model and searching the transition space of the automaton for error actions implements safety checking.

For simple languages like *Idealized Algol* (IA) [1], which have regular language game models, a finite-state automaton is constructed and a sound and complete safety check by exhaustive search *can* be realized, as in the first game-based model checker which was presented at SAVCBS 2003. The caveat is that for realistic types (e.g. 32-bit integers) the models are often too big, despite the black-box property.

More powerful features like recursive types and higher-order functions need infinite-state automata so state approximation and loss of soundness must be incorporated as usual.

Data Approximation The second games-based model checker, GameChecker [5] used *data approximation* and adapted the *CEGAR* (that is, "counterexample-guided approximation refinement") technique [4] to game models. This allows checking to begin with a very small model and gradually increase the precision of the data types in parts of the program that generate potential counterexamples. The results [6] show another success-in-principle: programs with realistic type signatures can be modelled and checked. But literal interpretation of the game-theoretic approach — build models from component models and pass the final product to a checker — makes analysis of large programs impracticable.

2. ...WITH MAGE

Mage Our new safety checker, Mage[1], makes several advances over the previous state-of-the-art that overcome some of the problems inherent to compositional black box models. These ideas, outlined below, give asymptotic improvements in the complexity of many safety checking problems. These big performance gains have been won by bending and breaking the game approach in various ways.

Lazy Safety Checking Models are big. But this should not be the barrier in safety checking because the result of safety checking is a verdict (and perhaps a counterexample); not a model. Thus, actually building a model then checking

[1] `http://www.cs.bham.ac.uk/~axb/games/mage/`.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM ISBN 978-1-59593-721-6/07/0009 ...$5.00.

it is very space inefficient. It is also very time inefficient if the model contains errors that show up early in the check. Mage generates parts of the model as they are demanded by the checker. And it stops as soon as it detects unsafety.

Symbolic Game Models The game models are regular languages represented as automata. To fit with lazy checking it is much better to implement models in an implicit — rather than constructed — form: an initial state and the next-state function suffice and we call this representation a symbolic model (cf. [2]). Symbolic composition is especially useful as it only considers parts of the model that are demanded by the checker: an integer function f model might have a different behaviour for each of its 2^{32} arguments but only those behaviours demanded by the possible values of the argument a are considered when generating the model of $f(a)$. Thus symbolic models are still *defined* compositionally but the checker can use information about the surrounding context to make a significant efficiency gain when searching the symbolic transitions.

Data Approximation We replaced integers with finite ranges. This breaks the soundness direction of full abstraction so in general only produces possible-counterexamples but can be very effective in eliminating error-free sub programs from the search and quickly detecting data-independent errors.

Approximated Counterexamples Data approximation adds behaviours to the model. Therefore counterexamples must be certified — i.e. is the image, under approximation, of an error in the unapproximated model. Model checkers usually analyse counterexamples with a SAT solver. Mage uses domain-specific knowledge to implement a simpler and more efficient solution: non-determinism on the path through the approximated model to the error indicates a possibly-false counterexample.

CEGAR Finding a possibly-false counterexample causes Mage to refine the re-check the model. Refinement means increasing the precision of the data approximations for those values that led to the counterexample. The symbolic model is refined simply by modifying the type annotation on affected variables. The model-check-certify-refine loop repeats until a true counterexample is found or every possibly-false counterexample is eliminated. Termination is guaranteed because each refinement makes a model strictly less approximate and ultimately the unapproximated models are finite.

Individuated Refinement It is a disadvantage to force different uses of the same variable to share the same approximation: approximate values needed at one site to generate unsafety are then considered at other sites, typically leading to more false counterexamples and more backtracking in the search and more refinement iterations than would otherwise happen. Mage identifies which variable site generated (or consumed) each value in a possible-counterexample. and refines the approximation used at each site individually.

Grey-box models To support the refinement and certification techniques we have to leak some information about internal actions. For certification this creates a constant overhead; for refinement the cost can be larger. So our models are not strictly black-box; merely as black as possible.

stack size	Mage	GameChecker	Blast
2	0.1	10.1	1.6
4	0.1	27.5	3.3
8	0.2	112.6	4.6
16	0.4	780.7	7.8
32	1.2	12,268.1	17.3
64	3.9	over 7 hours	43.7
128	13.9	-	145.3
256	54.8	-	space exhausted

Table 1: Stack overflow detection tests.

3. RESULTS

Stack Verification We compare Mage with the earlier CEGAR game-based checker GameChecker on the same verification problem. We also compare it with the powerful non-game-based model checker, Blast [7] (translating the problem from IA into C makes no semantic difference). Blast is a suitable non-game comparison because it also uses lazy modelling and refinement techniques and it represents the state of the art in verification based on *predicate abstraction* and it can verify significant applications such as device drivers. The problem is to discover contexts that lead to underflows and overflows in a stack of integers where the stack is represented by a finite array and the stack interface presents a push and a pop method that call exceptions when the empty stack is popped or a full stack is pushed.

Overflow Table 1 shows the time taken (on the same machine, in seconds) for the three tools to detect a context leading to an overflow for stacks of different sizes. The Mage times are roughly linear; GameChecker is exponential because it is dominated by model building; Blast is also roughly linear but suffers resource problems with larger stacks. Mage can handle stacks of thousands of elements.

Underflow For the underflow search problem the laziness of both Mage and Blast allow the counterexample “pop empty” to be discovered in a fraction of a second for stacks of billions of elementss. For GameChecker the need to build the model before checking causes similar (slightly faster) results to the overflow problem.

Future Prospects Results such as these suggest that the compositional games approach should be scalable to handle much larger software projects. Our research agenda is to extend the framework to a practical language such as C and then to combine the pure model checking with support from program analysis.

4. REFERENCES

[1] 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

[2] Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: SPIN. (2000) 113–130

[3] 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

[4] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169

[5] Dimovski, A., Ghica, D.R., Lazic, R.: Data-abstraction refinement: A game semantic approach. In: SAS. (2005) 102–117

[6] Dimovski, A., Ghica, D.R., Lazic, R.: A counterexample-guided refinement tool for open procedural programs. In: SPIN. (2006) 288–292

[7] Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast software verification system. In: SPIN. (2005) 25–26