MoonWalker: Verification of .NET Programs

Niels H.M. Aan de Brugh[1], Viet Yen Nguyen[2], and Theo C. Ruys[3]

[1] OCÉ, Venlo, The Netherlands
niels.aandebrugh@oce.com
[2] RWTH Aachen University, Germany
http://moves.rwth-aachen.de/~nguyen/
[3] University of Twente, The Netherlands
http://www.cs.utwente.nl/~ruys/

Abstract. MoonWalker is a software model checker for CIL bytecode programs, which is able to detect deadlocks and assertion violations in CIL assemblies, better known as Microsoft .NET programs. The design of MoonWalker is inspired by the Java PathFinder (JPF), a model checker for Java programs. The performance of MoonWalker is on par with JPF. This paper presents the new version of MoonWalker and discusses its most important features.

1 Introduction

This paper presents MoonWalker[1] 1.0 [18], a software model checker for the verification of CIL bytecode programs. CIL stands for Common Intermediate Language and is the platform independent bytecode used within Microsoft's .NET. MoonWalker targets programs compiled against the MONO development platform [16], an open source implementation of the .NET development platform.

MoonWalker is a software model checker that uses the *virtual machine* approach of verification: the effect of every CIL bytecode instruction is analysed by the tool. MoonWalker systematically explores all reachable states of the application under verification, which involves executing bytecode instructions, storing and restoring states, and checking for safety properties. During exploration, MoonWalker will check for deadlocks and assertion violations.

The approach of MoonWalker is inspired by the Java PathFinder (JPF) [12,15], a very successful software model checker for Java. JPF pioneered the concept of implementing a software model checker around a virtual machine. And although the object-oriented design and the actual implementation of MoonWalker (in C#) and organisation of the classes and algorithms are different, all credits for the verification approach should go to the developers of JPF.

With MoonWalker 1.0, however, there is now a competitive software model checker readily available for the .NET framework. An important advantage of CIL over Java bytecode is that CIL has been designed to be the target for many programming languages, not just C#. See [17] for a complete overview. Finally, version 1.0 of MoonWalker incorporates some new techniques not yet available in other model checkers.

[1] MoonWalker was previously known as MMC: the Mono Model Checker.

S. Kowalewski and A. Philippou (Eds.): TACAS 2009, LNCS 5505, pp. 170–173, 2009.
© Springer-Verlag Berlin Heidelberg 2009

XRT [5] is an alternative software model checker for .NET, which follows the same approach as JPF. XRT is not publicly available.

2 MoonWalker 1.0

The architecture of MoonWalker 0.5 has been outlined in [11] and the design and implementation of version 0.5 are described in detail in [1]. This section discusses the new features that have been added to MoonWalker over the past $1\frac{1}{2}$ years. Details can be found in [7], available from [18].

Several improvements have been made to MoonWalker 1.0 to enhance its usability, including a user-friendly error tracer, an extensive test framework, and an implementation of structured exception handling. Furthermore, two partial order reduction (POR) [4] techniques have been implemented into MoonWalker 1.0: (i) POR using object escape analysis (which is also used by JPF) and (ii) stateful dynamic POR. Finally, two novel techniques have been implemented in MoonWalker 1.0, which will be discussed in more detail below.

Memoised Garbage Collector. MoonWalker 0.5 used the well known Mark & Sweep algorithm (M&S) for garbage collection (GC). A drawback of M&S is that it is global: for each invocation (i.e. after each transition), the whole heap is visited twice to remove dead objects. In other words, M&S cannot exploit the locality of transitions within a (software) model checker.

For MoonWalker 1.0 we took a different approach, which is based on an incremental shortest-path algorithm for single-source directed graphs with positive weights [9]. We devised and implemented the Memoised Garbage Collector (MGC) algorithm, which uses information retrieved from changes between successive states to determine which objects should be garbage collected.

The basic idea is to track for each object in the heap its depth from the root elements (on the call stacks of the threads). Upon changes to the heap, the tracked depth of the changed objects become inconsistent, and their depths need to be recalculated. When the changes to the heap are small – which they usually are – only a small part of the heap needs to be traversed. If the depth of an object becomes infinity, we know that the object has become unreachable.

We know of one other software model checker which uses a non-global garbage collector: JNuke [2]. JNuke uses a generational garbage collection (GGC) as described in [3]. Although the objectives of both non-global GC algorithms are the same, the implementations are substantially different. MGC is provable precise [7], whereas GGC is not, because the latter exploits the heuristic that only new objects are likely to be garbage collected. For MoonWalker unpreciseness is undesired because this may cause the state matcher to determine that two semantically equivalent states are different [6].

MGC has a better time-complexity than M&S, which is the dominant garbage collection in use by software model checkers. Experiments showed that the use of MGC increases performance of about 10-25%, depending on the model and its state space. Details on MGC can be found in [7,8].

Collapsing Interleaving Information. The dynamic POR algorithm by Flanagan & Godefroid [4] only works correctly for *stateless* exploration. The issue lies in

the correct dynamic POR semantics upon a state revisit. A naive and incorrect stateful adaptation of dynamic POR would backtrack upon exploration of a revisited state. This is incorrect, because mutual dependencies between transitions in the state space below the revisited state and the current path to the revisited state would not be considered. This leads to over-aggressive reduction. Both [10] and [13] independently observed this, and proposed similar solutions. The idea is to mimic a stateless search upon a revisit by recalling all necessary *interleaving information* about the state space below the revisited state and inject the appropriate transitions in the working sets on the current DFS stack.

[10,13] observe that stateful dynamic POR uses a lot of memory and suggest (as future work) to compress the interleaving information used for stateful dynamic POR. MoonWalker 1.0 improves upon [13] by compressing the interleaving information by canonicalisation followed by collapse compression. The collapse compression step exploits the notion that the interleaving information of states do not change much between successive states. We reuse the structered state collapsion scheme that was already present in MoonWalker 0.5.

Experiments show (again depending on the model and the state space) that dynamic POR may reduce the memory consumption by a factor of two.

Implementation. The current version of MoonWalker is version 1.0. The total development of the tool took roughly two man years of work. The code base of MoonWalker 1.0 consists of 17k lines of C# code and constitutes 475Kb of source code. Both a binary and source distribution are available from [18].

MoonWalker 1.0 supports 74 CIL bytecode instructions. These are all possible instructions that can be emitted by MONO's C# 1.x compiler. Support for the last nine missing instructions is future work.

Experiments. Apart from using small experiments that synthesise a small scenario, we also used the *Java Grande Forum Benchmarks (JGF)* [14] for evaluating MoonWalker against JPF. JGF is a mature benchmark suite developed for the scientific community, which contains real life examples. Of the three multithreaded parallel benchmarks within this suite, we used the *MolDyn* benchmark (loc: 965, size: 26Kb) and *Raytracer* benchmark (loc: 1540, size: 49Kb). We ported these two benchmarks to C# for use with MoonWalker.

Results show that MoonWalker and JPF are on par in terms of performance. Differences between the two tools are small. Both tools typically explore about 1000-5000 states/sec. MoonWalker is faster in terms of states per second, but JPF is better at reducing the state space because its POR object escape analysis algorithm also uses locking information. The results also indicate that MoonWalker utilises memory relatively less efficiently than JPF. This is caused by the memory overhead incurred by stateful dynamic POR. Details on the experiments can be found in chapter 5 of [7].

3 Conclusions

In this paper we presented MoonWalker 1.0, a model-checker for CIL bytecode programs. Due to several refactorings, the design and implementation of

MoonWalker is clear, readable and extensible. We feel that MoonWalker is a useful platform in an academic environment where ease of experimentation with different implementations is an important virtue. To extend the usability of MoonWalker further, several improvements are planned:

- Further improvements to POR and state compression;
- Optimisations to the (memoised) garbage collector;
- Mixing of symbolic and concrete data;
- Multi-threaded and distributed version of MoonWalker;
- Case studies with other programming languages than C#;
- Support for C# 3.0 and .Net 3.5.

References

1. Aan de Brugh, N.H.M.: Software Model Checking for Mono. Master's thesis, University of Twente, Enschede, The Netherlands (August 2006)
2. Artho, C., Schuppan, V., Biere, A., Eugster, P., Baur, M., Zweimüller, B.: JNuke: Efficient Dynamic Analysis for Java. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 462–465. Springer, Heidelberg (2004)
3. Fargas, P.: Garbage Collection for JNuke. Master's thesis, ETH Zürich, Switzerland (September 2004)
4. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model Checking Software. In: Proc. of POPL 2005, pp. 110–121. ACM Press, New York (2005)
5. Grieskamp, W., Tillmann, N., Schulte, W.: XRT: Exploring Runtime for. NET - Architecture and Applications. ENTCS 144(3), 3–26 (2006); Proc. of SoftMC 2005
6. Iosif, R., Sisto, R.: Using Garbage Collection in Model Checking. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 20–33. Springer, Heidelberg (2000)
7. Nguyen, V.Y.: Optimising Techniques for Model Checkers. Master's thesis, University of Twente, Enschede, The Netherlands (December 2007)
8. Nguyen, V.Y., Ruys, T.C.: Memoised Garbage Collection for Software Model Checking. In: Knowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 201–214. Springer, Heidelberg (2009)
9. Ramalingam, G., Reps, T.W.: An Incremental Algorithm for a Generalization of the Shortest-Path Problem. Journal Algorithms 21(2), 267–305 (1996)
10. Ranganath, V.P., Hatcliff, J., Robby.: Enabling Efficient Partial Order Reductions for Model Checking Object-Oriented Programs. Technical Report SAnToS-TR2007-2, SAnToS Laboratory, CIS Department, Kansas State University (2007)
11. Ruys, T.C., Aan de Brugh, N.H.M.: MMC: the Mono Model Checker. ENTCS 190(1), 149–160 (2007); Proc. of Bytecode 2007, Braga, Portugal
12. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. ASE 10(2), 203–232 (2003)
13. Yi, X., Wang, J., Yang, X.: Stateful Dynamic Partial-Order Reduction. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 149–167. Springer, Heidelberg (2006)
14. The Java Grande Forum Benchmark Suite, `http://www.epcc.ed.ac.uk/research/activities/java-grande/`
15. Java PathFinder, `http://javapathfinder.sourceforge.net/`
16. The Mono Project, `http://www.mono-project.com/`
17. .NET Languages, `http://www.dotnetlanguages.net/`
18. Moonwalker, `http://www.cs.utwente.nl/~ruys/moonwalker/`