Zing: A Model Checker for Concurrent Software

Tony Andrews[1], Shaz Qadeer[1], Sriram K. Rajamani[1],
Jakob Rehof[1], and Yichen Xie[2]

[1] Microsoft Research
http://www.research.microsoft.com/zing/
[2] Stanford University

1 Introduction

The ZING project is an effort to build a flexible and scalable model checking infrastructure for concurrent software. The project is divided into four components: (1) a modeling language for expressing concurrent models of software systems, (2) a compiler for translating a ZING model into an executable representation of its transition relation, (3) a model checker for exploring the state space of the ZING model, and (4) model generators that automatically extract ZING models from programs written in common programming languages.

The goal is to preserve as much of the control-structure of the source program as possible. ZING's model checker exploits the structure of the source program, which is preserved in the model, to optimize systematic state space exploration. We believe that such an infrastructure is useful for finding bugs in software at various levels: high-level protocol descriptions, work-flow specifications, web services, device drivers, and protocols in the core of the operating system.

2 Architecture

We believe that the following features capture the essence of modern concurrent object oriented languages, from the point of building sound abstractions for model checking: (1) procedure calls with a call-stack, (2) objects with dynamic allocation, and (3) processes with dynamic creation, using both shared memory and message passing for communication. We designed ZING's modeling language to have exactly these features. ZING supports a basic asynchronous interleaving model of concurrency with both shared memory and message passing. In addition to sequential flow, branching and iteration, ZING supports function calls and exception handling. New threads are created via asynchronous function calls. An asynchronous call returns to the caller immediately, and the callee runs as a fresh process in parallel with the caller. Primitive and reference types, and an object model similar to C# or Java is supported, although inheritance is not supported. ZING also provides features to support abstraction and efficient state exploration. Any sequence of statements (with some restrictions) can be bracketed as atomic. This is essentially a directive to the model checker to not consider interleavings with other threads while any given thread executes an atomic sequence. Sets are supported, to represent collections where the ordering of objects is not important

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 484–487, 2004.

(thus reducing the number of potentially distinct states ZING needs to explore). A choose construct is provided that can be used to non-deterministically pick an element out of a finite set of integers, enumeration values, array members or set elements. An example ZING model that we extracted from a device driver, and details of an error trace that the tool found in the model can be found in our technical report [1].

Building a scalable model checker for the ZING modeling language is a huge challenge since the states of a ZING model have complicated features such as processes, heap and stack. We designed a lower-level model called ZING object model (or ZOM), and built a ZING compiler to convert a ZING model to ZOM. The compiler provides a clear separation between the expressive semantics of the modeling language, and a simple view of ZOM as labeled transition systems. This separation has allowed us to decouple the design of efficient model checking algorithms from the complexity of supporting rich constructs in the modeling language.

3 Model Checker

The ZING model checker executes the ZOM to explore the state space of the corresponding ZING model. Starting from the initial state, the model checker systematically explores reachable states in a depth-first manner. The biggest technical challenge, as with any model checker, is scalability. We have implemented several techniques that reduce the time and space required for state exploration.

Efficient State Representation. We observe that most state transitions modify only a small portion of the ZING state. By only recording the difference between transitions, we greatly cut down the space and time required to maintain the depth-first search stack. To further cut down on the space requirements, the model checker stores only a fingerprint of an explored state in its hash table. We use Rabin's finger-printing algorithm [3] to compute fingerprints efficiently.

Symmetry Reduction. A ZING state comprises the thread stacks, the global variables, and a heap of dynamically allocated objects. Two states are equivalent if the contents of the thread stacks and global variables are *identical* and the heaps are *isomorphic*. When the model checker discovers a new state, it first constructs a canonical representation of the state by traversing the heap in a deterministic order. It then stores a fingerprint of this canonical representation in the hash table.

Partial-Order Reduction. We have implemented a state-reduction algorithm that has the potential to reduce the number of explored states exponentially. This algorithm is based on Lipton's theory of reduction [12]. Our algorithm is based on the insight that in well-synchronized programs, any computation of a thread can be viewed as a sequence of transactions, each of which appears to execute atomically to other threads. During state exploration, it is sufficient to schedule threads only at transaction boundaries. If programmers follow the discipline of protecting each shared variable with a lock, then these transactions can be inferred automatically [6]. These inferred transactions reduce the number

of interleavings to be explored, and thereby greatly alleviate the problem of state explosion.

Summarization. The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. This is not the case for concurrent programs. ZING has an implementation of a novel model checking algorithm for concurrent programs that uses procedure summarization as an essential component [13]. Our method for procedure summarization is based on the insight about transactions mentioned earlier. We summarize within each transaction; the summary of a procedure comprises the summaries of all transactions within the procedure. The procedure summaries computed by our algorithm allow reuse of analysis results across different call sites in a concurrent program, a benefit that has hitherto been available only to sequential programs.

Compositional Reasoning. Stuck-freedom is an important property of distributed message-passing applications [14, 7]. This property formalizes the requirement that a process in a communicating system should not wait indefinitely for a message that is never sent, or send a message that is never received. To enable compositional verification of stuck-freedom, we have defined a conformance relation $\leq$ on processes with the following substitutability property: If $I \leq S$ and P is any environment such that the parallel composition $P \mid S$ is stuck-free, then $P \mid I$ is stuck-free as well. Substitutability enables a component's specification to be used instead of the component in invocation contexts, and hence enables model checking to scale. We have implemented a *conformance checker* on top of ZOM to verify the relation $I \leq S$, where I and S are ZING models.

4 Related Work

Model checking is an active research area [4]. The SPIN project [9] pioneered explicit-state model checking of concurrent processes. The SPIN checker analyzes protocol-descriptions written in the PROMELA language. Though PROMELA supports dynamic process creation, it is difficult to encode concurrent software in PROMELA due to absence of procedure calls and objects. Efforts have been made to "abstract" C code into PROMELA [10] to successfully find several bugs in real-life telephone switching systems, though no guarantees were given as to whether the generated PROMELA model is a sound abstraction of the C code. Over the past few years, there has been interest in using SPIN-like techniques to model check software written in common programming languages. DSPIN was an effort to extend SPIN with "dynamic" software-like constructs [11]. Model checkers have also been written to check Java programs either directly [18, 17, 15] or by constructing slices or other abstractions [5]. Unlike ZING, none of these approaches exploit program abstractions such as processes and procedure calls to do modular model checking. ZING supports a notion of conformance between two ZING models [7]. This feature of ZING is related to, but distinct from, the refinement checking feature of the FDR model checker [16, 8]. The SLAM project [2] has similar goals to ZING in that it works by extracting sound models from C

programs, and checking the models. SLAM has been very successful in checking control-dominated properties of device drivers written in C. Unlike ZING, it does not handle concurrent programs, and it is unable to prove interesting properties on heap-intensive programs.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. Technical report, Microsoft Research, 2004.
2. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
3. A. Broder. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152, 1993.
4. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
5. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*, pages 177–187. ACM, 2001.
6. C. Flanagan and S. Qadeer. Transactions for software model checking. In *SoftMC 03: Software Model Checking Workshop*, 2003.
7. C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *CAV 04: Computer-Aided Verification*, LNCS. Springer-Verlag, 2000.
8. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
9. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
10. G.J. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
11. R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN 99: SPIN Workshop*, LNCS 1680, pages 261–276. Springer-Verlag, 1999.
12. R. J. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.
13. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245–255. ACM, 2004.
14. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 166–179. Springer-Verlag, 2002.
15. Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
17. S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.
18. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICASE 00: Automated Software Engineering*, pages 3–12, 2000.