Basset: A Tool for Systematic Testing of Actor Programs

Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL, 61801, USA
{slauter2, rkumar8, marinov, agha}@illinois.edu

ABSTRACT

This paper presents Basset, a tool for systematic testing of JVM-based actor programs. The actor programming model offers a promising approach for developing reliable concurrent and distributed systems. Since the actor model is based on message passing and disallows shared state, it avoids some of the problems inherent in shared-memory programming, e.g., low-level dataraces involving access to shared data. However, actor programs can still have bugs that result from incorrect orders of messages among actors or processing of messages by individual actors. To systematically test an actor program, it is necessary to explore different message delivery schedules that might occur during execution. Basset facilitates such exploration and provides a generic platform that can support actor systems that compile to Java bytecode. Our current implementation of Basset supports testing of programs developed using the ActorFoundry library and the Scala programming language.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Verification

Keywords: Actors, JPF, Scala, State-space exploration

1. INTRODUCTION

The growing use of multicore and networked computing systems is increasing the importance of developing reliable concurrent and distributed code. Unfortunately, developing and testing such code is very hard, especially using shared-memory models of programming. The actor programming model provides an alternative where multiple autonomous agents communicate by exchanging messages. Each agent, called an *actor*, has an independent thread of control and its own local state [1].

A key challenge in testing actor programs is their inherent nondeterminism. Although actors do not communicate via shared memory, concurrency bugs may still occur as a result of an incorrect order of message arrivals at different actors. To test and model check actor programs, it is thus necessary to systematically *explore* different message schedules.

We developed Basset [3], a tool for systematic testing of actor programs that execute on a Java Virtual Machine (JVM). To leverage work in model checking, we built Basset as an extension for Java PathFinder (JPF), a popular explicit-state model checker for Java bytecodes [5]. Prior to our work, JPF did not have any direct support for actors. One goal of our work was to build a generic platform that can support several actor systems. There are over 20 actor-based languages and actor libraries for existing languages [3]. Basset is a generic tool and framework for testing actor systems that compile to Java bytecode. Providing support for a specific language requires only a small amount of new code to be written. We extended Basset for the Scala programming language and the ActorFoundry library for Java.

A question that arises is why build a new tool and not use JPF to directly check programs written against actor libraries such as Scala and ActorFoundry. These libraries include a complex, multi-threaded runtime system for execution of actor programs. While these runtime systems enable efficient *execution* of actor programs, because of the complexity and scheduling choices in the runtime system, they make *exploration* of the programs unfeasible or inefficient. JPF can execute the Scala library, but the resulting state space is huge: for example, exploration of the state space of a simple Scala `helloworld` application did not complete in one hour! Even after we simplified parts of the Scala library, JPF still took over 7 minutes to check `helloworld` [3].

Our design goal for Basset is *efficient exploration of the actor application itself* and not the exploration of the code inside actor libraries. Specifically, Basset provides a simplified abstraction of a typical actor runtime system that replaces much of the code in a given library. The modified actor library presents the same interface to the actor application but enables a much faster exploration. The result is a highly efficient system to test actor code for potential bugs due to message schedules. For example, Basset takes less than one second to check `helloworld`.

2. BASSET OVERVIEW

Basset is implemented as an extension to JPF. Exploration of actor programs using Basset can be performed *without* any modifications to the programs. All that is required to test an actor program is a simple test driver that is rarely more than a few lines in length. These drivers allow the developer to specify initial starting configurations that create actors and seed initial messages without exploring related message schedules. Since actors are often used to develop "open" systems, this allows the systematic exploration to be started after the system is setup and running. This capability can greatly reduce testing time.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
ACM 978-1-60558-791-2/10/11.

Basset provides several capabilities to support efficient state-space exploration: *actor state management* (keeping track of created and destroyed actors, as well as comparing states when stateful search is used), *actor execution* (managing actor threads), *message management* (scheduling and delivering messages, as well as tracking message causality when partial-order reduction is used), and *error checking*.

2.1. Actor state management: Each actor program creates several actors that compute as well as exchange messages. Basset keeps track of all actors created and destroyed during an execution, and maintains the internal state along with the status of each actor, including the following: processing a message (*busy*), waiting for a message (*idle*), or waiting on a reply of a synchronous message (*blocked*).

2.2. Actor execution: A critical aspect of any actor system is how to execute the actor code that processes each message. Semantically, each actor has its own thread of control. To support this model and to provide for efficient *exploration* (not *execution*) of actor programs, Basset uses a separate `ActorThread` object/thread for each actor instead of a thread pool. Exploring all possible fine-grained interleavings of instructions from these threads would be very costly and is not necessary because actors have no shared state. Hence, Basset uses the *macro-step semantic* [1] for actor execution: after Basset delivers a message to an actor, the actor processes the message by executing a sequence of instructions *atomically* until the next message receive point.

2.3. Message scheduling: At the heart of Basset are the message management and scheduling functions that drive the exploration of a subject program. Basset maintains a *message cloud* containing all messages that have been sent but not yet delivered to actors. Whenever a new cloud configuration contains more than one deliverable message, Basset nondeterministically chooses to deliver one of these messages to its corresponding receiver actor. In subsequent executions, Basset systematically explores the state space of the program by choosing different possible delivery schedules for the messages in the cloud. The tool facilitates reduction of the number of message delivery schedules that need to be explored by using either state comparison or dynamic partial-order reduction (DPOR).

State comparison: Basset can perform a stateful exploration, i.e., it stores visited states and checks whether a state has been seen previously by comparing it against a set of states [3]. A challenge for object-oriented programs (whose state include heaps with connected objects) is that states need to be compared for *isomorphism*. Using JPF's default state comparison, two states are equivalent when their heaps have the same shape among connected objects and the same primitive values, even if they have different object identities. Basset additionally provides a custom state comparison specialized for the actor domain. For example, when comparing actors, the top-level state items—actors and the message cloud—are sets. Standard comparison simply compares them at their concrete implementation level (say, as arrays or lists) and thus could find two sets with the same elements to be different due to the order of their elements. Basset's custom actor state comparison uses a sorting heuristic to help identify equivalent sets [3].

Partial-order reduction: As an alternative to stateful exploration, Basset includes support for dynamic partial-order reduction (DPOR) for actor programs. To avoid executing message schedules that are *equivalent*, DPOR algorithms dynamically identify situations where only a subset of the messages available for delivery need be considered. We have implemented multiple DPOR algorithms in Basset and have extended the tool's actor and message representations to include vector clocks that track causality among message (send and receive) events. Furthermore, since the effectiveness of DPOR is highly sensitive to the order in which messages (and their receiving actors) are ordered for delivery [4], Basset allows users to select from among several different message ordering heuristics when using DPOR.

2.4. Error checking: Basset provides several checks that can be applied during execution of actor programs. Basset can check for *state assertions* (expressed using arbitrary Java expressions) and for *undeliverable messages* at the end of an execution path (due to actors being terminated or blocked). Basset can also detect *deadlocks*. An obvious deadlock occurs when several actors are blocked, each waiting for another (in a cycle) to return from a synchronous call. Another type of deadlock can occur when the execution reaches a final program state where no alive actor can make progress. Since actor programs are open systems, such a final state is not necessarily a deadlock; it may be that actors are waiting for a new message from the environment. To check for deadlocks in such cases, Basset allows the user to "close" the system by providing a model of the environment (as another actor) [3].

3. CONCLUSIONS

We presented Basset, a tool for systematic testing of JVM-based actor programs. Basset's current implementation supports testing of programs developed using the ActorFoundry library and the Scala programming language. Basset is available at `http://mir.cs.illinois.edu/basset/`. The availability of Basset's generic exploration allowed us to quickly compare existing DPOR techniques for actor programs [4] and to experiment with mutation testing for actors [2]. We expect that making the tool publicly available will also help other researchers to test actors programs as well as to improve testing and model checking of actor programs.

Acknowledgments. We would like to thank Mirco Dotta, Milos Gligoric, Vilas Jagannath, and Samira Tasharofi for their help on this project. This material is based upon work partially supported by the NSF under Grant Nos. CCF-0916893, CNS-0851957, CCF-0746856, and CNS-0509321.

4. REFERENCES

[1] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[2] V. Jagannath, M. Gligoric, S. Lauterburg, D. Marinov, and G. Agha. Mutation operators for actor systems. In *Mutation*, 2010.

[3] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of Java-based actor programs. In *ASE*, 2009.

[4] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *FASE*, 2010.

[5] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.