A Framework for State-Space Exploration of Java-based Actor Programs

Steven Lauterburg
Dept. of Computer Science
University of Illinois
Urbana IL, 61801, USA
slauter2@illinois.edu

Mirco Dotta
I&C School
EPFL
Lausanne, Switzerland
mirco.dotta@epfl.ch

Darko Marinov and Gul Agha
Dept. of Computer Science
University of Illinois
Urbana IL, 61801, USA
{*marinov, agha*}*@illinois.edu*

***Abstract*—The *actor* programming model offers a promising model for developing reliable parallel and distributed code. Actors provide flexibility and scalability: local execution may be interleaved, and distributed nodes may operate asynchronously. The resulting nondeterminism is captured by nondeterministic processing of messages. To automate testing, researchers have developed several tools tailored to specific actor systems. As actor languages and libraries continue to evolve, such tools have to be reimplemented. Because many actor systems are compiled to Java bytecode, we have developed Basset, a general framework for testing actor systems compiled to Java bytecode. We illustrate Basset by instantiating it for the *Scala* programming language and for the *ActorFoundry* library for Java. Our implementation builds on Java PathFinder, a widely used model checker for Java. Experiments show that Basset can effectively explore executions of actor programs; e.g., it discovered a previously unknown bug in a Scala application.**

I. Introduction

The growing use of multicore and networked computing systems is increasing the importance of developing reliable parallel and distributed code. Unfortunately, developing and testing such code is very hard, especially using shared-memory models of programming which often results in bugs such as atomicity violations, dataraces, or deadlocks. An alternative for writing parallel and distributed code is message passing, where multiple autonomous agents communicate by exchanging messages. Actors provide a programming model for message-passing which enforces object-style data encapsulation.

In the *actor* programming model [1], [2], each computation entity (called an actor) has its own thread of control, manages its own internal state, and communicates with other actors by explicitly sending messages. An actor program may have different executions (even for the same input) based on the interleaving of messages exchanged between the actors, in much the same way as a multithreaded program may have different executions based on the interleaving of accesses to shared memory. Some actor-oriented programming systems include Axum [3], Charm++ [4], Erlang [5], E [6], Newspeak [7], Ptolemy II [8], Revactor [9], ThAL [10], Singularity [11], and the Asynchronous Agents Framework used in Microsoft Visual Studio 2010 image processing software [12].

While actor programming avoids some of the bugs inherent in shared-memory programming, e.g., low-level dataraces involving access to shared data, actor programs can still have bugs: for example, there may be an unsafe interleaving of messages to an actor, or incorrect sequential code within an actor. In this paper, we focus on the problem of automated testing of actor programs which have been compiled to Java bytecode. Such programs may be written in languages and libraries that include ActorFoundry [13], [14], Jetlang [15], Jsasb [16], Kilim [17], SALSA [18], and Scala [19]. Specifically, we develop *Basset*, a general framework which can be easily customized for different actor languages and libraries. Basset addresses the difficulty that actor languages and libraries, as well as testing and model checking techniques, continue to evolve as researchers and developers explore new variants.

Systematic testing and model checking can improve program reliability by automatically finding potential bugs. A key element of these approaches is *state-space exploration* (e.g., [20]) which searches through the possible executions of a program. Previous research on systematic checking of actor or distributed systems has focused on (stateless) checking of programs in one specific system [21]–[27]. Basset is also based on state-space exploration. However, Basset provides a *general exploration framework* that can support different actor systems with only a thin adaptation layer required for each system. The interface between the Basset core and adaption layers is designed both to allow *direct exploration of unmodified application code* and to ensure that optimizations for exploration (such as partial order reduction or actor state comparison, as explained later) can be made available to multiple actor systems by changing only the Basset core. Specifically, we consider actor programs written in the Scala language [19] (which compiles to Java bytecode) and those written using the ActorFoundry [13] library for Java (which also compiles to Java bytecode).

In order to leverage work in model checking, we built Basset on top of Java PathFinder (JPF), a popular explicit-state model checker for Java bytecodes [28], [29]. JPF was developed at NASA for checking programs written directly in the Java language. It has been used in numerous research

projects (e.g., [29]). JPF provides a specialized Java Virtual Machine that supports state backtracking and control over nondeterministic choices such as thread scheduling. Prior to our work, JPF did not have any direct support for actors, i.e., for high-level choices such as message scheduling. Note that although we use JPF, our techniques could use other explicit-state model checkers such as Bandera, BogorVM, CMC, JCAT, JNuke, SpecExplorer, or Zing.

A question that arises is why build a new framework and not use JPF to directly check programs written against actor libraries such as Scala and ActorFoundry. Those 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 impossible or inefficient. For instance, JPF cannot even execute all the ActorFoundry library, as the ActorFoundry uses Java networking libraries for exchanging messages between actors [24], [25]. JPF can execute the Scala library, but the resulting state space is huge: for example, exploration of the states of a simple Scala `helloworld` application did not complete in an hour! Even after we simplified parts of the Scala library, JPF took over 7 minutes to check `helloworld`.

Our design goal for Basset is *efficient exploration of actor application code itself* and not exploration of the actor libraries. Therefore, Basset does not check the actual library code but focuses instead on exposing potential bugs in the application code due to message scheduling, which is the source of non-determinism in actor-based programs. Specifically, the adaption layer in Basset replaces the implementation of an actor library with much simpler code that still provides the same interface to the actor application but enables a much faster exploration. The result is a highly efficient system to test actor code: Basset takes less than one second to check `helloworld`.

This paper makes the following contributions:

- **Framework:** We propose the concept of a general framework for exploration of actor programs.
- **Implementation:** We implement our general framework in a tool called Basset which uses the Java PathFinder model checker. We show how optimizations can be added to Basset by incorporating two known optimizations: dynamic partial-order reduction [20], [21] and state comparison/hashing [20], [30].
- **Instantiations:** We illustrate the flexibility of Basset by instantiating it for actor programs written in the ActorFoundry library, or in the Scala programming language, a popular new language used in systems such as Twitter [31]. These instantiations provide the first state exploration engines for the two actor systems, which are based on very different design decisions.
- **Experiments:** We present several experiments with Basset on nine actor programs. The results show that Basset can effectively explore executions of these programs. The results also illustrate that the existing optimizations for dynamic partial-order reduction and state comparison provide complementary benefits in exploration of actor programs. Our use of Basset discovered a previously unknown bug in the sample code provided at ScalaWiki, a popular web site for Scala [32]. We reported this bug, and the developer confirmed it; the current, updated version of the example on ScalaWiki is without the bug.

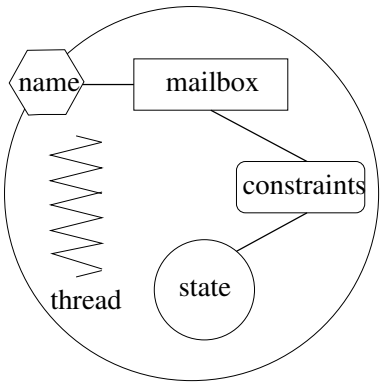

Figure 1. Schematic representation of an actor

II. Background

An actor is an autonomous concurrent object which interacts with other actors by sending messages. By default, these messages are *asynchronous*; other forms of communication, such as remote-procedure-call style synchronous messages, are defined in terms of asynchronous messages [1]. Each actor has a unique *actor name* (its virtual address) and a mailbox, as depicted in Figure 1. If an actor is busy, messages sent to it are queued in its mailbox. When an actor is done with processing a message, it checks its mail queue for another message. In response to processing a message, an actor may do one or more of three actions:

- **Send messages:** Messages may be to other actors or to itself, provided the sender knows the name of the recipient.
- **Create new actors:** Newly created actors have their own unique names and an associated mail queue. Upon creation, only the creator knows the name of the new actor, but names may be communicated in messages.
- **Update local state:** In some actor systems, the update is made by immediate assignment, in others the new local state is used to process the next message.

Each actor operates asynchronously. Consequently, if two actors send messages concurrently the order of arrival of messages is indeterminate. Moreover, no assumption is made about the routing of messages; therefore, two messages sent from the same actor to the same recipient may arrive in

any order.[1] Not requiring the order of messages between two actors to be maintained by default allows greater flexibility in the implementation: for example, a client may send requests to a receptionist which then forwards it to a stateless server. If the order of messages is not required to be preserved, the behavior of the system would be the same whether the receptionist forwards the requests to a single actor, or to a one of a collection of servers. However, message order can always be explicitly constrained; this requires imposing additional synchronization protocols. In some actor languages, message order between two actors can be easily expressed in terms of constraints (see below).

While actor systems do not guarantee in-order message delivery, they do guarantee that the messages are eventually delivered. In a real distributed implementation, messages may be lost, just as processors may crash or buffers may overflow in an implementation of a sequential program. Using an abstraction that guarantees message delivery enables us to reason about the liveness properties of actor systems. Lossy messages can always be modeled by explicitly representing channels as actors which nondeterministically lose messages. This is not hard to do. Although the behavior of an actor in response to a message is deterministic, there are a number of ways of essentially causing the behavior to be governed by a nondeterministic coin flip. The simplest is an actor which sends itself a message, causing the next message to be ignored rather than forwarded. Since this message would be shuffled with other messages, whether some message is lost or not would be nondeterministic.

Because of the asynchrony inherent in distributed systems, it is generally not feasible for the sender to know what the state of a recipient will be when it receives a message. For example, a free printer may ask a spooler for a print job, but the spooler may be empty. In many actor languages, the developer may specify synchronization *constraints* which are used to postpone requests until the recipient is in a state where it can process the request [33]. Actor runtimes implement these constraints by reordering messages. Moreover, actor languages typically provide higher level language constructs for "synchronous" or request-reply communication, where a value is returned by actor receiving a message; the sending actor waits until this value is received before carrying out further computation. These constructs can be translated into basic actor constructs [34].

Actors enable programmers to think in terms of objects as agents, a natural view of concurrency. Because an actor processes only one message at a time, access to an actor's state is *serialized*. Thus, actors avoid low level dataraces to variables in its state, and avoid the potential for deadlocks. In essence, actors raise the level of abstraction, allowing larger computation steps to be viewed as a logical unit.

[1]In some variants of actors, message order between pairs of actors is maintained.

Nevertheless, bugs may still occur in actor code: the interleaving of messages to an actor whose ordering should have been constrained may result in an incorrect behavior. The use of synchronous communication or synchronization constraints can result in a deadlock. Moreover, sequential code within an actor can have bugs. Some sources of bugs in actor programs are the result of the deficiencies in the current generation of actor languages and libraries. For example, a critical issue for actor programs is the cost of sending messages. In many actor languages and libraries, the cost of message passing is minimized by *transferring* ownership rather than by copying data. Unfortunately, these actor languages and libraries require the programmer to ensure that passing messages by transferring data ownership is correct: i.e., that the sender does not subsequently attempt to access the data. This requirement can be a source of bugs.

Many systems support actors using languages or libraries that generate executable Java bytecode [13], [15]–[19], [35]. This involves implementing Java classes for actor names, mail queues, threads, and state. Moreover, constraints which filter messages based on the state of an actor may be specified. The commonality of these structures motivates us to develop a unified, generic framework in which actor state exploration can be performed and optimized.

III. Example

To illustrate some key actor concepts and the Basset approach to exploration of actor programs, we use a simple example. Our example is a simplified version of a sample actor program available on ScalaWiki [32], a popular web site that provides several widely used resources for Scala, including code samples contributed by some developers of the Scala programming language. Using Basset, we discovered a bug in this sample code. Here, we present a version of this code implemented using Java and the ActorFoundry library, assuming that the readers are more familiar with Java than with Scala.

Figure 2 shows the code for our example. The main driver code first creates two actors—a server and a client—using the `createActor` method from ActorFoundry. This method takes the class of the actor (ActorFoundry heavily uses Java reflection) and, optionally, arguments for the class' constructor. The method creates an actor and returns an `ActorName` object that represents a handle to the actor. (Because ActorFoundry supports distributed applications and mobility of actor code, `createActor` does not return an actual reference to the created actor.) The main driver code then sends a message to the client to initiate the exploration. The `send` method implements an *asynchronous* send: the message `start` is sent to the client, and the `main` method continues execution (in this case it terminates right away, but the entire program keeps working as there are alive actors).

Our example `Server` actor simply stores and retrieves an integer value. (The actual server from the ScalaWiki code

```
class Driver {
  static void main(String[] args) {
    ActorName server = createActor(Server.class);
    ActorName client = createActor(Client.class,
                                   server);
    send(client, "start");
  }
}

class Server extends Actor {
  int value = 0;
  @message void set(int v) {
    value = v;
  }
  @message int get() {
    return value;
  }
  @message void kill() {
    destroy("server has finished processing");
  }
}

class Client extends Actor {
  ActorName server;
  Client(ActorName s) { server = s; }
  @message void start() {
    send(server, "set", 1);
    int v1 = call(server, "get");
    int v2 = call(server, "get");
    // assert (v1 == v2);
    send(server, "kill");
  }
}
```

Figure 2. Example code using ActorFoundry

was keeping track of an inventory of items with specified prices and quantities.) In ActorFoundry, each actor is a subclass of the `Actor` class. Each actor can process a set of messages as denoted with the `@message` annotation on the appropriate methods. In addition to storing and retrieving the value, the `Server` actor can also process a `kill` message, which instructs this actor to terminate itself as it invokes the `destroy` method from ActorFoundry.

Our example `Client` actor communicates with the server. The client first sends to the server an asynchronous message to store the value 1. The client then sends a message to the server to retrieve the value, using the `call` method for *synchronous* remote procedure call. Namely, after the client sends the `get` message, it blocks until it receives a reply from the server and then stores the return value into the appropriate variable. Note that the client retrieves the value twice, and in our example code, compares the return values when the `assert` is uncommented. (The actual code from ScalaWiki was also retrieving the inventory of items twice but performing two different computations on the inventory.) The client finally terminates the server.

The cause of problems in this example is the *order of message deliveries*. Actor systems, including ActorFoundry and Scala, do not guarantee in-order delivery of the messages; in other words, the default communication channels between the actors are *not FIFO*. However, anecdotal experience shows that assuming in-order delivery is a common cause of programming errors in many actor programs. In our example, for instance, the server need not process the message `set` before the first message `get`. Figure 3 shows some possible executions for our example program. Specifically, if the server interleaves processing of `set` between the two `get` messages, it will return inconsistent values for `v1` and `v2`. While this execution is not very likely (e.g., we ran our example code on the standard Scala run-time 1000 times, and it always processed `set` before `get`), it is possible. The program therefore has an *atomicity violation*. (The Scala developers confirmed the bug that we reported for their code from ScalaWiki.)

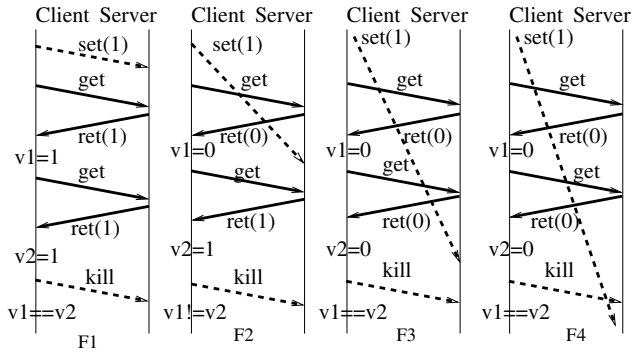

Figure 3. Possible message schedules and final states: correct, incorrect, correct, warning

Given an actor program, Basset can explore (all) different program executions due to different message orderings. Basset starts the execution from the `main` driver and explores non-deterministic choices that arise when several messages can be delivered. Basset provides two well known optimizations that can prune exploration: dynamic partial-order reduction [20], [21] and state comparison [20], [30]. For this example, we discuss how Basset performs a stateful search using Basset's customized state comparison for pruning.

Figure 4 shows the state space that Basset explores for our example code. Each state of an actor program consists of the *state of actors* (in our example, the field `value` in the server and the variables `v1` and `v2` in the client) and a *message cloud* (with all messages that have been sent but not yet delivered and can be thus delivered in any order). In Figure 4, a slash denotes an undefined state variable. Note also the `ret` messages that represent the return values of synchronous calls; these messages are not explicitly visible in the code, but ActorFoundry internally sends them.

In this example, Basset explores 20 states and finds *two potential bugs*. In the state labeled `E` (for "error") in Figure 4, the values of `v1` and `v2` differ, being 0 and 1, respectively. (If the `assert` is uncommented, Basset would report a bug and stop the exploration for that path.) Additionally, in the state labeled `F4` (for "final state 4"), Basset generates a *warning* since the server actor is not alive while there are undelivered messages for that actor. This case occurs when the server processes `kill` before `set`, as shown in Figure 3.

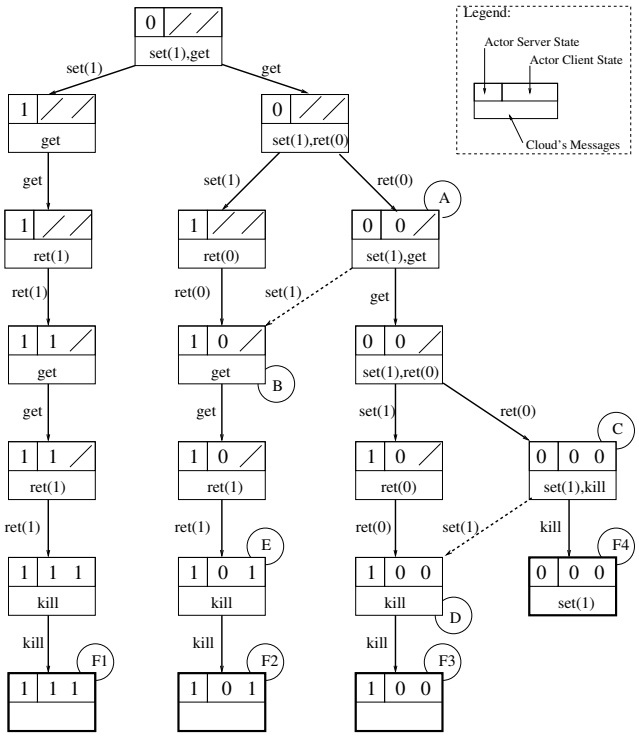

Figure 4. State space for the example program

Note that Basset explores only 20 states for this example due to a stateful search that compares states using a custom comparison for actor programs. This comparison allows Basset to detect that states `B` and `D` can be visited through various executions. Without the comparison, Basset would explore twice the executions both from `B` to `F2` and from `D` to `F3`, resulting in a total of 26 explored states.

As an alternative to stateful exploration, Basset also supports a dynamic partial-order reduction for actor programs. For this example, using the partial-order reduction resulted in a total of 18 explored states when messages were delivered to actors in the order in which the actors were created. We discuss state comparison and partial-order reduction in more detail in Section IV and our experiments with these techniques in Section VII.

IV. Framework

This section provides an overview of Basset's core, which contains the common, library-independent functionality for running actor programs and exploring different message schedules. The goal for Basset is to provide a platform *efficient for state-space exploration* but *not necessarily efficient for straightline execution* of actor programs. This goal affects the design decisions we made for the execution of actors. The three main components of the Basset core handle *actor states* (keeping track of created and destroyed actors, as well as comparing states when the stateful search is used), *actor execution* (managing actor threads), and *message management* (scheduling and delivering messages, as well as tracking message causality when partial-order reduction is used). We first provide more details about these three components, then discuss state space pruning, and finally present Basset's error detection facilities. The next section briefly discusses the API that Basset exposes to library-specific instantiations and presents the two instantiations created for ActorFoundry and Scala.

A. Actor states

Each actor program creates several actors that compute and exchange messages. Various actor libraries provide different specialized behavior for actors. The Basset core therefore does not create the concrete actors by itself but delegates that task to particular instantiations. The core only maintains generic information about actors, keeping track of all actors created and destroyed during an execution, and the status for each actor, which can be one of the following: executing, waiting for a general message, or blocked waiting on a reply for synchronous message (such as the `call` method shown in Figure 2). Basset uses all this information for efficient message scheduling (Section IV-C), during state comparison to prune redundant execution segments (Section IV-D), and to facilitate deadlock detection (Section IV-F).

B. 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. However, efficient implementations of actor libraries [17], [35] typically do not assign one native thread/process per actor and do not create a new thread whenever a new actor is created, since these operations are expensive; instead, they employ thread pools to reuse threads for new actors, migrate actors among threads, and/or use more lightweight parallel constructs, such as the Java Fork/Join Framework [36].

Since Basset aims for efficient *exploration* (not *execution*) of actor programs, Basset uses a separate `ActorThread` object/thread for each actor. Exploring all possible fine-grain interleavings of instructions from these threads would be very costly (see Section VII-A) and is not necessary when actors have no shared state. Hence, Basset uses the *macro-step semantic* [2], [21] for actor execution: after Basset delivers a message to an actor, the actor executes atomically until the next receive point (which either encounters a synchronous call or finishes the processing of the message and waits for a new message). The soundness of macro-step semantics is discussed elsewhere [2]. Similarly as for creating actors, Basset does not itself create actor threads, but delegates that task to particular instantiations, which provide the main control for the processing of one message.

C. Message management

At the heart of Basset are its message management and scheduling functions. Actors communicate by exchanging

messages. Basset again delegates the creation of concrete messages to instantiations, but it maintains a *message cloud* of all messages that were sent but not yet delivered to actors. The main loop in Basset controls the delivery schedule of these messages. Whenever the cloud contains more than one deliverable message, Basset non-deterministically chooses to deliver one of these messages to its receiver actor. Basset then systematically explores all possible program executions that arise from the delivery of the messages in the cloud, using either state comparison (Section IV-D) or dynamic partial-order reduction (Section IV-E) to prune the exploration. Currently the use of state comparison and dynamic partial-order reduction in Basset are independent and mutually exclusive. However, recent work [37], [38] proposes combining the two techniques, and we intend to explore this possibility for Basset.

It is important to point out that not all messages are deliverable at all times. One reason is that an actor can terminate itself (e.g., using `destroy` in ActorFoundry as shown in Figure 2). Another reason is that most actor libraries allow putting *constraints* on the messages that actors can receive. Scala, for instance, expresses these constraints by pattern matching on the messages [19]. If a message sent to an actor does not match any of its patterns, the message cannot be delivered until the behavior of the actor changes so that its new pattern matching accepts the message.

D. State comparison

Basset can perform a stateful exploration, checking whether a new state has been already visited previously, effectively comparing one state against a set of states. This is a standard operation in explicit-state model checking [20]. A challenge for object-oriented programs (whose state include heaps with connected objects) is that states need to be compared for *isomorphism* [39]–[41]. Namely, 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. Typical comparison of states for isomorphism involves linearizing the *entire* states into an integer sequence that normalizes object identities such that isomorphic states have equal sequences [39], [40]. The JPF tool, on top of which Basset is implemented, provides such state comparison.

In addition to JPF's default state comparison, Basset provides a custom comparison that has been specialized for the actor domain. For example, an additional challenge for actors is that the top-level state items—actors and message clouds—are sets. The usual linearization does not specially treat sets but simply compares them at the concrete level at which they are implemented (say, as arrays or lists) and thus could find two sets with the same elements to be different because of the order of their elements. To compare states of actor programs, we provided in Basset a known heuristic that first sorts set elements and then linearizes them as usual [42]. This heuristic offers more opportunity to identify equivalent sets and states but does not guarantee all equivalent states will be found: the sorting is done only for the elements without following any pointers from these elements, and thus does not handle arbitrary graph isomorphism [30].

E. Partial-order reduction

As an alternative to performing stateful exploration with path pruning, we adapted for Basset a dynamic partial-order reduction for actor programs proposed in dCUTE [21]. This reduction uses the happens-before relation [43] to avoid executing message schedules that can be identified as equivalent. The reduction identifies situations where only a subset of the messages available for delivery need be considered when non-deterministically choosing which message to deliver next. To facilitate the partial-order reduction, we extend the actor and message representations to optionally include vector clocks that track causality among message send and receive events. Since the benefit of this partial-order reduction is sensitive to the order in which messages (and their receiving actors) are ordered for delivery, Basset provides two different orderings (discussed in the experiments in Section VII).

F. Error checking

Basset provides several generic checks for executions of actor programs. As illustrated in Section III, 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 actors 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 [1], [2], [21]. 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). Two most common models are the one where the environment can send no new message (so any alive actors are deadlocked) and the one where the environment can send any new message (except a return from a synchronous call, so alive actors are never deadlocked unless waiting in cycle on synchronous calls). Basset provides these two models as defaults to choose from.

V. Instantiations

We next describe our instantiations of Basset for ActorFoundry and Scala. In each case, we started from the existing actor library and modified/simplified it with the following goals: (1) preserve the API of the library toward actor programs (such that we can check unmodified actor

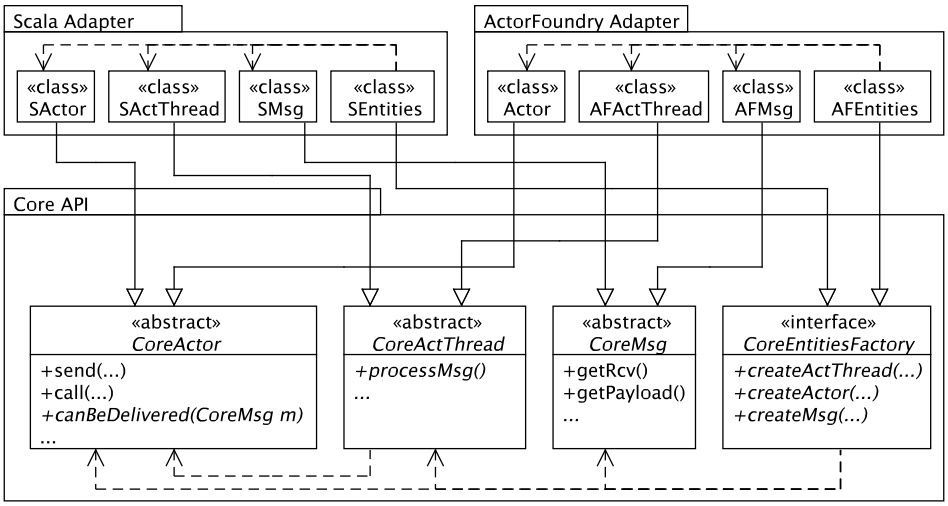

Figure 5. UML class diagram for the adapter layers

applications); (2) simplify the library, considering that we want fast exploration (for relatively small program states) and not necessarily fast executions (for relatively large program states, e.g., scaling up to thousands of actors); and (3) connect the library into the Basset framework to enable exploration. Our modifications removed some parts of libraries (e.g., distribution of actors across various computers, since our goal is to *check the actor applications not libraries* themselves) and replaced other parts (e.g., the `Actor` class in ActorFoundry, as described below). While these modifications of libraries may appear time consuming from their description, they were actually much easier to perform than building the general Basset framework.

Figure 5 shows the UML class diagram for connecting the two instantiations into the Basset core. The key entities that Basset manipulates are actors, actor threads, and messages. Basset does not directly create the objects for all these related entities but instead uses the Abstract Factory design pattern [44]. The core itself refers to the `CoreEntitiesFactory` interface, and each instantiation provides a concrete class that can create appropriate entities and also provides concrete classes for these entities. The concrete instantiation classes implement the abstract methods from the core classes, i.e., `canBeDelivered` and `processMsg`.

A. ActorFoundry

ActorFoundry is a Java library, and it was fairly straightforward to connect it to Basset. While adding the four classes shown in Figure 5, we removed certain library operations (most notably actor migration among computers) and mapped internal exceptions from Basset into ActorFoundry exceptions (e.g., sending a message for a nonexistent method). ActorFoundry does not expose messages or actor threads in the API to actor applications, so our adaptation layer could name those entities arbitrarily. However, ActorFoundry does expose actors in the API (more precisely in the internal API used within the library), so we had to preserve that the actor class be named `Actor`. Since this class had no superclasses in the original library, we could easily subclass it from the Basset's actor class and then provide both functionality required by the ActorFoundry library and by the Basset framework.

B. Scala

The changes for Scala were somewhat more involved, due to its compilation model (from the Scala source files to Java class files) and the rich API that the Scala actor library exposes to Scala applications. A key issue is that our adaptation layer for Scala needs to subclass the actor class from Basset and also needs to provide the actual interface of the Scala actor. To solve this issue, we used the Adapter design pattern [44] to translate the calls that a Scala application makes on an actor object into the appropriate calls to the Basset actor. Moreover, it was not possible to add a field to the existing Scala `Actor` trait (a trait can be thought of as a Java abstract class) to point to its corresponding Basset actor, because of the way that Scala compiles trait's fields [19]. Therefore, we maintain this correspondence as a map outside of the `Actor` trait. Each actor operation translates calls from the Scala world into the Basset world using this map.

VI. Implementation

We implemented the Basset framework on top of Java PathFinder (JPF), an extensible, explicit-state model checker for Java [28], [29]. We first describe the relevant parts of JPF and then present our changes.

JPF is written in Java and provides a specialized Java Virtual Machine (JVM) that supports state backtracking and

control over non-deterministic choices. The default non-deterministic choices in JPF are thread scheduling and explicit choices made in the code with the JPF library call `Verify.getInt` (analogous to `VS_toss` in VeriSoft [45]). The specialized JVM in JPF is an interpreter that executes the bytecodes of the application under exploration. JPF itself runs on top of a host, native JVM. JPF provides an interface, called Model Java Interface (*MJI*), for communication between the specialized JVM and the host JVM (analogous to the Java Native Interface for communication between Java and C in JVMs written in C).

The key change we made to JPF is in the control of thread scheduling. Recall that the Basset architecture puts each actor in its own thread. The actor code itself is in Java (more precisely, compiled to Java bytecode). Additionally, Basset has a main controller thread that decides which actor(s) should be executed at which point. We wrote the main controller itself in Java so that it runs on JPF's JVM (and not on the host JVM). All these threads are actual JPF/Java threads. Based on the macro-step semantics (Section IV-B), Basset enables only one of these threads at a time.

Note that the thread switch could *not* be implemented in pure Java executing on the JPF's JVM. To ensure that execution properly switched back and forth between actor threads and Basset's main controller thread, we needed greater control over thread switches than the Java language supports. Namely, the main loop in Basset proceeds as follows: the main controller chooses one actor to execute (more specifically, one message to deliver to an actor that then starts processing the message), and when that actor *blocks* (waiting to receive a message), the main controller should execute to schedule another actor. However, once the actor blocks, it cannot explicitly return control to the main controller at the Java level.

So, we made two modifications to JPF. First, we extended JPF, using the MJI interface to implement thread switches, effectively replacing the thread scheduler in JPF. Second, we optimized the core of Basset to eliminate the creation of JPF backtracking points when switching back and forth between the actor threads and the main controller thread. Since the JPF main loop is structured around executing only one thread in a transition, we modified the JPF core code to enable longer transitions. This allows us to consider the selection and delivery of a message by Basset's controller thread and the processing of that message by an application's actor thread as a single transition. To the best of our knowledge, this is the first JPF extension that considered state transitions with bytecodes executed by more than one thread.

VII. Experiments

We present several experiments using the Basset framework. We first briefly compare the state-space exploration of a trivial `helloworld` Scala application using Basset versus an exploration using the standard Scala library executing on JPF. We then describe the subject programs used to more quantitatively evaluate Basset for ActorFoundry and Scala. We finally present experimental results comparing the different state-space reduction options available in Basset. All experiments were performed using Sun's JVM 1.6.0_13-b03 on a 3.4GHz Pentium 4 workstation running Red Hat Enterprise Linux 5. We set the time limit to one hour, and we show partial results in cases where the exploration did not finish in an hour.

A. Basset versus original library

To illustrate the increased efficiency obtained by exploring an actor program using Basset instead of directly exploring an actor program and its library running on JPF, we ran an experiment to compare performance of these two options. Recall that our goal is to check actor applications, not actor libraries. However, the libraries already exist in Java bytecode, so it is natural to ask whether we can run them on JPF. Specifically, instead of developing Basset, could we have taken a Scala application with the existing Scala library and run it directly on JPF? Our experiments show that such direct exploration is possible but *extremely slow*, even after several simplifications to the library. The reason is that the Scala library is a complex, multi-threaded piece of code, and exploring it on JPF results in exploring a very large number of thread interleavings.

More concretely, we wrote in Scala a simple `helloworld` application that creates one actor and prints `Hello World`. Running this code with the unmodified Scala library on JPF did not finish in an hour! We then simplified the library by: (1) removing a timer thread, (2) disabling actor garbage collection, and (3) reducing the size of the thread pool that the library uses to execute actors. JPF still took *over 7 minutes* to explore this application. In contrast, the Scala instantiation of Basset takes *a fraction of a second* for this application. The key reasons for this speedup are that Basset uses a simplified framework with the macro-step semantics [2], [21] for exploration and does not interleave executions from different threads, and it does not explore the complex code of the Scala library on which Scala applications typically run.

Though we did not perform a similar experiment for ActorFoundry, we would expect similar results in that JPF would not be able to effectively explore the original library. In fact, the library code for ActorFoundry contains network calls that the publicly available JPF does not even support as they depend on native code in standard Java libraries. Projects by Artho and Garoche [24] and Barlas and Bultan [25] provide solutions for modeling some of these calls, but the original ActorFoundry library would still have a prohibitively large number of thread interleavings. Using the simplified library in our Basset framework, thread interleavings are manageable for exploring interesting programs.

Table I
COMPARING DIFFERENT STATE-SPACE REDUCTION TECHNIQUES IN BASSET.

		ActorFoundry					Scala				
Experiment		Resources		Statistics			Resources		Statistics		
Subject	State Reduction	Time (sec)	Memory (MB)	# of States	# of Msgs Delivered	# of Execs	Time (sec)	Memory (MB)	# of States	# of Msgs Delivered	# of Execs
fibonacci	None	16	89	769	768	168	28	110	768	767	168
	JPF Comp.	9	65	188	299	9	11	82	80	144	4
	Actor Comp.	8	71	105	184	6	9	85	43	75	2
	POR-high	15	106	495	494	102	13	100	147	146	32
	POR-low	10	90	289	288	64	33	160	768	767	168
leader	None	24	119	1467	1466	374	44	122	1467	1466	374
	JPF Comp.	17	95	671	864	106	19	93	255	341	42
	Actor Comp.	15	94	465	651	73	16	94	187	237	34
	POR-high	21	147	911	910	188	30	165	667	666	138
	POR-low	14	113	493	492	101	28	119	612	611	126
mergesort	None	1888	474	113067	113066	33264	2316	419	113066	113065	33264
	JPF Comp.	559	359	21450	32770	3080	73	131	1054	2729	20
	Actor Comp.	197	205	6081	10909	727	25	117	270	719	4
	POR-high	6	72	40	39	8	9	82	39	38	8
	POR-low	11	86	212	211	54	17	105	211	210	54
pi	None	2300	418	168646	168645	60480	3523	463	168645	168644	60480
	JPF Comp.	115	199	4436	7003	720	52	249	346	893	12
	Actor Comp.	60	156	1652	3376	120	39	240	188	554	4
	POR-high	22	130	734	733	105	37	197	733	732	105
	POR-low	10	86	166	165	24	15	136	165	164	24
quicksort	None	3601	522	176871	176870	38245	852	373	43438	43437	11088
	JPF Comp.	3601	516	83278	183734	3995	29	123	435	1196	5
	Actor Comp.	2075	465	42472	105877	2021	14	104	120	309	2
	POR-high	8	108	74	73	16	7	85	37	36	8
	POR-low	392	260	13281	13280	2772	33	164	912	911	210
scalawiki	None	-	-	-	-	-	640	372	22522	22521	4152
	JPF Comp.	-	-	-	-	-	215	207	6513	7303	1081
	Actor Comp.	-	-	-	-	-	179	217	5456	6267	836
	POR-high	-	-	-	-	-	923	382	22522	22521	4152
	POR-low	-	-	-	-	-	197	264	4644	4643	836
server	None	4	42	27	26	6	6	77	26	25	6
	JPF Comp.	6	41	24	24	5	6	75	23	23	5
	Actor Comp.	5	41	21	22	4	6	76	20	21	4
	POR-high	4	42	19	18	4	7	77	26	25	6
	POR-low	5	58	27	26	6	6	72	18	17	4
shortpath	None	178	238	10000	9999	3614	223	245	10000	9999	3614
	JPF Comp.	38	120	887	1772	140	35	126	534	1160	69
	Actor Comp.	18	110	261	608	28	20	115	230	556	22
	POR-high	13	89	287	286	98	16	114	287	286	98
	POR-low	50	154	1690	1689	408	58	212	1690	1689	408
spinsort	None	89	200	5046	5045	1152	118	234	5045	5044	1152
	JPF Comp.	22	99	528	861	31	18	103	317	508	24
	Actor Comp.	15	100	287	459	19	17	101	269	432	24
	POR-high	37	145	1290	1289	288	43	182	1289	1288	288
	POR-low	121	274	5046	5045	1152	145	181	5045	5044	1152

B. Subjects

Our Basset experiments use nine actor programs listed in Table I. The `server` subject is our running example described in Section III. Three of the subjects implement more complex algorithms and were previously used in the dCUTE study [21]: `leader` is an implementation of a leader election algorithm; `spinsort` is a simple distributed sorting algorithm, and `shortpath` is an implementation of the Chandy-Misra's shortest path algorithm [46]. The `fibonacci` subject computes the n-th element in the Fibonacci sequence. `mergesort` and `quicksort` are implementations of distributed sorting algorithms that use a standard divide-and-conquer strategy to carry out the computation. `pi` computes an approximation of the π number by splitting the task among a set of worker actors. Finally, `scalawiki` is the original client-server application previously available from the ScalaWiki website. Our use of Basset exposed an atomicity violation in this code, which has been corrected in the latest version of the example. We did not translate the entire `scalawiki` from Scala into ActorFoundry but only translated the simplified `server`.

All these subjects can be executed in the standard environments for Scala or ActorFoundry. No modification to the subjects' code was necessary to explore them using Basset.

C. State-space reduction

As discussed in Section VI, Basset provides two mechanisms for reducing the exploration of state space: state comparison and a partial-order reduction based on the happens-before relation. In addition to the default state comparison provided by JPF, we implemented a custom state comparison to improve the identification of previously visited states (Section IV-D). The abstraction we use for state comparison allows for more aggressive pruning of redundant message schedules, which, in turn, results in faster state-space exploration.

Table I shows the results of experiments comparing JPF's standard state comparison (JPF), the custom actor state comparison (Actor), and the dCUTE partial-order reduction implemented in Basset (POR-high and POR-low). For reference purposes, results without state comparison or partial-order reduction (None) have also been provided.

For each type of state-space reduction, we tabulate the total exploration time in seconds, memory usage in MB, the number of states identified during the entire exploration, the total number of messages (across all execution traces) that were delivered during the exploration, and the total number of executions. Effectively, the number of states and messages are the number of nodes and edges, respectively, in the state-space graph that Basset explores for these programs. The variations in numbers between ActorFoundry and Scala are due to differences in how the subjects were implemented and how the drivers establish the initial state.

Execution time typically improves as we progress through the three types of state comparison, from None to JPF to the Actor comparison. In all cases, the Actor comparison results in the fastest of these explorations. Memory utilization remains reasonable across all of the experiments, usually varying in line with the total number of explored states. Similar to reducing execution time, the Actor comparison reduces the number of explored states and the number of delivered messages. As the abstraction used by the state comparison is refined to consider only relevant state differences, the number of states and executions that can be pruned increases. As a result, the number of executions is not a particularly meaningful statistic when state comparison is used. The pruning of exploration can greatly reduce the number of executions that finish.

As previously mentioned in Section IV-E, the partial-order reduction is very sensitive to the order in which it chooses actors and messages for message delivery. To illustrate this, we ran experiments using two different orderings: POR-low delivers messages to actors in order in which the actors were created (from low to high actor ids), while POR-high delivers messages in the reverse order.

The results show that state comparison and partial-order reduction provide different speedups, with one or the other being better for different subjects. The differences in results between POR-low and POR-high show that it is worthwhile to investigate heuristics for comparing orderings of messages and actors. We believe that Basset provides an excellent research platform for such experiments on state-space exploration of actor programs, the same way that JPF has provided an excellent research platform for state-space exploration of Java programs.

VIII. Related Work

The work most related to Basset is on model checking actor programs. Sen and Agha present dCUTE [21] which checks actor programs written using a simplified actor library, by re-executing the programs for various message schedules. Both dCUTE and Basset use a dynamic partial-order reduction based on the happens-before relation to avoid exploring equivalent schedules. dCUTE combines this partial order reduction with a mixed concrete and symbolic execution for test generation [47]–[49]. In contrast, Basset provides a common framework for *stateful* exploration of Java-based actor libraries and handles *full actor libraries* for Scala and ActorFoundry (including dynamic creation and destruction of actors). Also, Basset is built on top of JPF and can reuse its functionality for state-space exploration (e.g., heuristics for ordering exploration).

Fredlund and Svensson present McErlang [22], a stateful model checker for actor programs written in the Erlang programming language [5]. McErlang, which is itself written in Erlang, modifies the concurrency system of the Erlang run-time library. A previous model checker for Erlang, etomcrl [23], checked Erlang programs by translating them into μCRL and using off-the-shelf model checkers, similarly as the very first version of JPF [50] checked Java programs by translating them into Promela and using SPIN [51]. Again, Basset does not focus on one language/library but provides a general framework built on an existing tool (JPF) and additionally incorporates several existing optimizations for exploration.

Bordini et al. present [52] a translation from AgentSpeak, a widely used agent-oriented programming language, into Java so that the original program could be verified using JPF. Agents in AgentSpeak share some similarities with actors. For instance, an agent communicates only by exchanging messages, and it has a private mailbox to queue up messages that cannot be processed immediately, just like actors. While their work focused on mapping AgentSpeak features into Java constructs, so that the resulting program can be executed in JPF, we instead focused on customizing JPF so that different Java-based actor systems can be tailored to Basset with a minimum effort.

Also related to Basset is work on checking distributed systems [24], [25], [53], [54]. In particular, Artho and Garoche [24] and Barlas and Bultan [25] provide frameworks for executing distributed Java code in JPF. A key problem is that such code uses network calls that JPF

does not support as they depend on native code from the Java standard libraries. These two projects solve this problem by instrumenting the bytecode [24] or providing stub classes [25]. These solutions are conceptually similar to Basset in that they replace/avoid the standard Java network library as Basset replaces actor libraries. The solutions would be also valuable for checking migration of actors. However, both solutions focus on low-level communication, whereas Basset focuses on high-level exploration of possible behaviors for actor programs.

Partial-order reduction is an important optimization for alleviating the state-space explosion in model checking [20], [26], [27], [55]–[57]. It uses static or dynamic analysis to avoid exploring certain message schedules altogether. As discussed in Section IV-E, Basset provides a dynamic partial-order reduction based on the happens-before relation. It also facilitates state-space reduction through the use of state comparison to determine when to prune exploration. At present, the partial-order reduction and the use of state comparison are mutually exclusive. Recent work by Yang et al. [37] and Yi et al. [38] proposes combining these two optimizations, which we plan to investigate in the future.

IX. Conclusions

We described Basset, a general framework for exploring Java-based actor programs. Basset is implemented on top of JPF, and we instantiated it for two systems: the Scala programming language and the ActorFoundry library. Experience with Basset suggests that a general purpose framework for automated testing of actors can be efficient and effective. Experimental results show that using Basset to explore the state space of actor program executions is more efficient than directly exploring the code and its libraries. Experiments also suggest that Basset can effectively explore executions of actor programs, as demonstrated by the discovery of a previously unknown bug in a sample Scala code available from the ScalaWiki web site (the bug was fixed after the authors confirmed our bug report).

We believe that Basset can serve as an excellent research platform for experiments on state-space exploration of actor programs. In the future, we expect Basset to be instantiated for more actor systems. We plan to investigate further capabilities and optimizations in Basset, and to use it to test other actor applications. Basset simplifies the development of tools for the automated testing of programs in new actor languages and runtime libraries, while at the same time making new techniques for testing readily available for all actor languages and libraries.

Acknowledgments

The authors would like to thank Stoyan Gaydarov, Bobak Hadidi, Vilas Jagannath, Rajesh Karmani, Viktor Kuncak, P. Madhusudan, Vasko Popstojanov, and Samira Tasharofi for discussions and other assistance during the course of this project. The authors would also like to thank the students from the Fall 2008 Concurrent Programming Languages class at the University of Illinois for their feedback on this work. This material is based upon work partially supported by the NSF under Grant Nos. CNS-0851957, CCF-0746856, CNS-0615372, and CNS-0509321.

References

[1] G. Agha, *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[2] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "A foundation for actor computation," in *Journal of Functional Programming*, 1997.

[3] Microsoft, "Axum webpage," http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx.

[4] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based On C," in *OOPSLA*, 1993, pp. 91–108.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG*. Prentice Hall, 1993, Second Edition.

[6] "E webpage," http://www.erights.org/elang/index.html.

[7] "Newspeak webpage," http://newspeaklanguage.org/.

[8] "Ptolemy II webpage," http://ptolemy.berkeley.edu/ptolemyII/.

[9] "Revactor webpage," http://revactor.org/.

[10] W. Kim, "ThAL: An actor system for efficient and scalable concurrent computing," Ph.D., University of Illinois at Urbana-Champaign, May 1997.

[11] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, 2007.

[12] Microsoft, "Asynchronous agents library," http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx.

[13] "ActorFoundry webpage," http://osl.cs.uiuc.edu/af/.

[14] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: A comparative analysis," in *PPPJ*, 2009.

[15] "Jetlang webpage," http://code.google.com/p/jetlang/.

[16] "Jsasb webpage," https://jsasb.dev.java.net/.

[17] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for Java," in *ECOOP*, 2008.

[18] C. A. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.

[19] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima, 2008.

[20] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[21] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *FASE*, 2006.

[22] L.-Å. Fredlund and H. Svensson, "McErlang: A model checker for a distributed functional programming language," in *ICFP*, 2007.

[23] T. Arts and C. B. Earle, "Development of a verified Erlang program for resource locking," in *FMICS*, 2001.

[24] C. Artho and P.-L. Garoche, "Accurate centralization for applying model checking on networked applications," in *ASE*, 2006.

[25] E. Barlas and T. Bultan, "NetStub: A framework for verification of distributed Java applications," in *ASE*, 2007.

[26] J. Yang, T. Chen, M. Wu, Z. Xu, H. L. Xuezheng Liu, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: Transparent model checking of unmodified distributed systems," in *NSDI*, 2009.

[27] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak, "CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems," in *NSDI*, 2009.

[28] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, April 2003.

[29] "Java PathFinder webpage," http://javapathfinder.sourceforge.net.

[30] C. Spermann and M. Leuschel, "ProB gets Nauty: Effective Symmetry Reduction for B and Z Models," in *TASE*, 2008.

[31] "Twitter on Scala," http://www.artima.com/scalazine/articles/twitter_on_scala.html.

[32] "ScalaWiki website," http://scala.sygneca.com/.

[33] C. Houck and G. Agha, "Hal: A high-level actor language and its distributed implementation," in *ICPP*, 1992.

[34] I. A. Mason and C. L. Talcott, "A semantically sound actor translation," in *ICALP*, 1997.

[35] P. Haller and M. Odersky, "Actors that unify threads and events," in *COORDINATION*, 2007.

[36] D. Lea, "A Java fork/join framework," in *Java Grande*, 2000.

[37] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Efficient stateful dynamic partial order reduction," in *SPIN*, 2008.

[38] X. Yi, J. Wang, and X. Yang, "Stateful dynamic partial-order reduction," in *ICFEM*, 2006.

[39] R. Iosif, "Exploiting heap symmetries in explicit-state model checking of software," in *ASE*, 2001.

[40] M. Musuvathi and D. L. Dill, "An incremental heap canonicalization algorithm," in *SPIN*, 2005.

[41] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *ISSTA*, 2002.

[42] M. d'Amorim, "Efficient explicit-state model checking of programs with dynamically-allocated data structures," Ph.D. thesis, 2007.

[43] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM*, 1978.

[44] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[45] P. Godefroid, "Model checking for programming languages using Verisoft," in *POPL*, 1997.

[46] K. M. Chandy and J. Misra, "Distributed computation on graphs: Shortest path algorithms," *Comm. ACM*, 1982.

[47] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005.

[48] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE*, 2005.

[49] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[50] K. Havelund, "Java PathFinder, A translator from Java to Promela," in *SPIN*, 1999.

[51] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, 1997.

[52] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "Verifying multi-agent programs by model checking," *Autonomous Agents and Multi-Agent Systems*, vol. 12, no. 2, March 2006.

[53] S. D. Stoller, "Model-checking multi-threaded distributed Java programs," in *SPIN*, 2000.

[54] D. Hughes, "A framework for testing distributed systems," in *P2P*, 2004.

[55] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state space exploration," in *CAV*, 1997.

[56] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, 2005.

[57] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.