Targeted Test Generation for Actor Systems

Sihan Li
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
sihanli2@illinois.edu

Farah Hariri
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
hariri2@illinois.edu

Gul Agha
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
agha@illinois.edu

Abstract

This paper addresses the problem of targeted test generation for actor systems. Specifically, we propose a method to support generation of system-level tests to cover a given code location in an actor system. The test generation method consists of two phases. First, static analysis is used to construct an abstraction of an entire actor system in terms of a *message flow graph* (MFG). An MFG captures potential actor interactions that are defined in a program. Second, a *backwards symbolic execution* (BSE) from a target location to an "entry point" of the actor system is performed. BSE uses the MFG constructed in the first phase of our targeted test generation method to guide execution across actors. Because concurrency leads to a huge search space which can potentially be explored through BSE, we prune the search space by using two heuristics combined with a feedback-directed technique. We implement our method in TAP, a tool for Java *Akka* programs, and evaluate TAP on the *Savina* benchmarks as well as four open source projects. Our evaluation shows that the TAP achieves a relatively high target coverage (78% on 1,000 targets) and detects six previously unreported bugs in the subjects.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases actors, symbolic execution, test generation, static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.8

Acknowledgements This work is supported in part by the National Science Foundation under grants NSF CCF 14-38982 and NSF CCF 16-17401. The authors would like to thank Krishna Kura for his help on transforming the benchmarks to Java *Akka* and thank Nadeem Jamali and the anonymous reviewers for their helpful comments and suggestions.

1 Introduction

We address the problem of targeted test generation for actor systems. Recall that an actor is an autonomous, concurrent agent which communicates with other actors using asynchronous messages. Asynchronous message-passing and state encapsulation (isolation) in actors make it easier to understand the message flow and facilitate scalability. State encapsulation prevents low-level data races and atomicity violations. Asynchronous message-passing avoids syntactic deadlocks [6, 7]. As a result, actor languages and frameworks–such as Erlang [9], Salsa [40], Scala/Java *Akka* [4, 2], and Orleans [3]–have gained in popularity, and have been used for scalable applications (for example, see [1, 3]).

© Sihan Li, Farah Hariri, and Gul Agha;
licensed under Creative Commons License CC-BY

A goal of testing programs is to detect violations of desired safety properties. Some safety properties such as “no dangling links” or “division by zero” are implicit. Others are explicitly stated in the form of assertions. Violations of safety properties happen if particular lines of the code can be reached with problematic data. Because concurrency leads to nondeterminism, figuring out if particular lines of the code can be reached is challenging. By taking advantage of the actor semantics, more effective testing tools may be developed. One approach [33, 22] is to combine *concolic testing* [34] with *partial order reduction* based on a macro-step actor semantics [8]. Unfortunately, given the very large number of potential message schedules in an actor system, concolic testing is sometimes ineffective in determining if a particular code location can be reached.

An alternate approach is to use a *targeted test generation* technique to try to generate tests that cover specific code locations.[1] Targeted test generation has the advantage that one does not explore paths leading to code locations that obviously cannot have problems. Previous research has developed techniques and tools based on symbolic execution for targeted test generation for sequential programs (e.g., [24, 18, 16, 25, 10, 15, 29, 13]).

In this paper, we propose a method for generating targeted tests for actor systems based on *backward symbolic execution* (BSE). The tests we generate are system-level test: they exercise a group of interacting actors rather than only an isolated actor. The goal is to find if a particular line can be reached through sending messages to the *entry point* of an actor system, where an entry point is a message handler of an actor which interacts with the external environment. In actor terminology, such actors are called *receptionists*. Each test consists not only of the messages received by each actor but also the order in which these messages are received. We start a BSE from the target code and explore only those paths that are relevant to reaching that target; the exploration continues until a feasible path to an entry point of the actor system is found.

In sequential programs, a call graph is used to guide the inter-procedural BSE [15, 29, 13]. In the actor context, we propose to use an abstraction of an actor system called *message flow graph* (MFG). An MFG captures interactions between actors and is useful to guide inter-actor BSE. We develop a sound whole-system static analysis to construct MFGs for actor systems.

One challenge in static MFG construction and BSE for actor systems is to handle actor operations such as message send/receive and actor creation. Even when an actor framework is written in a language like Java, analyses that treat these actor operations as normal methods would not work: if the actor semantics is ignored, BSE will explore the library methods that are used to implement an actor runtime. Because a library that implements an actor runtime contains complex multi-threading and networking code, symbolic execution would become infeasible (cf. [22]). In addition, a static analysis would not be able to establish connections between actors without understanding the meaning of these library methods. To solve this problem, we define formal semantic models of actor operations in both MFG analysis and BSE, and replace actual implementations of actor operations with the semantic models. Assuming that the actor library has been correctly implemented, we prevent our analysis from exploring the underlying library. This makes our analysis more efficient and thus scalable.

In general, it is computationally intractable to consider every possible message arrival schedule even if we explore only paths that are relevant to a single target. To efficiently navigate the search space, we use a depth-first search strategy combined with two heuristics

[1] Targeted test generation is sometimes called *directed* or *guided* test generation in the literature.

and a feedback-directed search technique. The depth-first strategy attempts to reach the entry point of the actor system as soon as possible. The two heuristics are as follows:

1. Each message handler is executed atomically so that search space is reduced due to the lack of the interleaved execution of message handlers. This heuristic applies the macro-step semantics in the Actor model [8], which follows from the fact that messages to a given actor are processed one at a time and that actors do not share state.
2. Low weights are assigned to transitions in BSE that introduce more actors to be explored, in order to avoid unnecessary explorations. The heuristic is based on the conjecture that most concurrency bugs may be triggered by considering interactions of a small number of actors. We do not have direct evidence of this conjecture. However, there is a previous finding that most concurrency bugs in multi-threaded programs may be triggered using only two threads [23].

As constraints are collected and solved, some paths turn out to be infeasible. In this case, we deduce an *unsatisfiable core*–a subset of the constraint clauses whose conjunction is unsatisfiable. Our feedback-directed technique uses these unsatisfiable cores to effectively drive BSE towards a feasible path. The technique is particularly useful in cases where BSE frequently hits infeasible paths.

We have implemented our method in a tool called TAP for the Java *Akka* framework [2], a popular library enabling actor-style programming in Java. However, our method can be applied to other actor frameworks or languages. We evaluate TAP on *Savina* [21], a set of 30 third-party actor benchmarks, as well as on four open source actor projects from GitHub. The evaluation results show that TAP is effective in covering targets, achieving 78% target coverage on a total of 1,000 targets. The heuristics and the feedback-directed technique together substantially improve the target coverage over random search. In addition, TAP detects six previously unreported bugs in the subjects, five of which are crash bugs caused by out-of-order message delivery.

The paper makes the following contributions:

- **The MFG concept and construction:** We introduce the MFG abstraction for actor systems and develop a sound static analysis to construct it.
- **Modeling of actor operations in BSE:** We formally define the full semantics of actor operations in BSE for actor systems.
- **Efficient path exploration:** We propose two search heuristics and a feedback-directed technique to efficiently navigate the generally huge search space in BSE of actor systems.
- **Implementation and evaluation:** We implement our method in TAP for Java *Akka*, and conduct evaluations on benchmarks and real-world projects that demonstrate TAP's effectiveness in target coverage and bug detection.

2 Background

We provide background on the Actor model and the Java *Akka* framework. We also describe the targeted test generation problem for actor systems in terms of the inputs and outputs.

2.1 The Actor model

In the Actor model [19, 6, 8], an actor is an agent of computation; it performs computations as a response to a message. An actor is characterized by an actor name, a local state, and behaviors. The actor name serves as the address of the actor in the system; it can be passed around to other actors so that they may send messages to it. The local state of an actor is

```java
public class Main {
    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("Banking");
        ActorRef serverActor = system.actorOf(Server.props());
        ActorRef clientActor = system.actorOf(Client.props(serverActor));
    }
}
public class Client extends UntypedActor {
    private double balance = 100;
    private ActorRef server;
    @Override
    public void onReceive(Object message) {
        if (message instanceof WithdrawMessage) {
            double amount = ((WithdrawMessage) message).amount;
            if(balance >= amount) {
                balance -= amount;
                server.tell(message);
            }
        } else if(message instanceof DepositMessage) {
            double amount = ((DepositMessage) message).amount;
            balance += amount;
            server.tell(mesage);
        }
    }
}
public class Server extends UntypedActor {
    private double balance = 100;
    @Override
    public void onReceive(Object message) {
        if (message instanceof WithdrawMessage) {
            double amount = ((WithdrawMessage) message).amount;
            assert(balance >= amount);
            balance -=amount;
        } else if(message instanceof DepositMessage) {
            double amount = ((DepositMessage) message).amount;
            balance += amount;
        }
    }
}
```

Figure 1 A simplified Bank Account example.

encapsulated within the actor – no external entity can change it directly. The only way to change the local state of an actor is to send it a message that triggers this actor to change its own state. Upon receiving a message, an actor can have the following three behaviors: (1) performing local computations (updating its local state), (2) sending messages to actors, or (3) creating new actors. Communication between actors is through *asynchronous* message passing – the sender does not block its computation waiting on the recipient to process the message, nor does it assume the order in which the recipient processes its incoming messages. Messages are immutable and processed by the recipient one at a time without interleaving. An actor system contains a group of actors. The subset of actors that can communicate with the external environment are called *receptionists*, and the other actors in the system are called internal actors.

2.2 Actors in Akka

Akka is a set of libraries for developing distributed and scalable systems on the Java Virtual Machine. It can be used in both Scala and Java. The core of *Akka* is the `akka-actor` library, which is an implementation of the Actor model. Figure 1 shows a simplified Bank Account example written using Java *Akka*. We use this example to illustrate important concepts of the Actor model in the context of *Akka*.

Actor Creation. To create actors, we need to first create the enclosing actor system (Line 3 in Figure 1), a container in which the actors run. Then we create actors that live in the system via the method `actorOf`. The example creates two actors: a client and a server (Lines 4-5). The `actorOf` method takes as input a configuration object (`props`) that specifies the options for creating an actor such as its type and arguments to its constructor, and returns an `ActorRef` object, which represent the address of the actor in the system. The `ActorRef` corresponds to the concept, actor name in the Actor model. Following the naming convention in *Akka*, we will use the terms actor reference and actor name interchangeably in this paper. Other actors can send a message to this `ActorRef`, and the actor identified by this `ActorRef` will receive this message. Note that other actors *cannot* directly access the local state of this actor (e.g., access fields, call instance methods) through the `ActorRef`.

Acquaintance Relations. Actor `A` knows of actor `B` if `A` has access to the actor reference (`ActorRef`) of `B`. At Line 5 in our example, we create a `clientActor` and pass it the `ActorRef` of the `serverActor` (now the client knows of the server and can send messages to it). The actor reference can also be sent as a message to inform other actors. Another type of acquaintance between actors is via receiving messages: when an actor receives a message, it can access the actor reference of the sender through the `getSender()` method. An actor can also get its own actor reference through the `getSelf()` method.

Sending and Processing Messages. Every actor must implement a message handler, the `onReceive` method. The `onReceive` method takes as input a message object, and is invoked upon receiving a message. Typically, different types of messages trigger different behaviors in the actor. For example, the `onReceive` of the `Client` actor (Lines 13-23) behaves differently on the `WithdrawMessage` and the `DepositMessage`. Messages are sent via calling the `tell` method on an `ActorRef` object (e.g., Line 17).

2.3 Problem Description

Actors model an *open system* – a system that may interact with its external environment. In order to preserve locality properties of actors, such interaction is through messages received by receptionist actors in the system and messages sent to external actors by actors in the system. Thus the entry points of the system are message handlers of receptionists. Examples of open systems in the real-world include Twitter, LinkedIn, Facebook Chat, and Halo 4, all of which have been implemented using actors.

The input to our problem includes: (1) the code under test, (2) a target code location, (3) a user defined set of receptionists of the system, and (4) a start configuration defining the initial acquaintance between actors. The output (if found) is a test case that covers the target. Such a test consists of messages sent to relevant actors as well as their arrival orders on each of these actors.

In our Bank Account example, the code under test is the `Client` and the `Server` actor classes; the receptionist is the `Client` actor as the client is the interface of the system for user interactions. The main method sets up the initial acquaintance that the client knows of the server. Suppose our target is the negation of the assertion at Line 32. One possible output test case that covers the target is as follows. The client receives a deposit message with the amount 50 and a withdraw message with the amount 120, in that order. Since the deposit message is received before the withdraw message, the condition at Line 15 is evaluated to true, and the client forwards both messages to the server. However, on the server side, the withdraw message somehow arrives before the deposit message, causing the assertion violation. This test case specifies the messages received by the client and the server

$$
\begin{aligned}
\mathsf{Class} &::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{C''\ f;}\ K\ \overrightarrow{M}\} \\
\mathbf{ActorClass} &::= \texttt{class}\ \mathbf{C}\ \texttt{extends}\ \mathbf{C'}\ \{\overrightarrow{\mathbf{C''\ f;}}\ \mathbf{K\ R}\ \overrightarrow{\mathbf{M}}\} \\
K \in \mathsf{Ctor} &::= C(\overrightarrow{C'\ f})\ \{\texttt{super}(\overrightarrow{f'});\ \overrightarrow{\texttt{this}.f''\ =\ f''';}\} \\
\mathbf{R} \in \mathbf{Receive} &::= \texttt{void onReceive}(\mathbf{C\ v})\ \{\overrightarrow{\mathbf{C'\ v';}}\ \overrightarrow{s}\} \\
M \in \mathsf{Method} &::= C\ m(\overrightarrow{C'\ v})\ \{\overrightarrow{C'v';}\ \overrightarrow{s}\} \\
s \in \mathsf{Stmt} &::= v = e;^{\ell} \mid \texttt{return}\ v;^{\ell} \mid \texttt{if}\ (e)\ \overrightarrow{s}\ \texttt{else}\ \overrightarrow{s'};^{\ell} \mid \mathbf{v}.\texttt{send}(\mathbf{v'});^{\ell} \\
e \in \mathsf{Expr} &::= v \mid (C)\ v' \mid v.f \mid v.m(\overrightarrow{v'}) \mid \texttt{new}\ C(\overrightarrow{v}) \mid v\ \mathsf{op}\ v' \mid \mathbf{aref} \\
\mathbf{aref} \in \mathbf{ARef} &::= \texttt{create}(\mathbf{C}.\texttt{class},\ \overrightarrow{\mathbf{v}}) \mid \texttt{self} \mid \texttt{sender} \\
v &\in \mathsf{Var}\ \text{is a set of variable names} \\
f &\in \mathsf{FieldName}\ \text{is a set of field names} \\
C &\in \mathsf{ClassName}\ \text{is a set of class names} \\
m &\in \mathsf{MethName}\ \text{is a set of method names} \\
\ell &\in \mathsf{Lab}\ \text{is a set of labels} \\
\mathsf{op} &\in \{+, -, *, /, <, >, ==, !=, \ldots, \texttt{instanceof}\}
\end{aligned}
$$

Figure 2 An actor language extending Featherweight Java.

as well as the message receiving orders on both actors. For illustration purposes, we do not assume the first-in-first-out (FIFO) message delivery between a pair of actors in this example. Given FIFO message delivery, the two messages could be routed through different actors, still creating nondeterminism in the arrival order at the server.

Note that we must specify the receptionists of an actor system in our problem settings. This requirement enforces system-level testing because internal actors can only be tested through receptionists. In our example, to cover the target we have to send messages to the client in order to trigger messages sent to the server. If all actors were potential receptionists, then every actor may receive messages directly from the external environment. In this case, each actor may be tested individually with all possible message sequences and no interaction between actors need be considered. The benefit of considering external messages only to designated actors is that it constrains the generated tests to those which would realistically occur in an actor system. While this means that system-level testing is required, it eliminates consideration of tests based on arbitrary messages to individual actors that would never be sent in a realistic system.

3 Actor Language

To formally describe our method, we define a simplified actor language by extending Featherweight Java [20] and adding actor constructs to it. We choose the Featherweight Java language for its simplicity and for the fact that our tool targets Java *Akka*. The formalism in this paper largely follows the conventions in previous work [20, 35, 26]. The actor constructs in our language resemble the counterparts in Java *Akka*. Although there have been formalizations of actor languages [8, 32], our formalization of the language is closely coupled with our analysis, and includes more details such as data store and context, which are required to specify our analysis.

$$
\begin{aligned}
\alpha \in \mathit{ActorMap} &= \mathit{ActorRef} \to \mathit{ActorState} \\
msg \in \mathit{Message} &= \mathit{ActorRef} \times \mathit{ActorRef} \times \mathit{Obj} \\
r \in \mathit{ActorRef} &\subset \mathit{Obj} \\
\varsigma \in \mathit{ActorState} &= \mathsf{Stmt} \times \mathit{Stack} \times \mathit{Store} \times \mathit{CallStack} \times \mathit{Context} \\
st \in \mathit{Stack} &= (\mathsf{Var} \rightharpoonup \mathit{Addr})^* \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathit{Obj} \\
o \in \mathit{Obj} &= \mathit{HContext} \times (\mathsf{FieldName} \rightharpoonup \mathit{Addr}) \\
cs \in \mathit{CallStack} &= (\mathsf{Stmt} \times \mathit{Context} \times \mathit{Addr})^* \\
a \in \mathit{Addr} &= (\mathsf{Var} \times \mathit{Context}) \cup (\mathsf{FieldName} \times \mathit{HContext}) \\
c \in \mathit{Context} &\text{ is an infinite set of regular contexts} \\
hc \in \mathit{HContext} &\text{ is an infinite set of heap contexts}
\end{aligned}
$$

Figure 3 Domains of actor maps and messages.

3.1 Syntax

Figure 2 describes the grammar of a simplified actor language. The language is in A-Normal form, where computations are syntactically sequentialized. For example, the statement `v = o.m(o.f)` is transformed to two statements `v1 = o.f; v2 = o.m(v1)` in A-Normal form. Such transformation brings our language closer to an intermediate language for simpler semantics definitions. Most of the notations in Featherweight Java are intuitive. We give a quick reminder of the less obvious conventions. A class declaration consists of a list of fields (we use an arrow to represent a list), a single constructor, and a list of methods. The constructor takes as input a list of arguments and assigns each argument to the corresponding field. Each statement in the language is assigned a distinct label. We augment Featherweight Java with *binary* expressions and *if* statements, which are later needed in the formalization of the BSE semantics. We omit the *loop* statement because loops are bounded and unrolled into *if* statements in our analysis. Such unrolling trades completeness for tractability and is standard practice in testing.

We now introduce actor constructs (highlighted in bold). Each actor class declaration must include exactly one `onReceive` method. This method takes a single input (message) and returns `void`. An actor creation operation `create`$(A$`.class`$, \overrightarrow{v})$ takes as input the class of the actor to be created C, followed by a list of arguments to the constructor of C, and returns the actor reference of the created actor. A message send operation v`.send`(v')`;`$^\ell$ sends the message v' to the actor reference v of the recipient actor.

3.2 Concrete Semantics

An instantaneous snapshot of an actor systems is called a *configuration*.[2] The semantics of our language is defined by a transition relation on configurations. A configuration is a tuple,

$$\langle \alpha \mid \mu \rangle$$

[2] Recall that actors are asynchronous: there is no unique global time. Thus an actor snapshot is with respect to some frame of reference, i.e., a causally consistent linearlization of a partial order.

Actor Creation

$$\langle \alpha \bullet r \mapsto (\, [\![v = \texttt{create}(C\texttt{.class},\ \overrightarrow{v'}\texttt{)}\texttt{;}^{\ell}]\!], st, \sigma, \_\,) \mid \_ \rangle \Rightarrow_C$$
$$\langle \alpha \bullet r \mapsto (\, \mathit{succ}(\ell), st, \sigma', \_\,) \bullet r' \mapsto \varsigma \mid \_ \rangle,\ \text{where}$$
$$r' \text{ is fresh} \qquad \sigma' = \sigma + [st(v) \mapsto r'] \qquad o'_i = \sigma(st(v'_i)) \qquad \varsigma = (\texttt{nil}, [], \sigma'', [], \texttt{nil})$$
$$\overrightarrow{f} = \mathcal{F}(C) \quad a_i = (f_i, hc) \quad o = (hc, [f_i \mapsto a_i]) \quad \sigma'' = [a_{this} \mapsto o, a_i \mapsto o'_i, a_{self} \mapsto r']$$

Message Sending

$$\langle \alpha \bullet r \mapsto (\, [\![v\texttt{.send}(v')\texttt{;}^{\ell}]\!], st, \sigma, \_\,) \mid \mu \rangle \Rightarrow_C$$
$$\langle \alpha \bullet r \mapsto (\, \mathit{succ}(\ell), st, \sigma, \_\,) \mid \mu \bullet \big(r, \sigma(st(v)), \sigma(st(v'))\big) \rangle$$

Message Receiving

$$\langle \alpha \bullet r \mapsto (\, \texttt{nil}, st, \sigma, cs, c\,) \mid \mu \bullet (r', r, o) \rangle \Rightarrow_C$$
$$\langle \alpha \bullet r \mapsto (\, s, st', \sigma', cs', c'\,) \mid \mu \rangle,\ \text{where}$$
$$c' \text{ is fresh} \qquad o_0 = \sigma(a_{this}) \qquad [\![\texttt{void onReceive}\ (C\ v)\ \{\overrightarrow{C'\ v'};\ \overrightarrow{s}\ \}]\!] = \mathit{rec}(\mathit{cls}(o_0))$$
$$s = \mathit{car}(\overrightarrow{s}) \qquad a = (v, c') \qquad a'_i = (v'_i, c') \qquad st' = \mathit{cons}([v \mapsto a, v'_i \mapsto a'_i], st)$$
$$cs' = \mathit{cons}((\texttt{nil}, c, \texttt{nil}),\ cs) \qquad \sigma' = \sigma + [a \mapsto o,\ a_{sender} \mapsto r']$$

Self Reference

$$\langle \alpha \bullet r \mapsto (\, [\![v = \texttt{self;}^{\ell}]\!], st, \sigma, \_\,) \mid \_ \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\, \mathit{succ}(\ell), st, \sigma + [st(v) \mapsto r)], \_\,) \mid \_ \rangle$$

Sender Reference

$$\langle \alpha \bullet r \mapsto (\, [\![v = \texttt{sender;}^{\ell}]\!], st, \sigma, \_\,) \mid \_ \rangle \Rightarrow_C$$
$$\langle \alpha \bullet r \mapsto (\, \mathit{succ}(\ell), st, \sigma + [st(v) \mapsto \sigma(a_{sender})], \_\,) \mid \_ \rangle$$

Figure 4 Concrete semantics for actor operations of the simplified actor language.

where α is an actor map that maps a finite set of actor references to actor states, and μ is a finite multi-set of pending messages. It is important to note that by modeling the pending messages as a multi-set, the order in which messages are sent is not preserved. As a result, our language semantics does not guarantee the FIFO message delivery between a pair of actors. We choose not to assume the FIFO message delivery in both the concrete language semantics and the BSE semantics in Section 5.1, because the FIFO semantics is not primitive in the Actor model [19, 6, 8]. However, one can easily accommodate the FIFO semantics in our models by replacing the multi-set with a data structure that preserves the message sending orders (e.g., a set of lists representing a sequence of messages, one list for each pair of a sender and a receiver). Since most real-world actor languages and frameworks guarantee the FIFO message delivery, we do implement the FIFO semantics in our tool.

The domains in a configuration are described in Figure 3. A message is a tuple consisting of the actor reference of the sender, the actor reference of the recipient, and the message content. An actor reference is an object that stores the location information of an actor. An actor state ς consists of a statement under execution, a data stack to store local variables, a data store of points-to relations, a call stack to track active method invocations, and a current execution context. A data stack *st* consists of a list of data frames, each of which maps local variables to addresses. A data store σ maps addresses to objects. A call stack *cs* consists of a list of call frames, and each call frame consists of the statement to return to,

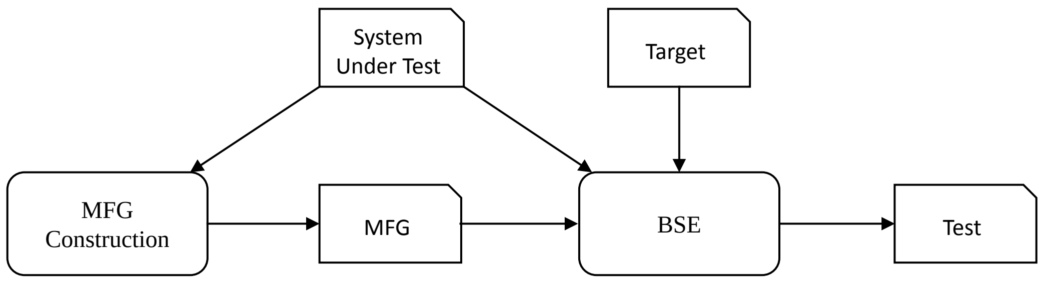

Figure 5 The overview of our two-phased test generation method.

the context to restore, and the address to store the return value. An object o consists of a heap context and a list of fields. An address is a location that holds an object. An address a consists of either a local variable and its regular context (allocated for a local variable) or a field and its heap context (allocated for a field). In the concrete semantics, every dynamic object instance has a unique heap context, and every dynamic method call has a unique regular context.

We express the concrete semantics of our language as a transition relation ($\Rightarrow_C$) from one configuration to another. Figure 4 shows the semantics of actor operations only. The semantics of local computations in an actor is similar to the normal semantics of Java, and thus omitted. For simplicity, we use underscore _ in our transition rules, to represent the remaining states in a tuple that are neither used nor updated in the transition. We use standard functions $car, cdr, cons, list$ to manipulate lists, and define a number of helper functions: $succ$ returns the next statement given the label of the current statement, $\mathcal{F}$ returns a list of field names for a given class, cls returns the class name of a given object, and rec returns the declaration of the `onReceive` method of a given class. We use the operator $\bullet$ to add an element to a set, and the notation $+$ to insert or update (if existing) entries in a map. We use `nil` as the null value for every domain. A *fresh* value means that a new value is generated from the corresponding domain. The symbols a_{this}, a_{self}, and a_{sender} represent reserved addresses to store the *this* object, the actor reference of itself, and the actor reference of a sender, respectively.

The *Actor Creation* rule says that a new actor is created with a fresh reference r' in the system. The actor has an initial state, where the current statement is `nil`. The *Message Sending* rule defines the asynchronous semantics of sending messages. The new message is put in the set of pending messages μ, and the sending actor continues its execution. Note that messages are immutable so that there are no concurrent writes on messages. The *Message Receiving* rule says that an actor can receive a message only when it is ready (i.e., the statement is `nil`). Upon receiving the message, the `onReceive` method is invoked, and the message is no longer pending and thus removed from μ. After executing the `onReceive` method, the statement is set to `nil`, signifying that the actor is ready to receive a message again. The *Self Reference* rule says that the actor reference of the *this* object is assigned to a local variable v. Similarly, the *Sender Reference* rule says that the actor reference of the message sender is assigned to a local variable v.

4 Message Flow Graph Construction

Our test generation method operates in two phases as shown in Figure 5. In the first phase, we use static analysis to construct a *message flow graph* (MFG), an abstraction of an actor system that models potential interactions (i.e., actor creation and communication) between

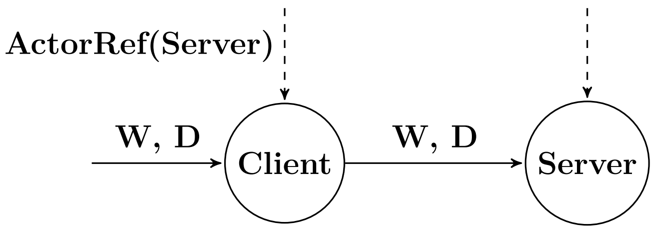

Figure 6 The MFG of the Bank Account example. The symbols W and D represent the withdraw message and the deposit message, respectively.

actors in the system. The input to our MFG analysis is the system under test including the code and the specified receptionists, and the output is an MFG of the system. In the second phase, we use BSE to generate a test that covers a given target. To generate tests that exercise multiple actors, BSE must go across actors. The MFG from the first phase is a key input that enables *inter-actor* BSE. After BSE reaches the entry of the message handler on an actor a, it queries the MFG to obtain actors that can send the required message to the actor a. Then BSE picks one potential sender, jumps to the exit of the message handler of the sender, and continues with the previous path constraint carried over. When a feasible path is found during the path exploration, we generate the test from the path constraint. We next explain the MFG construction and the BSE (in Section 5) in details.

An MFG is a directed graph between *abstract* objects, where an abstract object represents multiple concrete objects of the same class whose field values have been merged into a set. Specifically, a node in the MFG represents an abstract actor and a directed edge between two nodes means that the abstract actor represented by the source node either creates or sends a message to the abstract actor represented by the sink node. MFG edges are labeled with abstract constructor parameters for actor-creation edges and abstract messages for message-sending edges. Note that the MFG edges do not indicate the acquaintance between actors–it is possible that an actor a knows of another actor b, but there is no edge from a to b because a neither creates b nor sends a message to b.

An abstract object may be replaced by its class if there is only one abstract object per class. Figure 6 shows the MFG of the Bank Account example. There are two actors, `Client` and `Server`, in the graph. The symbols W and D represent the `WithdrawMessage` and the `DepositMessage`, respectively. Both actors are created (creation edges are represented with dashed arrows) by the external environment. The `Client` is initialized with an actor reference mapped to the `Server`, and it can send `WithdrawMessage` and `DepositMessage` to the `Server`. The `Client` is the only receptionist of the system and can receive `WithdrawMessage` and `DepositMessage` from the external environment.

To construct a MFG, we need to not only resolve the recipient of each message-sending site and the actor being created of each actor-creation site, but also pass along the messages and constructor parameters between actors. This is because the message and constructor parameters can affect the analysis of receiving actors. We use points-to analysis to compute the points-to sets for messages, constructor parameters, and actor references. In addition, we model the semantics of actor operations so that analysis information can be carried across actor boundaries. In particular, the actor creation operation conceptually creates two objects: an actor reference object and a corresponding actor object. Our analysis keeps track of such mappings to resolve the actor being created, and passes the points-to set of constructor parameters to this actor for instantiation.

$$
\begin{aligned}
\omega \in \Omega &= \mathit{ActorState} \times \mathit{Graph} \times \mathit{RefMap} \\
\gamma \in \mathit{RefMap} &= \mathit{ActorRef} \to (\mathsf{ClassName} \times \mathit{Obj}) \\
r \in \mathit{ActorRef} &\subset \mathit{Obj} \\
G \in \mathit{Graph} &= \mathit{ActorRef} \times \mathit{ActorRef} \times \mathit{Type} \rightharpoonup (\mathcal{P}(\mathit{Obj}))^* \\
\mathit{Type} &= \{\texttt{create}, \texttt{send}\} \\
\varsigma \in \mathit{ActorState} &= \mathsf{Stmt} \times \mathit{Stack} \times \mathit{Store} \times \mathit{CallStack} \times \mathit{Context} \\
st \in \mathit{Stack} &= (\mathsf{Var} \rightharpoonup \mathit{Addr})^* \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathcal{P}(\mathit{Obj}) \\
o \in \mathit{Obj} &= \mathit{HContext} \times (\mathsf{FieldName} \rightharpoonup \mathit{Addr}) \\
cs \in \mathit{CallStack} &= (\mathsf{Stmt} \times \mathit{Context} \times \mathit{Addr})^* \\
a \in \mathit{addr} &= (\mathsf{Var} \times \mathsf{MethodName} \times \mathit{Context}) \cup (\mathsf{FieldName} \times \mathit{HContext}) \\
\mathit{Context} &= \mathit{HContext} = \mathsf{Lab}
\end{aligned}
$$

Figure 7 State space of the small-step state machine.

Note that passing only the type of the message or constructor parameter between actors can result in unacceptable imprecision in our analysis. For example, a common case is that an actor reference `r` is sent as a message to a recipient actor `A`; `A` receives `r` and then sends a message to `r`. When resolving `r` in `A`, we only know that the type of `r` is `ActorRef`, but we know nothing about the actor that lives in `r`. Thus, we have to conservatively assume all actor classes in our system may live in `r`, and add a message-sending edge from `A` to every actor class. To avoid such imprecision, we need to pass along the points-to sets of messages and constructor parameters instead of their types. We next formally describe our analysis.

4.1 Analysis Semantics

We express the semantics of our analysis using small-step state machines, each modeling one abstract actor. Communication between actors is modeled by global states shared across state machines. The domain Ω of a state machine is defined in Figure 7. The reference map γ stores the mappings between actor references and the actors created in the system. The graph G records the actor-creation and message-sending events between actors. Specifically, G maps a tuple of a source actor reference, a sink actor reference, and an operation type to a list of points-to sets of messages or constructor parameters. Visually, an entry in the map can be seen as a directed edge, with the label being the list of points-to sets. The *ActorState* is similar to the one defined in concrete semantics of our language except that now the *ActorState* is an abstract state: the store maps an address to a set of objects rather than one object; the regular and heap contexts are a finite set of statement labels. An important design decision made by our analysis is that we create only one abstract actor object per actor class. That is, actors of the same class created in different sites are merged into one abstract actor object by merging the points-to sets of the corresponding fields. In this way, we only need to create one state machine per actor class, making our analysis faster and more scalable. The incurred imprecision can be refined by the BSE in phase II because our BSE distinguishes every concrete actor.

Actor Creation

$$((\llbracket v = \texttt{create}(C\texttt{.class},\ \overrightarrow{v'})\texttt{;}^{\ell}\rrbracket, st, \sigma, \_), G, \gamma) \Rightarrow_A ((succ(\ell), st, \sigma', \_), G', \gamma'), \text{ where}$$
$$(\gamma', r) = getRef(\gamma, C) \qquad \sigma' = \sigma \sqcup [st(v) \mapsto \{r\}] \qquad r' \in \sigma(a_{self})$$
$$G' = merge(G, [(r', r, \texttt{create}) \mapsto list(\sigma(st(v'_i)))])$$

Message Sending

$$((\llbracket v\texttt{.send}(v')\texttt{;}^{\ell}\rrbracket, st, \sigma, \_), G, \_) \Rightarrow_A ((succ(\ell), st, \sigma, \_), G', \_), \text{ where}$$
$$r \in \sigma(st(v)) \qquad r' \in \sigma(a_{self}) \qquad G' = merge(G, [(r', r, \texttt{send}) \mapsto list(\sigma(st(v')))])$$

Message Receiving

$$((\texttt{nil}, st, \sigma, cs, c), G, \_) \Rightarrow_A ((s, st', \sigma', cs', c'), G, \_), \text{ where}$$
$$o_0 \in \sigma(a_{this}) \qquad \llbracket\texttt{void onReceive } (C\ v)\ \{\overrightarrow{C'\ v'};\ \overrightarrow{s'}\}\rrbracket = rec(cls(o_0)) \qquad s = car(\overrightarrow{s'})$$
$$(hc_0, \_) = o_0 \qquad c' = hc_0 \qquad a = (v, \texttt{onReceive}, c') \qquad a'_i = (v'_i, \texttt{onReceive}, c')$$
$$st' = cons([v \mapsto a, v'_i \mapsto a'_i], st) \quad cs' = cons((\texttt{nop}, c, \texttt{nil}), cs) \qquad r \in \sigma(a_{self})$$
$$O_r = preds(G, r, \texttt{send}, \gamma) \qquad O \in \{car(G((r', r, \texttt{send}))) \mid r' \in O_r\}$$
$$\sigma' = \sigma \sqcup [a \mapsto O, a_{sender} \mapsto O_r]$$

Self Reference

$$((\llbracket v = \texttt{self;}^{\ell}\rrbracket, st, \sigma, \_), \_) \Rightarrow_A ((succ(\ell), st, \sigma \sqcup [st(v) \mapsto \sigma(a_{self})], \_), \_)$$

Sender Reference

$$((\llbracket v = \texttt{sender;}^{\ell}\rrbracket, st, \sigma, \_), \_) \Rightarrow_A ((succ(\ell), st, \sigma \sqcup [st(v) \mapsto \sigma(a_{sender})], \_), \_)$$

Figure 8 Abstract semantics for actor operations in MFG analysis.

The analysis semantics is defined by the transition relation $(\Rightarrow_A) \subset \Omega \times \Omega$. The analysis semantics of local computations is precisely the 1-object-sensitive points-to analysis [27]. We provide the transition rules for local computations in the appendix. Figure 8 describes transition rules for the actor operations. The *getRef* function checks if the given class C is in the value set of γ. If found, it returns itself and the key of the value. If not found, it adds an entry $r \mapsto (C, \texttt{nil})$ to γ, where r is fresh, and returns the updated γ' and r. Since only one abstract actor object is created per actor class, an actor class can appear in at most one tuple in the value set of γ. The *merge* function merges the labels of edges with the same source and sink. The *preds* function finds all predecessors of a given type for a node r in the graph G and returns the set of actor objects mapped by the predecessors in γ.

In the *Actor Creation* rule, instead of instantiating the actor object at the creation site, an actor-creation event is recorded and merged into the graph. Subsequently, when a state machine for this actor class is created, actor-creation events are used to instantiate the single abstract actor object for this class. Similarly in the *Message Sending* rule, a message-sending event is recorded and merged into the graph. The *Message Receiving* rule says that the *onReceive* method of the actor is invoked upon receiving a message. The graph G is queried to find the set of all possible senders O_r, and the set of all possible messages received by O. Note that when updating the call stack, we use `nop` instead of `nil` for the statement to return to. `nop` indicates no operation to be performed and stops the state machine. Otherwise, the state machine will not halt.

Algorithm 1: Iterative MFG construction.

```
   Input  : An Actor system $P$, a raw graph $G \in \mathit{Graph}$, and an actor reference map
            $\gamma \in \mathit{RefMap}$
   Output : A message flow graph of $P$
 1 $\mathit{worklist} \leftarrow [\,]$     $\mathit{factStore} \leftarrow [\,]$
 2 $\mathit{worklist}$.appendAll( $\gamma$.keySet() )
 3 while $\mathit{worklist}$ not empty do
 4     $r \leftarrow \mathit{worklist}$.removeFirst()
 5     $\mathit{beforeFacts} \leftarrow$ InEdges $(r,\ G)$
 6     if $\mathit{factStore}[r] \neq \mathit{beforeFacts}$ then
 7         $\mathit{factStore}[r] \leftarrow \mathit{beforeFacts}$
 8         $\mathcal{M}_r \leftarrow$ CreateStateMachine $(r, G, \gamma)$
 9         $\mathcal{M}_r$.execute()
10         $\mathit{worklist}$.appendAll( Successors $(r, G)$ )
11     end
12 end
13 return CollapseToMFG $(\gamma, G)$
14 Procedure CreateStateMachine $(r,\ G,\ \gamma)$
15     $(C, \_) \leftarrow \gamma(r)$     $\overrightarrow{f} \leftarrow \mathcal{F}(C)$     $a_i \leftarrow (f_i, \ell_C)$
16     $o \leftarrow (\ell_C, [a_i \mapsto f_i])$                                // actor allocation
17     $\gamma \leftarrow \gamma + [r \mapsto (C, o)]$                               // ref map update
18     $\overrightarrow{O} \leftarrow [\emptyset, \ldots, \emptyset]$                         // a list of points-to sets
19     foreach $(r', r'', \mathit{create}) \mapsto \overrightarrow{O'}$ in InEdges $(r, G)$ do
20         $O_i \leftarrow O_i \cup O'_i$
21     end
22     $\sigma \leftarrow [a_{this} \mapsto \{o\}, a_i \mapsto O_i, a_{self} \mapsto \{r\}]$
23     $\omega_0 \leftarrow ((\mathtt{nil}, [\,], \sigma, [\,], \mathtt{nil}),\ G, \gamma)$
24     Create $\mathcal{M}_r$ with the initial state $\omega_0$
25     return $\mathcal{M}_r$
26 End
```

4.2 MFG Construction Algorithm

Algorithm 1 shows our iterative algorithm to construct the MFG. The algorithm takes as input an actor system P, a raw graph G, and a reference map γ, and outputs an MFG graph. G and γ are initialized from the driver code that sets up the actor system. Initially, G contains actor-creation and message-sending events by the external environment, and γ contains the mappings for actors created by the external environment. For each actor class, one state machine is instantiated to model the abstract actor of this class. The algorithm maintains a *worklist* that keeps track of the abstract actors to be analyzed next as well as a *factStore* that stores the relevant data facts for each abstract actor. The data facts for an abstract actor are essentially the set of incoming edges of this actor node in G, and these facts affect the initial state of the state machine for this actor.

The algorithm starts with pushing the initial actors onto the *worklist* (Line 2), and iteratively analyzes these actors one at a time. Before the analysis, the algorithm computes the relevant data facts for this actor from G (Line 5). It then checks whether the facts are changed, by comparing the computed facts with the previous facts stored in *factStore*. If changed, the algorithm updates the facts for this actor in *factStore* (Line 7), analyzes this actor with these new facts by instantiating and running the state machine described in

Section 4.1 (Lines 8-9), and pushes all the successors of this actor node onto *worklist* (Line 10). Otherwise, the algorithm skips this actor because the execution of its state machine will yield the same result and will not change the global state G. This process continues until *worklist* is empty, indicating a fixed point is reached. The `CreateStateMachine` procedure is the only place where instantiations of abstract actors happen. The constructor parameters of multiple actor-creation edges are merged (Lines 18-21) and the results are used to initialize the fields of the abstract object (Line 22). Finally, the algorithm builds an MFG from G and γ by collapsing the abstract objects of nodes and labels into classes. If an object is an actor reference, we also encode the class of the underlying actor into the MFG.

4.3 Optimizations

Our analysis applies two lightweight yet effective optimizations to actor classes based on the code pattern in actor programs. Since actors often receive multiple types of messages and behave differently for each message type, a common code pattern in actors' `onReceive` methods is that an *if* statement is used at the top of its control flow to check the message type and process one type of message in one branch. In our running example, both the `Client` and the `Server` actors follow this pattern.

Our first optimization eliminates unreachable code based on the potential types of the message in our analysis. Specifically, we compute the potential types from the points-to set of the message and analyze only the branches of the top *if* statement that may be taken under these message types. Our second optimization is based on the idea that when a message must be of a certain type under some context, we can safely remove objects that are not an instance of this type from the points-to set of this message. The optimization works as follows: after entering a branch of the top *if* statement, we carry the corresponding type constraint of the message (obtained from the condition of the *if* statement) with our analysis. That is, whenever we query the points-to set of the message in this branch, an additional filter function $f : \mathcal{P}(Obj) \times \mathsf{ClassName} \rightarrow \mathcal{P}(Obj)$ is applied to the original points-to set to filter out objects that are not an instance of the given type. Our evaluation shows that these optimizations significantly reduce the size of the MFGs.

Example. Let us illustrate the optimizations using the `Client` actor in Figure 1. Suppose that the points-to set of the `message` parameter in the `onReceive` method contains only one `DepositMessage` message. Based on the first optimization, we only need to analyze the second branch of the *if* statement (Lines 20 - 22) instead of the whole method. To illustrate our second optimization, we now suppose that the points-to set of the `message` parameter contains a `WithdrawMessage` message and a `DepositMessage` message. Then both branches of the *if* statement must be analyzed. When analyzing its first branch (Lines 14-18), we know that the `message` parameter must be of the type `WithdrawMessage`. With this type constraint, we can remove the `DepositMessage` message from the points-to set in this branch because it is not an instance of the type `WithdrawMessage`. Hence, we can conclude that at Line 17, `message` must point to a `WithdrawMessage` message rather than may point to a `WithdrawMessage` message or a `DepositMessage` message. Similarly, the optimization can be applied to the second branch as well.

5 Test Generation

In phase II, we use backward symbolic execution to generate tests for the target. BSE starts from the target, and performs a backward exploration, searching for a feasible path to the entry points of the system. Constraints over the execution are collected and used to generate

$$\begin{aligned}
\alpha \in \mathit{ActorMap} &= \mathit{ActorRef} \rightarrow \mathit{ActorState} \\
\mathit{Event} &= \mathit{SendingEvent} \cup \mathit{CreationEvent} \\
\mathit{SendingEvent} &= \widehat{\mathit{ActorRef}} \times \widehat{\mathit{ActorRef}} \times \widehat{\mathit{Var}} \times \widehat{\mathit{Time}} \\
\mathit{CreationEvent} &= \widehat{\mathit{ActorRef}} \times \mathsf{ClassName} \times (\widehat{\mathit{Var}})^* \times \widehat{\mathit{Time}} \\
\varsigma \in \mathit{ActorState} &= \mathit{LocalState} \times \widehat{\mathit{Time}} \times \mathit{Requests} \\
\beta \in \mathit{LocalState} &= \mathsf{Stmt} \times \mathit{CallStack} \times \widehat{\mathit{Var}} \\
cs \in \mathit{CallStack} &= (\mathsf{Stmt} \times \widehat{\mathit{Var}} \times \widehat{\mathit{Var}})^* \\
Q \in \mathit{Requests} &= \widehat{\mathit{Var}} \times \widehat{\mathit{ActorRef}} \times \widehat{\mathit{Time}}
\end{aligned}$$

$\widehat{\mathit{Var}}, \widehat{\mathit{ActorRef}}, \widehat{\mathit{Time}}$ are sets of free variables in first order logic.

Figure 9 State space of the backward symbolic execution.

the test. The generated test consists of the messages sent to relevant actors as well as the message receiving orders.

The semantics of BSE is formally defined as a transition relation $\Rightarrow_S$ from one symbolic configuration to another symbolic configuration. A symbolic configuration is a tuple,

$$\langle \alpha \mid \mu \mid \phi \mid \chi \rangle$$

where α represents relevant actors in BSE and is a map from a finite set of actor references to actor states, μ is a finite set of pending events (including both actor creation and message sending events). ϕ is the path condition collected over the transitions, and χ is the set of external messages to the system. The domain of ϕ is the quantifier-free formulae in *first-order logic* (FOL) with equality. The domain of the remaining configuration is described in Figure 9. Note that $\widehat{\mathit{Var}}, \widehat{\mathit{ActorRef}}, \widehat{\mathit{Time}}$ are sets of free variables in FOL, which can hold values of primitives and references. A message-sending event consists of the actor reference of the sender, the actor reference of the recipient, the message, and the time when the message is sent. An actor-creation event consists of the actor reference of the actor being created, the type of the actor, and a list of constructor parameters, and the creation time. An actor state consists of a local state, the current local time of the actor, and a set of message requests.

Since BSE goes backwards, a message request under this context indicates that a certain message is required in order for the execution to reach this point, yet this message is not in the mailbox of that actor. For each message request, BSE attempts to find an actor that can send the corresponding message, and thus "fulfill" this request. The local state consists of the current statement, the call stack, and a variable representing the receiver object of the current method call. A message request consists of a message, an actor reference for the sender, and the time of receiving the message. The call stack consists of a list of call frames, and each call frame consists of the statement to return to, the variable of the return value, and the variable of the caller object. $\widehat{\mathit{Time}}$ is a set of integer variables.

To describe the BSE semantics, we add two additional types of statements to our language as indicators of reaching the entry of a method. We use $\texttt{entry}_R\texttt{;}$ as the first statement for every `onReceive` method, and use `entry;` as the first statement for all other methods. For space considerations, the formal semantics of local computations in BSE is described in the appendix.

Actor Creation

$$\langle \alpha \bullet r \mapsto ((\llbracket v = \texttt{create}(C\texttt{.class},\ \overrightarrow{v'})\texttt{;}^{\ell}\rrbracket, \_), \hat{t}, \_) \mid \mu \mid \phi \mid \_ \rangle \Rightarrow_S$$
$$\langle \alpha \bullet r \mapsto ((\mathit{pred}(\ell), \_), \hat{t'}, \_) \mid \mu' \mid \phi' \mid \_ \rangle, \text{ where}$$
$$\hat{t'}, \hat{r'} \text{ are fresh} \qquad \phi' = \phi[\hat{r'}/\hat{v}] \wedge \hat{t'} < \hat{t} \qquad \mu' = \mu \cup \{(\hat{r'}, C, \overrightarrow{\hat{v'}}, \hat{t})\}$$

Message Sending

$$\langle \alpha \bullet r \mapsto ((\llbracket v\texttt{.send}(v')\texttt{;}^{\ell}\rrbracket, \_), \hat{t}, \_) \mid \mu \mid \phi \mid \_ \rangle \Rightarrow_S$$
$$\langle \alpha \bullet r \mapsto ((\mathit{pred}(\ell), \_), \hat{t'}, \_) \mid \mu' \mid \phi' \mid \_ \rangle, \text{ where}$$
$$\hat{t'} \text{ is fresh} \qquad \phi' = \phi \wedge \hat{t'} < \hat{t} \qquad \mu' = \mu \cup \{(\hat{r}, \hat{v}, \hat{v'}, \hat{t})\}$$

Actor Entry-Existing Actor

$$\langle \alpha \bullet r \mapsto ((\llbracket \texttt{entry}_R\texttt{;}^{\ell}\rrbracket, \_), \hat{t}, Q) \mid \_ \mid \phi \mid \_ \rangle \Rightarrow_S$$
$$\langle \alpha \bullet r \mapsto ((\mathit{nil}, \_), \hat{t'}, Q') \mid \_ \mid \phi' \mid \_ \rangle, \text{ where}$$
$$\hat{t'} \text{ is fresh} \qquad \phi' = \phi \wedge \hat{t'} < \hat{t} \qquad \llbracket \texttt{void onReceive}(C'v')\{\overrightarrow{C''v''};\ \overrightarrow{s}\}\rrbracket = \mathit{method}(\ell)$$
$$Q' = Q \cup \{(\hat{v'}, \hat{r_{sender}}, \hat{t})\}$$

Actor Entry-New Actor

$$\langle \alpha \bullet r \mapsto ((\llbracket \texttt{entry}_R\texttt{;}^{\ell}\rrbracket, \_), \hat{t}, Q) \mid \_ \mid \phi \mid \_ \rangle \Rightarrow_S$$
$$\langle \alpha \bullet r \mapsto ((\mathit{nil}, \_), \hat{t'}, Q') \bullet r' \mapsto ((\mathit{nil}, [\,], \hat{v'_0}), \hat{t''}, [\,]) \mid \_ \mid \phi' \mid \_ \rangle, \text{ where}$$
$$\hat{t'}, \hat{t''}, \hat{r'}, \hat{v'_0} \text{ are fresh is fresh} \qquad \llbracket \texttt{void onReceive}(C'v')\{\overrightarrow{C''v''};\ \overrightarrow{s}\}\rrbracket = \mathit{method}(\ell)$$
$$C \in \mathit{predCls}(AC(r)) \qquad Q' = Q \cup \{(\hat{v'}, \hat{r_{sender}}, \hat{t})\} \qquad \phi' = \phi \wedge \hat{t'} < \hat{t} \wedge \hat{t''} < \hat{t}$$

Messaging Event Matching-Internal

$$\langle \alpha \bullet r \mapsto (\_, Q \bullet (\hat{v}, \hat{r_{sender}}, \hat{t})) \mid \mu \bullet (\hat{r'}, \hat{r''}, \hat{v'}, \hat{t'}) \mid \phi \mid \chi \rangle \Rightarrow_S$$
$$\langle \alpha \bullet r \mapsto (\_, Q) \mid \mu \mid \phi' \mid \chi \rangle, \text{where}$$
$$\phi' = \phi \wedge \hat{r} == \hat{r''} \wedge \hat{v} == \hat{v'} \wedge \hat{r_{sender}} == \hat{r'} \wedge \hat{t'} < \hat{t}$$

Messaging Event Matching-External

$$\langle \alpha \bullet r \mapsto (\_, Q \bullet (\hat{v}, \hat{r_{sender}}, \hat{t})) \mid \_ \mid \chi \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto (\_, Q) \mid \_ \mid \chi \cup \{(\hat{r}, \hat{v})\} \rangle$$

Creation Event Matching

$$\langle \alpha \bullet r \mapsto ((\texttt{nil}, \_, \hat{v_0}), \hat{t}, [\,]) \mid \mu \bullet (r', AC(r), \overrightarrow{\hat{v}}, \hat{t'}) \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \mid \mu \mid \phi' \mid \_ \rangle, \text{where}$$
$$\overrightarrow{f} = \mathcal{F}(AC(r)) \qquad \phi' = \phi \wedge \hat{r} == \hat{r'} \wedge \mathit{read}(\hat{v_0}, f_i) == \hat{v_i} \wedge \hat{t'} < \hat{t}$$

OnReceive Looping

$$\langle \alpha \bullet r \mapsto ((\texttt{nil}, \_), \_) \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((\mathit{last}(\overrightarrow{s}), \_), \_) \mid \_ \rangle, \text{where}$$
$$\llbracket \texttt{void onReceive}(C'v')\{\overrightarrow{C''v''};\ \overrightarrow{s}\}\rrbracket = \mathit{rec}(AC(r))$$

Self Reference

$$\langle \alpha \bullet r \mapsto ((\llbracket v = \texttt{self;}^{\ell}\rrbracket, \_), \_) \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((\mathit{pred}(\ell), \_), \_) \mid \phi[\hat{r}/\hat{v}] \mid \_ \rangle$$

Sender Reference

$$\langle \alpha \bullet r \mapsto ((\llbracket v = \texttt{sender;}^{\ell}\rrbracket, \_), \_) \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((\mathit{prev}(\ell), \_), \_) \mid \phi[\hat{r_{sender}}/\hat{v}] \mid \_ \rangle$$

Figure 10 Transition rules for actor operations in backward symbolic execution.

5.1 Semantics Of Actor Operations In BSE

Figures 10 shows the semantics of BSE for actor operations. We put a hat on a symbol to represent a free variable in ϕ. For example, we use $\hat{v}$ in ϕ to represent the corresponding variable v is free. Note that for variables with the same name in different execution contexts, we create distinct variables in ϕ to represent them. The notation $\phi[\hat{v'}/\hat{v}]$ means that every occurrence of $\hat{v}$ in ϕ is syntactically replaced by $\hat{v'}$. It is important to note that whenever such substitutions happen in ϕ, we also perform the corresponding substitutions in the rest of the symbolic configuration. For readability, we omit these subsequent substitutions in our transition rules. We use a number of helper functions in our transition rules. The function *pred* returns the previous statement of a given label, and the function *last* returns the last element of a given list. The function *method* returns the method that encloses the statement with the given label. The function *AC* returns the class name of the actor object mapped by the given actor reference. The function *read* takes as input a free variable representing an object and the field name, and returns the variable representing the field. The function *predCls* takes as input a class name, locates the node of this class in the MFG, finds the predecessors of the node, and returns a set of class name of the predecessors.

The *Actor Creation* rule and the *Message Sending* rule say that upon an actor-creation or message-sending operation, an actor-creation or a message-sending event is added to a pool of pending events μ. Every actor keeps a local time $\hat{t}$, and increases its local time when an actor operation is performed. Hence, the constraint $\hat{t'} < \hat{t}$ indicates that the operation at $\hat{t'}$ *happens before* the operation at t. The *Actor Entry* rules describe potential transitions when BSE reaches the entry of the `onReceive` method of an actor. Reaching the entry of the *onReceive* method implies that this actor must have been created and have received a message. Thus, in both *Actor Entry* rules, a corresponding message request is added to the set Q, indicating that the specific message is required in order for the execution to reach this point, and BSE needs to find an actor that sends the message. There are two possibilities concerning who may create this actor or send a message to this actor. The *Actor Entry-Existing Actor* describes one possibility that this actor is created by an existing actor in α, and the message is also sent from an existing actor; there is no need to introduce new actors in α. The *Actor Entry-New Actor* describes the other possibility: either the actor creation or the message send is done by actors not in α. As a result, a new actor is added to α. The MFG is queried to obtain the predecessors of this actor class, which is the set of actor classes that may create or send a message to this actor. Then an actor with the default initial state is created in α with its type being one of the predecessors. This is the only rule that introduces new actors to our exploration.

A message request is fulfilled either by a pending message event in μ sent from an actor inside the system or, if the actor is a receptionist, by a message sent from the external environment. The *Messaging Event Matching-Internal* rule describes the first case, in which the matched request and event are remove from Q and μ respectively, and a happens-before constraint between the message receive and send operations is added to ϕ. The *Messaging Event Matching-External* rule describes the second case, in which the request is removed from Q, and an external message is added to χ. The *Creation Event Matching* rule says that a pending actor-creation event is matched with an actor in α. Note that to match a creation event, the type of the actor must be the same as the type specified in the creation event, and the message request set Q of the actor must be empty, indicating all message requests are fulfilled. The *Receive Looping* rule says that an idle actor can start an execution from the exit of the *onReceive* method.

5.2 Path Exploration In BSE

The initial symbolic configuration is that the actor map α contains only one actor with the statement being the target, and the event pool μ contains the actor-creation events from the external environment. BSE starts with the initial configuration and takes one transition at a step. The computation branches when multiple transition rules can be matched on one configuration. BSE uses the depth-first search strategy for path exploration. At each branching point, we pick one transition from all enabled transitions, and check if the path constraints in the new configuration is satisfiable. If satisfiable, we continue the exploration on the new configuration; otherwise, we backtrack. The final accepting configurations are the ones with α being empty and ϕ being satisfiable. A system test can be constructed from the model of ϕ, the transition path, and the set of external messages.

Because actors in the configuration proceed their computations concurrently, almost any configuration has multiple enabled transitions. As a result, the search space in BSE is intractable. To address this problem, we propose two search heuristics and a feedback-directed search technique to efficiently find a feasible path in the huge search space.

Search heuristics. Our first heuristic is that BSE always explores a message handler atomically. In other words, once BSE starts a transition of local computations in a message handler of an actor, all transitions on other actors are disabled and BSE will keep exploring this message handler until reaching the entry of the message handler. As a result, the number of enabled transitions on each symbolic configuration is reduced. This heuristic leverages the atomicity of the macro-step semantics [8] in the Actor model–messages to a given actor are processed one at a time without interleaving. Macro-step is also enabled by the fact that the concurrent execution of message handlers on different actors need not be interleaved (i.e., messages to different actors can be sequentialized). This is because actors do not share states. Therefore, the heuristic is safe: it reduces the search space in BSE without missing any tests that can potentially cover the target.

Our second heuristic keeps the number of actors in the generated test small in order to avoid exploring unnecessary paths. This heuristic is based on the conjecture that most concurrency bugs may be triggered by considering interactions of a small number of actors. The conjecture is the result of a previous finding that most concurrency bugs in multi-threaded programs can be triggered using two threads [23]. With this conjecture, we assign different weights to transition rules for actor operations. When multiple transition rules are enabled on a configuration, the probability of picking a rule is based on its weight (rules with more weights have a higher chance of being picked). We give a much lower weight to the $\mathit{ActorEntry} - \mathit{NewActor}$ rule, which is the only rule to introduce new actors to a test. This is because introducing a new actor opens up a whole new search space – BSE has to find a feasible execution trace on this actor. In this way, we keep the number of actors in our test small, and avoid fruitless explorations. In addition, we give more weights to transition rules that consume pending events in the event pool μ so that message requests from actors can be fulfilled as soon as possible. Recall that a test is generated only when BSE reaches a final accepting configuration, where the actor map α must be empty. An actor is removed from α only when all of its message requests are fulfilled. Hence, fulfilling these message requests helps BSE find a test efficiently.

Feedback-directed search. Heuristics do not always work well. There are cases where a large number of transitions are enabled, but only a few of them can lead to a feasible path. If the heuristics do not bias towards these transitions, BSE will frequently hit infeasible paths.

```
 1  private int pingsLeft = 100;
 2  public void onReceive (Object message) {
 3     if(message instanceof PongMessage) {
 4        pongActor.tell(new PingMessage(), getSelf());
 5        pingsLeft --;
 6        if(pingsLeft == 0) {
 7           \\ target
 8        } ...
 9     } ...
10  }
```

Figure 11 An example from our subjects that illustrates the feedback-directed search technique.

The feedback-directed search technique guides BSE out from such undesirable situations by leveraging the unsatisfiable cores of the path constraint from the previous infeasible paths. An unsatisfiable core is a subset of clauses in the original constraint such that the conjunction of these clauses is unsatisfiable. To make the path constraint feasible, the clauses in the unsatisfiable core need to be changed. The idea of our feedback-directed technique is to drive the execution towards the code that changes the values of the variables in the unsatisfiable core, hoping that the changes will make the path constraint satisfiable.

Our feedback-directed technique has two steps. In the first step, we identify a set of code instructions that can potentially change the unsatisfiable core. We obtain the unsatisfiable core of the path constraint directly from the underlying SMT solver Z3 [14]. Then we extract all the variables from the unsatisfiable core, and map these variables to corresponding program variables. This can be done without additional overhead, because our symbolic execution keeps track of the mapping between the variables in path constraints and program variables. For each program variable, we identify a set of instructions that define this variable (definition sites). In our implementation, BSE is performed on an IR that is in the static-single-assignment form. Hence, there is only one definition site per variable. In the second step, we drive the execution to the definition sites identified in the first step. To do this, we compute the transitions that may lead to at least one of these definition sites. A transition may lead to a definition site if the statement transited to is reachable from the definition site in the inter-procedural control flow graph. We prioritize these transitions over the others.

Figure 11 shows the message handler of the ping actor in the Ping-Pong example. The `pingsLeft` field keeps track of the ping messages sent out, and is initially set to 100. To cover the target at Line 7, the ping actor has to receive 100 pong messages. Suppose that when BSE first reaches the entry of the message handler from the target, it chooses to jump to the constructor of the ping actor, meaning that only one pong message is received after creating this actor. Obviously, this path is infeasible. Its path constraint is $p = 100 \wedge p - 1 = 0 \wedge subType(type(m), PongMessage)$, where p and m map to the program variables `pingsLeft` and `message`, respectively. The unsatisfiable core of this path constraint is $\{p = 100, p - 1 = 0\}$, whose only variable maps to `pingsLeft`. Thus, Line 1 and Line 5 are identified as the definition sites for `pingsLeft`. Then BSE backtracks to the entry of the message handler, and picks the transition that jumps to Line 8, because it may lead to the definition site at Line 5. This transition indicates that the ping actor has received two pong messages. Note that BSE does not pick the transition that may lead to Line 1 (i.e., the transition that jumps to the constructor), because it has been explored previously, leading to an infeasible path. This process iterates 100 times and BSE finds a feasible path in which the ping actor receives 100 pong messages. Without this technique, in each iteration, BSE may try other messages that do not affect `pingsLeft`, thus making the search inefficient.

Table 1 Characteristics of the subjects in our evaluations.

Subjects	**LOC**	**Description**
Micro Bench.	50 - 200	8 well-known actor example programs
Concurrency Bench.	100 - 400	8 classic concurrency problems
Parallelism Bench.	200 - 1,000	14 realistic parallel applications
AkkaCrawler	715	A web crawler and indexer
Batch	1,309	A concurrent batch processing framework
Parallec	12,457	A parallel client firing requests and aggregating responses
Stone	20,935	An online game server framework

6 Implementation

We implement our method in a tool called TAP for actor systems developed with Java *Akka*. TAP is built on top of WALA [5], a static analysis infrastructure for Java. TAP transforms the Java bytecode of the system under test to WALA IR and performs analysis on WALA IR. The benefit of working on WALA IR is that one can directly use the basic built-in analyses provided by WALA. TAP uses multiple WALA built-in analyses such as class hierarchy analysis, call graph analysis, and points-to analysis. Since Scala *Akka* programs are also compiled to Java bytecode, TAP in principle may be used to analyze Scala *Akka* programs as well. However, Scala *Akka* has a different set of interfaces, and substantial engineering work is required to support Scala *Akka*. We plan to support Scala *Akka* in the future.

TAP consists of two major components, an MFG builder containing ∼4,000 lines of Java code and a BSE engine containing ∼11,000 lines of Java code. The implementation of the MFG builder closely follows the formalizations and the iterative MFG construction algorithm described in Section 4. A key part for MFG construction is resolving recipients and messages in message-sending sites. TAP maintains a map from an actor reference to a set of actor objects that are possibly referenced by it. This map is used to resolve `ActorRef` pointers. TAP queries WALA's points-to analysis to resolve all other pointers.

The BSE engine includes a backward symbolic interpreter on WALA IR as well as the search techniques. The interpreter implements a transition rule (similar to the semantic rules in our BSE formalization) for each type of statements in WALA IR. The actor library calls are interpreted using our semantic models so that TAP does not explore the actor library methods. The BSE engine forks a new symbolic configuration whenever the computation branches. TAP uses Z3 [14] as the off-the-shelf SMT solver for solving path constraints and computing unsatisfiable cores. An important deviation from the formalizations is that TAP implements the FIFO message delivery semantics, because our target actor framework, Java *Akka* guarantees the FIFO semantics. To implement the FIFO semantics, TAP models the pending messages as a set of lists rather than a multi-set. Each list models a FIFO communication channel between a pair of actors so that the message sending order is preserved.

7 Evaluation

We evaluate TAP on a set of third-party benchmarks called *Savina* [21] as well as four randomly selected open-source projects from GitHub. Our experiments consist of two parts: 1) the evaluation on the MFG construction analysis, measuring the size of the MFGs, analysis time, and the effectiveness of the optimizations; 2) the evaluation on the effectiveness of our test generation method.

Table 2 Comparison between the baseline MFG analysis and the optimized MFG analysis. The numbers for the three benchmark categories are averages.

Subjects	**Baseline Analysis**				**Optimized Analysis**			
	# Nodes	# Edges	# Labels	Time (s)	# Nodes	# Edges	# Labels	Time (s)
Micro	2.5	4.3	6.5	45	2.5	4.3	6.2	45
Concurrency	3.8	10.4	16.5	56	3.8	9.3	14.4	59
Parallelism	4.5	17.5	24.9	79	4.5	**15.8**	**19.2**	**72**
AkkaCrawler	3	6	15	57	3	6	12	55
Batch	5	12	31	85	5	**10**	**21**	**77**
Parallec	8	16	67	190	8	**13**	**46**	**131**
Stone	38	74	173	243	38	**58**	**121**	**169**

Table 1 describes the subjects used in our evaluation. The *Savina* benchmarks consist of 30 diverse programs written purely using actors. *Savina* has three categories: micro benchmarks with 8 well-known actor examples, concurrency benchmarks with 8 classic concurrency problems, and parallel benchmarks with 14 realistic parallel applications. *Savina* has been used in the actor community for various evaluation purposes, such as performance comparison of actor languages/frameworks [21, 12], actor profiling [31], and mapping from message passing concurrency to threads [39]. The original *Savina* does not have a Java *Akka* implementation. We transformed the Scala *Akka* implementation in *Savina* into Java *Akka* and used the transformed version in our experiment because TAP currently supports only Java *Akka*. We had at least two actor programmers double check that the transformed Java version is equivalent to the Scala version.

All four open source projects are written in Java using the Java *Akka* library. Most of their application logic is implemented in actors. AkkaCrawler is a parallel web crawler and indexer. Batch is a framework for concurrent batch processing. Parallec is a scalable asynchronous client, developed by eBay, for firing large numbers of HTTP/SSH/TCP/UDP requests and aggregating responses in parallel. Stone is a framework for developing online game servers. From all the actor-based Java *Akka* projects that we can find on Github, Parallec and Stone are among the largest projects. Some projects mix the Actor model with other concurrency models [36]. We exclude those projects from our evaluation because TAP does not handle other concurrency models such as threads. All our experiments ran on a quad-core machine with 16 GB of RAM, running a 64-bit Ubuntu 14 system.

7.1 Results on MFG Construction

To demonstrate the effectiveness of the optimizations described in Section 4.3, we compare the optimized MFG analysis to the one without optimizations in terms of the size of the MFGs and the time taken for MFG construction. We measure the size of an MFG using the number of nodes, the number of edges and the number of labels on all edges. Overall, 92% of the `onReceive` methods in our subjects match the code pattern for optimizations (i.e., the message handler has a top-level *if* statement that checks for the message type).

MFG Size. Table 2 shows the comparison results. The numbers for the three benchmark categories are averages because there are multiple projects in each category. On average, the optimized analysis reduces the number of edges by 11% and the number of labels by 23%. The number of nodes is not reduced because our analysis creates only one node per actor class. Recall that our optimizations are safe, indicating that all the reduced edges and labels are false positives. The results show that our optimizations substantially improve the precision of MFG analysis.

The results also show that the optimized analysis reduces a far larger percentage of edges and labels on larger projects. Table 2 highlights (in bold) cases where our optimizations significantly reduces the size of MFGs. For instance, the optimized analysis reduces edges by 19% and labels by 31% for the Parallec project, and reduces edges by 22% and labels by 30% for the Stone project. However, on small subjects such as the micro benchmarks, our optimizations do not produce a significant difference. The reason is that the computed points-to sets in larger projects are typically larger than those computed in smaller projects. Our optimizations often reduces the points-to set to only one element or a much smaller subset in a top-level branch. Therefore, the larger the points-to sets are, the more false positives are reduced. In summary, the optimized analysis has a bigger impact on larger projects.

Analysis Time. We ran the same experiment five times to obtain the average time taken by each analysis on each subject. An interesting observation is that the optimized analysis takes much less time than the baseline analysis does in projects where the optimized analysis reduces the MFG size significantly. For instance, on both Parallec and Stone projects, the analysis time drops about 30% with the optimizations. In other words, the optimized analysis produces more precise results with less time. Our investigation indicates that with smaller points-to sets, the iterative MFG construction algorithm reaches the fixed point faster: having larger points-to sets implies more candidate actors or messages, and this often leads to more iterations for the algorithm to converge. The overhead of our optimizations is negligible, because the optimized analysis performs only a simple structural check on the control flow graph of the `onReceive` method. As shown in the results, the two analyses take similar time on small projects such as the micro benchmarks and the AkkaCrawler project.

7.2 Results on Test Generation

To evaluate the effectiveness of our test generation method, we randomly selected basic blocks in actor classes as targets from all subjects, and for each target, we applied TAP to generate tests to cover it. To avoid biases, we evenly distributed the targets based on the size of actor classes in each project. In practice, the targets may be software patches [25], assertions, and suspicious code locations. In total, we selected 500 targets for the *Savina* benchmarks and 500 targets for the four open source projects. The effectiveness of our method is measured by the percentage of targets covered. A target is covered only when TAP finds a feasible path to the target within the given timeout.

Our problem settings require the specification of receptionists for each actor system. Unfortunately, such information is not specified in our subjects. Therefore, we manually inferred receptionists for each project from its drivers and tests. We set a timeout of ten minutes per target excluding the time for MFG construction. To compare our search techniques, we ran TAP using the following five settings: 1) **Random**, pick a transition randomly from all matched rules on a symbolic configuration; 2) **H1**, enable only the first heuristic; 3) **H2**, enable only the second heuristic; 4) **H1 + H2**, enable both heuristics; 5) **H + F**, enable both heuristics and the feedback-directed technique. All five settings used the depth-first search strategy.

Table 3 The target coverage results of running TAP with five settings.

Subjects	Targets	Random	H1	H2	H1 + H2	H + F
		# Cov (%)	# Cov (%)	# Cov (%)	# Cov (%)	# Cov (%)
Micro	97	52 (54%)	55 (57%)	59 (61%)	76 (78%)	82 (85%)
Concurrency	162	73 (45%)	91 (56%)	79 (49%)	114 (70%)	124 (77%)
Parallelism	241	86 (36%)	103 (43%)	143 (59%)	161 (67%)	173 (72%)
AkkaCrawler	39	21 (54%)	25 (64%)	28 (72%)	34 (87%)	35 (90%)
Batch	60	38 (63%)	43 (72%)	42 (70%)	51 (85%)	55 (92%)
Parallec	178	75 (42%)	81 (46%)	86 (48%)	91 (51%)	139 (78%)
Stone	223	64 (29%)	96 (43%)	107 (48%)	124 (56%)	167 (75%)
Total	1000	409 (41%)	494 (49%)	544 (54%)	651 (65%)	775 (78%)
Avg. time per target (s)		258	217	176	124	91

7.2.1 Target Coverage

Table 3 summarizes the results of running TAP with the five settings. Column 2 shows the number of targets selected for each subject. Columns 3-7 show the number and the percentage of the targets covered by the five settings, respectively. The last row shows the average time (in seconds) taken for covering a target in each setting excluding the time for MFG construction. Overall, the combination of heuristics and feedback-directed technique is effective in covering targets. Search heuristics increase the target coverage from 41% to 65%. The feedback-directed technique further increases the target coverage to 78%.

The Random setting does not work well. It times out in 228 out of 1000 cases. The major problem with Random is that it often introduces many unnecessary actors to path exploration. Introducing a new actor in a test is an expensive operation, because it opens up additional search space for TAP to find a feasible execution trace on the new actor. As a result, Random wastes lots of resources exploring traces for unnecessary actors, and takes longer time to cover a target. In additional, the tests generated by Random are typically larger in terms of the number of actors. The H1 setting suffers the same problem. However, it reduces the search space by sequentializing the execution of message handlers. As a result, the number of enabled transitions on each symbolic configuration in H1 is much smaller than that in Random. Due to the space reduction, H1 improves the target coverage to 49%.

The H2 setting improves Random by keeping the tests as small as possible to avoid exploring unnecessary space. Our experiment results show that in many cases, the target can be reached with no more than three actors. For example, many subjects use the master-worker pattern to implement parallelism. The workers proceed in parallel, and do not interact with each other. In such cases, it suffices to cover any target in the worker with only two actors: one master and one worker. Creating new workers only adds complexity to the problem. H2 is very efficient in covering such targets because it assigns a very low weight to transitions that introduce new actors.

The feedback-directed technique is particularly useful when our heuristics do not work well and BSE frequently hits infeasible paths. In our experiment, we find that there are a number of cases where covering the target requires creating multiple actors of the same class (e.g., comparing the IDs of actors). In these cases, the heuristics work poorly because they prefer to reuse the existing actor rather than create a new actor of the same class. As a result, the heuristics keep hitting infeasible paths in these cases. The feedback-directed technique is quite effective in guiding BSE to find a feasible path. For instance, in the case of checking for different IDs, it directly identifies that the ID field of the actor needs to be

```
 1  public void onReceive (Object message) {
 2      if (message instanceof TokenMessage) {
 3          TokenMessage token = (TokenMessage)message;
 4          if(token.hasNext()) {
 5              // bug: potential null de-reference on nextActor
 6              this.nextActor.tell(token.next(), getSelf());
 7          } ...
 8      } else if (message instanceof DataMessage) {
 9          this.nextActor = (ActorRef) ((DataMessage) message).data;
10      } ...
11  }
```

Figure 12 A bug caused by out-of-order message delivery in the ThreadRing benchmark.

changed, because the unsatisfiable core contains variables that map to this field. Since the only way to change the ID field of the actor is through its constructor, the feedback-directed technique prioritizes the transitions that introduce new actors to be explored first, and thus quickly finds a feasible path.

We analyze the cases in which TAP fails to cover the targets in the H + F setting. More than half of the cases are due to a lack of environment modeling (e.g., access to database and network). Such issues can be mitigated by adding models for calls to the environment. The rest of the cases are mainly due to timeouts for the exploration and complex constraints that Z3 fails to solve.

7.2.2 Bug Detection

By running TAP to cover these 1,000 targets, we are able to find six distinct bugs in our subjects. All six bugs are found in the *Savina* benchmarks in three projects. Five out of the six bugs are crash bugs. One bug is less critical: a non-crash warning from *Akka* regarding messages sent to actors that have been killed. We have confirmed that all bugs are triggered in both the original benchmarks and the transformed versions with our generated tests. We diagnose the six bugs and find that all five crash bugs are caused by out-of-order message delivery. Such bugs are hard to reveal locally because out-of-order message delivery is unlikely to happen locally. The other bug is caused by sending two stop messages to kill an actor, and the recipient actor kills itself after receiving the first stop message.

Figure 12 shows one crash bug found in the ThreadRing benchmark. There is a potential null de-reference on the `nextActor` field at Line 6. The ThreadRing system starts with a coordinator sending a DataMessage to each token passer to inform them the next passer and form a ring among them. The coordinator then sends a token to one passer in the ring, and then the token is passed from one passer to another in the ring. The passer sets its `nextActor` field at Line 9 upon receiving a `DataMessage` and sends the token at Line 6 upon receiving a `TokenMessage`. The assumption is that every passer must set the `nextActor` before sending the token (i.e., receive the `DataMessage` before the `TokenMessage`). Since the *Akka* framework guarantees FIFO message delivery, this assumption holds for the first passer. However, the assumption may not hold for the other passers. It is possible that the second passer receives the `TokenMessage` from the first passer before receiving the `DataMessage` from the coordinator. Although the `DataMessage` is sent before the `TokenMessage`, the two messages are sent by different senders, and may be delivered out of order. In this case, a null pointer exception is thrown in the second passer. TAP found this bug because the exceptional branch of Line 6 (WALA IR contains exceptional branches for potential null dereferences) happened to be chosen as a target. A simple fix to this bug is adding a null check on `nextActor` before passing the token.

8 Related Work

Testing Actors. The most related work on testing Actor systems is dCUTE [33]. dCUTE differs from TAP in three aspects. First, dCUTE's goal is to achieve overall coverage while TAP aims at covering target code locations. They can be used to complement each other. Second, dCUTE performs *forward* concolic execution while TAP does *backwards* symbolic execution without a side-by-side concrete execution. Lastly, dCUTE handles only a subset of actor operations. For example, it assumes that all actors have been created before execution, and thus does not handle dynamic actor creation. However, we provide a rigorous definition of the semantics of all actor operations in BSE.

BASSET [22] leverages a model checker to systematically explore message schedules in an actor system. BASSET assumes that input messages are given, and aims at exploring as many message schedules as possible on the given input. It uses state merging and dynamic partial order reduction (DPOR) to reduce the search space of message schedules. BITA [38] also explores possible message schedules for given input messages. It defines new *schedule coverage* criteria, and uses these criteria to guide the exploration to expose bugs. TRANSDPOR [37] proposes another DPOR technique that exploits the transitivity of the dependency relations between actors for schedule space reduction. TAP not only explores message schedules, but also generates message contents. These exploration techniques and space-reduction techniques can be integrated into TAP for more efficient test generation.

Targeted Test Generation. A number of targeted test generation techniques have been developed on sequential programs using both forward symbolic execution [24, 18, 16, 25, 10] and backward symbolic execution [15, 29, 13]. However, they cannot be directly applied to actor systems. Since an actor library often contains complex multi-threading and networking code, direct exploration of these actor library methods is impractical and the execution often fails to go across actors. Our work fills this gap by defining formal semantic models of actor operations in our analysis, and thus preventing our analysis from exploring the actor library.

Feedback-Directed Test Generation. Previous research has proposed using information from previous executions as feedback to guide test generation. RANDOOP [30] uses execution feedback from previous tests to avoid generating redundant and illegal inputs. Garg et al. [17] use the unsatisfiable cores from previous infeasible paths to generalize the reason for the infeasibility, and thus rule out more infeasible paths. We also use the unsatisfiable cores from infeasible paths, but we use them to guide BSE to efficiently find a feasible path.

Backward Symbolic Analysis. SNUGGLEBUG [11] uses backward symbolic *analysis* for computing inter-procedural weakest preconditions. The symbolic reasoning in their work is similar to ours except that their analysis works on all possible program paths to the target while our BSE aims at finding one feasible path.

Static Analysis of Actors. There has been previous work [28] on static analysis of actor programs to infer the ownership transfer of messages. This analysis works on individual actors (i.e., intra-actor), and does not model interactions between actors. Our MFG construction is a more complex whole-system analysis that requires modeling actor interactions.

9 Conclusion

We have presented a method for targeted test generation for actor systems based on BSE. Our method first constructs an MFG to capture the potential interactions between actors. Guided by the MFG, it starts BSE directly from the target to find a feasible path to the entry point of the actor system. We have provided high-level models for all actor operations and formally defined their semantics in our analysis to avoid analyzing the complex code in the actor library. To efficiently navigate the huge search space in BSE, we have proposed two heuristics and a feedback-directed search technique. We have implemented our method in Tap, and evaluated it on *Savina* and four open source projects. The evaluation results have shown that Tap is effective in targeted test generation for actor systems.

In the future, we plan to further improve our search techniques in BSE. One direction is to reduce the state space of message schedules using partial order reduction. The happens-before relation used in previous work is fairly coarse-grained. We plan to define a finer-grained partial order relation based on program analysis to further reduce the search space. Another direction is to leverage dynamic traces from existing tests to guide our explorations.

References

1 *Erlang Introduction.* http://erlang.org/faq/introduction.html.

2 *Java Akka.* https://doc.akka.io/docs/akka/current/actors.html?language=java.

3 *Orleans.* https://dotnet.github.io/orleans/index.html.

4 *Scala Akka.* https://doc.akka.io/docs/akka/current/index-actors.html?language=scala.

5 *Wala.* http://wala.sourceforge.net.

6 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

7 Gul Agha. Concurrent Object-oriented Programming. *Commun. ACM*, 33(9):125–141, 1990.

8 Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

9 Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

10 Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2011.

11 Satish Chandra, Stephen J Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. *ACM Sigplan Notices*, 44(6):363–374, 2009.

12 Dominik Charousset, Raphael Hiesgen, and Thomas C Schmidt. Caf-the C++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents and Decentralized Control*, pages 15–28. ACM, 2014.

13 Florence Charreteur and Arnaud Gotlieb. Constraint-based test input generation for java bytecode. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 131–140. IEEE, 2010.

14 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008.

15 Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 31–36. ACM, 2014.

16 Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Guided dynamic symbolic execution using subgraph control-flow information. In *International Conference on Software Engineering and Formal Methods*, pages 76–81. Springer, 2016.

17 Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 132–141. IEEE Press, 2013.

18 Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

19 Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

20 Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

21 Shams Imam and Vivek Sarkar. Savina-an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

22 Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 468–479, Washington, DC, USA, 2009.

23 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

24 Kin-Keung Ma, Khoo Yit Phang, Jeffrey Foster, and Michael Hicks. Directed symbolic execution. *Static Analysis*, pages 95–111, 2011.

25 Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

26 Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *ACM Sigplan Notices*, volume 45, pages 305–315. ACM, 2010.

27 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

28 Stas Negara, Rajesh K Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *ACM SIGPLAN Notices*, volume 46, pages 81–90. ACM, 2011.

29 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 616–628. ACM, 2015.

30 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007.

31 Andrea Rosà, Lydia Y Chen, and Walter Binder. Profiling actor utilization and communication in Akka. In *Proceedings of the 15th International Workshop on Erlang*, pages 24–32. ACM, 2016.

32 Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 275–299, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

33 Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.

34 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005.

35 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

36 Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do scala developers mix the actor model with other concurrency models? In *European Conference on Object-Oriented Programming*, pages 302–326. Springer, 2013.

37 Samira Tasharofi, Rajesh K Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems*, pages 219–234. Springer, 2012.

38 Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 114–124. IEEE, 2013.

39 Ganesha Upadhyaya and Hridesh Rajan. Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine. *ACM SIGPLAN Notices*, 50(10):840–859, 2015.

40 Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.

A Semantics of Local Computations in MFG Analysis

Figure 13 shows the transition rules for local computations in the MFG analysis. Since local computations concern only the actor state ς in ω, we omit other states in ω in our transition rules for better readability (the other states are the same on both sides of the rules). The operator $\sqcup$ is used to merge two maps by merging the values of the same key in both maps. The *dispatch* function takes as input an object and a method name, and returns the dispatched method[3]. The transition rules describe precisely the 1-object-sensitive points-to analysis [27]. The *Object Allocation* rule says that the heap context of an object is the label of its allocation site. The *Method Invocation* rule describes the context sensitivity. The rule says that the context used for analyzing a method is the heap context of the receiver object, which is the label of its allocation site.

B Semantics Of Local Computations In BSE

Figure 14 and Figure 15 show respectively the semantics of intra-procedural BSE and the semantics of inter-procedural BSE for local computations in an actor. Since local computations concern only the local state β and the path condition ϕ, we omit other states in the symbolic configuration in our transition rules for better readability. Note that *subType* is a predicate in FOL to check the sub-type relation, and *type* and $field$ are functions in FOL.

[3] Our language does not support method overloading, and thus a method can be dispatched based on the given object and its method name

Variable Reference

$$([\![v = v'\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),\ \text{where}\quad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(st(v'))]$$

Field Reference

$$([\![v = v'.f\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),\ \text{where}$$
$$(\_, [f \mapsto a_f]) \in \sigma(st(v')) \qquad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(a_f)]$$

Object Allocation

$$([\![v = \texttt{new}\ C(\overrightarrow{v'})\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),\ \text{where}$$
$$hc = \ell \quad \overrightarrow{f} = \mathcal{F}(C) \quad a_i = (f_i, hc) \quad o = (hc, [f_i \mapsto a_i])$$
$$\sigma' = \sigma \sqcup [st(v) \mapsto \{o\}, a_i \mapsto \sigma(st(v'_i))]$$

Method Invocation

$$([\![v = v_0\,.\,m(\overrightarrow{v'})\,;^{\ell}]\!], st, \sigma, cs, c) \Rightarrow_A (s, st', \sigma', cs', c'),\ \text{where}$$
$$M = [\![C\ m(\overrightarrow{C'\ v''})\ \{\overrightarrow{C''v'''};\ \overrightarrow{s'}\}]\!] = dispatch(o_0, m) \qquad o_0 \in \sigma(st(v_0)) \qquad (hc_0, \_) = o_0$$
$$c' = hc_0 \qquad a_i = (v''_i, m, c') \qquad a'_i = (v'''_i, m, c') \qquad st' = cons([v''_i \mapsto a_i, v'''_i \mapsto a'_i],\ st)$$
$$s = car(\overrightarrow{s'}) \qquad \sigma' = \sigma \sqcup [a_i \mapsto \sigma(st(v'_i))] \qquad cs' = cons((succ(\ell), c, st(v)),\ cs)$$

Return

$$([\![\texttt{return}\ v\,;^{\ell}]\!], st, \sigma, cs, c) \Rightarrow_A (s, cdr(st), \sigma', cdr(cs), c'), \text{where}$$
$$(s, c', a_{ret}) = car(cs) \qquad \sigma' = \sigma \sqcup [a_{ret} \mapsto \sigma(st(v))]$$

Casting

$$([\![v = (C)\ v'\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),\ \text{where}\quad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(st(v'))]$$

Figure 13 Abstract semantics for local computations in MFG analysis.

We also use a number of helper functions in our transition rules. The function *pred* returns the previous statement of a given label, and the function *last* returns the last element of a given list. The function *method* returns the method that encloses the statement with the given label. The function *callee* takes as input the label of a call site s, and returns the set of all possible callees. Specifically, it retrieves the signature *sig* of the called method from s, locates the enclosing method M of s in the call graph, and returns the set of all callees of M that match *sig*. The function *callsites* returns the set of all possible call sites of a given method.

In what follows, we explain the inter-procedural rules, which are more interesting. We assume that our language uses the *call-by-value* evaluation strategy. To perform inter-procedural BSE, a context-insensitive call graph is used to guide the execution. The entry point of the call graph is the message handler of the actor. As we execute a method m backwards, there are two possible cases regarding the target: 1) the target is outside m, indicating that BSE has previously reached the call site of m and has jumped from that call site to m, and the current call stack must be not empty; 2) the target is inside m, indicating that BSE starts from m and the current call stack must be empty. The first three rules in Figure 15 apply to the first case. The *Method Invocation* rule says that upon a method invocation, BSE queries the call graph for all possible callees of the invocation, jumps to the last statement of a possible callee, and adds the constraint that every parameter must

Variable Reference

$((\llbracket v = v' ;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[\hat{v'}/\hat{v}])$

Binary Expression

$((\llbracket v = v' \texttt{ op } v'' ;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi'),$ where $\phi' = \phi[(\hat{v'} \texttt{ op } \hat{v''})/\hat{v}]$

Field Reference

$((\llbracket v = v'.f ;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[read(\hat{v'}, f)/\hat{v}])$

Field Update

$((\llbracket v.f = v' ;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[update(f, v, v')/f])$

Casting

$((\llbracket v = (C)\ v' ;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[\hat{v'}/\hat{v}] \wedge subType(type(\hat{v}), C))$

Object Allocation

$((\llbracket v = \texttt{new } C(\overrightarrow{v'}) ;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi'),$ where

$\hat{v''}$ is fresh $\quad \overrightarrow{f} = \mathcal{F}(C) \quad \phi' = \phi[\hat{v''}/\hat{v},\ update(f_i, v'', v'_i)/f_i] \wedge type(\hat{v''}) == C$

If-True

$((\llbracket \texttt{if } (e)\ \overrightarrow{s} \texttt{ else } \overrightarrow{s'} ;^{\ell} \rrbracket, \_)\ \phi) \Rightarrow_S ((last(\overrightarrow{s}), \_), \phi \wedge \hat{e})$

If-False

$((\llbracket \texttt{if } (e)\ \overrightarrow{s} \texttt{ else } \overrightarrow{s'} ;^{\ell} \rrbracket, \_)\ \phi) \Rightarrow_S ((last(\overrightarrow{s'}), \_), \phi \wedge \neg\hat{e})$

Figure 14 Transition rules for intra-procedural backward symbolic execution.

Method Invocation

$((\llbracket v = v'.m(\overrightarrow{v''}) ;^{\ell} \rrbracket, cs, \hat{v}_0), \phi) \Rightarrow_S ((s, cs', \hat{v'}), \phi'),$ where

$M = \llbracket C\ m(\overrightarrow{C'\ v'''})\ \{\overrightarrow{s'}\} \rrbracket \quad M \in callees(\ell) \quad s = last(\overrightarrow{s'})$

$cs' = cons((pred(\ell), \hat{v}, \hat{v'}),\ cs) \quad \phi' = \phi \wedge \hat{v'''_i} == \hat{v''_i}$

Return-CallStack Not Empty

$((\llbracket \texttt{return } v ;^{\ell} \rrbracket, cs, \_), \phi) \Rightarrow_S ((pred(\ell), cs, \_), \phi[\hat{v}/\hat{v'}]),$ where $(\_, \hat{v'}, \_) = car(cs)$

Method Entry-CallStack Not Empty

$((\llbracket \texttt{entry} ;^{\ell} \rrbracket, cs, \hat{v}_0), \phi) \Rightarrow_S (s, cdr(cs), \hat{v'_0}), \phi),$ where $(s, \_, \hat{v'_0}) = car(cs)$

Return-CallStack Empty

$((\llbracket \texttt{return } v ;^{\ell} \rrbracket, [\,], \_), \phi) \Rightarrow_S ((pred(\ell), [\,], \_), \phi)$

Method Entry-CallStack Empty

$((\llbracket \texttt{entry} ;^{\ell} \rrbracket, [\,], \hat{v}_0), \phi) \Rightarrow_S ((pred(\ell'), [\,], \hat{v'}), \phi'),$ where

$s = \llbracket v = v'.m(\overrightarrow{v''}) ;^{\ell'} \rrbracket \quad \ell' \in callsites(M)$

$M = \llbracket C\ m'(\overrightarrow{C'\ v'''})\ \{\overrightarrow{s'}\} \rrbracket = method(\ell') \quad \phi' = \phi \wedge \hat{v'''_i} == \hat{v''_i}$

Figure 15 Transition rules for inter-procedural backward symbolic execution.

be equal to its corresponding argument of the callee (call-by-value). The *Return-CallStack Not Empty* rule says that the variable to which the return value is assigned at the call site is replaced with the return value in the path constraint. The *Method Entry-CallStack Not Empty* rule says that the execution returns to the call site, and the top frame is popped from the call stack. The last two rules in Figure 15 apply to the second case. The *Return-CallStack Empty* rule does not update the path constraint, because the caller is unknown at this point, so is the variable that would hold the return value. The *Method Entry-CallStack Empty* says that BSE queries the call graph for all possible callers of the current method, jumps back to a possible call site, and adds the constraint that every argument of the callee are equal to its corresponding parameter in the call site. Note that no constraint over the variable v that holds the return value is added to the path constraint, because once the execution returns to the call site, it moves backwards and will never use the variable v. The constraints over v do not affect covering the target, and thus need not be added.