Mutation Operators for Actor Systems

Vilas Jagannath, Milos Gligoric, Steven Lauterburg, Darko Marinov, and Gul Agha

Department of Computer Science
University of Illinois
Urbana IL, 61801, USA
{*vbangal2, gliga, slauter2, marinov, agha*}*@illinois.edu*

***Abstract*—Mutation testing is a well known technique for estimating and improving the quality of test suites. Given a test suite T for a system S, mutation testing systematically creates mutants of S and executes T to measure how many mutants T detects. If T does not detect some (non-equivalent) mutants, T can be improved by adding test cases that detect those mutants. Mutants are created by applying *mutation operators*. Mutation operators are important because they define the characteristics of the system that are tested as well as the characteristics that are improved in the test suite. While mutation operators are well defined for a number of programming paradigms such as sequential or multi-threaded, to the best of our knowledge, mutation operators have not been defined for the actor programming model. In this paper, we define and classify mutation operators that can be used for mutation testing of actor programs.**

I. Introduction

As the use of multicore and networked systems becomes more prevalent, the ability to develop reliable concurrent systems has become increasingly important. The development and testing of such systems has proven to be notoriously difficult. Developing code using a multi-threaded, shared-memory model can often result in programs that contain hard to identify concurrency bugs such as atomicity violations, deadlocks, and data races.

Approaches based on message passing, such as the actor programming model [1], offer a promising alternative. Actor programs consist of concurrent objects that communicate with each other by exchanging messages. The actor model is becoming increasingly popular, with systems based on the model including programming languages and libraries such as ActorFoundry [2], [3], Axum [4], Erlang [5], Kilim [6], Scala [7], Singularity [8] and the Asynchronous Agents Framework in Microsoft Visual Studio 2010 [9].

Although the actor programming model is not subject to some of the problems that plague the shared-memory model, actor programs can still have bugs. The order in which messages are delivered among actors is a source of nondeterminism that can result in deadlocks and other forms of erroneous behavior. Also, the sequential code that constitutes an actor's behavior can have bugs. Thus, actor programs still need to be tested, which requires writing and maintaining good test suites.

Mutation testing [10]–[12] is a well known technique for assessing and improving the quality of test suites. Mutation testing has been applied to a number of programming paradigms including object-oriented [13], [14] and multithreaded code [15], [16]. However, to the best of our knowledge, mutation testing has not previously been investigated for the actor programming model. In this paper, we *introduce* and *categorize* mutation operators for actor programs, and *demonstrate* how these new operators can be used to create mutations.

The rest of this paper is organized as follows. Section II provides an overview of actor systems. Section III introduces an example program that will be used to illustrate actor constructs. We also use snippets from the example to demonstrate our mutation operators. Section IV discusses common sources of bugs in actor programs. Sections V presents our proposed mutation operators for actor systems. Section VI discusses related work, and Section VII summarizes the paper and presents our plans for future work.

II. Background

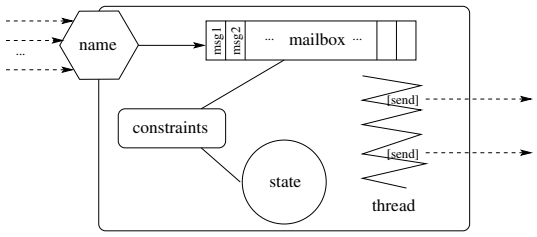

Figure 1. Schematic representation of an actor

Actors are autonomous concurrent objects with their own threads of control that communicate with each other via message passing. In response to received messages, actors can *send* messages to other actors, *create* new actors, and/or *update local state*. Each actor has its own thread of control that consists of a reactive loop that processes messages in its mailbox.

An actor is defined by specifying its local state, the set of message types that it can receive, and how it processes those messages. The definition can also contain a set of

constraints that can disable the receipt of certain messages under certain conditions. There is no shared/global state in an actor system; an actor can only access its own local state. Figure 1 illustrates the actor components discussed above. As discussed in Section I, there are many languages and libraries that support the development of actor systems. In this paper, we use the ActorFoundry [2] library for Java to illustrate our proposed mutation operators.

III. Example

We use a simple simulation of a banking system to illustrate the primitives used in actor programs to create actors and perform/regulate communication among actors. The same example is used in Section V to demonstrate the mutation operators introduced in this paper. The example is written using the ActorFoundry framework. All classes that implement actors are required to extend the `Actor` class which is provided by ActorFoundry. Actors are created using the `create` method. This method accepts the actor class and the parameters required for creation and returns the name of the created actor. Asynchronous messages between actors are sent using the `send` method that accepts the name of the destination actor, the name of the message to be sent and the parameters of the message. Messages that can be received/processed by an actor are implemented by methods annotated with the `@message` annotation. Both the `create` and `send` methods are part of the `Actor` class provided by ActorFoundry. Figure 2 shows the `BankActor` that maintains a collection of all opened `accounts` and accepts/processes messages that `openAccount` with an initial amount, `deposit` money into a given account, `withdraw` money from a given account, and `transfer` money between given accounts.

When a `deposit`, `withdraw`, or `transfer` message is received by a `BankActor`, it forwards this request to the appropriate `AccountActor` using the asynchronous `send` method. Figure 3 shows the `AccountActor` class that maintains the `balance` for an account and can receive `deposit`, `withdraw`, and `transfer` messages that modify the `balance` in various ways. The `withdraw` and `transfer` messages are constrained using the `withdrawDisabled` and `transferDisabled` constraints that ensure that those messages are not processed without a sufficient balance. Constraints in ActorFoundry are specified using a method annotated with the `@disable` annotation. The parameter provided to this annotation is the name of the message that is constrained. The constraint method accepts the same parameters as the message being constrained and returns `true` when the message should be disabled and `false` otherwise. ActorFoundry also provides the `call` method that can be used for synchronous communication between actors as shown in the `transfer` method.

```
class BankActor extends Actor {
  Map<Integer, ActorName> accounts;
  ...
  @message
  int openAccount (String fstName, String lstName) {
    return openAccount(0, fstName, lstName);
  }
  @message
  int openAccount (double initBal, String fstName, String lstName) {
    ActorName acc = create(AccountActor.class, nextId, initBal, fstName, lstName);
    accounts.put(nextId, acc);
    return nextId++;
  }
  @message
  void deposit (int accId, double amount) {
    ActorName acc = accounts.get(accId);
    send(acc, "deposit", amount);
  }
  @message
  void withdraw (int accId, double amount) {
    ActorName acc = accounts.get(accId);
    send(acc, "withdraw", amount);
  }
  @message
  void transfer (int accIdSrc, int accIdDst, double amount) {
    ActorName accSrc = accounts.get(accIdSrc);
    ActorName accDst = accounts.get(accIdDst);
    send(accSrc, "transfer", accDst, amount);
  }
  ...
}
```

Figure 2. Bank Actor

```
class AccountActor extends Actor {
  double balance = 0;
  ...
  AccountActor(int id, int balance, String fstName, String lstName) {
    this.id = id; this.balance = balance;
    this.fstName = fstName; this.lstName = lstName;
  }
  @message
  void deposit (double amount) {
    balance += amount;
  }
  @message
  void withdraw (double amount) {
    balance -= amount;
  }
  @disable(messageName = "withdraw")
  boolean withdrawDisabled (double amount) {
    return (amount > balance);
  }
  @message
  void transfer (ActorName accDst, double amount) {
    balance -= amount;
    call(accDst, "deposit", amount);
  }
  @disable(messageName = "transfer")
  boolean transferDisabled (ActorName accDst, double amount) {
    return (amount > balance);
  }
  ...
}
```

Figure 3. Account Actor

IV. Bug Patterns for Actors

In this paper, we introduce mutation operators for actor programs. For the operators to be effective, the modifications they make should be similar to common programmer errors or errors that are commonly found in actor programs (as per the competent programmer hypothesis and the coupling effect hypothesis [10], [17]). Hence, it is important to discuss the common programmer errors in actor-based programs.

While there has been classification and documentation of common programmer errors in shared-memory multi-threaded programs [18], [19], to the best of our knowledge, there has been no such classification or documentation of common programmer errors in actor based programs or distributed message-passing programs in general. However, there is a general agreement that bugs in distributed systems are usually caused by discrepancies related to the dynamic interfaces provided for communication between the distributed components [20], [21]. Our anecdotal experience with bugs in actor programs confirms this, and we expect that it holds in general as well.

In actor-based programs, the communication interface consists of the set of messages accepted by each actor and the signatures of each of those messages. The interface is dynamic because messages can be enabled or disabled dynamically at runtime using constraints, and actors can be dynamically created and destroyed. Other dimensions of the communication interface include two different synchronization strategies for sending messages (asynchronous or synchronous) and two different message argument sending strategies (send by value or send by reference). The actor mutation operators presented in this paper are targeted towards introducing changes to the usage of different parts of this communication interface in an actor program with the intent of introducing errors or unintended behavior.

V. Actor Mutation Operators

Category	Actor Mutation Operators
Messaging	**RSR** - Remove Send/Receive
	MMP - Modify Message Parameter
	RMP - Reorder Message Parameters
	MMN - Modify Message Name
	MMR - Modify Message Recipient
	CRT - Change (message) Reference Type
	CST - Change (message) Synchronization Type
Constraint	**RC** - Remove Constraint
	MC - Modify Constraint
Creation/Deletion	**RCD** - Remove Creation/Deletion
	MCP - Modify Creation Parameter
	RCP - Reorder Creation Parameters

Figure 4. Mutation operators for actor programs

In this section we introduce, categorize and demonstrate the mutation operators for actor programs. We follow a taxonomy structure similar to what has been used previously while introducing operators for sequential code for several programming languages [22]–[24], including Java [13], [25]–[29] and multi-threaded code [15], [16]. As shown in Figure 4, we have identified three categories of operators: *Messaging*, *Constraint*, and *Creation/Deletion*. The categories are organized according to the main features of actor programs.

In the following subsections we describe the operators in more detail and provide code snippets to demonstrate how the operators can be applied. The code snippets are based on the banking actor system described in Section III.

A. Messaging Related Operators

Messaging is the core feature of actor programs, as it forms the backbone for communication between various actors. The following operators modify this communication interface in various ways.

*1) **RSR** - Remove Send/Receive:* This mutation operator mimics the omission of messages by removing sends/receives. Note that removing receives is not possible in ActorFoundry since it does not support explicit receives. However, removing receives is possible in other actor systems such as Scala [7]. The following code snippets demonstrate an application of this operator in the `deposit` method of `BankActor`.

Original Code:

```
@message
void deposit (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  send(account, "deposit", amount);
}
```

RSR Mutant:

```
@message
void deposit (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  // send removed
}
```

*2) **MMP** - Modify Message Parameter:* This mutation operator modifies the value of a message parameter to another value of the same type. The modification is based upon various sequential mutation operators like replacing a variable reference with a constant value or incrementing a variable. The following code snippets demonstrate an application of this operator in the `withdraw` method of `BankActor`.

Original Code:

```
@message
void withdraw (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  send(acc, "withdraw", amount);
}
```

MMP Mutant:

```
@message
void withdraw (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  // amount replaced with constant
  send(acc, "withdraw", 0);
}
```

*3) **RMP** - Reorder Message Parameters:* This mutation operator reorders two parameters of a message. In order to avoid introducing compiler errors, **RMP** can only be applied when both the parameters are of the same type. Also, **RMP** can only be applied on messages with two or more parameters. The following code snippets demonstrate an application of this operator in the transmission of a `transfer` message to a `BankActor`.

Original Code:

```
{ ...
  send(bank, ”transfer”, accIdSrc, accIdDst, amount);
}
```

RMP Mutant:

```
{ ...
  // accIdSrc and accIdDst reordered
  send(bank, ”transfer”, accIdDst, accIdSrc, amount);
}
```

*4) **MMN** - Modify Message Name:* This mutation operator modifies the name of a message being sent. Application of **MMN** results in the transmission of a different message. Before **MMN** is applied, analysis needs to be performed to detect whether the recipient actor can receive any other messages with the same signature as the original message. Any such messages found during the analysis can be substituted instead of the original message. The following code snippets demonstrate an application of **MMN** in the `deposit` method of `BankActor`.

Original Code:

```
@message
void deposit (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  send(account, ”deposit”, amount);
}
```

MMN Mutant:

```
@message
void deposit (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  // deposit changed to withdraw
  send(account, ”withdraw”, amount);
}
```

*5) **MMR** - Modify Message Recipient:* This mutation operator modifies the recipient of a message. Applying **MMR** results in the message being sent to a different actor. Application of **MMR** requires analysis to find other actors in scope that can receive the message being sent. Any such actor can be substituted instead of the original actor. The following code snippets demonstrate an application of this operator in the `transfer` method of `BankActor`.

Original Code:

```
@message
void transfer (int accIdSrc, int accIdDst, double amount) {
  ActorName accSrc = accounts.get(accIdSrc);
  ActorName accDst = accounts.get(accDst);
  send(accSrc, ”transfer”, accDst, amount);
}
```

MMR Mutant:

```
@message
void transfer (int accIdSrc, int accIdDst, double amount) {
  ActorName accSrc = accounts.get(accIdSrc);
  ActorName accDst = accounts.get(accDst);
  // accSrc changed to accDst
  send(accDst, ”transfer”, accDst, amount);
}
```

*6) **CRT** - Send by Ref ↔ Send by Val:* This mutation operator changes a message sent by reference to a message sent by value and vice versa. It results in the same arguments being transmitted but with a different sending strategy. An application of this operator in the `withdraw` method of `BankActor` is demonstrated in the following code snippets.

Original Code:

```
@message
void withdraw (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  send(acc, ”withdraw”, amount);
}
```

CRT Mutant:

```
@message
void withdraw (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  // send changed to sendByRef
  sendByRef(acc, ”withdraw”, amount);
}
```

*7) **CST** - Sync Send ↔ Async Send:* This mutation operator changes a synchronous send to an asynchronous send and vice versa. It results in the same message being transmitted but with a different synchronization strategy. An application of this operator in the `deposit` method of `BankActor` is demonstrated in the following code snippets.

Original Code:

```
@message
void deposit (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  send(account, ”deposit”, amount);
}
```

CST Mutant:

```
@message
void deposit (int accId, double amount) {
  ActorName acc = accounts.get(accId);
  // send changed to call
  call(account, ”deposit”, amount);
}
```

B. Constraint Related Operators

While messaging defines the communication interface between actors, constraints allow actor programs to dynamically change this interface according to the state of the system. The following operators modify the constraints defined in an actor program.

*1) **RC** - Remove Constraint:* This mutation operator mimics the omission of a message constraint by removing a message constraint. An application of this operator on the `withdrawDisabled` constraint in `AccountActor` is demonstrated in the following code snippets.

Original Code:

```
@disable(messageName = ”withdraw”)
boolean withdrawDisabled (double amount) {
  return (amount > balance);
}
```

RC Mutant:

```
// removed annotation mapping this constraint
// method to the withdraw message
boolean withdrawDisabled (double amount) {
  return (amount > balance);
}
```

*2) **MC** - Modify Constraint:* This mutation operator modifies a message constraint. The modification is based on sequential operators like changing arithmetic, boolean and relational operators. An application of this operator on the `transferDisabled` constraint in `AccountActor` is demonstrated in the following code snippets.

Original Code:

```
@disable(messageName = "transfer")
boolean transferDisabled (ActorName accDst, double amount) {
  return (amount > balance);
}
```

MC Mutant:

```
@disable(messageName = "transfer")
boolean transferDisabled (ActorName accDst, double amount) {
  // changed > to <
  return (amount < balance);
}
```

C. Creation/Deletion Related Operators

Actor creation and deletion are important features of actor programs that modify the state of the system by enabling new behavior by creating actors and disabling existing behavior by deleting actors. The following operators modify the creation and deletion of actors.

*1) **RCD** - Remove Creation/Deletion:* This mutation operator mimics the omission of creation/deletion of an actor by removing a actor creation/deletion. The following code snippets demonstrate an application of this operator in the `openAccount` method of `BankActor`.

Original Code:

```
@message
int openAccount (double initBal, String fstName, String lstName) {
  ActorName acc = create(AccountActor.class, nextId, initBal, fstName, lstName);
  accounts.put(nextId, acc);
  return nextId++;
}
```

RCD Mutant:

```
@message
int openAccount (double initBal, String fstName, String lstName) {
  // creation replaced by null
  ActorName acc = null;
  accounts.put(nextId, acc);
  return nextId++;
}
```

*2) **MCP** - Modify Creation Parameter:* This mutation operator modifies the value of a creation parameter of an actor to another value of the same type. The modification of the parameter is based upon sequential mutation operators like replacing a variable reference with a constant of the same type, or replacing one variable reference with another variable reference of the same type. The following code snippets demonstrate an application of this operator in the `openAccount` method of `BankActor`.

Original Code:

```
@message
int openAccount (double initBal, String fstName, String lstName) {
  ActorName acc = create(AccountActor.class, nextId, initBal, fstName, lstName);
  accounts.put(nextId, acc);
  return nextId++;
}
```

MCP Mutant:

```
@message
int openAccount (double initBal, String fstName, String lstName) {
  // initBal parameter changed to 0
  ActorName acc = create(AccountActor.class, nextId, 0, fstName, lstName);
  accounts.put(nextId, acc);
  return nextId++;
}
```

*3) **RCP** - Reorder Creation Parameters:* This mutation operator reorders two parameters of an actor creation. In order to avoid introducing compiler errors, **RCP** can only be applied when both the parameters are of the same type. This is similar to the **RMP** operator. The following code snippets demonstrate an application of this operator in the `openAccount` method of `BankActor`.

Original Code:

```
@message
int openAccount (double initBal, String fstName, String lstName) {
  ActorName acc = create(AccountActor.class, nextId, initBal, fstName, lstName);
  accounts.put(nextId, acc);
  return nextId++;
}
```

RCP Mutant:

```
@message
int openAccount (double initBal, String fstName, String lstName) {
  // reordered fstName and lstName
  ActorName acc = create(AccountActor.class, nextId, initBal, lstName, fstName);
  accounts.put(nextId, acc);
  return nextId++;
}
```

VI. Related work

Mutation testing has been actively researched since its introduction [10]–[12]. In particular, mutation operations have been proposed and established for a wide array of languages including Ada, C, Cobol, C#, Fortran, Java, multi-threaded Java, and SQL. The survey report by Jia and Harman [30] provides an overview of the research that has been conducted on mutation testing including the mutation operators that have been introduced for various languages. However, to the best of our knowledge, no mutation operators have been introduced for actor systems or distributed message-passing systems in general.

The work that we found to be the closest to our contribution is the mutation testing of interfaces between distributed components [21]. In this work, Ghosh and Mathur introduce the concept of constraining mutation testing of large distributed systems to the interfaces between the various distributed components. However, they do not introduce specialized operators to perform the mutation. Instead, they reuse sequential mutation operators.

Another body of work that is related to our contribution is reliability testing of distributed systems using fault injection [31], [32]. Fault injection is a technique that is similar but complimentary to mutation testing. In fault injection, faults (modifications) are inserted into the *state* of a system at runtime, which is different from mutation testing that relies on inserting modifications into the *code*, either source or binary such as bytecode.

VII. Conclusion

In this paper we introduced and categorized new mutation operators for actor-based programs and demonstrated their use on ActorFoundry programs. In the future we plan to perform empirical studies to analyze the efficacy of the introduced operators. We also plan to implement a mutation testing framework for actor based programs that can use the introduced operators to automatically mutate actor programs and efficiently execute the test suite of an actor based program on multiple mutants.

Acknowledgements

We thank the fellow students of CS 524 and CS 598GA at the University of Illinois for constructive discussions on the material presented in this paper. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0916893 and CCF-0746856. Milos Gligoric was supported by the Saburo Muroga fellowship.

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] "ActorFoundry Webpage," http://osl.cs.uiuc.edu/af/.

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

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

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

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

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

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

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

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 4, no. 11, 1978.

[11] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE TSE*, vol. 3, no. 4, 1977.

[12] J. Offutt and R. Untch, "Mutation 2000: Uniting the orthogonal," 2001.

[13] S. Kim, J. Clark, and J. McDermid, "Class mutation: Mutation testing for object oriented programs," in *FMES*, 2000.

[14] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon, "Evaluation of mutation testing for object-oriented programs," in *ICSE*, 2006.

[15] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *MUTATION*, 2006.

[16] R. H. Carver, "Mutation-based testing of concurrent programs," in *ITC*, 1993.

[17] J. Offutt, "Investigations of the software testing coupling effect," *ACM TOSEM*, vol. 1, no. 1, 1992.

[18] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *IPDPS*, 2003.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[20] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *NSDI*, 2006.

[21] S. Ghosh and A. P. Mathur, "Interface mutation," *Wiley STVR*, vol. 11, no. 3, 2001.

[22] J. Offutt, J. Voas, and J. Payne, "Mutation operators for Ada," George Mason University, Tech. Rep., 1996.

[23] K. N. King and J. Offutt, "A Fortran language system for mutation-based software testing," *Wiley SPE*, vol. 21, no. 7, 1991.

[24] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford, "Design of mutant operators for the C programming language," Purdue University, Tech. Rep., 1989.

[25] S. Kim, J. Clark, and J. McDermid, "The rigorous generation of Java mutation operators using HAZOP," in *ICSSEA*, 1999.

[26] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *ISSRE*, 2002.

[27] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA 2009*.

[28] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble Java byte code to measure the effectiveness of unit tests," in *MUTATION*, 2007.

[29] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: A mutation system for Java," in *ICSE*, 2006.

[30] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," King's College London, Tech. Rep., 2009.

[31] R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders, "A global-state-triggered fault injector for distributed system evaluation," *IEEE TPDS*, vol. 15, no. 7, 2004.

[32] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE TSE*, vol. 16, no. 2, 1990.