# Deterministically Testing Actor-Based Concurrent Software

Piet Cordemans
KU Leuven - Technology campus Ostend
Zeedijk 101
8400 Ostend, Belgium
piet.cordemans@kuleuven.be

Eric Steegmans
KU Leuven - Department of Computer Science
Celestijnenlaan 200A
3000 Leuven, Belgium
eric.steegmans@cs.kuleuven.be

Jeroen Boydens
KU Leuven - Technology campus Ostend
Zeedijk 101
8400 Ostend, Belgium
jeroen.boydens@kuleuven.be

## ABSTRACT

Non-deterministic concurrent behavior of software prohibits the idempotent property of tests. XUnit frameworks traditionally do not offer support to deal with these concurrency issues which reduces the significance of unit testing concurrent software. In this paper we propose a tool which supports deterministic testing of concurrent software based on the Actor model. This tool reveals race conditions and seamlessly integrates with xUnit-like frameworks. In our approach, a Coloured Petri Net model is constructed per test as well as the code under test. This model allows isolation of concurrent behavior from the effective actor state. Subsequently, the state space is calculated and traces covering all states are constructed. Corresponding with these traces our tool issues test runs, guaranteeing full state space coverage of each test. Moreover, each failed trace can be backtracked, revealing valuable information concerning the race condition.

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*

## General Terms

Reliability

## Keywords

Concurrent software, Deterministic Testing, Actor Model

## 1. INTRODUCTION

In concurrent software two major issues exist. On the one hand data races occur when at least two write operations, access the same memory location concurrently and are not synchronization operations. On the other hand race conditions happen when at least two events have multiple orderings. When the correctness of a program depends on a specific ordering, the race condition becomes an issue.

Testing concurrent software is considered difficult because of two conflicting properties. Whereas concurrent software potentially exhibits non-deterministic behavior, software tests desirably execute in a deterministic fashion. Yang [11] described four challenges to deal with testing concurrent software. (1) Detecting unintentional races and deadlock; (2) forcing a path in the state space to be executed; (3) reproducing test execution; and (4) defining test coverage criteria.

By introducing concurrency, the state space quickly enlarges, a phenomena called state space explosion [3]. Non-deterministic behavior is introduced with a conventional scheduler as execution of a path in the state space is indeterminate at run-time in the test. Therefore, it is hard to guarantee state coverage while testing models of concurrent computation with mutable shared state.

### 1.1 Actor Model

The Actor model defines an actor as a concurrent entity which reacts to messages [2]. Upon receiving a message, an actor can (1) send a number of messages, (2) create a number of actors, (3) change its local state or (4) alter the behavior upon receiving a subsequent message. Messages received are stored in a mailbox from which the actor selects a message to react upon. Once a message is selected to be processed, the actor completes the corresponding action in a single atomic step. As long as messages are immutable, these are messages which do not change once created, the Actor model prevents data races as mutable data is only accessed in an Actor's local state. However race conditions are not prohibited by this model as the ordering of message handling is non-deterministic.

Lu et al. [9] reported that around one third of the non-deadlock concurrency bugs are due to a violation of the intended order by the programmer. Therefore, detecting race conditions requires meticulous testing of the state space, because these issues might exist in a single path of this state space.

### 1.2 Contributions

We expand on the ideas of applying state space exploration and the Actor model in the context of testing concurrent software. Our goal is to provide a deterministic testing



*A-TEST'15*, August 30-31, 2015, Bergamo, Italy
ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804327

technique for actor-based software which alleviates the effects of the state space explosion problem. More specifically, with this paper we tackle the challenges as posed by Yang:

- Allow automated unit tests to detect unintentional race conditions.
- Construct the state space of unit tests and forcing the execution of a test to follow a specific path in its state space.
- Provide information on paths leading to a failing test, in order to replay the paths of interest.
- Guarantee state coverage in the state space of the test.

Furthermore, we implement a lightweight tool implementing the model which seamlessly integrates with the specific run-time environment. More specifically, it integrates with an x-Unit and Actor model framework.

The paper is organized as follows: in section 2 we describe the model which allows to construct the traces in unit tests for software based on the Actor model. Then, in section 3 we describe a tool implementing the model.

# 2. ACTOR STATE SPACE EXPLORATION

In order to deterministically test actors, the state space of tests must be fully explored. However, to deal with the state space explosion problem, the concurrent behavior should be isolated from the state of the actors. This results in a state space which does not represent the local state managed by the actors. Rather it only contains the state of actor mailboxes and the different actor life cycle states. Once this state space has been constructed, paths of specific message ordering are composed.

## 2.1 Coloured Petri Net of the Actor Model

In our approach, we model the test and actors under test with a Coloured Petri Net (CPN) [7] model. This CPN model isolates concurrent behavior of actors and partitions the state space of the actor system. From this model, the state space can be constructed, as well as the minimum set of paths in order to visit each state at least once.

A CPN combines Petri Net (PN) modeling with features of high level programming languages. Most importantly, CPNs introduce the concept of a color set and token color which respectively describes place and token types. This type system allows to construct models which are more concise than regular PNs, while maintaining the possibility to decompose any CPN to a regular PN. PN models and by extent CPN models are well suited to model parallel computation, as their execution semantics are inherently non-deterministic.

Figure 1 represents a simplified CPN model of a single actor. *Idle*, *mailbox* and *processing* are states, while *receive* and *return* are transitions. Both *idle* and *mailbox* contain a token, respectively the *actor state* token and the *mailbox* token. The *mailbox* token is a list of messages. The arrows with annotations between transitions and states describe the behavior when firing the transition. For instance, when firing *receive*, the *actor state* token of *idle* is consumed, a message is consumed from the *mailbox* state, while the *mailbox* token is returned to the *mailbox* state and a tuple token of *message* and *actor state* is produced in the *processing* state.

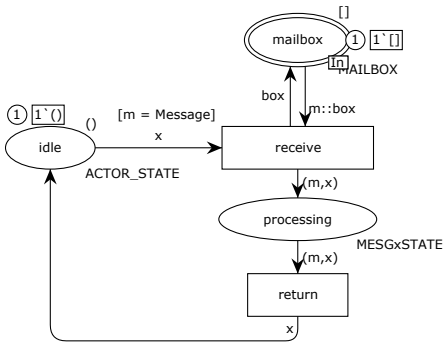


**Figure 1: Simplified CPN model of the Actor model**

### 2.1.1 Isolating Concurrent Behavior

Figure 2 is the generic CPN model of a single actor. Two tokens are always present in this model, one to depict actor state, while the other is the representation of its mailbox. The typical message reception procedure is as follows: upon reception of a message, a token is added to the mailbox token. This activates the respective receive transition, on condition that the actor state token is in the idle state. Subsequently, the token traverses to the processing state and is incremented to depict a new local actor state. While the token resides in the processing state, no other messages will be processed. After the processing state, the token for actor state returns to idle which enables the receive transitions to process a new message. After processing, the four resulting effects can be defined as follows:

1. Change of local state: local decisions result in a corresponding action. This includes either no continuation effect or one of the other resulting effects. Nevertheless, the local state of the actor does not affect the state space of concurrent behavior.
2. Change behavior upon reception of a subsequent message: by counting the number of received messages the subsequent concurrent behavior can be selected. However, the resulting effect does not affect the concurrent state space.
3. Send X messages: tokens are produced in the respective mailboxes of the recipients, as shown in the *Msg* branch of Figure 2.
4. Create N actors: tokens are produced in the activation places of the child actors. A token from the activation and the *notAlive* place are needed to activate the transition to the *idle* place. The activation place is an input socket in the hierarchical CPN model.

Once the message has been processed, its state on the one hand is modeled as either:

1. return the state token to the *idle* state,
2. return the state token to the *notAlive* state.

On the other hand the continuation behavior has one of three possible actions:

1. there is no effect on the test entity or any other actor,

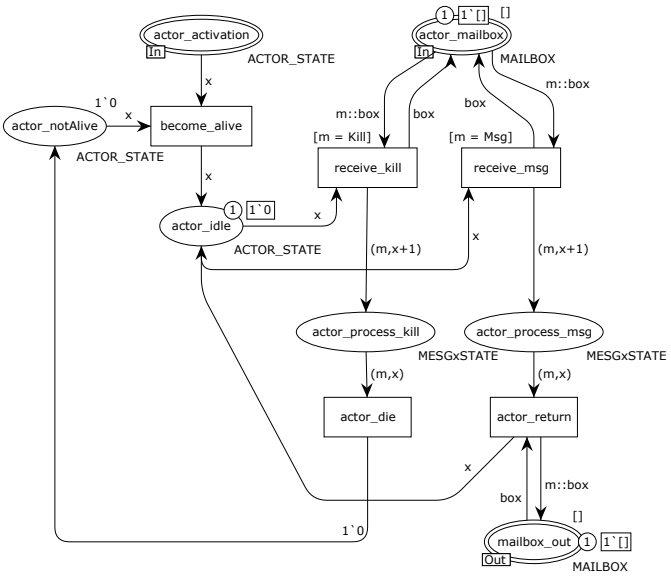


**Figure 2: CPN model of the Actor model**

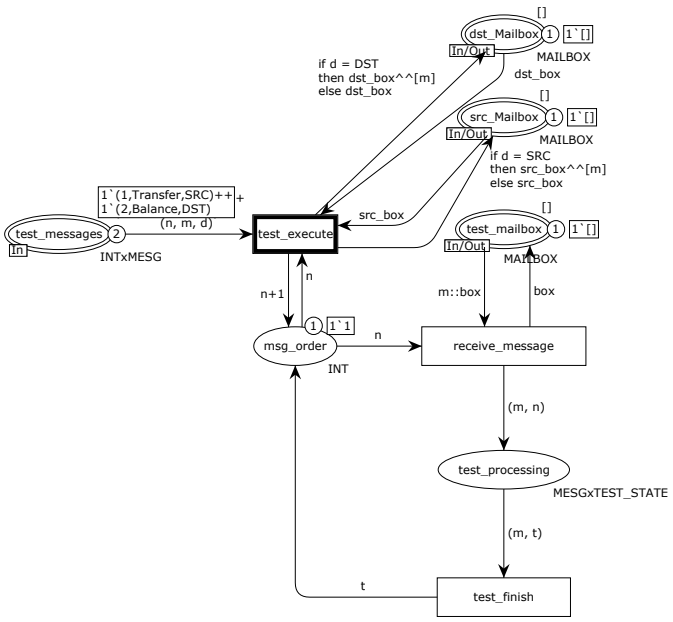


**Figure 3: CPN model of the test actor**

2. generate X tokens in Y mailbox places. With X and Y integers equal or larger than 1,

3. generate N tokens in M activation places. With N and M integers equal or larger than 1.

## 2.2 Hierarchical Test Model

The top level CPN contains all actors involved, the test actor, and a set of initial messages. The generic CPN model for actors provides two places accessible from the top level CPN model. These are the mailbox and activation place. Tokens are produced in these places from the continuation transitions of other actors.

Finally, the top level CPN model contains a representation of the xUnit test definition. This entity initiates the test and captures the results. Furthermore, the test behaves as an actor, because it has an implicit mailbox to allow message-based communication. Therefore, the CPN of the test definition is derived from the generic CPN of the Actor model. The generic CPN actor model can be reduced, because test actors do not need life cycle management. Instead, a test has a place which contains the set of messages to drive the test. Additionally, this place is represented in the top level CPN.

### *2.2.1 Constructing the Test State Space*

The state space can be deduced from the test CPN model. Each state represents a particular set of tokens, state and message tokens, at the corresponding places. Arcs between these places are transitions which upon activation reach the designated state. This is an implementation of the basic algorithm for state space construction as described by Jensen and Kristensen [7]. This algorithm generates the state space of concurrent behavior with regard to the test.

However, this set of states might consist of unreachable states. Namely, some states represent the path of the continuation of a local decision branch. Due to the test setup, only a single path is chosen in the set of possible continuations. On account of isolating concurrency from local actor state, it is impossible to indicate which states cannot been chosen in the model for concurrent behavior. However, by considering each state of the set of continuation states as valid, the effective continuation behavior will be identified at run-time.

## 2.3 Deterministic Traces

The purpose of constructing the state space of concurrent behavior is to determine message ordering in the test. In this state space, traces are designated to provide coverage of the state space of the test. Traces are paths in the state space graph which are chosen deterministically, guaranteeing coverage and reproducibility. As the number of traces is proportional to the run-time performance of the tool, the number of traces per test needs to be minimized.

When considering test coverage, two different viewpoints can be adopted. First is the coverage of the state space of the test, i.e. state space coverage with the set of messages and local actor state as defined by the test. This deals with the problem of non-deterministic test execution. Furthermore is the coverage of the test state space as part of the larger system. Namely which partition of the state space of the larger system is covered by the test.

### *2.3.1 State Space Coverage of a Single Test*

In order to guarantee determinism in a test with actors, state coverage of the concurrency state space is sufficient. Namely, non-determinism is introduced when multiple messages are bound to arrive at a single actor. Therefore, states with different message tokens at a mailbox place determine the effect of this non-determinism. Consequently, the occurring binding element which led to this state is insignificant for the purpose of identifying race conditions. Furthermore, outgoing arcs are either the sequential continuation effect of an actor, or an unrelated event to the current token in the actor state.

With respect to determinism guarantees, the minimum

set of traces to cover all states is proportional to the maximum number of messages in concurrent execution across all states. For instance, a test containing only actors forwarding a single message, will not contain any non-determinism. In effect, only a single trace will be generated for this test. On the other hand, a test with $n$ concurrent messages, effectively leads to $n!$ traces, as $n!$ represents the combinatorial set of message orderings. In general, in order to cover all states a minimum set of $n!$ traces will be needed with $n$ being the maximum number in a concurrency race. In order to construct these traces, a depth-first search algorithm for a directed acyclic graph is implemented.

### 2.3.2 *Partitioning the State Space*

Tests partition the state space of the system under test. Namely, a unit test consists of a limited set of actors and messages. Moreover, unit tests typically focus on a specific functionality of the system, thus most unit tests only explore a single logical path. Multiple tests are combined in a test suite to cover a larger set of states in the state space. This rationale does not change for concurrent software, however the strategy of state space exploration relies on the narrow focus of unit tests to be scalable. Namely, the combinatorial set of messages in concurrency, $n$, is defining for the worst case of the number of traces in the state space. With a limited set of messages and actors, the resulting set of traces, is limited, especially when considering the size of the state space of the system.

# 3. IMPLEMENTATION

In order to test the CPN model of the Actor model, a tool called ActorRunner has been developed. ActorRunner is implemented in Scala with the Akka actors library [6]. The x-Unit test runner of choice is ScalaTest and JUnit [1].

## 3.1 ActorRunner

The purpose of ActorRunner is to accept a set of traces and adapt the execution of unit tests to match these traces. This tool integrates with a conventional x-Unit framework and seamlessly intercepts and resends messages, in order to control message ordering. The general structure of this tool is illustrated in Figure 4.

For each test, ActorRunner starts with a set of traces which have been derived from the unit test and actors under test. For each trace, ActorRunner issues the x-Unit framework to run the test anew. Furthermore, ActorRunner instructs the marshal component with a list of states to conduct the test in a specific message ordering. Following these states, the test explores a specific path in the state space of the test. Finally, the test executes its assertions and determines whether the test passes or fails. Consequently, if there are traces which have not been executed yet, ActorRunner issues another test run with a different trace. Eventually, all the individual test results are aggregated. If a single trace of the test fails, the test is considered to fail altogether. Trace information is added to the test report, to facilitate debugging on the race condition. The coverage criteria, as defined in Section 2.3.1 ensure that the test runs deterministically, regardless of processor load, or properties of the non-deterministic scheduler.

The marshal component is an actor introduced to intercept all messages and resend them in compliance with the order of trace input. In order to intercept all messages, all

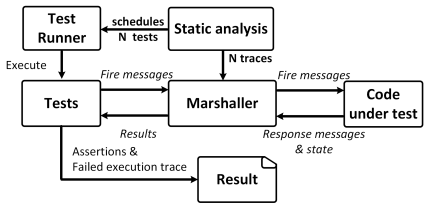


**Figure 4: ActorRunner takes a set of traces as input, schedules N tests and reorders messages as defined in the trace.**

actors are created under supervision of the marshal actor. Furthermore, once the test sends its first message the test actor reference is registered. At creation, instead of returning the real actor reference, a proxy is returned which redirects all messages to the marshal.

In the marshal all messages are gathered, as well as the current state in the trace is indicated. As the marshal is an actor, it acts upon the arrival of messages. Once the set of conditions have been obtained to advance to the next state, the marshal component advances. In effect, these conditions are defined in the following state of the trace. On the one hand these condition can be reached as messages arrive as defined in the following state of the trace. On the other hand one of the actors under test processes a message and the marshal actor internally continues to the next state.

## 3.2 Limitations

This approach is limited by the implementation of the Actor model. Namely, only race conditions can be detected, while data races should be prohibited by the Actor model itself. However, should a programmer violate against this condition, by sharing mutable data, sending mutable messages or no longer ensuring the Actor behavior as atomic, the deterministic state space exploration approach will not be able to detect these concurrency issues or correctly identify race conditions.

Furthermore, regarding liveness issues, such as deadlock or livelock, this approach will invoke the conditions leading to this behavior. However, depending on the properties of the testing framework, it will likely lead to a time-out, without any valuable debugging information. In effect, this approach is ineffective in detecting these issues.

# 4. RELATED WORK

State space exploration has been introduced by Edelstein et al. [5]. They proposed to explore the state space by rerunning existing tests, while manipulating thread interleaving. This technique allows to explore and replay different paths in the state space of the test. However, their approach suffers from the state space explosion problem, as each atomic operation can be interleaved. Moreover, each of the test runs is slowed down by the run-time performance cost of the multitude of context switches. Therefore, the tool provides a heuristic solution to detect concurrency problems which allows configuration of the number of context switches to explore.

Chess [10] is a tool which conducts state space exploration for .NET. However, this implementations did not deal with the state space explosion problem and relied on heuristics to indicate concurrency problems. Namely the number of ex-

plored thread interleaving is limited to constrain the number of paths explored.

State space exploration has been adopted in Basset [8] for Actor systems. Lauterburg et al. [8] continued on the idea of state space exploration. Instead of manipulating thread interleaving on the level of bytecode, they apply it to a higher level model, more precisely, the Actor model. Lauterburg et al. decided to build a model checker for actor programs based on Java Pathfinder [4]. However, instead of focusing on tests, their tool explores the state space of the whole actor program. By doing so, the state space explosion problem deteriorates, and to mitigate this effect, the tool is based on a heuristic to linearize the set of states. Furthermore, ActorRunner does not need to run an adapted JVM which Basset needs for Java Pathfinder.

# 5. FUTURE WORK

Firstly, decisions based on local state might change the concurrent behavior. Consequently, a similar test with different modalities might cover a different state space. Yet, as part of the concurrent state space analysis, these tests will partition a part of the state space of the test and even share some states. This is a possible optimization which might reduce the number of states and traces. Moreover when considering the test space over a multiple test span, it might be possible to indicate concurrency states which are not covered by the test suite, due to local state. This information might be included in a coverage report, so that a tester is aware that the test suite might be lacking some tests.

Secondly, the deterministic state space exploration approach requires an extensive case study, within a larger existing code base. This will allow to prove the feasibility and scalability of this approach. This will also require that some steps, such as CPN generation from code become automated, as described by the scheme in Section 2.1.

# 6. CONCLUSION

Due to indeterminism, the result of a test run on concurrent software is not reliable. In order to deal with this problem, this paper described an approach to conduct deterministic state space exploration for the Actor model. This approach allows to deterministically run a conventional test suite for concurrent software, as defined by the criteria of Yang. Coloured Petri Net models of tests isolate the concurrent behavior from the local state of the actors. This partitions the state space of the system, in which a limited set of traces allow to cover all states of tests. A marshal actor is introduced which reorders messaging in tests according to the generated traces. By aggregating the results of all traces, tests become deterministic. We implemented this approach in a tool called ActorRunner. This tool provides a proof of concept to introduce a seamless message scheduling system which in the context of unit testing is scalable, contains valuable debugging information and is effective to detect race conditions in a deterministic fashion.

# 7. REFERENCES


[1] Junit.org: Resources for test driven development, http://www.junit.org/.

[2] G. A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[3] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351. Springer, 1997.

[4] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder-second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*. Citeseer, 2000.

[5] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[6] P. Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.

[7] K. Jensen and L. Kristensen. *Coloured Petri Nets: modelling and validation of concurrent systems*. Springer, 2009.

[8] S. Lauterburg, M. Dotta, D. Marinov, and G. 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*, pages 468–479. IEEE Computer Society, 2009.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages*, 2008.

[10] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*, 2008.

[11] C.-S. D. Yang. *Program-based, structural testing of shared memory parallel programs*. PhD thesis, University of Delaware, 1999.