Jacco: More Efficient Model Checking Toolset for Java Actor Programs

Arvin Zakeriyan
University of Tehran
a.zakeriyan@ut.ac.ir

Ehsan Khamespanah
University of Tehran, Tehran, Iran
Reykjavik University, Reykjavik, Iceland
e.khamespanah@ut.ac.ir

Marjan Sirjani
Reykjavik University, Reykjavik, Iceland
marjan@ru.is

Ramtin Khosravi
University of Tehran, Tehran, Iran
r.khosravi@ut.ac.ir

Abstract

Actors provide concurrent, distributed, and event-driven autonomous objects which communicate asynchronously. Actor model benefits from higher level of scalability and actor programs are less error-prone in comparison to programs developed in other concurrent models. However, it does not prevent the racing and concurrency errors. So, to guarantee the correctness of mission critical actor programs, verification techniques like model checking are needed. Previously, Basset has been developed based on Java PathFinder, for model checking of Java actor programs. The message scheduling approach of Basset can cause false negative results as well as and early state space exploration. In addition, using Java PathFinder as the back-end model checker imposes the execution time inefficiencies. To resolve these issues, we developed Jacco as the direct model checking toolset for Java actor programs. We provided a new message scheduling approach and implemented it in Java. To illustrate how efficiently Jacco works, Basset and Jacco model checking results are compared for a number of case studies. We also used Jacco for the model checking of a real-world program in robotics systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

AGERE@SPLASH, 2015 Pittsburgh, Pennsylvania USA.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Categories and Subject Descriptors D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification - Assertion checkers, Formal methods, Model checking.

General Terms Theory

Keywords Actor Model, Java, Model Checking, Basset

1. Introduction

Actors are distributed, autonomous objects that interact by asynchronous message passing. The actor model was originally introduced by Hewitt [13] as an agent-based language and is later developed by Agha [9] as a mathematical model of concurrent computation. *Actors* are seen as universal primitives of concurrent computation [9]. Each actor provides a certain number of services which can be requested by other actors by sending messages to the provider. Messages are put in the mailbox of the receiver, the receiver takes a message from the mailbox and executes its corresponding service, possibly sending messages to some other actors.

The actor model provides a higher level of programming abstraction, prevents low-level data races, and provides a higher level of scalability. As a result, actor-based programs are less error-prone in comparison with programs which are developed based on other concurrency models. These features make actor model appropriate for the mission critical distributed and concurrent programs. A number of programming language vendors tried to provide language features and libraries for actor-based programming, e.g. Erlang [3] and Akka [2] as a well-known actor-based programming language and a Java library for actor-based programming, respectively. The developed libraries and languages have been used in many industrial projects, e.g. Twitter's message queuing system and Vendetta's game engine [16].

Actors, as the universal primitives of concurrent computation of an actor model, prevent low-level data racing because of data encapsulation; although, the mailboxes of actors are shared to allow the other actors putting messages into them. So, racing among actors, as the main difficulty of the development of concurrent programs, only may occur in access to the mailboxes. As a result, different orders of receiving messages may results in malfunctioning or deadlock. So, in order to guarantee the correctness of an actor program, different interleaving of receiving messages in an actor mailbox must be considered.

With all the benefits of the program testing approach, one can never guarantee the correctness of a program using testing; since, all of the possible orders of receiving messages are not considered in testing approaches. It is in contrast with the correctness requirements of safety-critical systems, where guarantee on program correctness is needed. Model checking, as an exhaustive exploration of the system's states, is an alternative to achieve guarantee on the correctness of systems.

Model checking is a formal verification approach aiming to verify correctness of systems with a very high-level of confidence. The importance of using model checking to assure reliability of systems has long been acknowledged. There are a number of toolsets for model checking of actor programs. Basset [15] as the model checker for Java actor programs and McErlang [12] as the model checker of Erlang programs are two well-known model checking toolsets of actor programs (Section 2). The major limiting factor in applying model checking for verification of real-world actor programs is the required amount of memory and time for storing and exploring the systems state spaces. Lauterburg et.al. in [16] proposed a reduction technique to reduce the number of explored states; however, it remained huge for real-world programs (Section 4).

In this paper, we implemented a new toolset for model checking of Java actor programs called Jacco (Section 3). In Jacco, we developed a new message scheduling approach which conforms the message scheduling of Java actor libraries like AKKA. We also provided a new approach for finding the similar states in state spaces, results in speedup in the model checking. This way, model checking of bigger case studies and real-world Java actor applications become possible. To illustrate the applicability of Jacco, we used Jacco for model checking of a distributed controller of a robotic system. This system is developed using Robotic Operating System (ROS), a frame-work for developing robotic systems which provides libraries and a toolset for developing distributed systems.

2. Model Checking of Actor Programs

Although using actor model makes the development of distributed systems easier, it presents a significant challenge for the testing and verification community. It is because of the fact that actor systems can nondeterministically exhibit exponentially many different interleaving of serving messages. This stems from the fact that multiple messages sent to an actor may be received in different orders, resulting in different configurations; however, usually only few specific interleavings and configurations lead to bugs. Although many testing approaches are developed to deal with this challenge [14, 18, 19], none of them guarantee the correctness of actor programs. In contrast, model checking ensures the program correctness by exhaustive checking of all of the possible interleavings and configurations. Model checking is a time and resource consuming process but this correctness guarantee is needed for mission-critical actor programs.

For Java actor programs, using the currently existing Java model checking toolsets like [10, 11, 20] is a straight forward solution. Java Path Finder (JPF) [20] is the most well-known model checking toolset and has been used in many real-world projects. JPF benefits from a special Java Virtual Machine (JVM) that provides complex control over low level interactions and also provides high level Java structures at low level code executions. JPF considers interleavings of Java byte code instructions, results in the finest possible granularity of interleaving. The main obstacle against using JPF and other general purpose Java model checking toolsets for verification of actor programs is the state space explosion problem. Since actor model provides higher level of abstraction, low level data races between different actors can not happen. As a result, there is no need for a fine-grain interleaving among the statements of the codes.

The other alternative is using direct model checking toolsets of Java actor programs. To the best of our knowledge, Basset [15] is the only existing model checking toolset of Java actor programs. Basset has been developed at the top of JPF, provides model checking facilities for Scala and ActorFoundry actor programs. Different reduction techniques have been developed to prune generated state space, relying on the encapsulation of actors for creating state space in coarser level of granularity. As we will show later, a pessimistic assumption in scheduler of Basset and using fine-grain back-end model checking toolset (JPF) is the main obstacle against using Basset for real-world case studies. These two issues increase both the state space size and the model checking time consumption.

In addition to the Java community, a toolset is developed for verification of actor programs which are developed by Erlang, called McErlang [12]. McErlang is a toolset for verification of Erlang distributed programs which provides support for a substantial part of the language. McErlang generates the state space in a fine-grain manner, so it suffers from state space explosion problem as well.

3. Jacco

In this section, we will show how the new approach for model checking of Java actor programs is implemented and how it improves the currently existing approaches. As mentioned in Section 2, Basset is the only toolset for model checking of Java actor programs. Basset currently provides support for Scala and ActorFoundry and it can be easily extended to support model checking of other actor libraries which execute over JVM. Here, prior to the detailed description of Jacco, we review the details of implementation of Basset to illustrate why Jacco is needed. Afterwards, we demonstrate the architecture of Jacco and its components.

3.1 Basset Message Scheduling

The message scheduling algorithm is the heart of model checking toolsets of actor programs, as scenarios with different interleavings of serving messages are created by message schedulers. Basset message scheduler stores all the on the fly messages (i.e. the messages which are sent but not yet delivered to their destination actors) in a message cloud. The message scheduler chooses one of the on the fly messages from the message cloud nondeterministically and delivers it to its destination actor. So, there is no assumption on the order of serving the received messages. Although in actor model there is no assumption on the ordering of the received messages, in real-world implementations of actor model it is a safe assumption to consider message ordering when an actor sends more than one message to another actor. It is because of the fact that the currently existing programming languages and libraries of actor model use TCP protocol to send messages. Using TCP protocol guarantees that the order of received data in destination is the same as its sending order. Based on this fact, actor libraries and languages clarify preserving the message ordering, e.g. for AKKA: "for a given pair of actors, messages sent directly from the first to the second will not be received out-of-order" [1]. So, the Basset scheduling algorithm causes false negative results in some cases. For example, in Listing 1 a special implementation of Ping/Pong system is presented where two pong messages are sent upon receiving a ping message. Using conditional statement of line 15, no ping message is sent upon receiving the first pong message and the only case of sending ping message is by receiving the second pong message. This way, the behavior of the model is the same as the normal Ping/Pong system. But, using Basset for analysis of this system results in a case where there is a queue overflow problem in the system. As there is no assumption on the execution order of messages in the message cloud of Basset, message scheduler maybe delivers the message which is sent by the statement `send(po, "pong", 1)` prior to the delivery of its previously sent message by the statement `send(po, "pong", -1)`. So, upon receiving the first pong message, a ping message is sent and the execution of this ping message results in sending two other pong messages. So, there is no need for consuming the second pong message at all to have progress in the program which results in queue overflow in Pong actor. However, this case is impossible in the real-world.

Listing 1. An special implementation of Ping/Pong model with two pongs in ActorFoundry.

```
1  public class Ping {
2      private Pong po;
3      @message
4      public void ping() {
5          send(po, "pong", -1);
6          send(po, "pong", 1);
7      }
8  }
9  public class Pong {
10     private Ping pi;
11     private int counter = 0;
12     @message
13     public void pong(int value) {
14         counter += value;
15         if (counter >= 0)
16             send(pi, "ping", null);
17     }
18 }
```

So, in a nutshell, in Basset there is no assumption in the delivery order of any messages while actor languages and libraries preserve the order of messages which are sent by an actor.

On the other hand, Basset is implemented over JPF tool set which model checks the programs at granularity of instruction level. But, this fined-grained level of granularity is not necessary for actor programs as actors do not have shared states. To avoid this performance penalty, Basset uses the macro-step semantics [8, 17] for an actor execution. In the macro-step semantics, after delivering a message to an actor, the actor executes its corresponding service atomically until the next receive point. This way, there is no need for fine-grain interleaving at instruction level. This approach reduces state space size, although, it does not affect time consumption significantly. We proposed an improvement from this direction, using direct model checking toolset. As Section 4 shows, using direct state space generation saves up to 99% of the time consumption.

3.2 Jacco Improvements

In order to consider all the possible interleaving of messages between actors and at the same time consider the timing order between received messages from an actor, in Jacco a set of queues are used where each queue contains the messages which are sent by a specific actor to another actor. This way, whenever the queue cloud contains more than one queue which contain deliverable messages, a message is chosen nondeterministically from the head of its corresponding queue and it is delivered to its destination actor. Figure 1 illustrates how messages are ordered in case of the example of Listing 1 in Jacco and Basset. The values `-1` and `1` are the parameters that are sent with the messages. As shown in Figure 1(b), there are four different queues for storing messages sent by any actor to the other actors. Notation `a->b` is used to address the queue of messages which are sent to actor “b” from actor “a”. In Jacco scheduler, both `send(po, "pong" , -1)` and `send(po, "pong" , 1)` are placed in one queue since they are sent from the same actor (Figure 1(b)). Consequently the timing order between them is preserved and `send(po, "pong", -1)` is the only message that can be delivered. However in Basset, no timing order is preserved and as a result, both messages are ready to deliver in the next step (Figure 1(a)).

Since actor model provides isolation for actors and actors do not have shared variables, we use macro-step semantics for actor verification like Basset. It means that, we do not interrupt the message execution. This assumption is valid since in Jacco, there is no synchronous communication among actors.

The other improvement in Jacco is in the way of storing the states and detecting repeated states. Considering many actor programs, we observed that in many actor systems, a number of auxiliary actors are instantiated to do partial calculations, then they are made garbage. This phenomena leads to creation of different states which have the same actors and the states of actors are the same except the actor ids. In addition, in some cases, there are some states which are different because of the fact that the order of the creation of actors is different; however, the states of actors are the same. In these cases, considering only one of these states satisfies verification requirements, as the behavior of an actor does not affected by its id. To this aim, we don’t consider the actors ids in the state of that actor. It should be noted that we just don’t consider the id of an actor in it’s own state but mapping between actor ids and their state are kept to consider different interleaving of actor executions.

3.3 Jacco Architecture and Implementation Issues

In order to directly verify any kind of actor programs without making any changes in their source codes, some modification on actor libraries are needed to connect the libraries to the model checking engine of Jacco. In addition, some features which are not related to the behavior of actor programs (e.g. handling communication between different actors on a geographically distributed nodes and fail over mechanism) must be removed. Note that, these modifications have to be minimized to keep the maintenance of libraries under control.

In case of Actor Foundry, we decided to perform modifications in the basic class which contains the actor behavior, called *actor class*. The main modification in *actor class* is that it must be inherited from Jacco `Actor` class. In addition, some minor modifications in the constructors of *actor class* are required. Performing modifications on constructors, we make sure that the instantiated actors are registered in the actor repository of the Jacco’s model checking engine. Finally, the `send` method of basic *actor class* has to be modified as the communications between actors must be tracked. The result library is uploaded in Jacco section in Rebeca home page [1][5]. The architecture of Jacco from this point of view is depicted in Package View presentation in Figure 2.

Currently, for the verification of actor programs, Jacco provides deadlock-freedom analysis and assertions-based model checking. To this aim, the state space of a program is created using BFS graph traversal algorithm. To implement BFS, a mechanism for detecting the repeated states must be implemented to avoid the creation of infinite state spaces. To this aim, during exploration of a state space, all the visited states are kept in a hash table. For a newly generated state, its hash value is computed; then, the set of all of the states with the same hash value are created from the hash table, called *set of possibly similar states*. To make sure that the newly generated state is new, its content is compared to all of the members of the possibly similar states set, considering the queue content of actors and the value of local fields. In some cases, the values of some of local fields are not a part of the state of the actor (their values do not impact actor actions and do not change during the time). We defined `exclude` Java annotation in Jacco API to allow developers to exclude a field from the state of an actor. For example it is common to have a universal serial id field in actor definition, which has the same value in its instances. Developers can use `exclude` annotation to remove this field from actor states to keep them smaller. It should be noted that if a field is excluded, the developer have to make sure that changing the value of the excluded field does not impact the state of the actor.

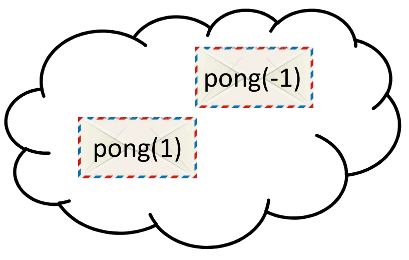
To reduce the memory consumption, the collapse technique of JPF is used in Jacco. To this aim, a static hash table is created to keep all of the local states of actors. Using linear probing for resolving hash collisions, open addressing is achieved and a unique hash key is assigned to the local state of each actor. This way, a system state is a set of pairs, where each pair is an actor id and the hash key of its state in the static hash table. Comparing two actor states is a simple comparison on hash keys of both states, which is less time consuming compared to the comparison on the actual content of actors states. Besides, since performing transition from one system usually does not change the state of the majority of actors; using collapse technique prevents keeping redundant data.

4. Experimental Results

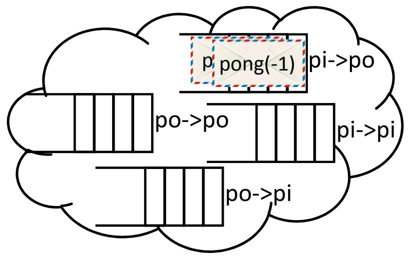
4.1 Jacco in Practice

This section presents experimental results of using Jacco for the analysis of a set of different case studies. The presented eight different case studies are a number of classic concurrent case studies in addition to a selection from the cases of [15]. The case studies are developed in ActorFoundry, so they can be model checked in both Jacco and Basset. The model checking results are presented in Table 1 for different

[1] `http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/Jacco`



(a) Organization of messages in message cloud of Basset



(b) Organization of messages in message cloud of Jacco

Figure 1. How messages in the example of Listing 1 are organized in Basset and Jacco message clouds (1 and -1 are the parameters of the message).

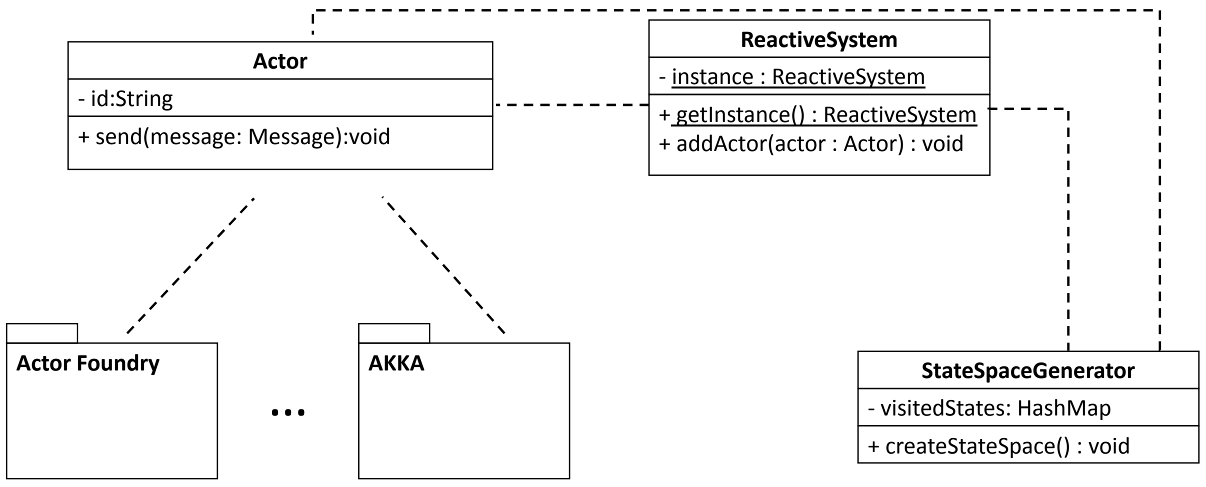

Actor
- id:String
+ send(message: Message):void
ReactiveSystem
- instance : ReactiveSystem
+ getInstance() : ReactiveSystem
+ addActor(actor : Actor) : void
Actor Foundry
...
AKKA
StateSpaceGenerator
- visitedStates: HashMap
+ createStateSpace() : void

Figure 2. Package view representation of Jacco Architecture.

configurations of each case study. The short description of these cases studies are presented below.

- **Fibonacci** is the actor based implementation of computing the nth element in the Fibonacci sequence.
- **Dining Philosophers** is the actor implementation of the classic dining philosophers problem, where each philosopher is modeled as an actor.
- **Merge sort**, **pipe sort**, and **quick sort** problems are the actor based implementation of sorting problems with the same names.
- **Pi** is the distributed implementation of computing the value of π number.
- **Shortest Path** problem is the distributed implementation of graph traversal for finding the shortest path from a vertex to the other vertices.
- **Chameneos** is the simplified version of the chameneos game. A chameneos lives alone, eating leaves in the forest, and training. When feeling ready for competition, a chameneos enters a mall, plays game with another chameneos, possibly mutates (i.e. changing color according to the neighbor chamenos), then returns to the forest. Chameneos game examines a population of chameneos for the possibility of reaching a state in which all chameneos have the same color; therefore, no one can mutate anymore.

As shown in Table 1, in all of the case studies the size of the state space and time consumption of Jacco model checker is far lower than of the Basset's. In the cases where the results are not present for Basset, it did not finish successfully (i.e. state space explosion). In Fibonacci the gained improvement is mainly because of the way of finding the repeated states which makes the state space very small. In this case study, for calculating the nth element, two new actors are created to calculate the values of $(n-1)$th and $(n-2)$th elements. Consequently, for calculating the $(n-1)$th element, two new actors have to be created to calculate the $(n-2)$th and $(n-3)$th elements and so on. So, almost all the elements are calculated twice. The implementation of Jacco for detecting repeated states does not count these two states as different states and this causes a big reduction in com-

Table 1. Comparing the size of the state space and time consumption in different case studies

experiment		Jacco		Basset		%Reduction
Case Study Name	configuration	# states	time	# states	time	
Fibonacci	4th element	43	< 1 sec	434	3 secs	90%
	5th element	753	2 secs	115K	7.5 mins	99%
	6th element	110K	1.5 min	-	-	-
Dining Philosophers	2 philosophers	22	$< 1sec$	176	2 secs	87%
	3 philosophers	110	$< 1sec$	5.8K	17 secs	98%
	4 philosophers	550	$< 1sec$	260K	14 mins	99%
	5 philosophers	2750	2 sec	-	-	-
Merge Sort	array of size 3	94	$< 1sec$	655	3 secs	86%
	array of size 4	5K	2 secs	25K	1 min	80%
	array of size 5	361K	1 min	566K	28 mins	37%
Pi	4 nodes	116	$< 1sec$	6K	15 secs	98%
	5 nodes	320	$< 1sec$	58K	2 mins	99%
	6 nodes	1094	1 sec	712K	32 mins	99%
	7 nodes	3K	3 sec	-	-	-
Pipe Sort	array of size 3	81	< 1sec	540	3 secs	85%
	array of size 4	744	< 1 sec	6K	18 secs	87%
	array of size 5	13K	$< 1sec$	107K	5 mins	88%
	array of size 6	361K	2 min	-	-	-
Quick Sort	array of size 3	94	$< 1sec$	655	3 secs	86%
	array of size 4	2K	1 sec	8K	25 secs	75%
	array of size 5	361K	1 min	553K	29 mins	33%
Shortest Path	graph with 3 nodes	59	$< 1sec$	1159	4 secs	95%
	graph with 4 nodes	161	$< 1sec$	10K	23 secs	98%
	graph with 5 nodes	855	$< 1sec$	192K	10 mins	99%
Chameneos	N = 1	382	$< 1sec$	29K	1 min	98%
	N = 2	574	$< 1sec$	54K	2 mins	99%
	N = 3	1146	1 sec	138K	5 mins	99%

parison to Basset. In the cases of dining philosophers and shortest path the number of active actors are the same in all of the states and the main reason for gaining reduction is in the message scheduling approach, described in Section 3.2.

4.2 Towards Real-World Case Studies

In order to illustrate the applicability of Jacco for the real-world case studies, we used Jacco for verification of distributed controller of a robotic system. This system is developed using Robotic Operating System (ROS). ROS provides a frame-work (including a set of libraries and toolset) for developing distributed robotic systems [6]. The goal of the frame-work is to encourage *collaborative* development by providing an infrastructure for orchestrating different modules together to make a system. Many complex and industrial robotic systems have been developed using ROS framework, e.g. Husky [4] robot and Turtlebot [7]. Using ROS, different components of systems run in an isolated environment and communications among them take place by asynchronous message passing. There are two types of message passing in ROS, which are Publisher/Subscriber message passing style and asynchronous Service call style. In Publisher/Subscriber style, program developers define some topics and different modules subscribe for the topics. These topics can use TCP or UDP for message transfer but The default message transport protocol in ROS systems is TCP and since TCP preserve order of the data, we can be sure that our assumption on preserving order of messages between actors is served. In Service call style, the communication takes place by sending a message from one module to another module and getting a reply. These features make ROS computation model very close to the actor model. Each processing unit of a ROS system can be viewed as an actor that performs some actions.

Based on this similarity, we provided an approach for transforming ROS elements to ActorFoundry actors to verify ROS systems. In this transformation, an actor is instantiated for each node and both types of communication styles in the ROS code are transformed into message passing among actors.

Since ROS currently does not support actors and the ROS systems are developed on Python or C++, we used the aforementioned approach to transform the control system of a Quadricopter from Python to ActorFoundry. The trans-

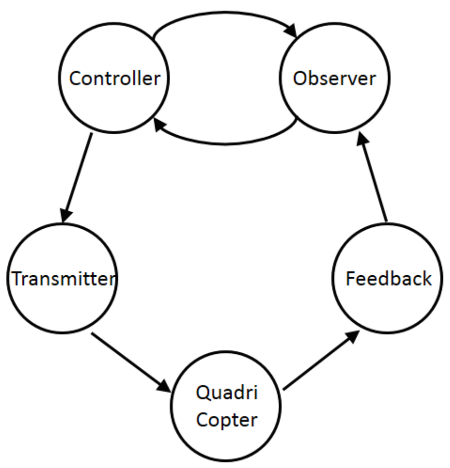


Figure 3. Data flow between nodes in quadricopter controller system.

formed system contains five actors *Controller*, *Transmitter*, *Quadricopter*, *Feedback*, and *Observer*. Each actor in this implementation is a representative of a computational node in the original system. Message handlers of the system had complex codes for calculating control values. Since our purpose was to verify this system against concurrency problems such as encountering deadlock, we abstracted computation and replaced them with random value generators. But the value of variables which are used in communication among actors remained the same as the original system. This way the system can be considered in two parts. The first part contains *Quadricopter* actor as the target device and *Transmitter* and *Feedback* as the interfaces of *Quadricopter* to the system controller. The second part contains *Controller* and *Observer*.

Controller is the actor that computes direction and other parameters for moving based on its received information. *Controller* sends the commands to the *Transmitter*. The *Transmitter* actor translates the received commands to the Quadricopter's commands. Upon receiving a command, Quadricopter moves to a new position. It has a camera recorder and some other sensors for measuring the environmental metrics. The measured data are sent to *Feedback* actor. *Feedback* actor translates received data to information which can be used by *Observer*. The sent information is validated by *Observer* and the required portion of it is sent to the *Controller*. This cycle creates the feedback loop of Quadricopter system. Figure 3 shows the data flow among different modules of Quadricopter system. We implemented this system in ActorFoundry and model checked it. Using Jacco, we verified the correctness of the existing implementation of the controller against deadlock and queue overflow properties. It results in a state space with 1,317,597 states and verified in 80 seconds.

5. Conclusion

In this paper, we presented Jacco as a new toolset for model checking of Java actor programs. Prior to our work, Basset has been developed for model checking of Java actor programs. We showed the message scheduling assumption of Basset makes it inappropriate for model checking of real-world Java actor programs; as it does not consider the order of received messages. This assumption is in contrast with the property of the existing actor libraries which guarantee the order of received messages in some conditions. This assumption causes false negative results in model checking of programs.

In addition, Basset has been developed on the top of Java Path Finder (JPF) which results in model checking inefficiencies. Since JPF verifies programs at byte-code instruction level granularity, the model checking time consumption is high, although Basset benefits from macro step semantics. To illustrate the applicability of Jacco, we used it for model checking of the problems which are previously model checked by Basset. Jacco dominates Basset results in both time consumption and state space size.

Currently, Jacco supports model checking against safety properties. As the future work, we are going to develop LTL and CTL model checking algorithms for Jacco. We also want to develop new other reduction techniques for Jacco to make it more suitable for verification of real-world programs. Besides, we want to verify some other control systems in robotics domain which are implemented on ROS, like Husky.

References

[1] *Akka Java Documentation*. http://akka.io/docs/.

[2] *Akka Programming Language Homepage*. http://www.akka.io.

[3] *Erlang Programming Language Homepage*. http://www.erlang.org.

[4] *Husky Robot Web Page*. wiki.ros.org/Robots/Husky.

[5] *Rebeca Home Page*. http://www.rebeca-lang.org.

[6] *ROS Web Page*. http://www.ros.org.

[7] *Turtlebot Web Page*. wiki.ros.org/Robots/Turtlebot.

[8] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[9] G. A. Agha. *ACTORS - A Model of Concurrent Computation in Distributed Systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[10] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 439–448. ACM, 2000.

[11] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[12] L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 125–136. ACM, 2007.

[13] C. Hewitt. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT, Apr. 1972.

[14] J. Hughes. Software testing with QuickCheck. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer, 2009.

[15] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha. A framework for state-space exploration of java-based actor programs. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 468–479. IEEE Computer Society, 2009.

[16] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In D. S. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.

[17] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[18] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In H. Giese and G. Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.

[19] S. Tasharofi, M. Pradel, Y. Lin, and R. E. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In E. Denney, T. Bultan, and A. Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 114–124. IEEE, 2013.

[20] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.