The Implementation of a Communication Deadlock Analysis Based on the Theory of Deadlock Analysis for the Actor Model

Johnny Xi, Harris Zhang ,Jack Xiang

August 4, 2019

Abstract. We present a unique approach for communication deadlock analysis for actor-model which has an under-approximated analysis result. Our analysis detects narrowly defined communication deadlocks by finding a cyclic dependency relation in a novel dependency graph called the *slave dependency graph*. The *slave dependency graph* is based on a new relationship between *Actors*, *slave dependency*, defined by us. After that, we implement this theory in Soot, an analysis tool for Java, and use it to analyze actor-based Java program realized by Akka, a Java library that allows actor-based programming. We argue that our analysis can detect a specific kind of communication deadlock with the precise result, but has many limitations.

1. Introduction

According to the development of theories across several decades, from the introduction of the concept 'actors' in Carl Hewitt's work in 1973, to the application of May-Happen-in-Parallel Analysis for the actor model in Flores Montaya's paper in 2013, various progress has been made in developing analysis theories for different types of deadlocks in actor models, including communication deadlock, behavioral deadlock and live deadlock[5]. According to the definitions of behavioral deadlock and livelock , there is a wide variety of causes of these kinds of deadlocks; hence, it is hard for researchers to specify them and think of a general solution, resulting in little progress on analyzing these kinds of deadlocks. Therefore, the studies on communication deadlock are relatively more common, which make it easier for us to learn about and do some research on this topic. However, as we dive deep into the field of communication deadlock analysis for actor models, we find that there are a lot more theoretic studies than practical implementation of theories on finding communication deadlock. As a result, we develop a novel approach involving a "Slave Graph" (based on changes to an existing approach), for finding communication deadlock in an *Actor System*. After that, we implement our analysis theory and use an analysis tool, Soot, to look for communication deadlocks in a certain actor-based programming toolkit in Java, Akka, by checking the definition statements of specific actor classes inside the program being analyzed.

2. Definition and Background Information

Actors are isolated concurrent entities that communicate through asynchronous messages and do not share state[5]. An actor responds to an incoming message by generating a finite set of communications sent to other actors, a new behavior (which will govern the response to the next communication processed) and a finite set of new actors created[1]. The actor-based concurrency model is generally seen as an enhancement on the current thread-based concurrency model. Since actors can only interact with each other through the mean of message exchange, the states of each actor could only be subjected to mutation from itself, unlike how multiple thread can access the same process in the thread model. This means that the actor model is an effective solution to common problems in multi-thread program, such as data race conditions as well as deadlocks. However, thoughtless design of algorithms using the actor model is still possible to result in deadlocks[5].

Soot is a static program analysis framework developed by the Sable Research Group at McGill University. It processes Java code into a 3-address intermediate language called "Jimple", which is suitable for static analysis. The modules in Soot support intraprocedural and interprocedural analysis, point-to analysis, def/use chain, and graph construction, but the users can also create their own analysis, like communication deadlock analysis in this research, based on the framework. Since Java by default does not support actor model concurrency, the programs we analyze in this research are Java codes written in Akka framework. Akka is an open-source toolkit developed by Lightbend that provide support for actor-model concurrency in both Java and Scala.

A communication deadlock is a condition in a system where two or more actors are blocked forever waiting for each other to do something. This condition is similar to traditional deadlocks known from thread-based programs[5]. However, we have a more specific situation of communication deadlock to analyze and have to narrow its definition. In our research, restricted communication deadlock occurs when two or more actors don't sent messages that are required by another actor and only declared by themselves, so that the actor system reaches a state that each actor in the actor system is waiting for another actor to send its message. An example of PingPong is given below:

Ping	Pong
1: Ping_msg	1: Pong_msg
2: receive(Pong_msg, x → {	2: receive(Ping_msg, x → {
3: sender().tell("ok")	3: sender().tell("ok")
4: })	4: })

Fig.1. Single example of restricted communication deadlock

In this example, *Actor* Ping and Pong both have the messages that are required by each other: Ping needs Pong_msg and Pong needs Ping_msg. However, neither of them will send those messages. Instead, they send "ok". Hence, they reaches a state of waiting for each other to send messages in line 2, which becomes a situation of restricted communication deadlock.

3. Development of Analysis Theory

3.1 Derivation from Dependency Graph

As we read through papers about finding communication deadlocks, we found a simple way to spot deadlocks, which is from the work of Flores-Montoya's team. They created a kind of state dependency graph G_S and state the definition of deadlock as following:

A program state S is deadlock if and only if when its dependency graph G_S contains a cycle[3].

However, we misunderstood the definition of "dependency graph G_S", and went straight into the wrong direction of looking for cycles in data dependency and control dependency graphs. In fact, G_S is actually built upon nodes like objects and task identifiers and whose edges are defined between objects and tasks, which is dramatically different from our work based on data dependency and control dependency graph.

At last, we found it couldn't solve restricted communication deadlock and decided to take its idea of finding cycles in a kind of dependency graph and built our own dependency graph.

3.2 Slave Dependency Graph

3.2.1 Semantics

This section presents the semantics of actor-based language:

An *Actor System* S is a set $S =$ `Actors` where `Actors` is the set of all created *Actors*.

An *Actor* is a term $A(a, M, RT < msg, t >)$ where a is the Actor identifier, M is the set of all messages created within the *Actor*, $RT < msg, t >$ is the set of all pairs of the message received and the message sent consequently, in which msg means the message received by this *Actor* and t means the message sent back by this *Actor* in response to the message it receives.

3.2.2 Definitions

Slave Dependency Relationship:
Given two Actors A_1 and A_2, we define their *slave dependency relatioinship*, A_1 is slave dependent on A_2 as follows:

$$A_1(a_1, MSG_1, RT_1 < msg, t_1 >)$$

$$A_2(a_2, MSG_2, RT_2 < msg, t_2 >)$$

$$\frac{\exists\, x, RT_1[x].msg \in MSG_2 \wedge \exists\, y, RT_2[y].msg \in MSG_1 \wedge \forall z, RT_2[y].t_2 \neq RT_1[z].msg}{A_1 \rightarrow A_2}$$

This type of relation corresponds to the situation when A_1 wants A_2'message and send itself's message to A_2 whereas A_2 reseive the message from A_1 and don't send back its message.

Slave Dependency Graph:
Given an *Actor System* $S =$ `Actors`, we define its *slave dependency graph* G_S whose nodes are the existing *Actors* and whose edges are slave dependency relationship defined above.

Restricted Communication deadlock:
An *Actor System* S has a restricted communication deadlock iff its *slave dependency graph* contains a cycle.

An example could be PingPong mentioned in the background information, in which PingPong can be expressed as follows:

$$A_{Ping}(Ping, \{Ping\_msg\}, \{< Pong\_msg, \text{“}ok\text{”} >\})$$

$$A_{Pong}(Pong, \{Pong\_msg\}, \{< Ping\_msg, \text{“}ok\text{”} >\})$$

Hence,

$$\frac{Pong_{msg}{=}Pong_{msg} \wedge Ping_{msg}{=}Ping_{msg} \wedge ''ok'' \neq Pong_{msg}}{A_{Ping} \rightarrow A_{Pong}}$$

$$\frac{Ping_{msg}{=}Ping_{msg} \wedge Pong_{msg}{=}Pong_{msg} \wedge ''ok'' \neq Ping_{msg}}{A_{Pong} \rightarrow A_{Ping}}$$

3.3 Slave Chain

After we generate a *slave dependency graph* for the actor model program, another algorithm will verify if the *slave dependency graph* has a cycle. Since the dependency graph is a collection of pairs of *Actors* in slave dependent relationship and we want to find a cycle in the graph, we merge these pairs into a chain called "Slave chain". Here, we consider these pairs as a short chain of two *Actors*. We merge the *Slave Chain* with the next short chain *iff* the tail of the *Slave Chain* equals the head of the next one. After all the possible merges are finished, we look at the final product of *Slave Chain*. If the head of the *Slave Chain* equals its tail, which means the *Actors* in the chain are in a cyclic slave dependency relationship, we consider that the *Actors* in this chain are in a restricted communication deadlock.
For example in *slave dependency graph* G_S:

$$A \to B \quad B \to C \quad C \to A$$

G_S contains a restricted communication deadlock. As B is the tail of the first pair as well as the head of the second pair, we merge them into a *Slave Chain*:

$$A \to B \to C \quad C \to A$$

We then merge the slave chain and the next pair for the same reason:

$$A \to B \to C \to A$$

Since the final *Slave Chain* $A \to B \to C \to A$ has its head and tail the same *Actor* A, and according to the theory above, the *Actors* in the chain, A, B, C, are in the state of restricted communication deadlock.

Continue, using the example of PingPong, as we have determined their slave dependency relationship: $A_{Ping} \to A_{Pong}, A_{Pong} \to A_{Ping}$, we find that the *Actor* at the tail of the first chain is also at the head of the second one. We then merge them together into $A_{Ping} \to A_{Pong} \to A_{Ping}$. Since the head is also the tail in this chain, we conclude that the *Actors* A_{Ping} and A_{Pong} are in restricted communication deadlock.

4. Implementation of the Analysis Theory

4.1 Pseudocode of Analysis Program

Algorithm 1 Psuedocode of internalTransform()

```
1: <List> statements = soot.getStatement()
2: for StatementSequence ss in statements do
3:     if ss contains pattern <@this x1 ; @parameter0 x2 ; new x3> then
4:         Add Pair(x2, x3) to RecvSendMapping of ActorGraph x1
5:     end if
6:     if ss contains x1.lambda$createreceive$<type x2...>" then
7:         Add Pair(x2, null) to actor x1's RecvSendMapping
8:     end if
9: end for
```

internalTransform() is automatically excuted by Soot, extracting statements within a class. In our implementation, we turn Soot to whole-program mode, so that Soot can exact statements from every classes of Actors.

a) In line 1, we transform every line of code being analyzed into a statement by using Soot. We then store them in a list for further analysis.

b) In Line 2, StatementSequence is a set of consecutive statements in a program.

c) In line 3, we use the pattern we found from the source code of .match() function. As long as StatementSequence match this pattern, we can determine *Actor*'s name and the receiving and sendiing of an actor, where the *Actor*'s name is after `@this`, the sending message is after `@parameter0` and receive message is after `new`.
For example, the output of Soot shows the source code of *Actor* Celcius:

```
[junit] r0 := @this:  test1.Celsius
[junit] r1 := @parameter0:  test1.AristotleAristotle_msg
[junit] $r3 = new test1.Celsius$Celsius_msg
```

d) In line 4, we add the receiving message and sending message pair to ***RecvSendMapping***, which is a list of pairs describing how the *Actor* will react to the incoming message. ***ActorGraph*** is a data structure generally describing the characteristic of an *Actor*, which is a class contains *Actor*'s' name and ***RecvSendMapping***.

e) In line 6, we also found a pattern to make sure we can find all of the receiving messages whether it sends the correct message or not. Here the predication means the *Actor* doesn't send the right message.

f) In line 7, we do the similar thing in line 4, whereas we add `null` to its send message.

Algorithm 2 Psuedocode of findDependency()

```
1: graphList = find all actors and create their ActorGraph
2: store RecvSendMapping in corresponding ActorGraph
3: for actor1, actor2 in graphList do
4:     if one of the actor1's recvMsgType matches actor2 then
5:         if one of the actor2's recvMsgType matches actor1 then
6:             if actor2 is not telling actor1 its recvMsg then
7:                 Add Pair(actor1, actor2) to slaveDependency
8:             end if
9:         end if
10:    end if
11: end for
12: for Pair(p1, p2) in slaveDependency do
13:    interval = new List();
14:    if p1.getValue() equals p2.getKey() then
15:        Add p1.getValue() to interval
16:        Add Pair(Pair(p1.getKey(),p2.getValue()),interval) to slaveChains
17:    end if
18: end for
19: for Pair p in slaveChains do
20:    if p.getKey() is a pair with the same key and value then
21:        Report p
22:    end if
23: end for
```

findDependency() is the method that search among *slave dependency graph* and reports restricted communication deadlocks. It is executed only once.

a) In line 1, we find all of the *Actors* by finding keyword `new` and `AbstractActor`, and store them into corresponding created ***ActorGraph***. ***graphList*** is a list of all ***ActorGraph*** in the *Actor System*.

b) In line 2, we store ***RecvSendMapping*** in corresponding ***ActorGraph***.

c) In lines 3-11, we check if the two *Actors* are in slave dependent relationship and add them to ***slaveDependency*** in a pair. ***slaveDependency*** is a list of pairs in which the key of the pair is the slave *Actor* and the value is the master *Actor*.

d) In line 13, we create a list ***interval*** in order to store the *Actors* that are being merged.

e) In line 15, we add the interval*Actor* to ***interval***.

f) In line 16 we add the merged slave chain to ***slaveChains***. ***slaveChains*** is a collection of pairs describing a series of *Actors* that are slave dependency relationship.

g) In line 19-23, if we eventually get a Pair in ***slaveChians*** that has Pair(a,a), which means a circle in our *slave dependency graph*, the program will report an error by showing all the *Actors* in this chain are in the state of restricted communication deadlock.

4.2 Restrictions

For the above analysis to be employed in an actor model program, the program must satisfy two restrictions:

Restriction 1:

The message must be sent by the *Actor* that it is belonged to. Although in Akka, messages can technically be sent by *Actors* that does not own that particular message. However, since we couldn't find the right code showing the receving message is sent by which *Actor*, we have to restrict the message being sent. If a message can only be sent by one *Actor*, it is much easier for us to trace back where the message is from.

Restriction 2:

In in each method match() of an *Actor* class, there is only one tell() function, whose caller of must be sender(). If there are multiple tell functions, no matter who the receiver is, we shall assume an *Actor* with two sender.tell(a). If the sender can reply to a, and the message it sent can also be replied by the first *Actor*, then it will do the two sender.tell(a) twice, because it received the same message twice. Thus, the *Actor*'s reply of the sender would increase exponentially with a base of two in this simplest base case. It would soon cause the entire program to go into a chaos either because of an infinite loop or recursion. It would be meaningless to write such codes in an *Actor*. Secondly, if we assume an *Actor* has a tell method with a caller "A", where A can react to the kind of message the *Actor* sent — it still can't imply that the message the *Actor* sent is strictly owned by itself. If we cannot identify it, we will not be sure whether the *Actor* A is dependent on the first *Actor* or the actual owner of the message or actually. Hence, it is also a meaningless piece of code.

4.3 Test Cases

Test Case 1:

Ping	Pong

Ping_msg
receive(Pong_msg, x → {
  sender().tell("ok")
})
	Pong_msg
	receive(Ping_msg, x → {
	  sender().tell("ok")
	})

Fig.2. Psuedocode of Test Case 1

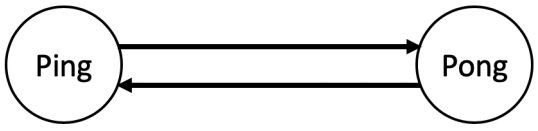

Fig.3. Slave Dependency Graph of Test Case 1

Test Result (20 Trails):

All Errors Reported	100%
Partial Errors Report	0%
No Error Report	0%

Test Case 2:

Aristotle	Beethoven	Celsius

A_msg
receive(B_msg, x →{
  sender().tell("ok")
})
	B_msg
	receive(C_msg, x →{
	  sender().tell("ok")
	})
	receive(E_msg, x→{
	  sender().tell(B_msg("ok"))
	})
		C_msg
		receive(D_msg, x →{
		  sender().tell(C_msg("ok"))
		})
		receive(A_msg, x →{
		  sender().tell("ok")
		})

Darwin	Einstein

D_msg
receive(A_msg, x →{
  sender().tell(D_msg("ok"))
})
receive(E_msg, x→{
  sender().tell("ok")
})
	E_msg
	receive(D_msg, x →{
	  sender().tell("ok")
	})
	receive(C_msg, x →{
	  sender().tell(E_msg("ok"))
	})

Fig.4. Psuedocode of Test Case 2

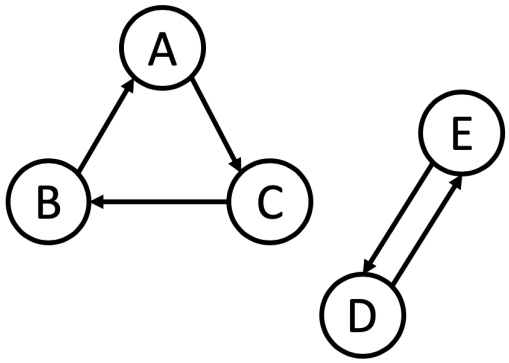

Fig.5. Slave Dependency Graph of Test Case 2

Test Result (20 Trails):

All Errors Report	75%
Partial Errors Report	15%
No Error Report	10%

5. Related Work and Conclusions

Deadlock detection in concurrency programs is a well-studied topic. Although most of those studies focus on the traditional thread-based concurrency model, some studies still have significant influences on us so that we decided to bring their ideas into our own study. Flores-Montoya et al. have presented an analysis that finds deadlocks in concurrency programs through dependency graph construction, and the deadlock situation occurs when the dependency graph contains a circle. Although our analysis also looks for deadlocks by finding a cyclic relation in a graph, we use a different graph, *Slave Dependency Graph*. On the other hand, the work of Flores-Montoya et al. pays more attention on the thread-based concurrency programs. Nevertheless, there exist some works that are directly related to communication deadlock detection in actor model programs. Maria Christakis and Konstantinos Sagonas proposed a static analysis that could detect both Communication deadlock and Behavioral deadlock in the Erlang language. The analysis detects deadlocks by generating a control flow graph and finds whether the graph is a closed figure or not. If the graph is not closed, the whole program is considered to have a deadlock[4]. This approach has demonstrated its feasibility by finding communication deadlocks in some open source libraries, but the approach we use is unique one.

We develop a novel dependency graph, *slave dependency graph*, in which we can find restricted communication deadlocks of an actor-based program, and demonstrate the algorithm's feasibility by the implementation of building an analysis program. The analysis program can detect a specific kind of restricted communication deadlock in an actor-based Java program with Akka toolkit.

There are two limitations in our research. The first one is about the imperfect precision of our analysis program. From the result of the test cases, we can see as the number of *Actors* in an *Actor System* increases, the percentage of failing report increases notably. This is due to the internalTransform() method provided by Soot, which extracts the key information of an *Actor* from the Java source code. It seems this method is run in multi-threaded for analyzing all actor classes, which may cause the pattern we found fails to construct ***ActorGraph*** with right information. Thus, the analysis program cannot report all the restricted communication deadlocks correctly from time to time. According to the test result, we believe the percentage of failing report would increase significantly, and we would have a smaller possibility to identify all the restricted communication deadlocks in a huge *Actor System*. Therefore, our analysis program is more suitable for small *Actor System*. The second limitation is due to our narrow definition of restricted communication deadlock. Since we have such a strict definition for restricted communication deadlock, the slave dependency relationship ignores situations. For example, an *Actor* receives a message and send its message to another *Actor*. This restriction of definition makes our program only able to perform a limit under-approximation analysis of communication deadlock in an *Actor System*.

References

1 Agha, Gul A. *Actors: A Model Of Concurrent Computation In Distributed Systems*. MIT Artificial Intelligence Laboratory, 1987.

2 Albert, Elvira et al. "May-Happen-In-Parallel Analysis For Actor-Based Concurrency". *ACM Transactions On Computational Logic*, vol 17, no. 2, 2015, pp. 1-39. *Association For Computing Machinery (ACM)*, doi:10.1145/2824255.

3 Flores-Montoya, Antonio E. et al. "May-Happen-In-Parallel Based Deadlock Analysis For Concurrent Objects". *Formal Techniques For Distributed Systems*, 2013, pp. 273-288. *Springer Berlin Heidelberg*, doi:10.1007/978-3-642-38592-6_19.

4 Maria Christakis and Konstantinos Sagonas. Static Detection of Deadlocks in Erlang. In Draft Proceedings of the Twelfth International Symposium on *Trends in Functional Programming (TFP'11)*, pages 62-76, 2011. *Department of Computer Systems and Computing, Universidad Complutense de Madrid.*

5 Torres Lopez, Carmen et al. "A Study Of Concurrency Bugs And Advanced Development Support For Actor-Based Programs". *Lecture Notes In Computer Science*, vol 10789, 2018, pp. 155-185. *Springer International Publishing*, doi:10.1007/978-3-030-00302-9_6.