Verification of complex systems in Stainless

Romain Ruetschi

Version 0.1
December 2017

Abstract

(TODO: Abstract)

Master Thesis Project under the supervision of
Prof. Viktor Kuncak & Dr. Jad Hamza
Lab for Automated Reasoning and Analysis LARA - EPFL

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Contents

1 Introduction

(TODO: Introduction)

Related Works

(TODO: Related Works)

2 Motivation

(TODO: Motivation)

3 Verifiying Actor Systems

3.1 The Actor Model

(TODO: Actor Model)
[1]

3.1.1 Message

In our framework, messages are modelled as instances of the `Msg` trait.

```
abstract class Msg
```

3.1.2 Actor Reference

Each actor is associated with a unique and persistent reference, modelled as an instance of the `ActorRef` trait.

```
abstract class ActorRef
```

3.1.3 In-flight Messages

In-flight messages are represented as a product of the `ActorRef` of the destination actor, and the message itself.

```
case class Packet(dest: ActorRef, payload: Msg)
```

3.1.4 Actor Context

When a message is delivered to an actor, the latter is provided with a context, which holds a reference to itself, and a mutable list of `Packet`s to send.

```
case class ActorContext(
  self: ActorRef,
  var toSend: List[Packet]
)
```

3.1.5 Behavior

A behavior specifies both the current state of an actor, and how this one should process the next incoming message. In our framework, these are modelled as a subclass of the abstract class `Behavior`, which defines a single abstract method `processMsg`, to be overriden for each defined behavior.

Using the provided `ActorContext`, the implementation of the `processMsg` method can both access its own reference, and register messages to be sent after the execution of the method is complete. It is also required to return a new `Behavior`

```
abstract class Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior
}
```

3.1.6 Transition

Whenever a message is processed, we record the transition between the previous state of the system and the one after the message has been been processed. We keep track of this information as an instance of the `Transition` class, which holds: the `Msg` that has been delivered, its sender and receiver, the new behavior of the destination actor, and the list of `Packet`s the destination actor wants to send.

```
case class Transition(
  from: ActorRef,
  to: ActorRef,
  msg: Msg,
  newBehavior: Behavior,
  toSend: List[Packet]
)
```

3.1.7 Actor System

The state of the Actor system at a given point in time is modelled as a case class, holding the behavior associated with each actor reference, the list of in-flight messages between any two actors, as well as a trace of the execution up to that point, modelled as a list of `Transition`s.

```
case class ActorSystem(
  behaviors: CMap[ActorRef, Behavior],
  inboxes: CMap[(ActorRef, ActorRef), List[Msg]],
  trace: List[Transition]
)
```

The `ActorSystem` class is equipped with a `step` method, which takes a pair of `ActorRef` as arguments, and is in charge of delivering the oldest message found in the corresponding inbox, and which returns the new state of the system after the aforementioned message has been processed.

```
def step(from: ActorRef, to: ActorRef): ActorSystem
```

3.2 Operational Semantics

We formulate the small-step operational semantics of our Actor model in Figure 1, where s : `ActorSystem` is an Actor system, m : `Msg` is a message, n, n_{to}, n_{from} : `ActorRef` are references, b, b' : `Behavior` are behaviors, ps : `List[Packet]` a list of packets to send, t : `Transition` is a transition, c : `ActorContext` is a context, and $\emptyset_n$: `ActorContext` is the empty context for an actor whose self-reference is n, defined as $\emptyset_n := \texttt{ActorContext}(n, \texttt{Nil})$.

$$\frac{\nexists m \in s.\texttt{inboxes}(n_{from}, n_{to})}{\langle s.\texttt{step}(n_{from}, n_{to})\rangle \longrightarrow \texttt{s}} \quad \text{(STEP-NOMSG)}$$

$$\frac{\exists m \in s.\texttt{inboxes}(n_{from}, n_{to}) \qquad \langle s.\texttt{deliverMsg}(n_{to}, n_{from}, m)\rangle \Rightarrow (b, ps, t)}{\langle s.\texttt{step}(n_{from}, n_{to})\rangle \longrightarrow s \uplus (n_{to} \mapsto b, \ldots, t)} \quad \text{(STEP)}$$

$$\frac{\langle s.\texttt{behaviors}(n_{to}).\texttt{processMsg}(m, \emptyset_{n_{to}})\rangle \longrightarrow (b, c)}{\langle s.\texttt{deliverMsg}(n_{to}, n_{from}, m)\rangle \longrightarrow (b, c.\texttt{toSend}, t)} \quad \text{(DELIVER-MSG)}$$

$$\frac{b.\texttt{processMsg}(m, \emptyset_{n_{to}}) = [i_1, \ldots, i_n, b'] \qquad \emptyset_{n_{to}} \vdash \langle [i_1, \ldots, i_n]\rangle \Rightarrow c}{\langle b.\texttt{processMsg}(m, \emptyset_{n_{to}})\rangle \longrightarrow (b', c)} \quad \text{(PROCESS-MSG)}$$

$$\frac{}{\langle \texttt{Nil}, c\rangle \Rightarrow c} \quad \text{(I-NIL)}$$

$$\frac{\langle i, c\rangle \Rightarrow c'}{\langle i \;\texttt{::}\; is, c\rangle \Rightarrow \langle is, c'\rangle} \quad \text{(I-CONS)}$$

$$\frac{}{\langle n \;\texttt{!}\; m, c\rangle \Rightarrow (b',\ c.\texttt{copy}(\texttt{toSend} \mapsto \texttt{(}n, m\texttt{)} \;\texttt{::}\; c.\texttt{toSend}))} \quad \text{(I-SEND)}$$

Figure 1: Operational semantics

3.3 Proving Invariants

After having defined an Actor system with our framework, one might want to verify that this system preserves some invariant between each step of its execution. That is to say, for an `ActorSystem` s, any two `ActorRef` n, m, and an invariant `inv: ActorSystem` $\rightarrow$ **`Boolean`**, if `inv`(s) holds, then `inv`$(s.$`step`$(n, m))$ should hold as well. We express this property more formally in Figure 2.

$$\forall s : \texttt{ActorSystem},\ n : \texttt{ActorRef},\ m : \texttt{ActorRef}.\ \texttt{inv}(s) \implies \texttt{inv}(s.\texttt{step}(n, m))$$

Figure 2: Invariant preservation property

This property can be trivially expressed in PureScala as:

```
def inv(s: ActorSystem): Boolean = {
  /* ... */
}

def preserveInv(s: ActorSystem, n: ActorRef, m: ActorRef): Boolean = {
  require(inv(s))
```

```
  inv(s.step(n, m))
} holds
```

When encoutering such a definition, Stainless will generate a verification condition equivalent to Figure 2, which will then be discharged to Inox and the underlying SMT solver.

3.4 Reasoning About Traces

(TODO: Traces)

3.5 Case studies

Replicated Counter (increment)

As a first and very simple case study, we will study an Actor system which models a replicated counter, which can only be incremented by one unit. This system is composed of two actors, a primary counter whose reference is `Primary()`, and a backup counter whose reference is `Backup()`. Each of these reference is associated with a behavior: the primary counter reference with an instance of `PrimaryB`, and the backup counter reference with an instance of `BackupB`, both of which hold an integer, representing the value of the counter. Whenever the primary actor receives a message `Inc()`, it forwards that message to the backup actor, and returns a new instance of `PrimaryB` with the counter incremented by one. When the backup actor receives an `Inc()` message, it just returns a new instance of `BackupB` with the counter incremented by one. The corresponding PureScala implementation can be found in Listing 1.

```
case class Primary() extends ActorRef
case class Backup()  extends ActorRef

case class Inc() extends Msg

case class PrimaryB(counter: BigInt) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Inc() =>
      Backup() ! Inc()
      PrimaryB(counter + 1)
  }
}

case class BackupB(counter: BigInt) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Inc() => BackupB(counter + 1)
  }
}
```

Listing 1: Replicated counter implementation (increment)

```scala
case class Primary() extends ActorRef
case class Backup()  extends ActorRef

case class Inc() extends Msg
case class Deliver(c: BigInt) extends Msg

case class PrimaryB(counter: BigInt) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Inc() =>
      Backup() ! Deliver(counter + 1)
      PrimaryB(counter + 1)

    case _ => Behavior.same
  }
}

case class BackupB(counter: BigInt) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Deliver(c) => BackupB(c)
    case _          => Behavior.same
  }
}
```

Listing 2: Replicated counter implementation (deliver)

Replicated Counter (deliver)

Lock Service

Listing 3

Leader Election

(TODO: Leader election)

Key-value store

(TODO: KV store)

3.6 Executing an Actor System with Akka

4 Strong Eventual Consistency with CRDTs

(TODO: CRDTs)

5 Biparty Communication Protocols

(TODO: Bipart)

6 Conclusion

(TODO: Conclusion)

7 Future Work

(TODO: Future Work)

```scala
case class Server() extends ActorRef
object Server {
  case class Lock(agent: ActorRef) extends Msg
  case class Unlock(agent: ActorRef) extends Msg
}

case class Agent(id: BigInt) extends ActorRef
object Agent {
  case object Lock extends Msg
  case object Unlock extends Msg
  case object Grant extends Msg
}

// The head of ‘agents‘ holds the lock, the tail are waiting for the lock
case class ServerB(agents: List[ActorRef]) extends Behavior {
  def isLocked: Boolean = agents.nonEmpty

  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Server.Lock(agent) if agents.isEmpty =>
      agent ! Agent.Grant
      ServerB(List(agent))

    case Server.Lock(agent) =>
      ServerB(agents :+ agent)

    case Server.Unlock(agent) if agents.nonEmpty =>
      val newAgents = agents.tail
      if (newAgents.nonEmpty) newAgents.head ! Agent.Grant
      ServerB(newAgents)

    case _ =>
      Behavior.same
  }
}

case class AgentB(holdsLock: Boolean) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Agent.Lock =>
      Server() ! Server.Lock(ctx.self)
      Behavior.same

    case Agent.Unlock if holdsLock =>
      Server() ! Server.Unlock(ctx.self)
      AgentB(false)

    case Agent.Grant =>
      AgentB(true)

    case _ =>
      Behavior.same
  }
}
```


Listing 3: Replicated counter implementation (deliver)

A References

[1] S. Yasutake and T. Watanabe, “Actario: A framework for reasoning about actor systems,” in *Workshop on Programming based on Actors, Agents, and Decentralized Control*, AGERE 2015, 2015.