Verification of complex systems in Stainless

Romain Ruetschi

Version 0.1
December 2017

Abstract

(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

(Introduction)

Related Works

(Related Works)

2 Motivation

(Motivation)

3 Verifiying Actor Systems

3.1 The Actor Model

(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 **??**.

(Operational semantics)

$$\emptyset_n := \texttt{ActorContext}(n, \emptyset)$$

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

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

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

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

$$\frac{}{\langle \texttt{Nil}, c\rangle \Rightarrow c} \tag{I-NIL}$$

$$\frac{\langle i, c\rangle \Rightarrow c'}{\langle i \;\texttt{::}\; is, c\rangle \Rightarrow \langle is, c'\rangle} \tag{I-CONS}$$

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

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` → **`Boolean`**:

$$\frac{\texttt{inv}(s)}{\texttt{inv}(s.\texttt{step}(n, m))} \tag{PRESERVE-INV}$$

3.4 Reasoning About Traces

(Traces)

3.5 Executing an Actor System with Akka

4 Strong Eventual Consistency with CRDTs

(CRDTs)

5 Biparty Communication Protocols

(Bipart)

6 Conclusion

(Conclusion)

7 Future Work

(Future Work)

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.