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)

2 Program Verification with Stainless

(TODO: Describe Stainless/Inox pipeline)

3 Symbolic Evaluation of Inox Programs

3.1 Motivation

While one would expect the following program to successfully verify in a fair amount of time, Stainless is actually unable to automatically prove the verification condition corresponding to `test`'s postcondition.

```scala
def foldLeft[A](list: List[A], z: B)(f: (B, A) => B): B = list match {
  case Nil() => z
  case Cons(x, xs) => foldLeft(xs, f(z, x))(f)
}

def insert[A, B](values: List[(A, B)], map: Map[A, B]) = {
  foldLeft(xs, map) {
    case (acc, (k, v)) => acc.updated(k, v)
  }
}

def test(map: Map[String, Int]): Boolean = {
  val xs = List("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
  val res = insert(xs, map)
  res("b") == 2
}.holds
```

On the other hand, if one were to manually unfold `insert`, then unfold `foldLeft` as much as possible in the program above, one would end up with the following definition for the `test` function, the postcondition of which would then promptly verify.

```scala
def test(map: Map[String, Int]): Boolean = {
  val res = map.updated("a", 1).updated("b", 2).updated("c", 3).updated("d", 4)
  res("b") == 2
}.holds
```

Indeed, while the `map` variable is kept abstract, the list of values `xs` we want to insert is not, and it is thus possible to simplify the initial program, yielding to a much simpler verification condition.

It is of course unreasonable to ask users to manually perform such a transformation, especially with large programs, hence why we decided to investigate a way to do so automatically, as part of the verification pipeline.

(TODO: ...)

Let's first take a look at how the *PureScala* program above looks when expressed in Inox' input language:

```scala
def foldLeft[A](list: List[A], z: B)(f: (B, A) => B): B =
  if (list.isInstanceOf[Nil[A]]) {
    z
  } else {
    val x  = list.asInstanceOf[Cons[A]].head
    val xs = list.asInstanceOf[Cons[A]].tail
    foldLeft(xs, f(z, x))(f)
  }
}

def insert[A, B](values: List[(A, B)], map: Map[A, B]) = {
  foldLeft(xs, map, (acc: Map[A, B], kv: (A, B)) => acc.updated(kv._1, kv._2))
}

def test(map: Map[String, Int]): Boolean = {
  val xs = List("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
  val res = insert(xs, map)
  res("b") == 2
}.holds
```

3.2 Semantics

The symbolic evaluator keeps track of the current *path condition*, denoted by Δ. Figure 1 lists the evaluation rules for the symbolic evaluator. Since most of those are fairly straightforward, we will only focus on rules (21), (22) and (23), which pertain to functions invocations.

3.3 Case Studies

(TODO: Case Studies)

3.4 Conclusion

(TODO: Conclusion)

$$\frac{e \in \Delta}{\llbracket e\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{1}$$

$$\frac{\neg e \in \Delta}{\llbracket e\,;\,\Delta \rrbracket \longrightarrow \texttt{false}} \tag{2}$$

$$\frac{}{\llbracket \lambda x_1, \ldots, x_n.\, e\,;\,\Delta \rrbracket \longrightarrow \lambda \llbracket x_1\,;\,\Delta \rrbracket, \ldots, \llbracket x_n\,;\,\Delta \rrbracket.\, \llbracket e\,;\,\Delta \rrbracket} \tag{3}$$

$$\frac{}{\llbracket l \;\texttt{==>}\; r\,;\,\Delta \rrbracket \longrightarrow \llbracket \neg l \;\texttt{||}\; r\,;\,\Delta \rrbracket} \tag{4}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow \texttt{true}}{\llbracket \texttt{if (}c\texttt{) } t \texttt{ else } e\,;\,\Delta \rrbracket \longrightarrow \llbracket t\,;\,\Delta \cup c \rrbracket} \tag{5}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow \texttt{false}}{\llbracket \texttt{if (}c\texttt{) } t \texttt{ else } e\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \cup c \rrbracket} \tag{6}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow c' \qquad \llbracket t\,;\,\Delta \cup c' \rrbracket \longrightarrow t' \qquad \llbracket e\,;\,\Delta \cup \neg c' \rrbracket \longrightarrow e' \qquad t' = e'}{\llbracket \texttt{if (}c\texttt{) } t \texttt{ else } e\,;\,\Delta \rrbracket \longrightarrow t'} \tag{7}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow c' \qquad \llbracket t\,;\,\Delta \cup c' \rrbracket \longrightarrow t' \qquad \llbracket e\,;\,\Delta \cup \neg c' \rrbracket \longrightarrow e' \qquad t' \neq e'}{\llbracket \texttt{if (}c\texttt{) } t \texttt{ else } e\,;\,\Delta \rrbracket \longrightarrow \texttt{if (}c'\texttt{) } t' \texttt{ else } e'} \tag{8}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow \texttt{true}}{\llbracket \texttt{assume(}p\texttt{, } e\texttt{)}\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \rrbracket} \tag{9}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow \texttt{false}}{\llbracket \texttt{assume(}p\texttt{, } e\texttt{)}\,;\,\Delta \rrbracket \longrightarrow \texttt{assume(false, } \llbracket e\,;\,\Delta \rrbracket} \tag{10}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow p'}{\llbracket \texttt{assume(}p\texttt{, } e\texttt{)}\,;\,\Delta \rrbracket \longrightarrow \texttt{assume(}p'\texttt{, } \llbracket e\,;\,\Delta \cup p' \rrbracket\texttt{)}} \tag{11}$$

$$\frac{\texttt{T}_2\texttt{: ADTType} \qquad \neg\texttt{isSort(T}_2\texttt{)}}{\llbracket \texttt{C(T}_1\texttt{, } a_1, \ldots, a_n\texttt{).isInstanceOf[T}_2\texttt{]}\,;\,\Delta \rrbracket \longrightarrow \texttt{T}_1\texttt{.id == T}_2\texttt{.id}} \tag{12}$$

$$\frac{\texttt{T: ADTType} \qquad \texttt{isSort(T)}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{13}$$

$$\frac{\texttt{T: ADTType} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e' \qquad \texttt{isInstanceOf(e', T, }\Delta\texttt{) == Some(}b\texttt{)}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow b} \tag{14}$$

$$\frac{\texttt{T: ADTType} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e' \qquad \texttt{isInstanceOf(}e'\texttt{, T, }\Delta\texttt{) == None}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow e'\texttt{.isInstanceOf[T]}} \tag{15}$$

$$\frac{\texttt{T: ADTType}}{\llbracket e\texttt{.asInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \rrbracket\texttt{.asInstanceOf[T]}} \tag{16}$$

Figure 1: Operational semantics of the symbolic evaluator

$$\frac{\text{(TODO: This is overly simplistic)}}{[\![\, \texttt{let x:T = } v \texttt{ in } e \,;\, \Delta \,]\!] \longrightarrow [\![\, e[\mathsf{x}/v] \,;\, \Delta \,]\!]} \tag{17}$$

$$\frac{}{[\![\, \neg\, e \,;\, \Delta \,]\!] \longrightarrow \neg\, [\![\, e \,;\, \Delta \,]\!]} \tag{18}$$

$$\frac{[\![\, f \,;\, \Delta \,]\!] \longrightarrow \lambda \mathsf{x}_1 \texttt{:} \mathsf{T}_1 \texttt{,...,} \mathsf{x}_n \texttt{:} \mathsf{T}_n \,.\, b \qquad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow [\![\, e'_i \,;\, \Delta \,]\!],\, i \in \{1 \dots n\}}{[\![\, f(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow [\![\, b[\mathsf{x}_1/e'_1, \dots, \mathsf{x}_n/e'_n] \,;\, \Delta \,]\!]} \tag{19}$$

$$\frac{[\![\, f \,;\, \Delta \,]\!] \longrightarrow f' \qquad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow [\![\, e'_i \,;\, \Delta \,]\!],\, i \in \{1 \dots n\}}{[\![\, f(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow f'(e'_1, \dots, e'_n)} \tag{20}$$

$$\frac{\neg\, \texttt{isRecursive(id)} \quad \texttt{id.params} = \langle \mathsf{x}_1, \dots, \mathsf{x}_n \rangle \quad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow [\![\, e'_i \,;\, \Delta \,]\!],\, i \in \{1 \dots n\}}{[\![\, \texttt{id}(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow [\![\, \texttt{id.body}[\mathsf{x}_1/e'_1, \dots, \mathsf{x}_n/e'_n] \,;\, \Delta \,]\!]} \tag{21}$$

$$\frac{\texttt{id.body} \Downarrow \Delta \uplus \{\, \mathsf{x}_i \mapsto e'_i \,|\, 1 \leq i \leq n \,\} \quad \texttt{id.params} = \langle \mathsf{x}_1, \dots, \mathsf{x}_n \rangle \quad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow [\![\, e'_i \,;\, \Delta \,]\!],\ i \in \{1 \dots n\}}{[\![\, \texttt{id}(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow [\![\, \texttt{id.body}[\mathsf{x}_1/e'_1, \dots, \mathsf{x}_n/e'_n] \,;\, \Delta \,]\!]} \tag{22}$$

$$\frac{[\![\, e_i \,;\, \Delta \,]\!] \longrightarrow [\![\, e'_i \,;\, \Delta \,]\!],\, i \in \{1 \dots n\}}{[\![\, \texttt{id}(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow \texttt{id}(e'_1, \dots, e'_n)} \tag{23}$$

Figure 1: Operational semantics of the symbolic evaluator

4 Verifiying Actor Systems

4.1 Motivation

(TODO: Motivation)

4.2 The Actor Model

(TODO: Actor Model)

4.3 Our Framework

Message

In our framework, messages are modelled as constructors of the `Msg` abstract class.

```
abstract class Msg
case class Hello(name: String) extends Msg
```

Actor Reference

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

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

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)
```

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]
)
```

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
}
```

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, and the list of in-flight messages between any two actors.

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

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
```

4.4 Operational Semantics

We formulate the small-step operational semantics of our Actor model in Figure 2, 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, c : `ActorContext` is a context, and $\emptyset_n$: `ActorContext` is the empty context for an actor whose self-reference is n, defined as $\emptyset_n :=$ `ActorContext`$(n,$ `Nil`$)$.

4.5 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 3. This property can be easily expressed in PureScala, as shown in Listing 1.

When encoutering such a definition, Stainless will generate a verification condition equivalent to Figure 3, which will then be discharged to Inox and the underlying SMT solver.

4.6 Case studies

Replicated Counter 1

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()`.

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

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

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

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

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

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

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

Figure 2: Operational semantics

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

Figure 3: Invariant preservation property

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

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

Listing 1: Invariant preservation theorem in PureScala

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 a positive 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 2.

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

case class Inc() extends Msg

case class PrimaryB(counter: BigInt) extends Behavior {
  require(counter >= 0)

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

case class BackupB(counter: BigInt) extends Behavior {
  require(counter >= 0)

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

Listing 2: Replicated counter implementation (increment)

Given such a system, one might want to prove that the following invariant is preserved between each step of its execution:

```scala
def invariant(s: ActorSystem): Boolean = {
  s.inboxes((Backup(), Backup())).isEmpty && {
    (s.behaviors(Primary()), s.behaviors(Backup())) match {
      case (PrimaryB(p), BackupB(b)) =>
        p.value == b.value + s.inboxes(Primary() -> Backup()).length
      case _ => false
    }
  }
}
```

Listing 3: Replicated counter invariant (increment)

This invariant specifies that the `Backup()` actor does not send itself any messages, that both actors have the proper corresponding behavior, and that, last but not least, the value of the primary counter is equal to the value of the backup counter added to the number of messages that are yet to be delivered to the backup actor.

```
def preserveInv(s: ActorSystem, n: ActorRef, m: ActorRef): Boolean = {
  require(invariant(s))
  invariant(s.step(n, m))
} holds
```

Listing 4: Replicated counter theorem (increment)

We can now define the actual theorem we want Stainless to prove for us:
(TODO: Rep Counter Inc Result)

Replicated Counter 2

Listing 5 shows a variant of the previous case study, where instead of having the primary actor forward the `Inc()` message to the backup actor, it instead delivers it with its new value.

```
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 5: Replicated counter implementation (deliver)

The invariant now reads slightly differently, as can be seen in Listing 6, and can no longer be proven automatically by Stainless on its own. Listing 5 shows the full proof.
(TODO: Rep Counter Del Result)

4.7 Spawning Actors

(TODO: Name uniqueness)

```
def invariant(s: ActorSystem): Boolean = {
  validBehaviors(s)                           &&
  noMsgsToSelf(Primary()).isEmpty             &&
  noMsgsToSelf(Backup()).isEmpty              &&
  noMsgsToSelf(Backup() -> Primary()).isEmpty && {
    val PrimBehav(p) = s.behaviors(Primary())
    val BackBehav(b) = s.behaviors(Backup())
    val bInbox       = s.inboxes(Primary() -> Backup())

    p.value >= b.value && isSorted(bInbox) && bInbox.forall {
      case Deliver(Counter(i)) => p.value >= i
      case _                   => true
    }
  }
}
```

Listing 6: Replicated counter implementation (deliver)

Up until now, our framework has only been able to model Actor systems with a static topology, ie. systems where no new actors besides the ones that are statically defined can be spawned. Let's now attempt to enrich our model to account for dynamic topologies.

To this end, we modify the `ActorRef` definition to include both a name and an optional field holding a reference to its parent `ActorRef` if any. We also add a new constructor of the `ActorRef` data type, which will be assigned to actors spawned from another actor.

```
abstract class ActorRef(
  name: String,
  parent: Option[ActorRef]
)

case class Child(name: String, getParent: ActorRef)
  extends ActorRef(name, Some(getParent))
```

In order for actors to spawn other actors, by specifying their name and associated initial behavior, we modify the `ActorContext` class as follows:

```
case class ActorContext(
  self: ActorRef,
  var toSend: List[Packet],
  var toSpawn: List[(ActorRef, Behavior)]
) {
  def spawn(behavior: Behavior, name: String): ActorRef = {
    val id: ActorRef = Child(name, self)
    toSpawn = toSpawn :+ (id, behavior)
    id
  }
  /* ... */
}
```

As can be seen in the listing above, the context now keeps track of the names and behaviors of the actors to be spawned, and provides a `spawn` method which is in charge of constructing the `ActorRef` of the spawned actor, storing it along with the behavior within the context, and returning the newly generated reference.

Let's now update the case study in Listing 2 to accommodate these changes, while noting that we are not making use of this new feature yet. Only the two `ActorRef` definitions need to be touched, becoming:

```scala
case class Primary() extends ActorRef("primary", None())
case class Backup()  extends ActorRef("backup", None())
```

Unfortunately, when we now feed the updated benchmark to Stainless, the latter is be unable to prove the very same theorem it previously had no issue whatsoever with. All is not lost though, as turning on the symbolic evaluator described in Section 3 enables Stainless to verify the program in less than 10 seconds.

Listing **??** defines a simple system with a dynamic topology, where one actor, deemed `Primary`, waits for a `Spawn` message to spawn a child actor, and change its behavior from `BeforeB` to `AfterB` in order to keep track of the reference to the child. The invariant we would to verify holds here, states that, if the `Primary` actor has behavior `BeforeB()`, then the behavior associated with the `ActorRef` of its child actor must be `Stopped`. On the other hand, if the `Primary` actor has behavior `AfterB(child)`, then the behavior associated with `child` must be `ChildB`. This test case verifies promptly, provided the symbolic evaluator is enabled.

4.8 Executing an Actor System with Akka

While the verification of Actor systems is in itself an interesting endeavour, it is not of much use unless one is able to run these systems, potentially in a distributed environment. In the Scala ecosystem, the most widely used real-world Actor system is Akka (TODO: REF). Listing 7 shows a shallow shim which allows to run an Actor system developed with our framework within Akka, with only a few alterations to the original program.

(TODO: Explain shim)

To this end, one must define a subclass of `ActorSystem`, and provide an implementation of its `run` method. Within this method, one can spawn new top-level actors, get a reference to those, and send them messages. Listing 8 shows such an implementation for the replicated counter described in Section 4.6.

```
case object Primary extends ActorRef("primary")
case object Spawn extends Msg

case class BeforeB() extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Spawn =>
      val child = ctx.spawn(ChildB(), "child")
      AfterB(child)
  }
}

case class AfterB(child: ActorRef) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case _ => Behavior.same
  }
}

case class ChildB() extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case _ => Behavior.same
  }
}

def invariant(s: ActorSystem): Boolean = {
  s.behaviors(Primary) match {
    case BeforeB() =>
      s.isStopped(Child("child", Primary()))
    case AfterB(child) =>
      s.behaviors(child) == ChildB()

    case _ => false
  }
}

def theorem(s: ActorSystem, from: ActorRef, to: ActorRef): Boolean = {
  require(invariant(s))
  invariant(s.step(from, to))
} holds
```

```scala
import akka.actor

type ActorRef = actor.ActorRef

case class ActorContext(self: actor.ActorRef, ctx: actor.ActorContext)

class Wrapper(var behavior: Behavior)
  extends actor.Actor with actor.ActorLogging {

  implicit val ctx = ActorContext(self, context)

  def receive = {
    case msg: Msg =>
      log.info(s"${behavior}: ${msg}")
      behavior = behavior.processMsg(msg)

    case _ => ()
  }
}

abstract class ActorSystem(val name: String) {
  lazy val system = actor.ActorSystem(name)

  def spawn(behavior: Behavior, name: String): actor.ActorRef = {
    system.actorOf(actor.Props(new Wrapper(behavior)), name = name)
  }

  def run(): Unit
}
```

Listing 7: Akka shim for our Actor system framework

```scala
@extern
object System extends ActorSystem("rep-counter-sys") {
  def run(): Unit = {
    val backup  = spawn(BackupB(0), "backup")
    val primary = spawn(PrimaryB(0, backup), "primary")

    primary ! Inc()
  }
}

@extern
def main(args: Array[String]): Unit = {
  System.run()
}
```

Listing 8: Akka shim for our Actor system framework

5 Biparty Communication Protocols

5.1 Motivation

(TODO: Motivation)

5.2 Session Types

5.3 Session Types in PureScala

5.4 Linear Types

5.5 Case Studies

5.6 Conclusion

6 Conclusion

(TODO: Conclusion)

7 Future Work

(TODO: 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.

[2] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, “Verdi: A framework for implementing and formally verifying distributed systems,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’15, (New York, NY, USA), pp. 357–368, ACM, 2015.

[3] G. Agha and P. Thati, *An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language*, pp. 26–57. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[4] C. Hewitt and H. Baker, “Applications of the laws for communicating parallel processes,” in *IFIP Working Conf. on the Formal Desc. of Prog. Concepts*, 1977.

[5] J. He, P. Wadler, and P. Trinder, “Typecasting actors: From akka to takka,” in *Proceedings of the Fifth Annual Scala Workshop*, SCALA ’14, (New York, NY, USA), pp. 23–33, ACM, 2014.

[6] P. Haller and A. Loiko, “Lacasa: Lightweight affinity and object capabilities in scala,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 272–291, ACM, 2016.

[7] P. Haller and F. Sommar, “Towards an Empirical Study of Affine Types for Isolated Actors in Scala,” *ArXiv e-prints*, Apr. 2017.

[8] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

[9] R. Neykova and N. Yoshida, “Multiparty session actors,” *Logical Methods in Computer Science*, vol. 13, no. 1, 2017.

[10] A. Scalas and N. Yoshida, “Lightweight Session Programming in Scala,” in *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (S. Krishnamurthi and B. S. Lerner, eds.), vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 21:1–21:28, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[11] P. Wadler, “Propositions as sessions,” in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’12, (New York, NY, USA), pp. 273–286, ACM, 2012.

[12] P. Wadler, “Linear types can change the world!,” in *PROGRAMMING CONCEPTS AND METHODS*, North, 1990.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[14] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *CoRR*, vol. abs/1610.00502, 2016.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, *Conflict-Free Replicated Data Types*, pp. 386–400. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

[18] M. Orchard and N. Yoshida, *Session Types with Linearity in Haskell.* River publishers, 2017.