ReActor: A notation for the specification of actor systems and its semantics*

Rodger Burmeister

rodger.burmeister@tu-berlin.de

Abstract: With the increasing use of the actor model in concurrent programming there is also an increased demand in precise design notations. Precise notations enable software engineers to rigorously specify and validate the non-deterministic behavior of concurrent systems. Traditional design notations are either imperative, too concrete, or do not support the actor model. In this paper, we present a new, TLA-inspired specification language called ReActor that supports a declarative style of specification and selected programming language features in combination. For ReActor a precise operational semantics is defined in terms of action interleavings. We propose ReActor to be used in abstract design specification and as a supplement to existing design notations, especially if a sound notion of concurrent objects is required.

1 Introduction

The paper presents a supplemental notation for describing actor systems and their behavior in a rigorous way. The notation comes with a precise semantics and supports a declarative style of description that allows abstract design specifications.

Actor systems consist of concurrent objects that are called actors. They can be efficiently executed on parallel and distributed systems due to their concurrent nature. Each actor has its own exclusive state that either is represented by variables (e.g. Scala) or tail recursive function parameters (e.g. Erlang). Variables and parameters can reference other actors. These actor references can change over time (dynamic object topology). Actors can activate other actors by sending them asynchronous messages. An actor processes its messages sequentially one at a time. The effect of a message is described by sequential statements. Statements can alter the state of an actor, create new actors, send new messages, or enable termination.

While the actor model is basically the same in all actor languages there are differences in the way messages are handled. Formal notations like Rebeca [SMdB05], Temporal Actor Logic [Sch01], Algebra of Actors [GZ01], or Simple Actor Language [AT04, AMST97, Agh86] model pending messages as sets and allow to process them in any order. Programming languages like Erlang [Arm07] and Scala [OAC+04] use queues and simplify the implementation of sequential protocols by features that we call *selective receive* and

*Preparation of this work was supported in part by German Research Foundation (DFG) grant HE 6088/1-1.

relative message order preservation. The former provides a receiving actor with the ability to define guards for picking a matching message from the pending ones. The latter feature ensures that messages sent from one actor to another actor are transmitted (but not necessarily processed) in their sending order.

Formal actor notations with their generic actor model can be used to specify and verify actor models and their properties. Programming features like *selective receive* and *relative message order preservation* are usually not supported but simplify the implementation of sequential protocols. Without these features additional acknowledge-messages are necessary to establish a defined message order. Adding *selective receive* and *relative message order preservation* to formal specification languages would not only improve the readability of a specification but also slim the interfaces of the actors.

Another shortcoming of existing actor notations is the imperative style of description. Messages are described as concrete sequences of statements. In early phases of software design such detailed descriptions are not available. Describing messages in a more abstract way would leave implementation details open to the programmer and later design phases.

In our previous work, we presented an abstract model of an actor-based observer pattern in TLA$^+$ and used the model checker TLC to verify essential safety and liveness properties [BH12]. While our model was perfectly able to express the pattern's logic it was hard to keep it seperate from the underlying actor model with its features as TLA$^+$ has no explicit support for objects or message passing.

In this paper, we present a declarative notation for the specification of actor systems called ReActor that improves this situation. It "hides" redundant details of the actor model in its semantics and keeps the application's logic seperate. The notation features *selective receive* and *relative message order preservation*. Messages are described by pre and post conditions and give a cumulative abstraction of what a message is required to do. An operational small step semantics describes precisely how actor objects and cumulative abstractions blend together. We use Lamport's Temporal Logic of Actions (TLA) [Lam02] for declarative expressions in ReActor and for the definition of its semantics.

We assume the reader has a basic knowledge in reading operational semantics [Plo04] and TLA$^+$ specifications [Lam02]. Section 2 introduces ReActor by a small example. Abstract data structures for actors, ReActor specifications, and actor state transition systems are presented in Section 3. In Section 4, we define the semantics of ReActor models by defining their reachable system states inductively.

2 A blinking light example specified in ReActor

ReActor is a declarative specification language for describing actors and their behavior. In this section the notation and its structure is explained by a small example.

A ReActor system specification, also called model, consists of several actor modules, each describing a class of actor objects. An actor module consists of declarations and different kinds of specification schemata. Declarations define constant and variable symbols which

can be used in local expressions to refer to an actor's local state. Schemata are visual templates in the style of TLA$^+$ and Object-Z [RD00]. They are used to specify interface operations, internal actions, initial conditions, state invariants, and temporal properties. At the expression level ReActor uses a subset of the notation and syntax of TLA$^+$ enriched by some actor-specific elements.

In Figure 1, we present a system specification for an actor system that models the behavior of some lights. The system specification consists of two kinds of actor modules one specifying a blinking light and the other the environment. A blinking light actor represents a light that can be turned on and off by sending a *Switch* message. An environment actor models the user of several blinking lights. It can instantiate or release lights and switch their status.

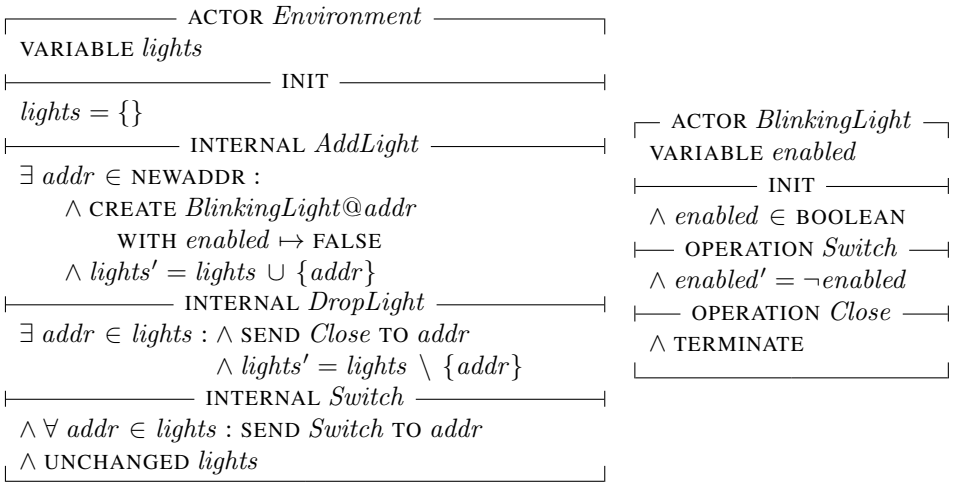

Figure 1: A simple actor system specification using ReActor with an environment actor that creates, releases and switches lights on the left and a blinking light actor that either toggles its status or depletes on external requests on the right. Vertically aligned conjunctions remove the need for nested parentheses.

The blinking light module declares one local state variable, an initial state predicate and two interface operations. The initial state predicate defines that a light is initially either enabled or disabled. Initialization schemata are state-level predicates[1] that logically combine constants, constant operators, and unprimed variables.

Operation schemata like *Switch* or *Close* relate incoming message events to local behavior. An operation schema is a transition-level predicate that logically relates constants, constant operators, primed and unprimed variables, and actor-specific expressions. The *Switch* operation negates a light's status. The *Close* operation defines that an actor has reached its end of life and that it is eventually depleted. An actor's end of life is claimed by using the actor-specific keyword TERMINATE.

[1] TLA$^+$ differs between constant-, state-, transition- and temporal-level expressions.

The environment module declares the state variable $lights$, an initialization schema, and three internal action schemata. The variable $lights$ holds references to attached instances of $BlinkingLight$ at runtime. Initially it is assumed to be empty. The set is populated and depopulated by the internal actions $AddLight$ and $DropLight$. Internal actions in difference to operations do not rely on external message events but on local state preconditions only. We use them to stimulate the system initially, or to model events from the environment, or to describe internal actor transitions.

The internal action $AddLight$ specifies that a new $BlinkingLight$ instance is created and that its reference is added to the set of attached lights. Actors are created by picking an unbound address ($addr \in$ NEWADDR) and binding it to a concrete instance definition. An instance is defined and bound by a CREATE expression. A CREATE expression takes an actor type (the name of an actor module), an address, and an initial state definition (the equivalent to a constructor) as parameters.

The internal action $DropLight$ removes one of the attached lights and sends it a $Close$ message, which then eventually is released. The dispatch of messages is defined by using the SEND expression. A SEND expression takes a message (optionally with parameters) and the receiver's address as input. It defines that a message is eventually delivered to a receiver's inbox. Messages sent to unbound addresses eventually get dropped. The universal quantifier is used in the $Switch$ schema to send appropriate switch requests to all attached lights.

In this example, we focused on behavioral modeling aspects of ReActor only. Beside operation and internal action schemata ReActor also supports the specification of safety and liveness requirements. Both are not further detailed here.

3 Data structures for actors, actor schemata and actor systems

In this section abstract data structures for actors, ReActor specifications, and actor state transition systems are presented. Hierarchical TLA$^+$ type definitions are used to describe these data structures. ReActor specifications are related to actor state transition systems in Section 4.

In the following definitions messages, actor identifiers (addresses), actor type identifiers (class names), values assignable to variables and constants, and names of variables and constants (symbols) are modeled by constant sets:

CONSTANTS $Message, ActorID, TypeID, Value, Identifier$

The concrete values depend on the concrete ReActor model and must be derived from there.

3.1 Actor instances

Actors are unique, stateful objects that react on incoming message events. We model actor instances as TLA$^+$ records with a globally unique address aid, a type reference tid, a local state lst, and an inbox mbx:

$$Actor \triangleq [aid : ActorID, tid : TypeID, lst : LocalState, mbx : Inbox]$$

The type reference relates an actor instance to a ReActor specification module. The local state is a function that maps constant and variable identifiers to concrete values. We assume $Value$ to be a constant set of all assignable entities including references to other actors:

$$LocalState \triangleq [Identifier \to Value]$$

In ReActor actors are able to process their messages in their relative sending order. Therefore, the inbox is modeled as a sequence of messages instead of a commonly used set:

$$Inbox \triangleq Seq(Message)$$

3.2 ReActor specifications

In ReActor a specification is defined by a set of specification modules. A specification module defines the state and behavior for a concrete type of actor. It declares a set of local state identifiers and different kinds of schemata.

$$\begin{aligned} ActorSchema \triangleq [&tid : TypeID,\ decl : \text{SUBSET } Identifier,\ ini : StatePredicate,\\ &int : InternalAction,\ op : Operation] \end{aligned}$$

Each specification module defines a globally unique actor type tid. This identified is used to relate an actor instance to a concrete specification module. A set of identifiers declares the names of constants and variables that can be used in schema expressions.

An initialization schema constraints the type of initial states by some state predicate expression. A state predicate function maps the universe of all local states to their validity values and abstracts state predicate expressions in our data structure. The domain of the local state function must include all declared variable and constant identifiers.

$$StatePredicate \triangleq [LocalState \to \text{BOOLEAN}]$$

Internal action schemata and operation schemata represent boolean-valued state transition expressions that either rely on an actor's local state only or on incoming messages too. We represent the former kind of schemata by an *internal action* and the latter ones by a so called *operation*:

$$\begin{aligned} InternalAction &\triangleq [LocalStep \to \text{BOOLEAN}]\\ Operation &\triangleq [InboxStep \to \text{BOOLEAN}] \end{aligned}$$

An internal action relates *local steps* to boolean values while an operation assigns boolean values to so called *inbox steps*. Both kinds of steps relate two succeeding actor states (pre and $post$) to their transition's effect.

$$LocalStep \triangleq [pre : LocalState,\ post : LocalState,\ eff : Effect]$$
$$InboxStep \triangleq [pre : LocalState,\ msg : Message,\ post : LocalState,\ eff : Effect]$$

Inbox steps depend on a concrete message in an actor's inbox while local steps do not. An effect defines outgoing messages, newly created actor instances, and an actor's status of termination. Newly instantiated actors must be bound to free actor identifiers uniquely.

$$Effect \triangleq [out : Outbox,\ new : \text{SUBSET } Actor,\ eol : \text{BOOLEAN}]$$

In ReActor operations and internal actions may claim that several messages are sent in one step, for instance message X and message Y to actor A. Messages, that are sent in one step, can be transmitted in any order and are represented as a bag for each potential receiver:

$$Outbox \triangleq [ActorID \to Bag(Message)]$$

If the relative order between X and Y is important then X and Y must be sent in different steps. ReActor preserves sending order between messages sent in different steps only.

3.3 A structure for actor systems

An actor system describes all potential states and state transitions in terms of a concrete model. It is defined by a set of system states and a set of actor schemata:

$$ActorSystem \triangleq [sst : \text{SUBSET } SystemState,\ mod : \text{SUBSET } ActorSchema]$$

A model relates succeeding system states and defines proper system steps. A system state defines the state of the overall system at a point in time. It consists of a concrete configuration of instantiated actors and a concrete configuration of pending messages:

$$SystemState \triangleq [act : \text{SUBSET } Actor,\ buf : MessageBuffer]$$

Pending messages are messages that are sent but not delivered to the receivers' inboxes yet. They must be deployed in their relative sending order for each pair of communicating actors because of the targeted feature of message order preservation. Therefore, we relate a sequence of bags of messages to each combination of sending and receiving actors:

$$MessageBuffer \triangleq [ActorID \times ActorID \to Seq(Bag(Message))]$$

The bag encapsulates a number of unordered messages sent within a local or inbox step. The sequence keeps order between messages that are sent in different steps. Assigning such a sequence to each potential pair of communicating actors preserves relative message order globally.

4 Operational semantics of ReActor

In this section, we present the semantics of ReActor in terms of reachable system states. We first summarize important features verbally and then present a precise, formal description.

4.1 Informal description

In ReActor actors are stateful concurrent objects. An actor can either process an operation or an internal action but only one at a time. Enabled internal actions and operations are selected and processed non-deterministically. An internal action is enabled if its state precondition is fulfilled. An operation is enabled if its state precondition is fulfilled and a corresponding message was send before. Actors with continuously enabled internal actions or operations must eventually be processed (fair scheduling). Each message can be processed only once. Send messages are guaranteed to eventually be delivered. Local messages have no special priority and are treated like any other message. The keywords NIL and SELF represent the empty reference value and the address of an actor itself.

Internal actions and operations represent atomic system steps. Atomic system steps may be applied in any (interleaving) order as long as the application of each step is valid. Internal actions and operations may alter the local state of an actor, create new actors, send messages, or enable termination. Messages can be sent to known actors only. Actors are known if their addresses were communicated during initialization, as a parameter of a message, or as the result of a locally created actor. Enabled termination eventually leads to the release of an actor. Addresses of released actors can be reused.

Messages sent in a single atomic step (either internal action or operation) can be received in any order. Messages sent in different atomic steps are received in their sending order as long as the sending and receiving actors are the same (relative message order preservation). An actor processes its enabled operations in correspondence to the receiving order of its pending messages. Messages that do not correspond to an enabled operation stay unchanged (selective receive). Messages with an invalid destination may be dropped at any time. This may happen if a destination was terminated before an already sent message was received and processed.

A system can start with any number of well initialized actors. An actor is well initialized if the initialization predicate is fulfilled. Initially there are no pending messages, means communication must be established by internal actions.

4.2 Formal description

After giving an informal summary of ReActor's semantical details, we will now define them more formally. For this, we define the set of all valid system states sst for any

arbitrary system model mod inductively. We will start with valid system initializations (induction basis), and derive and add successor states appropriate to the model consecutively (inductive step). We use inference rules in the style of Plotkin [Plo04] to derive initialization and target states. The following rule reads as "P implies that s is a valid system state":

$$\frac{P}{s \in sst}$$

For ReActor there are five such rules: one for deriving initial system states from the model, one for processing enabled internal actions, one for processing enabled operations, one for delivering messages, and one for dropping messages with invalid destinations. Nondeterminism is covered by the fact that more than one rule may be applied in any of the system states.

Each of the rules expresses its premise and implication in a TLA style of notation. We use our own definitions to make the rules easier to read. You will find the definitions in alphabetical order in Annex A. Each definition is fully specified in terms of TLA$^+$. We used the model checker TLC, type assumptions, and different input values to test all our rules and definitions.

We assume fair scheduling for all transition rules. Actors and transitions that are continuously enabled must eventually be applied. In addition, we assume that model mod is well formed. A model is well formed if its structure and behavioral definitions are consistent. For example the domain of a local state function must match the variable and constant identifiers of the related actor type. Due to lack of space, we do not give a full description of well formed models here; but we believe that most of these predicates are obvious by the design of our language.

The first rule – the induction basis – defines the initial system states of a model mod:

$$\frac{\begin{array}{c} s \in \mathit{SystemState},\ s.\mathit{buf} = \mathit{EmptyMessageBuffer}, \\ \mathit{ActorsHaveUniqueIDs}(s.\mathit{act}),\ \forall\ \mathit{act} \in s.\mathit{act} : \mathit{ActorInit}(\mathit{act}, \mathit{mod}) \end{array}}{s \in sst}$$

In ReActor a system can start with an arbitrary number of well initialized actors that each must have a globally unique address. An actor is well initialized if its state satisfies the actor's initialization predicate and if the actor's mailbox is empty. Also the global message buffer is initially assumed to be empty. Actors are related to their corresponding type definitions in the model by using their type identifier. Each system state s that applies to the first rule is a valid starting point for a system run and for our inductive definition of reachable system states.

With having defined the set of initial system states, we can now expand it by adding all target states of our atomic system transitions inductively. The first transition rule defines all target states for the valid application of an enabled internal action:

$$\frac{s \in sst,\ a \in s.act,\ ls \in EnabledLocalSteps(s, a, mod),\ s' = EvalLocalStep(s, a, ls)}{s' \in sst}$$

An internal action is enabled if there is an actor a that satisfies the internal precondition of the corresponding local step. We define the target state by evaluating the enabled local step's description and the system's source state. We evaluate only one enabled local step at a time (small step semantics). The definition of $EvalLocalStep$ adds a bag of send messages to the global message buffer, removes itself from the system's pool of instantiated actors if terminated, also adds new created actors, and changes the actor's local state accordingly to the specification. Each system state s and succeeding target state s' matching the rule are in the set of reachable system states.

The second transition rule defines the target states for the application of enabled operations:

$$\frac{s \in sst,\ a \in s.act,\ is \in FirstEnabledInboxStep(s, a, mod),\ s' = EvalInboxStep(s, a, is)}{s' \in sst}$$

In difference to internal actions an operation requires a corresponding inbox message. The message that was received first and corresponds to an enabled operation has the highest application priority (selective receive). Applying an operation requires that the corresponding message is removed from the inbox of the actor. All transitional effects are captured by the $EvalInboxStep$ definition of the target state.

The two rules for applying internal actions and operations both add sent messages to the global message buffer. The next rule defines the delivery of a pending message stored in the global message buffer to its receiver's inbox:

$$\frac{s \in sst,\ srp \in Pending(s.buf),\ msg \in \text{DOMAIN } FirstBag(s.buf, srp),\ Rcv(srp) \in ActorIDs(sst.act),\ s' = EvalTransmission(s, srp, msg)}{s' \in sst}$$

We deliver only one pending message from an arbitrary sender to an arbitrary receiver at a time. In our rule srp represents such a tuple of sending and receiving actor. The global message buffer stores pending messages for each of these tuples as a sequence of bags. A bag encapsulates messages sent in an internal action or operation step. We transmit the messages of each tuple's front bag first to preserve the order between messages sent in different steps. Delivered messages are enqueued to the end of the receivers inbox and removed from the global message queue. The $EvalTransmission$ definition describes both these effects in terms of a target system state.

The last transition rule defines how to deal with pending messages if the receiver does not exist anymore. Programming languages like Erlang silently drop such messages. In the same manner the sending of messages always succeeds in ReActor even if the receiving actor is not available anymore:

$$\frac{s \in sst,\ srp \in Pending(s.buf),\ msg \in \text{DOMAIN } FirstBag(s.buf, srp),\ Rcv(srp) \notin ActorIDs(sst.act),\ s' = EvalDrop(s, srp, msg)}{s' \in sst}$$

If a pending message targets for a non-existing receiver it is eventually removed from the message buffer without any further effects. Only one message is removed from the global message buffer at a time. The proper processing of send messages is a property of the model and requires that a receiver stays until all its pending messages were processed.

The presented rules together inductively define the set of all reachable system states sst and all valid system state transitions for a given ReActor model mod. A list of all used helper definitions can be found in Annex A.

5 Related work

Our work contributes to the field of actor system specifications and actor semantics. We relate our work to three representative and important works in this fields.

Agha et al. present an actor language and one of the most referenced operational actor semantics [AMST97] in the field. The language describes actors in difference to ReActor at a lower, more concrete level using an extension of the λ-calculus. External actors and internal receptionists represent interfaces to open environments, while ReActor assumes a fully closed system specification here. Their semantics is defined by a transition relation on actor configurations. Transition steps are defined for each low-level actor operation. We compose the low-level actor operations of each message to a more abstract but semantically also more complex atomic transition step. The message buffer is modeled as a generic bag while we use sequences of bags to support message order preservation. Their language and its semantics aims at general application and does not treat with features of popular actor programming languages.

Fredlund et al. present a small step operational semantics for a subset of the programming language Erlang [Fre01] and an improved version that also covers distributed nodes [SF07]. The semantics is separated into two parts, one defines the meaning of functional expressions and the other defines the global weaving of local actor behavior. We use an abstract characterization for our schema expressions and define the global weaving of local actor behavior only. The difference between our abstract characterization and their functional expressions is the granularity the system needs to be modeled in. Our semantics is closely inspired by Erlang and uses the same fundamental abstractions; e.g. mailboxes are modeled as sequences of messages. We add local messages to the global set of pending messages while Erlang delivers these messages immediately. We do not model dead processes explicitly but remove them instantly. References to acquaintances are defined explicitly by Fredlund and enables reasoning about locality. They implemented their semantics into the model checker McErlang [FS07] that can be used to examine real world Erlang programs and prove liveness and safety properties.

The language Rebeca aims at closing the gap between formal verification and real pro-

grams [Sir06]. The core of the language are so called reactive object templates. A template describes the data structure and the behavior of an actor and corresponds to our actor schemata. The semantics is defined in an interleaving manner by defining a labeled transition system [SMSdB04]. Controlled behavior for unusual situations is left unspecified, e.g. for cases where a message should be sent to a depleted actor. In difference to our work Rebeca proposes an imperative view to actor operations while we use a declarative approach here. Rebeca describes actor systems in terms of closed system specifications. Open system specifications are supported by components that together implement a closed model. While we use internal actions to specify environmental behavior, Rebeca relies on messages and obligatory initialization actions completely. Different model checker back ends can be used to verify Rebeca implementations.

6 Conclusion and future work

In this paper, we proposed a design notation and a semantics for the specification of actor systems. The notation is called ReActor and supports a declarative style of description and selected features of actor programming languages in combination. A declarative style of descriptions allows for a more abstract description than in existing actor notations. Implementation details that are unknown in early phases of software design are left open to the programmer and later design phases. Features like *selective receive* and *relative message order preservation* are known from actor programming languages like Erlang and Scala. Both simplify the specification of sequential protocols in asynchronous environments. Supporting them in ReActor helps to slim object interfaces and to improve readability. We detailed the structure of ReActor specifications and presented a precise semantics that defines system states inductively. The semantics of ReActor systems was specified and mechanized in the Temporal Logic of Actions. Basic assumptions like types were tested using the model checker TLC. We propose ReActor for specifying actor systems on an abstract but precise level and as a supplement to existing design notations like the UML. Our semantics can be used to strengthen concurrent object notations that lack in a precise semantics for object management or message passing.

Overall, we contribute a notation that enables declarative specification of actors and their behavior without fixing implementation details. Future work includes the connection of ReActor to different automatized verification back ends. We also work on better tool support and plan case studies for different kinds of applications.

A TLA$^+$ definitions

The following definitions complement the rules of ReActor's semantics. We used TLA$^+$ and its base modules (sequences, natural numbers etc.) to define them. The model checker TLC was used to test basic assumptions about these definitions.

$$ActorIDs(conf) \triangleq \{act.aid : act \in conf\}$$

$$
\begin{array}{l}
ActorInit(act, mod) \triangleq \\
\quad \land\ act.tid \in \{m.tid : m \in mod\} \\
\quad \land\ \text{LET } schema \triangleq \text{CHOOSE } m \in mod : m.tid = act.tid \\
\quad\quad \text{IN} \quad \land\ \text{DOMAIN } act.lst = schema.decl \\
\quad\quad\quad\quad \land\ schema.ini[act.lst] \equiv \text{TRUE} \\
\quad\quad\quad\quad \land\ act.mbx = \langle\rangle
\end{array}
$$

$$
ActorsHaveUniqueIDs(conf) \triangleq \forall\, a, b \in conf : (a.aid = b.aid \Rightarrow a = b)
$$

$$
\begin{array}{l}
BagDecrement(bag, msg) \triangleq \\
\quad \text{IF } bag[msg] = 1 \text{ THEN } [m \in (\text{DOMAIN } bag \setminus \{msg\}) \mapsto bag[m]] \\
\quad\quad\quad\quad\quad\quad \text{ELSE } [bag \text{ EXCEPT } ![msg] = bag[msg] - 1]
\end{array}
$$

$$
EmptyMessageBuffer \triangleq [srp \in (ActorID \times ActorID) \mapsto \langle\rangle]
$$

$$
\begin{array}{l}
EnabledInboxSteps(sst, act, mod) \triangleq \\
\quad \text{LET } schema \triangleq \text{CHOOSE } m \in mod : m.tid = act.tid \\
\quad \text{IN} \quad \{inbox\_step \in \text{DOMAIN } schema.op : \\
\quad\quad\quad\quad \land\ schema.op[inbox\_step] \equiv \text{TRUE} \\
\quad\quad\quad\quad \land\ inbox\_step.pre = act.lst \\
\quad\quad\quad\quad \land\ inbox\_step.msg \in Range(act.mbx) \\
\quad\quad\quad\quad \land\ ActorIDs(inbox\_step.eff.new) \cap ActorIDs(sst.act) = \{\}\}
\end{array}
$$

$$
\begin{array}{l}
EnabledLocalSteps(sst, act, mod) \triangleq \\
\quad \text{LET } schema \triangleq \text{CHOOSE } m \in mod : m.tid = act.tid \\
\quad \text{IN} \quad \{local\_step \in \text{DOMAIN } schema.int : \\
\quad\quad\quad\quad \land\ schema.int[local\_step] \equiv \text{TRUE} \\
\quad\quad\quad\quad \land\ local\_step.pre = act.lst \\
\quad\quad\quad\quad \land\ ActorIDs(local\_step.eff.new) \cap ActorIDs(sst.act) = \{\}\}
\end{array}
$$

$$
EvalDrop(sst, srp, msg) \triangleq [act \mapsto sst.act, buf \mapsto Receive(sst.buf, srp, msg)]
$$

$$
\begin{array}{l}
EvalInboxStep(sst, act, istep) \triangleq \\
\quad \text{LET } post\_act \triangleq \text{IF } istep.eff.eol \text{ THEN } \{\} \text{ ELSE} \\
\quad\quad\quad \{[act \text{ EXCEPT } !.lst = istep.post, \\
\quad\quad\quad\quad\quad\quad !.mbx = WithoutFirst(istep.msg, act.mbx)]\} \\
\quad \text{IN} \quad [act \mapsto (sst.act \setminus \{act\}) \cup post\_act \cup istep.eff.new, \\
\quad\quad\ buf \mapsto Send(istep.eff.out, act.aid, sst.buf)]
\end{array}
$$

$$
\begin{array}{l}
EvalLocalStep(sst, act, lstep) \triangleq \\
\quad \text{LET } post\_act \triangleq \text{IF } lstep.eff.eol \text{ THEN } \{\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{ELSE } \{[act \text{ EXCEPT } !.lst = lstep.post]\} \\
\quad \text{IN} \quad [act \mapsto (sst.act \setminus \{act\}) \cup post\_act \cup lstep.eff.new, \\
\quad\quad\ buf \mapsto Send(lstep.eff.out, act.aid, sst.buf)]
\end{array}
$$

$$
\begin{array}{l}
EvalTransmission(sst, srp, msg) \triangleq \\
\quad \text{LET } rcv \triangleq \text{CHOOSE } a \in sst.act : a.aid = Rcv(srp) \\
\quad\quad post\_rcv \triangleq [rcv \text{ EXCEPT } !.mbx = Append(rcv.mbx, msg)] \\
\quad \text{IN} \quad [act \mapsto (sst.act \setminus \{rcv\}) \cup \{post\_rcv\}, \\
\quad\quad\ buf \mapsto Receive(sst.buf, srp, msg)]
\end{array}
$$

$$
\begin{array}{l}
FirstEnabledInboxStep(sst, act, mod) \triangleq \\
\quad \text{LET } enabled\_steps \triangleq EnabledInboxSteps(sst, act, mod) \\
\quad \text{IN } \{s \in enabled\_steps : \forall\, t \in (enabled\_steps \setminus \{s\}) : \\
\qquad FirstIndexOf(t.msg, act.mbx) > FirstIndexOf(s.msg, act.mbx)\}
\end{array}
$$

$$
\begin{array}{l}
FirstIndexOf(msg, mbx) \triangleq \\
\quad \text{CHOOSE } n \in \text{DOMAIN } mbx : \\
\qquad \land\ mbx[n] = msg \\
\qquad \land\ \neg\exists\, m \in \text{DOMAIN } mbx : (m < n \land mbx[m] = msg)
\end{array}
$$

$$Pending(buf) \triangleq \{srp \in \text{DOMAIN } buf : buf[srp] \neq \langle\rangle\}$$

$$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$$

$$Rcv(srp) \triangleq srp[2]$$

$$
\begin{array}{l}
Receive(buf, srp, msg) \triangleq \\
\quad \text{LET } post\_bag \triangleq BagDecrement(FirstBag(buf, srp), msg) \\
\quad \text{IN } [buf \text{ EXCEPT } ![srp] = \text{IF } post\_bag \neq EmptyBag \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } \langle post\_bag\rangle \circ Tail(buf[srp])) \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE } Tail(buf[srp])]
\end{array}
$$

$$
\begin{array}{l}
Send(out, aid, buf) \triangleq \\
\quad [srp \in \text{DOMAIN } buf \mapsto \\
\qquad \text{IF } (Snd(srp) = aid) \land (out[Rcv(srp)] \neq EmptyBag) \\
\qquad\quad \text{THEN } Append(buf[srp], out[Rcv(srp)]) \text{ ELSE } buf[srp]]
\end{array}
$$

$$Snd(srp) \triangleq srp[1]$$

$$
\begin{array}{l}
WithoutFirst(msg, mbx) \triangleq \\
\quad \text{LET } F[seq \in Seq(Range(mbx))] \triangleq \\
\qquad \text{IF } Head(seq) = msg \text{ THEN } Tail(seq) \\
\qquad\qquad\qquad\qquad \text{ELSE } \langle Head(seq)\rangle \circ F[Tail(mbx)] \\
\quad \text{IN } F[mbx]
\end{array}
$$

References

[Agh86] Gul Agha. Actors: A Model of Concurrent Computation In Distributed Systems. Technical Report 844, Massachusetts Institute of Technology, 1986.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Programmers, 2007.

[AT04] Gul Agha and Prasanna Thati. An Algebraic Theory of Actors and its Application to a Simple Object-Based Language. In *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 26–57. Springer, 2004.

[BH12] Rodger Burmeister and Steffen Helke. The Observer Pattern applied to actor systems: A TLA/TLC-based implementation analysis. In Tiziana Margaria, Zongyan Qiu, and Hongli Yang, editors, *Proceedings of the Sixth International Conference on Theoretical Aspects of Software Engineering*, pages 193–200. IEEE CS Press, 2012.

[Fre01] Lars-Ake Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology Stockholm, 2001.

[FS07] Lars-Ake Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International conference on functional programming (ICFP 2007)*, pages 125–136. ACM Press, 2007.

[GZ01] Mauro Gaspari and Gianluigi Zavattaro. An Actor Algebra for Specifying Distributed Systems: the Hurried Philosophers Case Study. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 428–444. Springer, 2001.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Sofware Engineers*. Addison-Wesley, 2002.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, Ecole Polytechnique Federale de Lausanne, 2004.

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004.

[RD00] Gordon Rose and Roger Duke. *Formal Object Oriented Specification Using Object-Z.* Palgrave Macmillan, 2000.

[Sch01] Susanne Schacht. Formal Reasoning about Actor Programs Using Temporal Logic. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2001.

[SF07] Hans Svensson and Lars-Ake Fredlund. A More Accurate Semantics for Distributed Erlang. In *Proceedings of the ACM SIPGLAN 2007 Erlang Workshop*, pages 37–42. ACM Press, 2007.

[Sir06] Marjan Sirjani. Rebeca: Theory, Applications, and Tools. In *Proceedings of the 5th international conference on Formal methods for components and objects (FMCO'06)*, volume 4709 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2006.

[SMdB05] Marjan Sirjani, Ali Movaghar, and Frank de Boer. Integrating Model Checking and Deduction for an Actor-Based Language. *Scientia Journal*, 12(1):55–65, 2005.

[SMSdB04] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank de Boer. Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.