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

Rodger Burmeister
Software Engineering Group
Berlin Institute of Technology, Germany
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 specifications and as a supplement to existing design notations, especially if a sound notion of concurrent objects is required.

## 1 Introduction

The execution environments of software have changed significantly in the last two decades. There are trends towards parallel hardware designs, that could not longer be ignored by general software developers. We observe the shift from single processor cores with higher clock rates to many processor cores. Also, computers get continuously connected by high-bandwidth, low-latency networks, which enable access to remote resources as if they where local. Both trends – many-core and distributed architectures – require programming abstractions that can handle concurrent resources in a scalable and maintainable way.

One such abstraction is the so called *actor model* [AMST97, AH87]. Actors are stateful, concurrent objects that use asynchronous, non-blocking *message passing* for inter-object communication. In contrast to threads there is no shared memory and no error-prone mechanisms for synchronizing memory access. Each actor has its own exclusive state space 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 references can change over time and lead to a dynamic object topology. Actors can activate other actors by sending them messages. Each actor evaluates its messages sequentially, one at a time. Evaluating a message can lead to a local state change, the creation of new actors, further messages, or to an actor's end of life.

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

Messages can be transmitted and evaluated (by individual actors) in parallel. The concrete order in which messages are evaluated may vary for each system run and lead to different behaviors and results. Race conditions and deadlocks may occur in some, rare message order combinations. Testing all potential message order combinations is usually impossible for real world systems due to state space complexity and difficulties in process orchestration. A far better and viable approach would be to specify the actor processes abstractly and validate their combined behavior with the help of tool-supported formal methods, either in the design or implementation phase of the software cycle.

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 [Sir06], Temporal Actor Logic [Sch01], Algebra of Actors [GZ01], or Simple Actor Language [AT04, AMST97, Agh86] model pending messages as sets and allow them to be processed in any order. Programming languages like Erlang [Arm07] and Scala [HO07] use queues and simplify the implementation of sequential protocols by features that we call *selective receive* and *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 evaluated) in their sending order.

Formal actor notations with a 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, as these additional messages can be omitted. Another shortcoming of existing actor notations is the imperative style of description. Messages are usually described as concrete sequences of statements. In early phases of software design such implementations 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 a concurrent version of the observer pattern and an implementation of the actor model in TLA$^+$. The effect of messages was modeled abstractly by using pre- and post-conditions. We used the model checker TLC to verify essential safety and liveness properties [BH12] for a limited number of actor-objects. While our model was well-suited to express the pattern's logic it was hard to keep it separate from the underlying actor model as TLA$^+$ has no explicit support for objects or message passing. In this paper, we therefore came up with a special purpose specification language, called ReActor, that hides many details of the actor model in its semantics. Figure 1 illustrates the artifacts and steps of our framework. In this paper, we focus on Step 1, the interpretation and the semantics of ReActor specifications.

In ReActor, we describe each class of objects by an individual specification module. Details like object management and message passing are embedded into the semantics and do not interfere with an application's logic. Both *selective receive* and *relative message order preservation* are supported by ReActor. Actions are described abstractly by pre- and post-

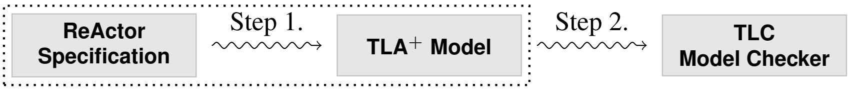

Figure 1: In Step 1, we translate an ReActor-system-specification into a TLA$^+$ model. In Step 2, we use automated model checking techniques to verify safety and liveness properties of concurrent object-based systems. In this paper, we focus on Step 1 and here especially on ReActor's semantics.

conditions. The impact of a message is called *effect*. An operational small step semantics describes how actor objects, messages, and effects globally blend together. We (re)use Lamport's syntax of the Temporal Logic of Actions (TLA) [Lam02] for our 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. An abstract syntax and structure for actors, ReActor specifications, and actor state transition systems is presented in Section 3. In Section 4, we define the semantics of ReActor models by defining their 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 2, we present a 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 creates and releases lights and switches their status.

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.

Operations like $Switch$ or $Close$ relate incoming message events to local behavior. An *operation* is a transition-level predicate that logically relates constants, constant operators,

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

```
┌──────────── ACTOR Environment ────────────┐
 VARIABLE lights
├──────────────────── INIT ─────────────────┤
 lights = {}
├────────── INTERNAL AddLight ──────────────┤
 ∃ addr ∈ NEWADDR :
      ∧ CREATE BlinkingLight@addr
          WITH enabled ↦ FALSE
      ∧ lights' = lights ∪ {addr}
├────────── INTERNAL DropLight ─────────────┤
 ∃ addr ∈ lights : ∧ SEND Close TO addr
                   ∧ lights' = lights \ {addr}
├────────── INTERNAL Switch ────────────────┤
 ∧ ∀ addr ∈ lights : SEND Switch TO addr
 ∧ UNCHANGED lights
└───────────────────────────────────────────┘
```

```
┌─ ACTOR BlinkingLight ─┐
 VARIABLE enabled
├──────── INIT ─────────┤
 ∧ enabled ∈ BOOLEAN
├── OPERATION Switch ───┤
 ∧ enabled' = ¬enabled
├── OPERATION Close ────┤
 ∧ TERMINATE
└───────────────────────┘
```

Figure 2: A simple ReActor model 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.

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 released. An actor's end of life is claimed by using the actor-specific keyword TERMINATE.

The environment module declares the state variable *lights*, an initialization schema, and three internal actions. The variable *lights* holds references to attached instances of blinking lights. Initially it has 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 blinking light is created and that its reference is added to the set of attached lights. Actors are created by choosing a free address ($addr \in$ NEWADDR) and binding it to a concrete instance definition. An instance is defined and bound by a CREATE expression. It 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. The dispatch of messages is defined by using the SEND expression. A SEND expression requires a message (optionally with parameters) and the receiver's address as input and defines that a message is eventually delivered to a receiver's inbox. Messages sent to unbound addresses will be dropped eventually. The universal quantifier in the *Switch* action is used to send a switch request to all attached lights.

In this small example, we focused on modeling behavioral aspects in ReActor. Beside operations and internal actions ReActor also supports the specification of safety and liveness requirements.

## 3 Abstract syntax of actors, ReActor specifications, and actor systems

In this section, we present an abstract syntax of actors, ReActor specifications, and actor systems. For each, we define an abstract data type in TLA$^+$. We need these types to introduce ReActor specifications and their semantics in Section 4. In the following, we assume constant sets for messages, actor identifiers (addresses), actor class identifiers, values (assignable to variables and constants), and symbols for naming variables and constants:

$$\text{CONSTANTS } \mathit{Message}, \mathit{ActorID}, \mathit{ClassID}, \mathit{Value}, \mathit{Symbol}$$

The concrete type of these sets depend on the specific ReActor model. For our blinking light example from Section 2 the symbol type is $\{\text{“}\mathit{enabled}\text{”}, \text{“}\mathit{lights}\text{”}\}$. Both variable names from both the actor modules are included in this global type. The value type of $\mathit{enabled}$ is BOOLEAN, because a light can either be enabled or disabled. The value type of $\mathit{lights}$ is SUBSET $\mathit{ActorID}$, because the environment references a variable set of blinking light objects. The overall value type is therefore the union of both these types. In the same way, we can derive the other constants from our specification.

### 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 $\mathit{aid}$, a class identifier $\mathit{cid}$, a local state $\mathit{lst}$, and an inbox $\mathit{mbx}$:

$$\mathit{Actor} \triangleq [\mathit{aid} : \mathit{ActorID}, \mathit{cid} : \mathit{ClassID}, \mathit{lst} : \mathit{LocalState}, \mathit{mbx} : \mathit{Inbox}]$$

A *class identifier* relates an actor instance to a concrete actor module. An instance of a blinking light for example always references the blinking light module. The *local state* is a function that maps constant and variable symbols to concrete values. We assume $\mathit{Value}$ to be a constant set of all assignable entities including references to other actors:

$$\mathit{LocalState} \triangleq [\mathit{Symbol} \rightarrow \mathit{Value}]$$

For a blinking light the value of the variable $\mathit{enabled}$ is either TRUE or FALSE. 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:

$$\mathit{Inbox} \triangleq \mathit{Seq}(\mathit{Message})$$

A typical inbox for a blinking light actor may look like $\langle \text{“}\mathit{Switch}\text{”}, \text{“}\mathit{Close}\text{”} \rangle$. This sequence of input messages leads to first switching and thereafter releasing the light. We assume that messages with different parameters are modeled as individual entities of the message type. In our small example messages do not have any parameters.

## 3.2 ReActor specifications

In ReActor a *system specification* consists of set of class descriptions, which we call *actor modules*. In our example, we defined two such modules one for the environment and another for actors of the type $BlinkingLight$. A actor module defines the state and behavior for a concrete type of actor. It first declares a set of symbols for variables and constants, and then defines different kinds of properties and actions, each represented by its own schema:

$$\begin{aligned} ActorModule \triangleq\ [&cid : ClassID,\ decl : \textsc{subset}\ Symbol,\ ini : StatePredicate, \\ &int : InternalAction,\ op : Operation] \end{aligned}$$

The blinking light module of our example consists of a variable declaration, an initialization predicate, and two operations. Each actor module relates to a globally unique class identifier $cid$. This identifier is used to relate an actor instance to a concrete specification module. Declared symbols can be used within any local schema to reference to the variables' or constants' values.

An initialization predicate constrains the type of initial states by some state predicate. A *state predicate* maps the universe of local states to their validity values. In our example, an environment configuration is mapped to TRUE only, if the variable $lights$ is empty. Using such predicate functions enables us to abstract any kind of predicate expressions in our abstract syntax:

$$StatePredicate \triangleq [LocalState \rightarrow \textsc{boolean}]$$

A predicate's local state function must include all combinations of declared symbols and value assignments. In our environment example, where $lights$ can reference an arbitrary number of blinking lights, this may lead to infinite domain types. *Internal actions* and *operations* are predicates that define if a local state transition is valid or not. Internal actions rely on an actor's local state only, while operations require an inbox message, too. Both define the validity for the universe of all potential pre- and post-state combinations. A valid transition is permitted by the specification:

$$\begin{aligned} InternalAction &\triangleq [LocalStep \rightarrow \textsc{boolean}] \\ Operation &\triangleq [InboxStep \rightarrow \textsc{boolean}] \end{aligned}$$

In our example, we used both kinds of transition predicates. In the environment, we have internal actions like $AddLight$ that only rely on the precondition but not on incoming messages. We call this kind of transitions *local step*. In contrast, a blinking light operation like $Switch$ can happen only if the precondition is fullfilled and if there is a corresponding message in the actor's inbox. This kind of transition we call *inbox step*. Both kinds of steps relate a pair of succeeding actor states ($pre$ and $post$) to their transition's effect:

$$\begin{aligned} &LocalStep \triangleq [pre : LocalState,\ post : LocalState,\ eff : Effect] \\ &InboxStep \triangleq [pre : LocalState,\ msg : Message,\ post : LocalState,\ eff : Effect] \end{aligned}$$

An *effect* defines the actors created during a step, the messages sent during a step, and if a step has lead to an actor's end of life. All actors that are created during a step must be bound to a free actor identifier to preserve the uniqueness of actor-identifier relation:

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

The only effect of our environment's action $AddLight$ is that a new blinking light actor is created: $[new \mapsto \{[cid \mapsto \text{“}BlinkingLight\text{”}, mbx \mapsto \langle\rangle, \ldots]\}, \ldots]$. In ReActor operations and internal actions can claim to send several messages to a target. For example, we could have specified an environment action $SwitchAndClose$ that sends a switch and a close message to each attached blinking light in the same step. Messages that are sent in the same step and to the same receiver can be received in any order. For the $SwitchAndClose$ action this implies that a receiving blinking light can either process the close or the switch message first, which of course is no useful behavior. However, in ReActor we model such unordered sets of messages as a bag for each potential receiver:

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

If the relative order between messages is important then these messages must be sent in different steps, as we do for $Switch$ and $DropLight$ in our environment specification. For these actions ReActor preserves the relative order of outgoing messages, as they are sent in different steps.

## 3.3 Actor systems and their semantics

After defining the abstract syntax of actors and actor modules, we now need to relate them. For this, we introduce an abstract representation of an actor system. An *actor system* describes all semantical states and state transitions in terms of a concrete model. It consists of a set of valid system states $sst$ and of a set of actor modules $mod$:

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

The actor modules together define a *model*. The internal actions and operations of that model applied to an initial set of actor instances define how a system can evolve. Our blinking light example always starts with a well initialized environment actor and continues in accordance to the actors' transition predicates. Each transition predicate relates succeeding system states and defines proper system steps. We will use them in the next section to derive target systems states from preceding ones. A *system state* defines the state of the overall system at a point in time. It consists of a configuration of instantiated actors and a configuration of pending messages:

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

For our blinking light system the system state includes the states of all potential environment and blinking light configurations and all configurations of messages in transit. Messages in transit are messages that are sent but not delivered to the receivers' inboxes yet. They are stored in a global *message buffer*. This buffer is part of the semantical state of a system and consists of a message queue for each pair of sender and receiver:

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

Each queue contains a series of message bags. Bags encapsulate a number of unordered messages sent within a single local or inbox step. Succeeding bags define messages sent in different steps. An environment $E$ that has just sent an switch message to an attached blinking light $L$ is buffered as $[\langle E, L\rangle \mapsto \langle \ldots, [\text{“}\mathit{Switch}\text{”}] \mapsto 1\rangle, \ldots]$. We are using queues of bags to implement the targeted feature of *message order preservation*. The concurrent delivery and evaluation of messages is part of ReActor's semantics and will be detailed in the next section.

## 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 informally 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 evaluated (fair scheduling). Each message can be evaluated only once. Sent messages are guaranteed to be delivered. Local messages have no special priority and are treated like any other message. The empty reference and the identity of an actor are represented by the keywords NIL and SELF.

Internal actions and operations represent atomic system steps. Atomic system steps can be interleaved in any order as long as the precondition of each step is fullfilled. 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 identifiers were communicated during initialization, as a parameter of a message, or as the result of a locally created actor. In the blinking light example the environment knows the lights it has created, but the lights do not know the environment as they do not keep any reference to it. Termination eventually leads to the release of an actor. Identifiers 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 evaluates 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*). A blinking light object with an inbox $\langle \text{“}\mathit{Switch}\text{”}, \text{“}\mathit{Close}\text{”}\rangle$ always evaluates its enabled switch operation first. It is enabled because the precondition of the switch operation is always TRUE. Messages with an invalid destination may be dropped at any time. This may happen if a receiving actor was released before an already sent message was transmitted.

A system can start with any number of well initialized actors. An actor is *well initialized* if its initialization predicate is fulfilled. In the blinking light example, we start with a single environment object. The global message buffer always starts to be empty. Internal actions, like the environment's $Switch$ must be used to trigger initial communication.

## 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 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 to describe the initialization and derive 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 evaluating an enabled internal action, one for evaluating an enabled operation, one for delivering a pending message, and one for dropping a message with an invalid destination. Non-determinism is covered by the fact that more than one rule may be applied in any of the system states. Figure 3 depicts how each rule contributes to the definition of a system's state space.

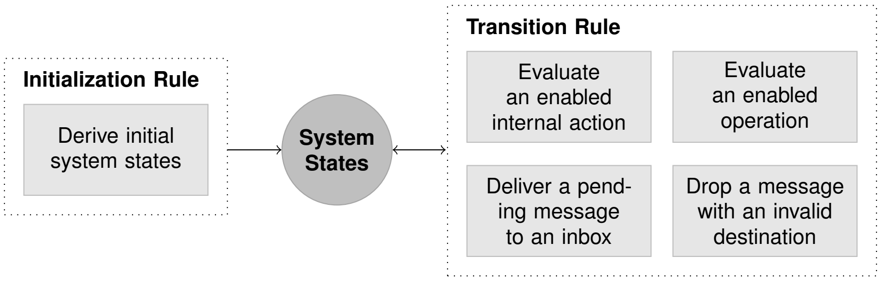
Initialization Rule
Derive initial system states
System States
Transition Rule
Evaluate an enabled internal action
Evaluate an enabled operation
Deliver a pending message to an inbox
Drop a message with an invalid destination

Figure 3: The semantics of a model is defined by first deriving all initial system states and then evaluating a transition rule for each system state inductively.

Each of the rules expresses its premise and implication in TLA$^+$ syntax. All related TLA$^+$ definitions are listed in alphabetical order in Annex A. 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 data structure and behavioral definitions are consistent. For example the domain of a local state function for a blinking light must match its variable and constant symbols. Due to lack of space, we do not give a full description of well formed models here; but we believe that most of these conditions are obvious by the design of our language.

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

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

A system can start with any number of well initialized actors that each must have a globally unique identifier. Our blinking light model always starts with one well initialized environment actor. 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 has to be empty. Actors are related to their corresponding actor specifications in the model by using their annotated class identifier. Each system state $s$ that satisfies the rule is a valid starting point for a system run and for our inductive definition of a system's state space. The only valid initialization state in our blinking light example consists of an empty message buffer and an environment actor with no attached lights.

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 evaluation of enabled internal actions:

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

An internal action is enabled if there is an actor $a$ that satisfies the internal precondition of the corresponding local step. For the blinking light's environment action *DropLight*, we assume that there is at least one attached light ($\exists\ \mathit{addr} \in \textsc{newaddr} : \ldots$). We define a system's target state by applying an enabled local step and its effect to the corresponding source state. In each system step, we evaluate only one enabled local step at a time (small step semantics). The definition of *EvalLocalStep* implements the evaluation of an enabled local step. It adds newly created actors, enqueues sent messages, releases an actor if it was terminated, and updates the local state accordingly to the local step's specification. For our environment's internal action *AddLight* it redefines a (source) system state by adding a newly created blinking light to the global actor configuration and by updating the environment's local set of lights. Each system state $s$ and succeeding target state $s'$ matching the rule have to be in the system's state space.

The second transition rule defines the target states for the evaluation of enabled operations in the same way we did for the internal actions:

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

In contrast 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*). A blinking light object with $\langle$"$\mathit{Switch}$", "$\mathit{Close}$"$\rangle$ as inbox, will always evaluate the switch message first. In the case that the first message cannot be evaluated because of an unfullfilled precondition, the next enabled message gets evaluated. If a operation is evaluated then its corresponding message is removed from the actor's inbox. All transition effects are described by the $\mathit{EvalInboxStep}$ definition of the target state. For a blinking light's $\mathit{Switch}$ operation it removes the first "$\mathit{Switch}$" message from the actor's inbox and updates the enabled status of that light. All other aspects of the preceding system state stay unchanged.

The rules for evaluating internal actions and operations both add sent messages to the global message buffer. The next rule defines how buffered messages are delivered to their receivers' inboxes:

$$\frac{\begin{array}{c} s \in sst,\ srp \in \mathit{Pending}(s.buf),\ msg \in \textsc{domain}\ \mathit{FirstBag}(s.buf, srp), \\ \mathit{Rcv}(srp) \in \mathit{ActorIDs}(sst.act),\ s' = \mathit{EvalTransmission}(s, srp, msg) \end{array}}{s' \in sst}$$

ReActor delivers only one pending message from any sender to any receiver at a time. In our blinking light example, we deliver either a pending $\mathit{Switch}$ or $\mathit{Close}$ message from the buffer to an blinking light's inbox. The symbol $srp$ represents an arbitrary 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 that are sent in different steps. Lets assume $\langle[$"$\mathit{Switch}$" $\mapsto 1], [$"$\mathit{Close}$" $\mapsto 1]\rangle$ to be the buffered messages of two environment actions ($\mathit{Switch}$ and $\mathit{Close}$) for any blinking light, then all messages of the front bag $[$"$\mathit{Switch}$" $\mapsto 1]$ must be transmitted first. Delivered messages are enqueued to the end of the receiver's inbox and removed from the buffer's message queue. The $\mathit{EvalTransmission}$ 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. For example, the environment action $\mathit{SwitchAndClose}$ from Section 3.2 can lead to situations where a blinking light was already released, but where a $\mathit{Switch}$ message is still pending. 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:

$$\frac{\begin{array}{c} s \in sst,\ srp \in \mathit{Pending}(s.buf),\ msg \in \textsc{domain}\ \mathit{FirstBag}(s.buf, srp), \\ \mathit{Rcv}(srp) \notin \mathit{ActorIDs}(sst.act),\ s' = \mathit{EvalDrop}(s, srp, msg) \end{array}}{s' \in sst}$$

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 sent messages is a property of the model and requires that a receiver stays until all its pending messages were evaluated.

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 $mod$. All used helper definitions are listed 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 contrast to ReActor at a lower, more concrete level using an extension of the $\lambda$-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 message buffer 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 programs [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 modules. 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 messages should be delivered to invalid destinations. In contrast 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

With the trends towards many-core and distributed architectures, engineers also need reliable tools to validate their concurrent software designs. In this paper, we described the formal foundation for an actor-oriented specification and analysis framework. The presented 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 an operational semantics that defines system states inductively. The semantics of ReActor systems was specified and mechanized in TLA$^+$. 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. In the future, we assume an infrastructure that enables the specification of early actor-system designs and the automated validation of safety and liveness properties. Essential features of actor systems are embedded into the semantics of the specification language. This includes the organization of message buffers, the management of object instances, the administration of the address space, and the scheduling of operations and internal actions. With the TLA$^+$-based implementation and validation of an actor-based observer pattern [BH12], we already demonstrated the applicability of our approach.

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.

## References

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

[AH87] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, Computer Systems Series, pages 49–74. MIT Press, 1987.

[AMST97] G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

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

[AT04] G. Agha and P. 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 *LNCS*, pages 26–57. Springer, 2004.

[BH12] R. Burmeister and S. Helke. The Observer Pattern applied to actor systems: A TLA/TLC-based implementation analysis. In *Proc. of the 6th Intern. Conference on Theoretical Aspects of Software Engineering*, pages 193–200. IEEE Press, 2012.

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

[FS07] L.A. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proc. of the 12th ACM SIGPLAN Intern. Conference on Functional Programming (ICFP 2007)*, pages 125–136. ACM Press, 2007.

[GZ01] M. Gaspari and G. Zavattaro. An Actor Algebra for Specifying Distributed Systems: the Hurried Philosophers Case Study. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *LNCS*, pages 428–444. Springer, 2001.

[HO07] P. Haller and M. Odersky. Actors That Unify Threads and Events. Technical Report LAMP-REPORT-2007-001, Ecole Poytechnique Federale de Lausanne, 2007.

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

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

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

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

[SF07] H. Svensson and L.A. Fredlund. A More Accurate Semantics for Distributed Erlang. In *Proc. of the ACM SIPGLAN 2007 Erlang Workshop*, pages 37–42. ACM Press, 2007.

[Sir06] M. Sirjani. Rebeca: Theory, Applications, and Tools. In *Proc. of the 5th Intern. Conference on Formal Methods for Components and Objects*, volume 4709 of *LNCS*, pages 102–126. Springer, 2006.

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

## 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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