Step-wise Development of Provably Correct Actor Systems

Bernhard K. Aichernig[0000−0002−3484−5584] and
Benedikt Maderbacher[0000−0002−5834−352X]

Graz University of Technology, Austria

Abstract. Concurrent and distributed software is widespread, but is inherently complex. The Actor model avoids the common pitfall of shared mutable state and interprocess communication is done via asynchronous message passing. Actors are used in Erlang, the Akka framework, and many others. In this paper we discuss the formal development of actor systems via refinement. We start with an abstract specification and introduce details until the final model can be translated into an actor program. In each refinement, we show that the abstract properties are still preserved. Agha's classical factorial algorithm serves as a demonstrating example. To the best of our knowledge we are the first who formally prove that his actor system computes factorials. We use Event-B as a modelling language together with interactive theorem proving and SMT solving for verification.

Keywords: Actors, Refinement, Proof-based Development, Formal Method, Event-B, Verification.

1 Introduction

Modern computer systems rely heavily on concurrent and distributed software. Classic techniques using shared mutable state and explicit synchronization mechanisms are not ideal for these tasks. Instead, many systems are written using techniques that are designed to handle the challenges inherent to concurrent programs. A model that is widely used in this area are actor systems [19]. They are based on asynchronous communication via message passing. Each actor has its own memory and state that is isolated from the rest of the world. All interaction is done by sending messages between actors. This concept has been implemented in various programming languages such as Erlang [6,5] as well as in frameworks for other languages such as Akka [21] for Scala [27] and Java [7]. Many well-known distributed systems use various actor implementations in their backend. This includes network infrastructure by Cisco [9] and Ericsson's telecommunication systems [18]. The messenger WhatsApp uses Erlang on its servers [31,24]. Other usages of actors include various online games, for example LeagueOfLegends [11]. Actor systems can also be used to describe other distributed systems such as IoT devices.

Actor systems by design help to prevent many common bugs in concurrent programming, such as data races and many forms of deadlocks, but they do not guarantee that the software is correct. There are still many possibilities to introduce errors in software written with actors. The usage of such systems in critical areas such as communication systems makes them an attractive target for formal methods. Formal methods use mathematics and logic to model and analyse hardware and software. They aim to find errors or certify the conformance to a specification. This techniques help to create software with fewer errors. Large companies, such as Amazon [26] and Microsoft [8], use formal methods to improve the quality of their software. In this paper we will explore how the formal method Event-B [1] can be used to verify actor systems. Here, we will focus on one classical example, further examples can be found in Maderbacher's master thesis [23].

Actor Systems. The main component of the actor systems concurrency model are so called actors. These are similar to processes or threads but they cannot access any shared memory. Each actor can have its own local memory. Actors communicate by sending messages. An actor who receives a message can do three kinds of actions: (1) it can send messages to other actors, (2) create new actors, or (3) change its own state or behaviour. A behaviour defines how an actor reacts to messages. While an actor performs computations triggered by one message, no other message can interrupt it. This allows actors to avoid the classic data race problem [28,10].

Event-B. Event-B is a modelling language and formal method based on set theory. One writes a model that captures the important behaviours of a system, instead of directly verifying a computer program. An Event-B model contains machines and contexts. A machine has a set of variables that define the state and guarded events that can change this state. An initial event defines the initial values of the variables. Contexts contain the static definitions of a model, including carrier sets, constants, and axioms. The models represent a discrete transition system: the initial state is defined via the initial event. The transitions are formed by enabled events with their guard expression evaluating to true in the current state. If more than one guard is enabled, the choice is non-deterministic. If no guard is enabled the system terminates or deadlocks depending on the interpretation. A model is developed by using step-wise refinement. At each step a new machine is created that is a refinement of the previous one. It is a more concrete version that contains more details and is closer to the modelled system. For each step a formal proof is required to demonstrate that this refinement relation holds. The Rodin Platform [2], an Event-B IDE based on Eclipse, supports the development and refinement of models with automatic generation and partial discharging of mathematical proof obligations.

Next, we introduce our demonstrating example in Section 2. Then, in Section 3 we discuss the modelling of actors in Event-B. Section 4 presents our formal development starting from the mathematical definition of the problem and ending in a correct actor system. Next, in Section 5 we briefly discuss a

Listing 1: Factorial with Scala's actor library Akka Typed.

```scala
final case class Request(value: Int, replyTo: ActorRef[Result])
final case class Result(value: Int)

val fact: Behavior[Request] = Behaviors.receive { (c, m) =>
  m.value match {
    case 0 => m.replyTo ! Result(1)
    case n =>
      val cont = c.spawnAnonymous(cont(m.value, m.replyTo))
      c.self ! Request(m.value - 1, cont)
  }
  Behaviors.same }

def cont(i:Int, cust:ActorRef[Result]): Behavior[Result] =
  Behaviors.receive { (c, m) =>
    cust ! Result(i * m.value)
    Behaviors.same }
```

truly concurrent extension of the previous actor system. Section 6 surveys related work. Finally, in Section 7 we discuss the results and draw our conclusions.

2 Demonstrating Example

As a demonstrating example we will develop Agha's classical factorial algorithm with actors [3]. The algorithm works recursively, but the computation is not solely done by one function. Instead, it works by creating continuations for each step. Each of these continuations is represented as its own actor. Additionally, there is one actor called *fact* that receives requests by customers, to calculate the factorial for a certain number. In response to these requests, it starts the continuation actors to do the actual work.

An implementation of this algorithm, using Scala [27] and Akka Typed [22], can be seen in Listing 1. The program contains two types of behaviours, *fact* and *cont*. There exists exactly one actor with the behaviour *fact*, therefore we will also refer to it as *fact*. It receives as a request the value that shall be processed and the address of the recipient of the result. If the number is 0, it will immediately send the result 1 to the recipient, otherwise it creates a new actor with the *cont* behaviour. This new actor keeps as state the value and the recipient of the request. After the new actor is created, *fact* sends itself an updated request, containing the decremented value and the newly created continuation as recipient. The *cont* actors await the result of the factorial of the number below the one stored by them. Once this is received, it is multiplied by the stored number and the result is sent to the stored recipient.

Figure 1 shows a sequence diagram computing the factorial of 3. At first, only the actor *Factorial* exists. It receives a request with the number 3 and the recipient address *c*. As a result the continuation actor *m* is created with the state 3 and *c*. The actor *Factorial* also sends itself the new request message containing 2 and the address of *m*. This is repeated two more times and the actors *m'* and *m"* are created. When *Factorial* receives the request with the value 0, it responds to the newest actor *m"* with the result 1. This triggers a chain of result messages. The actor *m"* computes the value 1 and sends it to *m'*. This continues till *m* sends the final result 6 to the customer who sent the original request.

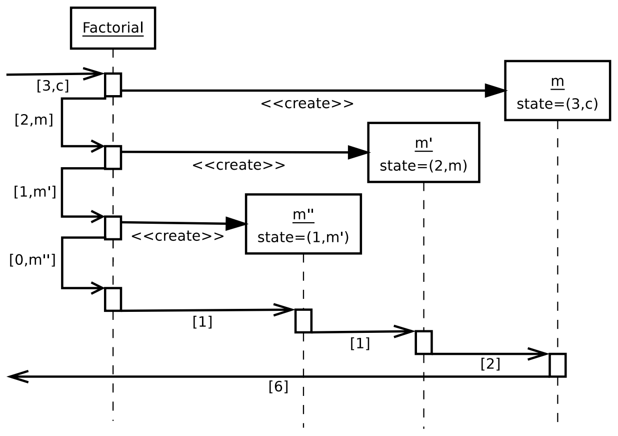

Fig. 1: Sequence diagram computing the factorial of 3 with actors.

Computing the factorial function in this way is not more efficient than using a sequential program. It might even consume more memory because the number of actors is linear in the size of the problem. One possible advantage of this program is that it can distribute the computation of multiple calls to factorial over multiple processors, instead of doing them one after another. In this case the factorial actor receives not only a single request, but multiple requests over time. The created continuation actors are distributed and can do all computations independently. The general pattern of using actors to represent continuation can also be used for more complicated computations. Thus, the techniques used to verify this case study, might be applied to other more complex distributed applications.

3 Modelling Actor Systems

To model actor systems in Event-B [1], it is necessary to define when a model represents an actor system. This requires us to assign Event-B constructs to all components of an actor system, we want to study. The two most important of these components are actors and messages.

Actors have unique identifiers. Hence, we define an Event-B context with a carrier set *ACTOR_ID*. It contains identifiers for dynamically created actors, represented as natural numbers, and two special identifiers. The first one is *final_id* for the customer (main) actor who is outside the modelled system and who, in our example, should receive the final result. To simplify computations, it is represented by the number -1. In addition, we use *invalid_id* if a variable of type *ACTOR_ID* is not used at some point in time. These identifiers are formally defined as Event-B constants with the following axioms:

`axm_0`: $ACTOR\_ID = \mathbb{N} \cup \{-1, invalid\_id\}$
`axm_1`: $final\_id = -1$
`axm_2`: $invalid\_id \notin \mathbb{N}$
`axm_3`: $final\_id \neq invalid\_id$

To represent the actors, multiple new variables are introduced into an Event-B machine: *num_actors* stores the number of existing actors and *actor_id* is a set that stores all actor identifiers which are currently in use. These variables are defined by these invariants:

`inv_1`: $num\_actors \in \mathbb{N}$
`inv_2`: $actor\_id = 0 \mathrel{..} (num\_actors - 1)$

Meaning that exactly the actor identifiers from 0 to $num\_actors - 1$ are valid and a newly created actor will get the next larger number as its identifier.

Actors may have a state. For example, the continuation actors in our example store their value and the target actor who will receive their response. Both of these state variables are represented as functions from *actor_id* to their respective types:

`inv_3`: $cont\_actors\_target \in actor\_id \to (actor\_id \cup \{final\_id\})$
`inv_4`: $cont\_actors\_value \in actor\_id \to \mathbb{N}_1$

Actors communicate asynchronously via message queues. Hence, the most general model would map actor ids to sequences (arrays) of messages. However, we may postulate assumptions in order to simplify the model and consequently verification. For example, for proving that the actor model computes the recursive definition of a factorial function, we may (initially) assume a slow environment, where a new request is only issued after a response has been received. With such a synchronized behaviour, at most one message at a time can exists in our factorial actor system. Hence, it is sufficient to use a single set of variables for all components of messages. The Boolean variable *msg_exists* stores if an unprocessed message exists. The variables *msg_content* and *msg_recipient* store the values of a message. If no message exists, the value of *msg_recipient* will be *invalid_id*. The variable *active_actor* stores the actor which receives the current message if a message exists. Otherwise, it is the identifier of the last actor that was created. We give the formal definition of these variables:

`inv_5`: $msg\_exists \in BOOL$

inv_6: $msg\_recipient \in actor\_id \cup \{final\_id, invalid\_id\}$
inv_7: $msg\_content \in \mathbb{N}_1$
inv_8: $msg\_exists = FALSE \Leftrightarrow msg\_recipient = invalid\_id$
inv_9: $active\_actor = num\_actors - 1$

Note, that we prefer to decompose a message into separate variables over defining a composite message type. Naturally, one would describe a message as a tuple of its fields, e.g., the pair $msg \in (actor\_id \cup \{final\_id, invalid\_id\}) \times \mathbb{N}_1$. The reason for our flat encoding is that the provers tend to have less difficulties with basic data types.

In contrast, in a fully concurrent model, we need to keep track of individual computation requests via a $REQUEST\_ID$. Hence, we model the message queues of the continuation actors as follows[1]:

invC_1: $cont\_mail\_msg\_content \in (ACTOR\_ID \times REQUEST\_ID) \mathrel{+\!\!\!\!\rightarrow} \mathbb{N}_1$

This is the most general model, where each actor has a set of messages to be processed. In this model, too, it is beneficial to split the message queue of an actor into separate queues per message type.

Having discussed the representation of actors in Event-B, we are going to develop the actor model of the factorial.

4 Step-wise Development

In this section we detail the development of the sequential actor model where the environment issues the next computation requests after receiving the result of the previous one. Our formal development of the provably correct factorial actor system follows a refinement strategy. We start with the standard recursive definition of factorial. Then, in five refinements, the details necessary for an actor system are added.

The initial model consists of an event that computes the factorial in one step. The first refinement changes the one step computation into an iterative algorithm. In the second refinement the used memory is made explicit in the form of a stack. We also separate the creation of the memory cells from performing the computation. Refinement 3 is the first one that resembles an actor system. At that point the stack elements are replaced by actors and the computations are triggered by messages. However, the process of creating the actors is still controlled by an iterative program. Refinement 4 turns this last part into an actor, controlled by sending updated messages to itself. Refinement 5 changes the shared mailbox to one mailbox per actor.

4.1 Specification

The specification is captured in an initial model comprising a context that defines the function *fact* and a machine (Figure 2) that uses this function. The context defines two constants, the recursive factorial function *fact* and the *input*:

[1] An arrow with a vertical bar is Event-B's notation for a partial function.

axm0_0: $fact \in \mathbb{N} \rightarrow \mathbb{N}_1$
axm0_1: $fact(0) = 1$
axm0_2: $\forall n \cdot n \in \mathbb{N} \Rightarrow fact(n + 1) = (n + 1) * fact(n)$
axm0_3: $input \in \mathbb{N}$

The initial machine has only a single variable *result* of type $\mathbb{N}$. This variable is used to store the final result of the computation. For brevity, we do not display the variable definitions in the model listings, but show events only. There are two events:

The *Initialisation* event sets the value of *result* to 0.
The *Finish* event contains one action
act0_0: $result := fact(input)$
that assigns *result* to the result computed by the *fact* function. This event contains no guard and may be repeated.

This model is sequential as it assumes that one factorial is computed after each other. A simulation consists of the execution of the *Initialisation* event followed by an unrestricted number of computation steps without any effect. Execution of the *Finish* event computes the factorial in one step. For simplicity, the input is a constant as we are only interested to prove that the refined actor system computes a factorial of an arbitrary input.

4.2 Refinement 1

The first refinement (Figure 3) splits the computation into multiple steps. The new variables *tmp_result* and *val* are introduced to hold the intermediate state. The *result* variable stays part of the machine. The algorithm will do the calculation beginning with the smallest number 1. In each consecutive step the next factorial number is calculated based on the previous one which is stored in *tmp_result*. The number of remaining steps is stored in *val*. The types of this new variables are $\mathbb{N}$ for *val* and $\mathbb{N}_1$ for *tmp_result*.

MACHINE m0
EVENTS
Initialisation
begin
  act0_0: $result := 0$
end
Finish ⟨ordinary⟩ $\widehat{=}$
begin
  act0_0: $result := fact(input)$
end
END

Fig. 2: Events of the Specification.

```
MACHINE m1
EVENTS
Initialisation
begin
   act1_0: $result := 0$
   act1_1: $val := input$
   act1_2: $tmp\_result := 1$
end
ComputeStep ⟨convergent⟩ $\widehat{=}$
when
   grd1_0:  $val > 0$
then
   act1_0: $val := val - 1$
   act1_1: $tmp\_result := tmp\_result * (input - val + 1)$
end
Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish
when
   grd1_0:  $val = 0$
then
   act1_0: $result := tmp\_result$
end
END
```

Fig. 3: Events of Refinement 1.

The *Initialisation* assigns the variable *val* to *input*. We need to do as many steps as the value of the input. To start the computation properly, *tmp_result* is initialised to 1 the multiplicative identity. As in the previous machine, *result* is set to 0.

The event *ComputeStep* is new and we have to show that it is *convergent*. This means that the event must not be enabled infinitely often possibly preventing the other events. The guard states that this event can only be executed if val is not 0, meaning that there is still work to do. The two actions decrement *val* and update *tmp_result* to the next factorial number.

The event *Finish* refines the event of the same name. It now contains a guard. Also, instead of assigning the final result directly, the variable *tmp_result* is assigned to *result*. This event can now only be executed if there are no more computations to do and instead of doing the computation itself, the result is just copied.

In order to demonstrate that this machine is indeed a refinement of the previous machine, we need to confirm that all events refine their corresponding abstract event. It is also required to show that all convergent events are really convergent. That is, there exists a variant, an expression bounded from below which is decreased by every execution of the convergent event.

The event *ComputeStep* is new and the refinement relation is thus trivially satisfied, if we can shown convergence. The variant for this machine is the variable *val*. It is a natural number and thus cannot get infinitely small and it is decremented in *ComputeStep*. Thus, it turns out that *ComputeStep* is indeed convergent.

To justify that *Finish* refines its predecessor event, we need to demonstrate that whenever *val* is 0, the value in *tmp_result* is the correct final result. We can prove this with the following invariants added to the model:

`inv1_2`: $val \leq input$
`inv1_3`: $tmp\_result = fact(input - val)$

We need to prove that these invariants are preserved by all events. The initialisation satisfies both invariants. More interesting is the event *ComputeStep*. Before the event is executed $tmp\_result = fact(input - val)$ and afterwards $tmp\_result' = fact(input - val) * (input - val + 1) = fact(input - val')$ which proves that the invariant inv1_3 is preserved. Here, we use standard notation: primed variables refer to values after event execution while unprimed variables denote values before execution. Constants are always unprimed. The proof of inv1_2 follows from the observation that *val* is only decremented and *input* is a constant.

Finally, we prove deadlock freedom. The corresponding theorem states that at least one event must always be enabled, or expressed differently, the disjunction of all guards must be a valid expression:

`thm_DLF`: ⟨theorem⟩ $(val > 0) \vee (val = 0)$

This theorem follows directly from the type definition invariant of *val*, stating that $val \in \mathbb{N}$. All proof obligations for this refinement can be discharged by the included automatic solvers [17,14]. No manual proofs are required.

4.3 Refinement 2

In the second refinement (Figure 4) the computation is split into two phases. First, all numbers are pushed on a stack, afterwards they are multiplied. This brings us one step closer to the actor system, where first actors are created, then they process messages to perform the actual computation.

For this stack-based model, we need to introduce several new variables: *counter* tracks how many more elements need to be pushed, *stack* is a function that models the stack and *stack_pointer* is the current size of the stack. They are defined by the following invariants:

`inv2_1`: $stack \in \mathbb{N} \pfun \mathbb{N}_1$
`inv2_2`: $stack\_pointer \in \mathbb{N}$
`inv2_3`: $0 \,..\, (stack\_pointer - 1) \subseteq dom(stack)$
`inv2_4`: $counter \in \mathbb{N}$

```
MACHINE m2
EVENTS
Initialisation
begin
   act2_0: $result := 0$
   act2_1: $counter := input$
   act2_2: $stack := \varnothing$
   act2_3: $stack\_pointer := 0$
   act2_4: $tmp\_result := 1$
end
Call ⟨convergent⟩ $\widehat{=}$
when
   grd2_0:  $counter > 0$
   grd2_1:  $tmp\_result = 1$
then
   act2_0: $counter := counter - 1$
   act2_1: $stack\_pointer := stack\_pointer + 1$
   act2_2: $stack(stack\_pointer) := counter$
end
Return ⟨convergent⟩ $\widehat{=}$
refines ComputeStep
when
   grd2_0:  $counter = 0$
   grd2_1:  $stack\_pointer > 0$
then
   act2_0: $tmp\_result := tmp\_result * stack(stack\_pointer - 1)$
   act2_1: $stack\_pointer := stack\_pointer - 1$
end
Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish
when
   grd2_0:  $counter = 0$
   grd2_1:  $stack\_pointer = 0$
then
   act2_0: $result := tmp\_result$
end
END
```

Fig. 4: Events of Refinement 2.

In `inv2_3` the *dom* function is used to get the domain of a function. It means that *stack* is defined for all $\mathbb{N}$ up to, but not including *stack_pointer* which points to the next free space in the stack. The variables *result* and *tmp_result* are the same as in the previous machine. The variable *val* is no longer visible.

The event *Initialisation* sets *counter* to *input*, *stack* to an empty set, *stack_pointer* to 0. The old variables *result* and *tmp_result* are initialised as before to 0 and 1.

The event *Call* is a new convergent event. It is responsible for pushing the numbers on the stack. The guard states that there are numbers left and that the computation has not started. This is needed to satisfy some invariants. The actions push the value of counter and decrement it. To establish that this event is convergent, the variant *counter* is used.

The event *Return* is a refinement of *ComputeStep*. Its guard requires that all elements are pushed and that the stack is non-empty. When executed, it pops one element and multiplies it with *tmp_result*. The value of *tmp_result* is the same as in the previous refinement. The difference is that now it is computed based on a stack element and not based on a simple variable.

The event *Finish* is almost the same as in the previous refinement. Only the guard is slightly different.

To establish the refinement relationship, we need to relate the new variables to the old ones of the more abstract model. This relation is defined in so called gluing invariants as follows:

inv2_5: $\forall n \cdot n \in dom(stack) \Rightarrow stack(n) = input - n$
inv2_6: $stack\_pointer + counter = val$
inv2_7: $counter = 0 \Rightarrow val = stack\_pointer$
inv2_8: $counter \neq 0 \Rightarrow val = input$

The invariant `inv2_5` allows us to know the value on the stack, which is important for the proof obligations related to the *return* event. The other three invariants state how *counter*, *stack_pointer*, and *val* are related. While the event *Call* is executed, the value of *val* stays at *input*. At the same time *counter* and *stack_pointer* are decremented and incremented, but always both. Once *counter* is 0 and the execution of *return* starts, the *stack_pointer* takes the role of *val*. The invariant `inv2_7` follows directly from `inv2_6`, it could be marked as a theorem. Using these invariants, all proof obligations can be discharged by the automatic solvers [17,14]. The deadlock freedom theorem

thm_DLF: ⟨theorem⟩ $(counter > 0 \wedge tmp\_result = 1) \vee$
$(counter = 0 \wedge stack\_pointer > 0) \vee (counter = 0 \wedge stack\_pointer = 0)$

is also proven automatically.

MACHINE m3
EVENTS
Create ⟨convergent⟩ $\widehat{=}$
refines Call
when
grd3_0: $counter > 0$
then
act3_0: $counter := counter - 1$
act3_1: $actor\_id := 0 \mathrel{..} num\_actors$
act3_2: $cont\_actors\_target(active\_actor + 1) := active\_actor$

```
    act3_3: $cont\_actors\_value(active\_actor + 1) := counter$
    act3_4: $active\_actor := active\_actor + 1$
    act3_5: $num\_actors := num\_actors + 1$
end
Created ⟨convergent⟩ $\widehat{=}$
when
    grd3_0: $counter = 0$
    grd3_1: $msg\_exists = FALSE$
    grd3_2: $active\_actor = input - 1$
then
    act3_0: $msg\_exists := TRUE$
    act3_1: $msg\_recipient := active\_actor$
    act3_2: $msg\_content := 1$
end
Compute ⟨convergent⟩ $\widehat{=}$
refines Return
when
    grd3_1: $msg\_exists = TRUE$
    grd3_2: $msg\_recipient \neq final\_id$
    grd3_3: $msg\_recipient = active\_actor$
then
    act3_0: $msg\_recipient := cont\_actors\_target(msg\_recipient)$
    act3_1: $msg\_content := msg\_content * cont\_actors\_value(msg\_recipient)$
    act3_2: $num\_actors := num\_actors - 1$
    act3_3: $actor\_id := 0 \mathinner{..} num\_actors - 2$
    act3_4: $cont\_actors\_target := \{msg\_recipient\} \lhd\!\!\!- cont\_actors\_target$
    act3_5: $cont\_actors\_value := \{msg\_recipient\} \lhd\!\!\!- cont\_actors\_value$
    act3_6: $active\_actor := cont\_actors\_target(msg\_recipient)$
end
Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish
when
    grd3_1: $msg\_exists = TRUE$
    grd3_2: $msg\_recipient = final\_id$
    grd3_3: $msg\_recipient = active\_actor$
then
    act3_0: $result := msg\_content$
end
END
```

Fig. 5: Events of Refinement 3.

4.4 Refinement 3

With the third refinement (Figure 5), we start to introduce actors. The stack, used in the previous refinement, is now represented as actors and the computation phase is controlled by messages sent between these actors. Actor identifiers and the number of existing actors are defined as described in Section 3.
In this refinement step only the memory for the computation is represented as actors. This corresponds to the behaviour *cont* in Listing 1. The state of these actors consists of *value* and *target* as defined in Section 3. Also the simplified

model for messages for these actors has been presented in Section 3. Furthermore, the two variables *result* and *counter* are taken from the previous refinement. The machine consists of five events, one more than in the previous refinement.

In the *Initialisation* event most variables are set to empty or default values. The variable *active_actor* is set to -1, meaning that no dynamic actor exists at that point. For brevity the initialisation event is not shown in Figure 5.

The event *Create* refines the event *Call*. It creates continuation actors. The guard is a subset of the guard of *Call*. When executed, the counter is decremented and a new continuation actor is created. To create this new actor the *num_actors* variable is incremented, the *actor_id* variable is extended, and the state is added to *cont_actors_target* and *cont_actors_value*. Additionally, the *active_actor* variable is set to the id of the newly created actor.

The newly introduced event *Created* is enabled when the counter reaches zero, but no message was sent to a continuation actor. It is responsible for starting the computation by sending the first message to the continuation actor that was created last. To do so, the *msg_exists* flag is set to *true* and the other *msg* variables are filled. This event is introduced in this refinement and marked as convergent. So we need to provide a suitable variant. We know that this event is only executed once and it is the only event that changes *msg_exists*. To build a variant out of this Boolean variable, we need an auxiliary function that turns the Boolean value into an integer and decreases when the input changes from $false$ to *true*. The following definition is part of the context:

axm3_4: $boolToNat \in BOOL \rightarrow \mathbb{N}$
axm3_5: $boolToNat(TRUE) = 0$
axm3_6: $boolToNat(FALSE) = 1$

By using it, the variant can be defined as *boolToNat(msg_exists)*.

The event *Compute* is the receive function of the continuation actors. It is enabled whenever there exists a message for one of these actors. It also contains two additional guards to keep the system synchronized. When the event is executed, a message is sent to the stored target. The message contains the product of the stored value and the value received via the latest message. This corresponds to the *cont* behaviour in the actor algorithm in Listing 1. Additionally, the actor who processed the message is deleted, as there will be no more messages for it to process. This is done by removing it from the *cont_actors* functions and from the *actor_id* set.

The *Finish* event corresponds to the customer who receives the final result. It is enabled if a message arrives at this customer. When executed, the variable *result* is set to the received result in the message.

In order to demonstrate that this third machine is a refinement of the second machine, we need to provide some gluing invariants. These relate the now invisible variables of the stack system, to the new variables of the actor system. The roles of *tmp_result* and *stack_pointer* are now taken by *msg_content* and *num_actor*. In fact, these variables are equivalent to its predecessors, they are just renamed to be a better fit for describing an actor system. The content of the continuation actors is equivalent to stack frames in the second refinement. The gluing invariants are formally stated as:

inv3_8: $msg\_exists = TRUE \Rightarrow counter = 0$
inv3_11: $msg\_content = tmp\_result$
inv3_12: $num\_actors = stack\_pointer$
inv3_14: $\forall x \cdot x \in dom(cont\_actors\_value) \Rightarrow stack(x) = cont\_actors\_value(x)$
inv3_15: $\forall x \cdot x \in dom(cont\_actors\_target) \Rightarrow cont\_actors\_target(x) = x - 1$

As for all the previous machines, we need to provide a deadlock freedom theorem. In this case we need to provide additional invariants to prove it because our existing invariants are not strong enough. The value of *msg_recipient* needs to be derived correctly from the other information known in a guard. There is no way to guarantee its values independently.

inv3_16: $msg\_exists = TRUE \Rightarrow msg\_recipient = active\_actor$
inv3_17: $(counter = 0 \wedge msg\_exists = FALSE) \Rightarrow active\_actor = input - 1$

With this additional invariants the deadlock freedom theorem, i.e. the disjunction of all guards equals true, can be proven. The proof obligations from the invariants and the deadlock freedom theorem are all automatically discharged by the solvers [17,14]. The only manual intervention was the creation of the two additional invariants for deadlock freedom.

4.5 Refinement 4

In the fourth refinement (Figure 6), we replace the counter variable by a message. This message is sent by the factorial actor to itself. It corresponds to the *Request* message and the *fact* actor in Listing 1. To model this message, we introduce a new channel consisting of the variables *msgC_exists* and *msgC_content*. The *msgC* prefix expresses that these variables belong to the message that sends the counter. The variable *counter* from the previous refinement is removed, all other variables stay the same. The variables are defined, including the gluing invariant, as follows:

inv4_0: $msgC\_exists \in BOOL$
inv4_1: $msgC\_content \in \mathbb{N}$
inv4_2: $msgC\_content = counter$

The number and names of the events are unchanged, compared to the previous refinement.

The *Initialisation* is the same as before, except for the new variables.

The event *Create* is modified to handle the new message. Instead of checking the value of the *counter*, the existence of the message and its value are checked. The decremented *counter* is not updated directly, but instead sent as a message. The *msgC_exists* flag is already *true*, thus unchanged, and the message content is written to *msgC_content*.

```
MACHINE m4
EVENTS
Create ⟨convergent⟩ ≙
refines Create
when
   grd3_0:  $msgC\_exists = TRUE$
   grd3_1:  $msgC\_content > 0$
then
   act3_0: $msgC\_content := msgC\_content - 1$
   act3_1: $actor\_id := 0 \,..\, num\_actors$
   act3_2: $cont\_actors\_target(active\_actor + 1) := active\_actor$
   act3_3: $cont\_actors\_value(active\_actor + 1) := msgC\_content$
   act3_4: $active\_actor := active\_actor + 1$
   act3_5: $num\_actors := num\_actors + 1$
end
Created ⟨convergent⟩ ≙
refines Created
when
   grd3_0:  $msgC\_exists = TRUE$
   grd3_1:  $msgC\_content = 0$
   grd3_2:  $msg\_exists = FALSE$
   grd3_3:  $active\_actor = input - 1$
then
   act3_0: $msg\_exists := TRUE$
   act3_1: $msg\_recipient := active\_actor$
   act3_2: $msg\_content := 1$
   act3_3: $msgC\_exists := FALSE$
end
END
```

Fig. 6: Events of Refinement 4. The events *Compute* and *Finish* are the same as in Figure 5.

The event *Created* is also modified to work with the counter message. The guard now checks the existence of the message and whether its content is 0. An additional action supplements the actions of the event. After the last counter message was handled, the channel will be empty, as this event does not create a new one. Thus, the value of the *msgC_exists* flag needs to be changed.

The events *Compute* and *Finish* are the same as in the previous refinement.

The proofs for refinement and deadlock freedom are done automatically [17,14]. To establish the deadlock freedom theorem, we need this additional invariant:

inv4_3: $msgC\_exists = FALSE \Rightarrow msg\_exists = TRUE$

4.6 Refinement 5

The fifth and last refinement (see Appendix) changes the mailboxes to arrays and uses separate ones for each actor. This follows the technique for truly concurrent systems described in Section 3 and gives a model that better resembles an actor system.
We introduce the new variables for the mailboxes of the *fact* and *cont* actors. They replace the variables *msg_exists*, *msg_content*, *active_actor*, *msgC_exists* and *msgC_content*. Their types are defined by the following invariants:

inv5_0: $fact\_mail\_msgC\_content \in \mathbb{N} \rightarrow\!\!\!\!\!\!+ \mathbb{N}$
inv5_3: $fact\_index\_msgC \in \mathbb{N}$
inv5_4: $cont\_mail\_msg\_content \in (ACTOR\_ID \times \mathbb{N}) \rightarrow\!\!\!\!\!\!+ \mathbb{N}$
inv5_6: $cont\_index\_msg \in \mathbb{N}$

The state variables as well as the *result* and *actor_id* variables are unchanged compared to the previous refinement. The events are adapted to the new message encoding in a relatively straightforward way. There is no change in the processing logic.
To satisfy the refinement condition, we need gluing invariants to link the old mailbox variables to the new ones. Note that the model still adheres to the restriction that there can be only one message in all the continuation actor mailboxes. This message must be in the mailbox of the actor identified by the now hidden *active_actor* variable. The Boolean *exists* flags are replaced by using an empty set instead. This gives us these gluing invariants:

inv5_1: $msgC\_exists = TRUE \Leftrightarrow$
$ran(fact\_mail\_msgC\_content) = \{msgC\_content\}$
inv5_2: $msgC\_exists = FALSE \Leftrightarrow fact\_mail\_msgC\_content = \varnothing$
inv5_7: $msg\_exists = TRUE \Leftrightarrow ran(cont\_mail\_msg\_content) = \{msg\_content\}$
inv5_8: $msg\_exists = FALSE \Leftrightarrow cont\_mail\_msg\_content = \varnothing$
inv5_9: $\exists n \cdot msg\_exists = TRUE \Rightarrow$
$dom(cont\_mail\_msg\_content) = \{active\_actor \mapsto n\}$

Again, with these invariants all proofs are found fully automatically.

5 Concurrent Version

The previous model has one major limitation: it can only perform the computation once and, hence, behaves like a sequential program. Even though the actor program in Listing 1 can compute the solution for multiple requests, these requests can also occur while the previous computation is still running. In that case the two computations are performed concurrently and can be interleaved. In this section we adapt our previous factorial model to handle concurrent requests like the Scala version.
It is not possible to realize this as a refinement of the previous machine. Instead we create a new specification machine and refine it to an actor system as in

```
MACHINE m0c
EVENTS
Initialisation
begin
  act0_0: $tasks := \varnothing$
  act0_1: $results := \varnothing$
end
Start ⟨ordinary⟩ $\widehat{=}$
any
  input
  task
where
  grd0_0: $input \in \mathbb{N}$
  grd0_1: $task \notin dom(tasks)$
then
  act0_0: $tasks(task) := input$
end
Finish ⟨ordinary⟩ $\widehat{=}$
any
  task
where
  grd0_0: $task \in dom(tasks)$
  grd0_1: $task \notin dom(results)$
then
  act0_0: $results(task) := fact(tasks(task))$
end
END
```

Fig. 7: Events of the concurrent specification.

the previous section. The concurrent machines follow the same structure as before, but we introduce a task identifier to associate each continuation actor and message with a task.

The concurrent specification (Figure 7) contains variables for *tasks* and for *results*. The *start* event expects as parameters an input and a unique task identifier, it adds these to the set of tasks. Analogue to the *Finish* event in the previous section the *Finish* event here calculates the factorial number in one step. Instead of accessing the constant *input* it processes one of the tasks that do not yet have an associated result. The newly computed number is inserted into the results. That way the model is able to handle arbitrary many requests instead of only one.

All refinements closely follow the ones from the previous section. The events are similar, but they all expect a task parameter to know which task is processed. Variables and invariants need to be lifted to the set of tasks. An invariant that previously had the form $\varphi(input, result)$ becomes in this model

$\forall task : \varphi(tasks(task), results(task))$. Except for the newly added *Start* event all machines contain the same events as in the sequential case.
Changing all of the variables to functions leads to some proofs requiring manual intervention. Table 1 shows how many proof obligations where generated for each refinement and how many of them where done automatically. We can see that 29 out of 305 proof obligations required interactive proof. This is contrast to the sequential actor model where all proofs were done automatically. This demonstrates the effect of more complex data structures (here functions) to proof automation.

Element	Total	Auto	Manual
ctx0c	2	1	1
ctx3c	0	0	0
m0c	8	8	0
m1c	26	24	2
m2c	50	47	3
m3c	129	110	19
m4c	33	33	0
m5c	57	53	4
Σ	305	276	29

Table 1: Proof statistics for the concurrent model.

6 Related Work

Type systems have been used in conjunction with actors. Charalambides et al. [12] apply session types to actor systems. This has been extended to also prove liveness properties of actor systems [13]. Our method on the other hand uses refinement to develop a model in multiple steps. The specification is built gradually and the model is separate from a possible program.
Rebeca is a modelling language and model checker for actor systems [30,29]. However, Rebeca cannot deal with the dynamic creation of actors necessary for the factorial case study. Another actor modelling language is ABS [20]. It is an executable and formally specified language based on the active object variant of actor systems. ABS has been used in large industrial case studies [4]. KeY-ABS [15,16] allows tool-based reasoning about ABS specifications. However, ABS does not support refinement.
Musser and Varela [25] developed an actor theory in the Athena proof assistant. Using Athena, they proved properties about actor systems like uniqueness of addresses or fairness. Their theory supports the creation of actors and the exchange of actor identifiers. Another implementation of actor systems was done

in the Coq proof assistant by [32]. They also modelled Agha's factorial example [3] but without a complete correctness prove. Their system can export Erlang code and they proved uniqueness for their address generation and fairness. Both of these works use correctness properties as theorems. However, they do not use stepwise refinement or any other iterative process to develop the final program from the specification.

7 Conclusion

In this paper we studied the formal development of actor systems in Event-B using refinement. Starting from a mathematical recursive specification, we have proven with five refinement steps that Agha's classical factorial actor system is correct. We have also proven deadlock-freeness and convergence from which the termination of a single computation follows. With the assumption that requests are issued synchronously, we could keep the actor model flat and all proofs could be resolved automatically — once the necessary invariants have been added. Our actor models use a naming scheme and actor code could be generated from it, in principle, although this has not been implemented.
To the best of our knowledge, we are the first who formally verified that Agha's factorial actor system implements its recursive definition. Furthermore, we think that we are the first who developed actor systems in Event-B. The example might be simple, but it shows how recursive definitions can be turned into actor systems. Furthermore, the case study demonstrates the proof power of the available provers. The key to this high automation is to keep the actor model as simple as possible: we exploited the synchronous nature of the recursive definition and kept the actor model flat, avoiding composite data structures.
The full development of the truly concurrent factorial model discussed in Section 5 can be found in [23]. It uses the insights from the synchronous development and shows that with the more involved data structures we loose proof automation: 9.5% of the 305 proof obligations needed manual intervention, which is still acceptable. Maderbacher also develops a messaging client-server system which demonstrates the applicability of the method beyond the factorial example.
We strongly believe that abstract models and refinement are essential to the development of dependable distributed systems. An abstract model provides the necessary global view and complexity needs to be added incrementally. Starting at the actor or code level is too late and one has difficulties in stating the correctness properties. This is demonstrated by the observation that we seem to be the first who formally proved the correctness of the classical factorial actor system — which was quite surprising to us.

Acknowledgement. This work is supported by the TU Graz LEAD project Dependable Internet of Things in Adverse Environments. The authors wish to thank the three anonymous reviewers for their constructive feedback in order to improve the paper.

Appendix

The complete model of the final actor model computing a factorial number (Refinement 5).

MACHINE m5
REFINES m4
SEES ctx5
VARIABLES

result
num_actors
actor_id
cont_actors_target
cont_actors_value
fact_mail_msgC_content
fact_index_msgC
cont_mail_msg_content
cont_index_msg

INVARIANTS

inv5_0: $fact\_mail\_msgC\_content \in \mathbb{N} \nrightarrow \mathbb{N}$
inv5_1: $msgC\_exists = TRUE \Leftrightarrow ran(fact\_mail\_msgC\_content) = \{msgC\_content\}$
inv5_2: $msgC\_exists = FALSE \Leftrightarrow fact\_mail\_msgC\_content = \varnothing$
inv5_3: $fact\_index\_msgC \in \mathbb{N}$
inv5_4: $cont\_mail\_msg\_content \in (ACTOR\_ID \times \mathbb{N}) \nrightarrow \mathbb{N}$
inv5_6: $cont\_index\_msg \in \mathbb{N}$
inv5_7: $msg\_exists = TRUE \Leftrightarrow ran(cont\_mail\_msg\_content) = \{msg\_content\}$
inv5_8: $msg\_exists = FALSE \Leftrightarrow cont\_mail\_msg\_content = \varnothing$
inv5_9: $\exists n \cdot msg\_exists = TRUE \Rightarrow dom(cont\_mail\_msg\_content) = \{active\_actor \mapsto n\}$

EVENTS
Initialisation
begin

act5_0: $result := 0$
act5_5: $num\_actors := 0$
act5_6: $actor\_id := \varnothing$
act5_7: $cont\_actors\_target := \varnothing$
act5_8: $cont\_actors\_value := \varnothing$
act5_9: $fact\_mail\_msgC\_content := \{0 \mapsto input\}$
act5_10: $fact\_index\_msgC := 1$
act5_11: $cont\_mail\_msg\_content := \varnothing$
act5_13: $cont\_index\_msg := 0$

end
Create ⟨convergent⟩ $\widehat{=}$
refines Create
any

content
index

where

grd5_0: $index \in dom(fact\_mail\_msgC\_content)$
grd5_1: $fact\_mail\_msgC\_content(index) = content$
grd5_2: $content > 0$
then
act5_0: $fact\_mail\_msgC\_content := \{fact\_index\_msgC \mapsto content - 1\}$
act5_1: $fact\_index\_msgC := fact\_index\_msgC + 1$
act5_2: $actor\_id := 0 \mathbin{..} num\_actors$
act5_3: $cont\_actors\_target(num\_actors) := num\_actors - 1$
act5_4: $cont\_actors\_value(num\_actors) := content$
act5_6: $num\_actors := num\_actors + 1$
end
Created ⟨convergent⟩ $\widehat{=}$
refines Created
any
index
where
grd5_0: $\{index\} = dom(fact\_mail\_msgC\_content)$
we need to guarante that there is only one msg, because of the previous machines
grd5_1: $fact\_mail\_msgC\_content(index) = 0$
grd5_2: $cont\_mail\_msg\_content = \varnothing$
grd5_3: $num\_actors = input$
then
act5_0: $cont\_mail\_msg\_content := \{(num\_actors{-}1 \mapsto cont\_index\_msg) \mapsto 1\}$
act5_3: $fact\_mail\_msgC\_content := \{index\} ⩤ fact\_mail\_msgC\_content$
end
ContCompute ⟨ordinary⟩ $\widehat{=}$
refines Compute
any
actor
index
where
grd5_0: $\{actor \mapsto index\} = dom(cont\_mail\_msg\_content)$
grd5_1: $actor \neq final\_id$
then
act5_1: $cont\_mail\_msg\_content := \{(cont\_actors\_target(actor) \mapsto cont\_index\_msg) \mapsto (cont\_mail\_msg\_content(actor \mapsto index) * cont\_actors\_value(actor))\}$
act5_2: $num\_actors := num\_actors - 1$
act5_3: $actor\_id := 0 \mathbin{..} num\_actors - 2$
act5_4: $cont\_actors\_target := \{actor\} ⩤ cont\_actors\_target$
act5_5: $cont\_actors\_value := \{actor\} ⩤ cont\_actors\_value$
end
Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish
any
actor
index
where
grd5_0: $\{actor \mapsto index\} = dom(cont\_mail\_msg\_content)$
grd3_1: $actor = final\_id$

```
then
  act3_0: result := cont_mail_msg_content(actor ↦ index)
end
END
```

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B **12**(6), 447–466 (2010)
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
4. Albert, E., de Boer, F.S., Hhnle, R., Johnsen, E.B., Schlatte, R., Tarifa, S.L.T., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: An industrial case study using Real-Time ABS. Service Oriented Computing and Applications **8**(4), 323–339 (2014)
5. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Programmers, The Pragmatic Bookshelf, second edition edn. (2013)
6. Armstrong, J., Virding, R., Williams, M.: Concurrent Programming in ERLANG. Prentice Hall (1993)
7. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison-Wesley (2000)
8. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2999, pp. 1–20. Springer (2004)
9. Bevemyr, J.: How Cisco is using Erlang for intent-based networking (2018), https://youtu.be/077-XJv6PLQ, Code Beam Stockholm
10. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 68–78. ACM (2008)
11. Cesarini, F.: Which companies are using Erlang, and why? (2019-09-11), https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html
12. Charalambides, M., Dinges, P., Agha, G.: Parameterized concurrent multi-party session types. In: Kokash, N., Ravara, A. (eds.) Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012. EPTCS, vol. 91, pp. 16–30 (2012)
13. Charalambides, M., Palmskog, K., Agha, G.: Types for progress in actor programs. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 11665, pp. 315–339. Springer (2019)

14. Clearsy: Atelier B (2016), https://www.atelierb.eu/en/atelier-b-tools/
15. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. pp. 517–526. Lecture Notes in Computer Science, Springer (2015)
16. Din, C.C., Tarifa, S.L.T., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings. pp. 217–233. Lecture Notes in Computer Science, Springer (2015)
17. Dharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT Solvers for Rodin. In: Derrick, J., Fitzgerald, J.S., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7316, pp. 194–207. Springer (2012)
18. Ericsson: Erlang celebrates 20 years as open source (2018-05-31), https://www.ericsson.com/en/news/2018/5/erlang-celebrates-20-years-as-open-source
19. Hewitt, C.: Actor model of computation: Scalable Robust Information Systems. arXiv (2010), http://arxiv.org/abs/1008.1459
20. Johnsen, E.B., Hhnle, R., Schfer, J., Schlatte, R., Steffen, M.: ABS: a core language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers. Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2010)
21. Lightbend: Akka Documentation (2019), https://doc.akka.io/docs/akka/2.5/index.html
22. Lightbend: Akka Typed Documentation (2019), https://doc.akka.io/docs/akka/2.5/typed/actors.html
23. Maderbacher, B.: Proof-Based Development of Actor Systems. Master's thesis, Graz University of Technology, Institute of Software Technology (December 2019), supervisor: Bernhard K. Aichernig
24. Metz, C.: Why WhatsApp Only Needs 50 Engineers for Its 900M Users. WIRED (2015), https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/
25. Musser, D.R., Varela, C.A.: Structured reasoning about actor systems. In: Jamali, N., Ricci, A., Weiss, G., Yonezawa, A. (eds.) Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013. pp. 37–48. ACM (2013)
26. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Communications of the ACM **58**(4), 66–73 (2015)
27. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc (2008)
28. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems **15**(4), 391–411 (1997)

29. Sirjani, M.: Power is overrated, go for friendliness! Expressiveness, faithfulness, and usability in modeling: The actor experience. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10760, pp. 423–448. Springer (2018)
30. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and Verification of Reactive Systems using Rebeca. Fundamenta Informaticae **63**(4), 385–410 (2004), http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05
31. WhatsApp: 1 million is so 2011 (2012-01-06), https://blog.whatsapp.com/196/1-million-is-so-2011
32. Yasutake, S., Watanabe, T.: Actario: a framework for reasoning about actor systems. Tech. rep., Tokyo Institute of Technology (2015)