# Verification of Actor Systems Needs Specification Techniques for Strong Causality and Hierarchical Reasoning *

## — Position Paper —

Arnd Poetzsch-Heffter, Ilham W. Kurnia, Christoph Feller

University of Kaiserslautern, Germany
{poetzsch,ilham,c_feller}@cs.uni-kl.de

**Abstract.** Actor languages become increasingly popular for modeling and programming concurrent and distributed system. Although several foundational proof systems are available for actor systems, we claim that there is a need for higher-level specification and verification techniques. To clarify and substantiate this position, we introduce a simple actor language together with a non-trivial example and show how language-based specification techniques, generalizing ideas from object-oriented specifications, can provide more structure and modularity. We explain the semantics of the specification constructs based on strong causality relations of incoming and outgoing messages and illustrate their use. Furthermore, we discuss how verification techniques can be constructed for such specifications. The paper finishes with a discussion of related and future work.

## 1 Introduction

For analysis, checking, and verification, many modern programming languages are complemented by specification languages that allow the formulation of program-specific properties (e.g., Java and JML [22], C# and Spec# [5], Ada and Anna [25]). Whereas these specification languages mainly focus on sequential programming, researchers more and more investigate multi-threading and other concurrency aspects of such programming languages. On the other hand, there is a trend to develop modeling and programming languages directly based on concurrency primitives, e.g., based on actors or processes (e.g. [3,18,31,7,20,19]). A strong argument for these approaches is that their concurrency constructs usually come with a well-understood theory and often foundational reasoning technique. However, the development of specification and high-level reasoning techniques for these languages is less advanced than for the classical programming languages.

In this paper, we focus on actor languages in which a system or component is described as a dynamically varying set of actors. An actor can create other actors and communicate with other actors via asynchronous messages. When used for

* This work is partially supported by the EU project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models* and the RS3 project *MoVeSPAcI: Modular Verification of Security Properties in Actor Implementations*.

modeling purposes, actor languages need specification languages to state and verify properties of the modeled systems. We claim that the specification techniques and corresponding verification methods to develop such specification languages are not sufficiently understood. Formulated positively, we aim to convey that the development of specification techniques for actor languages is an interesting and fruitful research area.

More precisely, the *position of this paper* is that the following questions still wait for satisfying solutions:

1. What are the central specification constructs for actors?
2. How do we achieve scaling from single actors to actor systems?
3. How can we build higher-level reasoning techniques based on such actor specifications?

In the following, we explain these questions in turn by comparing them to the specification techniques for classical programming languages. In addition, we shortly introduce the approaches to these questions that we will further elaborate in the rest of the paper to clarify the stated position.

The central specification construct for programming languages centers around procedures/methods and is based on pre- and postconditions (see, e.g., [28]). A pre-/post-specification nicely abstracts from the execution sequence of a procedure by talking only about its initial and final states. In object-oriented (OO) languages, pre-/post-specifications of methods are complemented by object invariants (see, e.g., [24]). What are the corresponding central abstractions to specify actors living in a concurrent environment? Our approach is to specify the reaction to incoming messages.[1] As a new concept, we will propose to use so-called *strong causality relations* between incoming messages and the resulting outgoing messages as a generalization of pre-/post-specifications.

In sequential settings, procedures hierarchically structure the complete system runs. One call might lead to subcalls. The behavior of the main call can be derived from the behaviors of the subcalls. Similar to OO settings, the scaling in actor programs is less clear. Specification techniques for OO programs often use ownership concepts for scaling from single objects to object structures (see, e.g., [32,5]). Our approach is similar in that we will propose to specify the behavior of components consisting of groups of actors. In particular, we aim to specify such components with the same technique as actors.

There are foundational theories how to reason about actor systems (see, e.g., [1,14,2]). These theories directly build on the traces generated by the actor systems. This is very flexible and powerful. However, for practical purposes, it can get quite complex and elaborate. Are there higher-level, easier to handle verification methods for reasoning based on a specification technique, maybe even at the expense of losing relative completeness? In this paper, we will only discuss the topic in relation to the presented specification techniques.

[1] How this approach can be extended to spontaneous actions will be discussed below.

```
interface Client { void receive( Value );   }

interface Server { void serve( Client, CompTask ); }

interface Worker {
   void do( CompTask, Int );
   void propagateResult( Value, Client );
}

actor class AServer implements Server {
   void serve( Client c, CompTask t ) {
      Int    numberSubtasks = taskSize(t); // numberSubtasks >= 1
      Worker w = new AWorker();
      w.do( t, numberSubtasks );
      w.propagateResult( null, c );
   }
}
```

**Fig. 1.** Interfaces and the actor class `AServer`

*Paper outline.* To clarify and substantiate the sketched position, we discuss the questions mentioned above in a concrete setting. Sect. 2 describes a simple actor language that we call AJ and a non-trivial example written in AJ. In Sect. 3, we present and discuss a candidate for a specification technique and apply it to the running example. In Sect. 4, we discuss verification aspects. Sect. 5 contains summarizing discussions, further comparisons to related work, and our aims in future work.

## 2 Actor Language and Actor Systems

To have a sufficiently clear background for the following discussion on specification and verification, we informally introduce the core actor language AJ together with a client-server example. The server takes a computation task, splits it up into a number of subtasks that are concurrently executed, collects and merges the results, and sends the final result back to the client (this is a simplified version of the ring example treated in a case study by Arts and Dam [4]).

Actors share many properties with objects. In particular, they are declared using classes (we use the keyword **`actor class`**), can be dynamically created, implement interfaces, have an actor-local state expressed in terms of instance variables, and are addressed via a typed reference. Thus, for these aspects, we can use a syntax similar to OO programming languages. For example, Fig. 1 shows the interfaces `Client`, `Server`, and `Worker` together with an implementation of the `Server`-interface.

*Reacting to messages.* The central difference of actors and objects is the treatment of messages and methods. In AJ, a *message* consists of a name and typed parameters,

but must not have a return type/value[2], and the semantics of message sending is different. A statement of the form `r.m(p1,p2)` is executed by sending the message `m` with actual parameters `p1` and `p2` to the receiver actor `r`. Such a send-operation is non-blocking; execution directly continues with the next statement. Thus, in general, a message send leads to concurrent behavior. For each of its messages, an actor has a *body* that describes how it reacts to a message. For example, an actor of class `AServer` (see Fig. 1) reacts on a message `serve` as follows: It determines the number of subtasks into which task `t` should be split, creates a worker actor, and sends first a `do-` and then a `propagateResult` message to the worker. For AJ, we make the following simplifying assumptions (we discuss the assumptions in Sect. 5):

A1 Messages sent from one sender to the same receiver are transmitted in order.
A2 Actors are single-threaded, i.e., they work on at most one message body at a time.
A3 Message bodies are executed without interruption.
A4 The execution of message bodies must terminate. This is an obligation of the programmer. (If an actor should be non-terminating, it can send itself a message at the end of the message body.)

*Receiving and selecting messages.* It remains to explain what happens on a message receive. We assume that actors are input enabled (cf. [27, p. 257]) and have an infinite input queue. Messages are *selected* from the queue essentially in a FIFO manner. However if they have a guard that evaluates to false, their selection is postponed. Thus, an actor has control over the execution of incoming messages. Message selection is fair for messages with true guards. In Fig. 2, the actor class `AWorker` uses a guard to select a `propagateResult`-message only if a result is available.

*Further constructs.* In addition to the actor-related aspects, we assume some basic language constructs usually present in functional programming languages. In the example, we use the types `CompTask` and `Value` and the functions:

```
compute   :  CompTask ⟶ Value
taskSize  :  CompTask ⟶ Int
firstTask :  CompTask × Int ⟶ CompTask
restTask  :  CompTask × Int ⟶ CompTask
merge     :  Value × Value ⟶ Value
```

where `compute(t)` computes the result of `t`; `taskSize(t)` yields a number of subtask in which `t` could be reasonably partitioned; `firstTask(t,n)`, for $n \geq 1$, chops off the last $n-1$-th part of `t`, `restTask(t,n)` returns the chopped part; and `merge` merges results. We assume in particular:

```
taskSize(t) ≥ 1
n > 1 → compute(t) = merge(compute(firstTask(t,n)),compute(restTask(t,n)))
compute(t) = compute(firstTask(t,n))
merge(v,null) = v
merge(v,merge(w,x)) = merge(merge(v,w),x)
```

[2] I.e., return types are always **void**.

```
actor class AWorker implements Worker {
   Value  myResult = null;
   Worker nextWorker = null;

   void do( CompTask t, Int n ) {
      if( n > 1 ) {
         nextWorker = new AWorker();
         nextWorker.do( restTask(t,n), n-1 );
      } else {
         nextWorker = null;
      }
      myResult = compute( firstTask(t,n) );
   }

   void propagateResult( Value v, Client c )
      guard  myResult != null
   {
      if( nextWorker == null ) {
          c.receive( merge(myResult,v) );
      } else {
          nextWorker.propagateResult( merge(myResult,v), c );
      }
   }
}
```

**Fig. 2.** Actor class `AWorker`

Actor systems are started by creating actors on computers and start their activities or connect them to activities in the environment, for example to user interfaces.

We designed AJ in such a way that it supports the core features of actor programming and is easy to understand for the object community in order to simplify the bridge to the OO specification community. A comparison of AJ in the context of related work is contained in Sect. 5.

## 3 Specifications of Functional Properties

Even if described by a simple language such as AJ, actor systems can become very complex with almost unmanageable behavior. Of course, in theory we can capture this behavior in form of trace sets and formalize properties as predicates on these sets. However, for practical purposes, we would like to have a more modular and more structured approach. Modularity would allow us for example to prove properties about our server actors of Sect. 2 without knowing all possible contexts in which the servers are used. The hierarchical structuring should help us to develop specifications for typical programs. Our position is that improvements w.r.t. modularity and hierarchical techniques are necessary to master actor systems.

To clarify and substantiate the above claim, we describe how such improvements could look like. We focus on the basic ideas and make several simplifying assumptions. Furthermore, we try to reuse OO specification technology where appropriate. Our central goal is

> *a specification technique that works for single actors and scales to groups and components consisting of many actors.*

The reason for looking at groups of actors is that the user of an actor is rarely interested in how the actor does its work. For example, a client using our server is not interested how the server splits the work and assigns it to several workers and how these do their work concurrently; the client is only interested in getting the correct result. In the following, we only talk about actor groups and not actor components[3], but we believe that the main aspects of what we develop also hold for components built from actors. Before we turn back to groups, we consider the specification of single actors.

### 3.1 Specifying Single Actors

Our basic specification construct is designed to express the behavior of an actor in reaction to a method selection. The construct has the following form:

*input_message* $\Longrightarrow$ set of *actor_creation*;
REGEXP(*output_messages*)
**assume** *boolean_expression*
**assert** *boolean_expression*

We call the specification construct a *reaction rule*, because it can be read as defining a transition in reaction to an input: When the actor receives the *input_message*, i.e., queued, and the assume clause evaluates to true, it can make the following transition. The actor creates the specified set of actors, sends the output messages as described by the regular expression over the *output_messages*, and guarantees that the assert clause holds in the poststate. Note that assumption A4 guarantees that a poststate exists. Using regular expressions to describe the sent output messages is just a convenience. In this paper we will only use finite sequences of output messages and assume, in particular, that the Kleene-star cannot be used.

The expression in the assume clause can refer to parameters of the input message and the instance variables of the actor. The default for missing assume clauses is **`true`**. The expression in the assert clause can refer to parameters of the input message and to the instance variables of the actor in the pre- and poststate. The value of an instance variable `v` in the prestate is denoted by **`old`**`(v)`. The default for missing assert clauses asserts that all instance variables `v` are unchanged, i.e., `v == old(v)`.

Before we discuss the construct further, let us consider specifications for our server and worker as examples.
The behavior of an actor `AServer` given in Fig. 1 is represented in the specification given in Fig. 3. On selection of a `serve` message, a server creates a new actor of

[3] mainly to stay out of a component discussion

```
actor spec AServer {
   this.serve( c, t ) ==>  w <- new AWorker;
                           w.do( t, taskSize(t) );
                           w.propagateResult( null, c )
   assumes c != null
}
```

**Fig. 3.** AServer actor specification

type `AWorker`, names it `w` and sends two messages to the newly created worker. This selection assumes that both message parameters are non-null. The workers are more interesting. It uses the instance variables of the actor class in the specification (according to JML terminology, they are `spec-public`; cf. [23]).

```
actor spec AWorker {
   Value  myResult = null;
   Worker nextWorker = null;

   this.do( t, n ) ==> empty
   assumes  n == 1
   asserts  myResult == compute(t) && nextWorker == null

   this.do( t, n ) ==> w <- new AWorker;
                       w.do( restTask(t,n), n-1 )
   assumes  n > 1
   asserts  myResult == compute( firstTask(t, n) ) && nextWorker == w


   this.propagateResult( v, c ) ==> c.receive( merge(myResult,v) )
   assumes  myResult != null && nextWorker == null

   this.propagateResult( v, c ) ==>
                nextWorker.propagateResult( merge(myResult,v), c )
   assumes  myResult != null && nextWorker != null
}
```

**Fig. 4.** `AWorker` actor specification

The figure above states the behavior of the `AWorker` actor in Fig. 2. On selection of a `do` message with actual parameter `n > 1`, a worker creates another worker and asks it to `do` the remaining $(n-1)$ parts of the computation task. The worker itself computes the first part and updates its state accordingly. On selection of a `do` message with actual parameter `n == 1`, a worker computes the given task; it does not have any effect on the environment.

If the return from a procedure/method execution is considered as sending an outgoing message back to the original caller, our specification construct can be understood as a generalization of pre-/post-specifications. The central differences are that the number of outgoing messages can also be zero or several and that actor creation has to be explicitly specified as it affects the environment.

*Semantical aspects.* We assume that the semantics of actor programs is defined as sets of traces where a trace is a possibly infinite sequence alternating between configurations and events. A *configuration* captures the set of actors and their states. There are three types of *events*: output events for message sending, input events for message selection, and creation events for actor creation. The distinction between output and input events is necessary, because a message that never satisfies its guard is never selected. Events have an identity[4] in the sense that no events in a trace are equal.

The semantics of specifications is based on the semantics of actor systems. A reaction rule is interpreted as a mechanism expressing a causality relation between an input event $e_i$ and a set $E = \{e_{c_1}, \ldots e_{c_m}, e_{o_1}, \ldots, e_{o_n}\}$ of creation and output events where causality means that

> *whenever we see an event $e_i$, eventually all events of $E$ will appear in the subsequent trace*

In this weak form of causality, it might happen that for two input events $e_i$ and $e_i'$ the caused event sets $E$ and $E'$ have common events, i.e., that an output or creation event is "caused" by two different input events. In practice, this would lead to the undesirable situation that sending five times the same request to a server might result in only one response. Furthermore, this weak form is in contrast to our goal to generalize pre-/post-specification, as the actor specifications above guarantee that the number of calls always corresponds to the number of returns[5]. Most importantly, actor implementations never generate traces where an event is caused by two different input events. That is why we argue for a *strong causality* semantics in which

> *every event e either has a unique event that caused e or is an* external *event*

where output events are caused by the corresponding input event and external events are coming from outside the considered system (e.g., for starting up systems or for interacting with an unknown environment).

*Discussion.* From a view point of actor implementations, strong causality seems to be a natural basis for specifications. That is why it was a bit surprising to us to notice that strong causality cannot be expressed in classical temporal logic. To be more precise, when we project down the traces to the communicating actors (e.g., between a server and a client), the trace prefixes are non-regular. Specifying

[4] for example, the number of their position in the trace

[5] Recall that we assume termination

the strong causality needs counting facilities (see [21] for a temporal logic with counting).

The above specifications for `AServer` and `AWorker` slightly abstract from the behavior of the actor implementations and simplify the reasoning about these actors. However, from a client point of view, they are not of much interest. A client is not interested in how the server does its work, but wants to know that the server eventually returns the correct result for every request. That is, the client is interested in the behavior of the group of actors consisting of the server and the dynamically created workers. This is the topic of the next subsection.

### 3.2 Specifying Actor Groups

As illustrated above, there are typical scenarios in which a user of an actor is not interested in the work directly performed by the actor itself, but wants to have guarantees about the behavior that the actor achieves together with its helper actors. This is one reason to develop specifications for actor groups. Another reason is that group specification are important to formulate induction invariants for proofs about actor groups of varying sizes (in our example, the number of workers varies depending on the input). In summary, our position is that

> *specifications for actor groups are needed both as contracts for more complex actor groups and as important construct for reasoning.*

We illustrate both aspects. We like to stress that the aspects are closely related as specifications of single actors and actor groups are needed to prove specifications of larger actor groups in a hierarchical manner.

```
group spec AServer {
   this.serve(c,t) ==> c.receive( compute(t) )
   assumes  c != null
}
```

**Fig. 5.** AServer group specification

A central goal in the design of specifications of actor groups is that they are similar to specifications of single actors. For example, the specification of the group owned by an `AServer`-actor in Fig. 5 almost looks like an actor specification. It expresses exactly what a client wants to know. To achieve our design goal, we make two simplifying assumptions in this paper:

G1 Every actor *a owns* a group. The members of *a*'s group are just the actors directly or indirectly created by *a*.
G2 Every group only exposes the reference of the owner to the group environment.

```
group spec AWorker {
   boolean  working = false;
   CompTask myTask = emptyTask;

   this.do( t, n ) ==> empty
   assumes  n >= 1
   asserts  myTask == t && working == true

   this.propagateResult( v, c ) ==> c.receive( merge(compute(mytask), v) )
   assumes  working
}
```

**Fig. 6.** `AWorker` group specification

The used notions of grouping are similar to simple ownership disciplines in OO programming (e.g., [33]). Because of assumption G1, we do not have to introduce constructs in AJ to define groups. Groups are defined implicitly. To be more realistic, one could distinguish at creation sites whether the new actor should be created in the local group or in the environment (ABS uses such a technique [19]). However, for illustrating our position, our primitive grouping mechanism is sufficient. Assumption G2 is a restriction on the actor programs we are allowed to write, and it cannot be checked automatically. In general, an actor group can expose references of several actors of the group to the environment. For example, the `receive`-message could pass a reference of a worker to the client. To eliminate G2, one would need to generalize the specification construct for actor groups.

Whereas the specification of an `AServer`-group illustrates a typical client-server contract, the specification of an `AWorker`-group in Fig. 6 describes the behavior of an `AWorker`-actor $w$ together with the workers created by $w$. To express this behavior, the specification uses the variables `working` and `myTask`. The boolean flag `working` is a model variable that abstracts the instance variable `myResult`. It is true iff `myResult != null`. We use the model variable here to hide the implementation details of the worker. Variable `myTask` is a ghost variable storing the task worked on by the actor group. It is needed to formulate the reaction rule for message `propagateResult`. Model and ghost variables are well-investigated techniques in OO specification (see, e.g., [8]).

*Semantical aspects.* Group specifications look syntactically similar to single actor specifications. The semantics is also similar except that group internal events are hidden. That is why the specification of the `do` message does not list any caused events. More precisely, a reaction rule has to specify all output events with a receiver outside the group that are directly or indirectly caused by an input event. The reaction rule for message `propagateResult` is an example of an indirectly caused event, because the propagation might iterate through a number of other workers before the result is received by the client.

## 4 Modular Verification

In this section, we discuss how the developed specification techniques can be used for verification. The aim is to *illustrate* how a higher-level verification technique that does not work on the semantics or trace level can be conceived. To achieve this, we will leave many details undiscussed and do not claim that we present a formal proof. We describe how group specifications can be verified from other group and actor specifications. Thus, we have a modular, hierarchical verification technique in which we can use the verified specifications of actors and subgroups to prove enclosing groups. We look into the two essential cases:

- Verifying a group by composing the specifications of the subgroups and actors.
- Verifying a group by induction over its parameter values.

In this paper, we do not discuss the task of verifying that actor implementations satisfy their specifications. This is the classical problem that can be handled by adapted Hoare-style or dynamic logics.

To aid the presentation, the verification sketches will be accompanied with proof outlines. A proof outline has the form of a sequence whose elements are separated by the symbol ==>*. Each element consists of a list with state assertions, actor creations and output messages. State assertions are expressed as a predicate enclosed in curly brackets, representing the value of each variable of the actors and the message parameters. Messages have the same format as in the specifications. The symbol ==>* should be read as a combination of the transitive, reflexive closure of ==> and logical implication.

*Compositional verification of group specification.* In our server example, we have two groups that we need to verify: the `AServer`-group and the `AWorker`-group. Here, we verify the specification of the `AServer`-group (Fig. 5) to show compositional verification using the specifications of the `AWorker`-group (Fig. 6) and the `AServer`-actor (Fig. 3).

The property that we have to prove is that whenever an instance of the `AServer`-group receives a `serve(c,t)` request to compute the task `t`, the server sends back to the non-null client `c` the corresponding computation result. The proof outline for the `AServer` group is given in Fig. 7. The first step of the outline is directly obtained from the specification of message `serve` in Fig. 3. Using the group specification of the created worker and the property of `taskSize` (cf. Sect. 2) gives us the state expression. The next two steps are obtained by unfolding the `do`- and `propagateResult`-specifications of Fig. 6. Finally, we use a property of `merge`.

*Inductive verification of group specification.* Inductive verification of group specifications is in general more complex. In simple cases, the group specification is sufficient as an induction invariant. However, in more complex cases, structural invariants are needed in addition. For example, the proof of the worker group specification of Fig. 6 needs an invariant that links the construction of the actor structure with the result propagation.

```
    { c!= null }
    this.serve(c,t)
==>*
    w <- new AWorker
    w.do( t, taskSize(t) );
    w.propagateResult( null, c )
==>*
    { w.working==false && w.myTask==emptyTask && n==taskSize(t) && n>=1 }
    w.do( t, n );
    w.propagateResult( null, c )
==>*
    { w.myTask == t  &&  w.working == true }
    w.propagateResult( null, c )
==>*
    { w.myTask == t  &&  w.working == true }
    c.receive( merge(compute(w.myTask),null) )
==>*
    c.receive( compute(t) )
```

**Fig. 7.** Proof outline for the server verification

To discuss inductive proofs, we consider the actor group in Fig. 8 computing the factorial of a natural number. The basic proof outline of the induction step is as follows:

```
   { n>0 }
   this.fac(n,c)
==>*
   { n>0 && argument==n  &&  myCaller==c }
   f <- new AFactorial
   f.fac(n-1,this)
==>*
   { n>0 && argument==n  &&  myCaller==c }
   this.return(factorial(n-1))
==>*
   { n>0 && argument==n  &&  myCaller==c }
   c.return(factorial(n-1)*n)
==>*
   c.return(factorial(n))
```

In the first step, we use the property of the actor specification. Then, we use the group specification and finally the actor specification again. Of course, this form of reasoning has to be complemented with mechanisms controlling undesired outside interaction either by always working with new actor instances or by blocking incoming messages as long as critical computations are going on in a group.

```
interface Caller { void return( Int ); }
interface Factorial extends Caller { void fac( Int n, Caller c ); }

actor class AFactorial implements Factorial { ... }

actor spec AFactorial {
   Int argument;
   Caller myCaller = null;

   this.fac( n, c ) ==> c.return(1)
   assumes  n == 0

   this.fac( n, c ) ==> f <- new AFactorial
                        f.fac( n-1, this )
   assumes  n > 0
   asserts  argument == n && myCaller == c

   this.return(res) ==> myCaller.return( res*argument )
   assumes  myCaller != null
}

group spec AFactorial {
   this.fac( n, c ) ==> c.return(factorial(n))
   assumes n >= 0
}
```

**Fig. 8.** An actor group computing the factorial function

## 5 Discussion of Related and Future Work

Actors provide a popular model for distributed and concurrent systems. Although researchers have worked for over twenty years on actor languages and verification, there are still interesting and open questions. In particular, we claim that the specification and verification techniques for non-local actor properties in dynamically evolving actor systems are not sufficiently developed. Technically, we proposed

- to use specifications that are based on strong causality and not on classical temporal logic and
- to support reasoning over hierarchical actor groups or components.

Next, we broaden the discussion of related work and mention plans for future work.

### 5.1 Related Work

*Actor languages.* Many actor languages are available (to name only some: ABS [19], AmbientTalk [11], CoBoxes [36], Creol [20], E [29], Erlang [3]) and the family is still growing. Furthermore, there existed a variety of actor libraries for

programming languages. AJ can be seen as a simplified version of Creol [20] and ABS [19], without futures and return values and without the cooperative scheduling mechanism. However, many of the features can be expressed using additional state and self-messages, i.e., messages that are sent from an actor to itself. As with Creol or CoBoxes, it can be proved that the type system guarantees that type correct AJ actors know the messages that are sent to them.

*Specification.* Whereas in the OO community there is a large corpus of work about specification languages and techniques (see [17]), this is not true in the actor community. In [16], a technique for specifying and dynamically checking the interaction behavior of single actors is described. The technique uses attribute grammars for specifying the possible protocols an actor can engage in.

History-based specification techniques as described in, e.g., [9,14] enable one to specify safety properties based on trace prefixes in a modular way. As strong causality property is a liveness property, these techniques cannot specify it without any extensions. An approximation of the intended `AServer`-group behavior (Fig. 5) is as follows.

```
ok([]) = true
ok(h; m) = true, WHERE m != c.resp(_)
ok(h; c.resp(res(t))) = okcnt(h; c.resp(res(t)), c.resp(res(t)), 0) && ok(h)

okcnt([], _, n) = (n >= 0)
okcnt(h; m, c.resp(res(t)), n) = okcnt(h, c.resp(res(t)), n)
  WHERE m != c.resp(res(t)) && m != this.req(t, c)
okcnt(h; this.req(t, c), c.resp(res(t)), n) = okcnt(h, c.resp(res(t)), n+1)
okcnt(h; c.resp(res(t)), c.resp(res(t)), n) = okcnt(h, c.resp(res(t)), n-1)
```

The execution invariant predicate `ok` keeps track of the number of requests and responses with the same parameters, assuming one cannot distinguish them (i.e., the events lack identities).

To represent liveness properties, an extension in form of *ready set* is presented in [9]. This set represents the set of events an actor (group) is ready to accept next. More investigation is needed to compare strong causality with ready set.

The notion of strong causality also appears in form of session types [37] and has been applied to a similar client/server context as described above [15,13]. For example, using the syntax from MOOSE ([12]) the interaction of the `AServer`-group using a session type as given below.

```
session ClientRequest = begin.!CompTask.?Value.end
```

A client request is represented as the emittance of a task and receiving a value. Communication between a client and a server is done over a channel, and channels can be passed around creating the possibility of higher-order session. Because an interaction between actors is represented within a session, the client reference does not need to be sent. However, the types are an overapproximation of the intended behavior. In the example above, there is no information that the session indeed occurs exclusively between a client and a server (because channels can be passed

around), and whether the resulting value should have any correspondence to the task until the `ClientRequest` type is coupled to the implementation. Our aim is to provide a precise specification of how a system and its components should interact. Furthermore, strong causality can be used to specify the link between events that do not occur within one session.

Somewhat further than session types lies $\pi$-calculus [30], which provides hierarchical means to specify concurrent systems, a necessity stated in our position. An adaptation to the actor setting will require an introduction of identities.

Other works dealing with specification of system level functional properties covering data and control aspects include the following. Broy [6] proposes a stream-based approach; in the architecture description language (ADL) field, we have, for example, SOFA [35] and Rapide [26]; and the specification of open distributed system in Erlang by Dam, Fredlund and Gurov [10]. As stated in [10], the literature mostly covers aspects of static systems, with fixed amount of component instances interacting within the execution of a system, lacking the instance creation aspects.

*Verification.* Dovland, Johnsen and Owe [14] and Ahrendt and Dylla [2] developed a Hoare logic and a dynamic logic, respectively, for Creol [20]. Formulation of a property of a simple program already requires a large formula, making it hard to be applied directly to specify system-wide property. Closely related is the work of Dam, Fredlund, and Gurov on verification of dynamically created Erlang processes [10]. They used a logic based on the first-order $\mu$-calculus, but did not consider additional specification techniques.

### 5.2 Future Work

This paper introduced AJ as a core language acting as the base for specification and verification. The core language needs to include just the essential features that affect the specification and verification techniques. It would be interesting to analyze how other features like cooperative multi-tasking or futures as in ABS [19] could be expressed in AJ. On the other hand, we plan to extend our approach to directly support the modeling language ABS that has all these features in the language.

As hierarchical system is a necessary part of verifying dynamic actor systems, the specification language needs to deal with actor groups/components. The cases we have dealt with so far use a single actor as the entry point to a component. In general, a component may contain more than one actor which readily interact with its user. It is harder to compose the causality relations of actors of such a component for verification purposes.

Handling the strong causality specification construct requires an expressive logic, such as a higher-order logic. To strengthen our verification approach, we have embedded some parts of our formalization in Isabelle/HOL [34]. We are continuing this work to obtain full coverage of our specification and verification technique.

*Acknowledgement* The authors thank Mads Dam for discussion on strong causality and verification of non-local actor properties. In particular, he suggested to use a simplified version of the problem described in [4] as a running example. The authors also thank the anonymous referees for their valuable input.

## References


1. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
2. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Science of Computer Programming (2011)
3. Armstrong, J.: Erlang. Commun. ACM 53, 68–75 (2010)
4. Arts, T., Dam, M.: Verifying a distributed database lookup manager written in Erlang. In: World Congress on Formal Methods. pp. 682–700 (1999)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS. pp. 49–69. Springer (2004)
6. Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. 53(10), 1758–1782 (2010)
7. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: POPL. pp. 123–134 (2004)
8. Cheon, Y., Leavens, G.T., Sitaraman, M., Edwards, S.H.: Model variables: cleanly supporting abstraction in design by contract. Softw., Pract. Exper. 35(6), 583–599 (2005)
9. Dahl, O.J., Owe, O.: Formal development with ABEL. In: VDM Europe (2). pp. 320–362 (1991)
10. Dam, M., Fredlund, L.Å., Gurov, D.: Toward parametric verification of open distributed systems. In: COMPOS. pp. 150–185 (1997)
11. Dedecker, J., Cutsem, T.V., Mostinckx, S., D'Hondt, T., Meuter, W.D.: Ambient-oriented programming in AmbientTalk. In: ECOOP. pp. 230–254 (2006)
12. Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Objects and session types. Inf. Comput. 207(5), 595–641 (2009)
13. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A distributed object-oriented language with session types. In: TGC. pp. 299–318 (2005)
14. Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of dynamic systems: Component reasoning for concurrent objects. ENTCS 203(3), 19–34 (2008)
15. Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: ESOP. pp. 74–90 (1999)
16. de Gouw, S., de Boer, F., Vinju, J.: Prototyping a tool environment for run-time assertion checking in JML with communication histories. In: FTfJP (2010)
17. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. Tech. Rep. CS-TR-09-01a (2010)
18. Inmos Corp: Occam Programming Manual. Prentice Hall Trade (1984)
19. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: FMCO 2010. LNCS, Springer (2011), to appear
20. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365(1-2), 23–66 (2006)
21. Laroussinie, F., Meyer, A., Petonnet, E.: Counting LTL. In: TIME. pp. 51–58 (2010)
22. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
23. Leavens, G.T., Cheon, Y.: Design by contract with JML (2006)
24. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: ECOOP. pp. 491–516 (2004)
25. Luckham, D.C., von Henke, F.W., Krieg-Brückner, B., Owe, O.: ANNA - A Language for Annotating Ada Programs, Reference Manual, LNCS, vol. 260. Springer (1987)

26. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Software Eng. 21(4), 336–355 (1995)
27. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
28. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (1997)
29. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency among strangers. In: TGC. pp. 195–229 (2005)
30. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
31. Misra, J.: A discipline of multiprogramming. ACM Comput. Surv. 28 (1996)
32. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. Ph.D. thesis, FernUniversität Hagen (2001)
33. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for alias and dependency control. Tech. Rep. 279, Fernuniversität Hagen (2001)
34. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
35. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. 28(11), 1056–1076 (2002)
36. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: ECOOP 2010. pp. 275–299. LNCS, Springer (2010)
37. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. pp. 398–413 (1994)