A Relational Trace Logic for
Simple Hierarchical Actor-Based Component Systems

Ilham W. Kurnia
University of Kaiserslautern
ilham@cs.uni-kl.de

Arnd Poetzsch-Heffter
University of Kaiserslautern
poetzsch@cs.uni-kl.de

Abstract
We present a logic for proving functional properties of concurrent component-based systems. A component is either a single actor or a group of dynamically created actors. The component hierarchy is based on the actor creation tree. The actors work concurrently and communicate asynchronously. Each actor is an instance of an actor class. An actor class determines the behavior of its instances. We assume that specifications of the behavior of the actor classes are available. The logic allows deriving properties of larger components from specifications of smaller components hierarchically.

The behavior of components is expressed in terms of traces where a trace is a sequence of events. A component specification relates traces of input events to traces of output events. Generalizing Hoare-like logics from states to traces and from statements to components, we write \( \{ p \} \ C \{ q \} \) to mean that if an input trace satisfies \( p \), component \( C \) produces output traces satisfying \( q \); that is, \( p \) and \( q \) are assertions over traces. Such specifications are partial in that they only specify the reaction of \( C \) to input traces satisfying \( p \).

This paper develops the trace semantics and specification technique for actor-based component systems, presents important proof rules, proves soundness of the rules, and illustrates the interplay between the trace semantics, the specification technique and the proof rules by an example derived from an industrial Erlang case study.

Categories and Subject Descriptors F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs — generalized Hoare logics

General Terms Design, Theory, Verification

1. Introduction
In this paper, we develop a specification and reasoning technique for component-based open distributed systems. Distributed systems are realized by dynamically growing collections of actors [1] that communicate with other actors via asynchronous messages. In particular, we want to enable reasoning about functional properties of open systems, that is, about systems working in an environment for which we do not have an implementation or a precise specification.

The basis of our approach is a two-tier verification as shown in Fig. 1. The two-tier verification avoids the complex task of directly reasoning about system properties on the implementation level. Following the approach of Creol [19] and ABS [20], we assume that actors are implemented using the object-oriented concept of classes. A class determines how all instances of the class behave.

To reflect the concept of classes at the specification level, we use specifications of actor classes, called actor class specifications, which allow specifying properties about the behavior of all instances of a class implementation. In the first tier of our verification approach, the actor implementation is verified against a specification of the actor. This task is not considered in this paper, but addressed in Din et al. [14] and Ahrendt and Dylla [3].

The aim of this paper lies in the second tier, i.e., to use actor class specifications to verify properties of small components and to use these component specifications to verify larger components and open systems. A component is formed hierarchically by following the actor creation
tree. Starting from the initial actor, a component consists of all actors transitively created by the initial actor. The exact formation of a component is influenced by how the unknown environment interacts with the component. Hence, an open system can be considered as a component.

To achieve the foregoing goal, we characterize open actor systems in terms of a trace semantics, introduce a specification technique based on the traces that does not refer to any implementation, and derive a logic that utilizes this specification technique. An execution of an actor system can be represented by a trace of observable events [18]. The advantage of dealing only with traces is the abstraction from the actual state representation of the system. The semantics of actors and components is expressed by trace sets. When the actor or the component represented by the trace set is known, each trace in the set can be split into input and output traces representing the events it receives and produces, respectively.

Based on this semantics, we develop a specification technique relating input traces to output traces. Formally, a specification consists of a finite set of Hoare-like triples [17]. A triple \( \{p\} D \{q\} \) denotes that if an input trace satisfies \( p \), component \( D \) produces output traces satisfying \( q \). The component \( D \) either denotes the behavior of a single actor of class \( C \), or denotes the (external) behavior of groups of actors with an initial actor of class \( C \). Notice that a triple specifies the behavior of a component only for inputs satisfying \( p \). These input conditions express assumptions about the usage of the component and help to focus the reasoning.

We show the usage of the specification technique by means of a proof system for a simple form of composition that we call *daisy chain* composition. It allows a component to dynamically create a new component, but forbids the new component to call back to its creator. We show through an example taken from an Erlang case study [6] how to verify properties of larger components by only using the specifications of smaller components.

**Paper Structure.** The following section describes the language background and our running example. Section 3 gives the definitions of actor classes and components and how their trace sets are characterized and composed. Sections 4 and 5 present the specification technique and the proof system, along with their application to the running example. Section 6 discusses the related work. Section 7 concludes and describes future work.

2. Language Background and Example

To have a sufficiently clear background for the following discussion on specification and verification, we informally introduce a core actor language ActJ together with an example for illustrating our approach. Following the design of the modeling languages Creol [19] and ABS [20], ActJ uses classes to describe actors (we use the keyword *actor class*). Actors can be dynamically created, implement interfaces, have an actor-local state expressed in terms of instance variables, and are addressed via a typed reference.

As a running example, we use a variant of the client-server setting treated in an industrial Erlang case study by Arts and Dam [6]. The server receives requests from the client, where each request contains a task. The server system is to respond to the requests with the appropriate computation results. To serve each request, the server creates a worker and pass on the task to be computed. As a task can be divided into multiple chunks, more concurrency can be introduced in the following way. Before each worker processes the first chunk of the task, it creates another worker to which the rest of the task is passed on. When the computation of the first task chunk is finished, the worker merges the previous result with this computation result and passes on the merged result to the next worker. Eventually all chunks of the task are processed, and the last worker sends back the final result to the client. The structure of the request processing forms a daisy chain as illustrated in Fig. 2. In the actor setting (where all clients, servers and workers are actors), the client name needs to be passed around as well so that the last worker can return the task computation result to the client. This example illustrates unbounded actor creation and non-trivial concurrency.

Figures 3 and 4 illustrate how the server scenario can be implemented in ActJ, which uses a Java-like syntax. The central actor class is AServer implementing the interface **Server**: receiving message **serve** with a task \( t \), it deals with the computation task \( t \) and makes sure that a response is sent to the client (cf. the interface **Client**). To enable concurrent execution of tasks, the server delegates the

---

**Figure 2.** Daisy chain structure of the server system

**Figure 3.** Interfaces and the actor class AServer

---

1 Icons in the figure are taken from http://www.iconarchive.com
actor class AWorker implements Worker {
  Value myResult = null;
  Worker nxtWorker = null;
  do(CompTask t) {
    if (taskSize(t) > 1) {
      nxtWorker = new AWorker();
      nxtWorker.do(restTask(t));
    } else { nxtWorker = null; }
    myResult = compute(firstTask(t));
  }
  propagateResult(Value v, Client c)
    guard myResult != null {
      c.response(merge(myResult, v));
    } else {
      nxtWorker.propagateResult(merge(myResult, v), c);
    }
} }

Figure 4. Actor class AWorker

A task is assigned to a dynamically created worker (interface Worker). If the task has more than one chunk (taskSize(t) > 1), the worker delegates the rest of the task to a newly created worker and works on the first chunk. By a series of propagateResult messages, initiated by the server, the results of the different chunks are collected, merged, and the final result is sent back to the client.

Reacting to messages. In ActJ, a message consists of a method name and typed parameters. A message is produced when a statement of the form r.m(p) is executed. This statement, known also as a method call, sends the message m to the receiver actor r where m is the method name with a list of parameters p. The parameters can be data values or actor names. Such a send-operation is non-blocking; execution directly continues with the next statement. Thus, in general, a message send leads to concurrent execution. 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. 3) reacts to a message serve(c, t) as follows: It creates a worker actor, sends first a do and then a propagateResult message to the worker. We assume that the execution of message bodies must terminate.

Receiving and selecting messages. It remains to explain what happens on a message receive. We assume that actors have an unbounded input queue and are input enabled (cf. [21, p. 257]); i.e., actors can always accept new input. Messages are selected from the queue primarily in a FIFO manner, but 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 (weakly) fair for messages with true guards, meaning that a message whose guard is infinitely often evaluated to true will eventually be picked for processing.

In Fig. 3, the actor class AWorker uses a guard to select a propagateResult message only if a result is available.

Further constructs. As well as the actor-related aspects, ActJ supports recursive data types and function definitions for handling data (as in functional programming languages). In the running example, we assume appropriate definitions for the data types CompTask and Value and the total functions:

\begin{align*}
\text{compute} &: \text{CompTask} \rightarrow \text{Value} \\
\text{taskSize} &: \text{CompTask} \rightarrow \text{Int} \\
\text{firstTask} &: \text{CompTask} \rightarrow \text{CompTask} \\
\text{restTask} &: \text{CompTask} \rightarrow \text{CompTask} \\
\text{merge} &: \text{Value} \times \text{Value} \rightarrow \text{Value}
\end{align*}

where compute(t) computes the result of t; taskSize(t) yields a number of chunks in which t could be partitioned; firstTask(t) returns the first chunk of t; restTask(t) returns the rest of t; and merge merges results. We assume the following properties:

\begin{align*}
\text{taskSize}(t) &\geq 1 \\
\text{taskSize}(t) &> 1 \rightarrow \text{compute}(t) = \text{merge}(\text{compute}(\text{firstTask}(t)), \\
&\quad \text{compute}(\text{restTask}(t))) \\
\text{taskSize}(t) &= 1 \rightarrow \text{compute}(t) = \text{compute}(\text{firstTask}(t)) \\
\text{merge}(\text{null}, v) &= v
\end{align*}

A task consists of at least one chunk; computing a non-primitive task is the same as merging the result of computing the first task with the computation of the rest of the task; computing a single task chunk is the same as computing the first task of the chunk; and merging with null with some value v returns v.

Actor systems are started by creating actors and start their activities or connect them to activities in the environment, e.g., to user interfaces. In this perspective, we deal with open systems because of the interaction with their environments.

3. Semantics of Actors and Components

An execution of an actor system can be represented by a trace of observable events [8, 13, 18]. The functional behavior of an actor system is represented by a trace set. Taking the work of Agha et al. [2] and Vasconcelos and Tokoro [30] as a guide, we consider the trace sets of actors and components in an open system setting. More precisely, we characterize the trace sets with respect to the most general environment, i.e., the environment that provides all admissible behaviors. For the sake of simplicity, we assume that each actor has a fair chance to do its computation and there is a type system handling the correctness of types of events and their content.

This section is divided into two subsections. The first subsection deals with the foundation of traces, namely what the trace events are, the basic operations one can perform on traces, what valid traces are, and what the trace set of the individual actors are. The second subsection defines what a component is, based on a composition of the traces of the actors contained in the component. To focus on the interaction with its environment, we define
Table 1. Helper predicates and functions

<table>
<thead>
<tr>
<th>Predicate/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pref(s)</td>
<td>The set of all prefixes of sequence s.</td>
</tr>
<tr>
<td>class(a)</td>
<td>Returns the class of actor a.</td>
</tr>
<tr>
<td>isMtd(m)</td>
<td>Checks if message m is a method call.</td>
</tr>
<tr>
<td>callee(e)</td>
<td>Returns the callee of event e.</td>
</tr>
<tr>
<td>caller(e)</td>
<td>Returns the caller of event e.</td>
</tr>
<tr>
<td>msg(e)</td>
<td>Returns the message of event e.</td>
</tr>
<tr>
<td>acq(t)</td>
<td>Returns the accumulated actor names exposed in each method call event in t.</td>
</tr>
<tr>
<td>cr(t)</td>
<td>Returns the set of actors created in t.</td>
</tr>
<tr>
<td>t↓_L</td>
<td>Projects t to non-external events of a set of actors L. The operator can also take callee or caller as an extra parameter.</td>
</tr>
<tr>
<td>exposedTo(t,A)</td>
<td>The set of traces $\text{Traces}{{a}}$ where $\text{class}(a) = C$.</td>
</tr>
<tr>
<td>[[C]]</td>
<td>The set of traces $\text{Traces}{[L]}$ where the initial actor of L is of class C.</td>
</tr>
<tr>
<td>ext(T)</td>
<td>Extracts the largest visible subset of local actors from a boxed trace set T.</td>
</tr>
<tr>
<td>split(t,L)</td>
<td>Splits t into input and output traces $(t_i, t_o)$ based on local actors L.</td>
</tr>
</tbody>
</table>

a boxing operator on a component to hide all internal interaction. However, this boxing operator is, for the proof system, too strong. In particular, we want to know how the subcomponents interact with each other. For this purpose, we provide a glass box view [8] p. 5, given the structure of the component. An informal summary of numerous helper predicates and functions is given in Table 1 as a quick reference.

3.1 Trace Foundation

Traces are represented using the finite sequence data structure $\text{Seq}(T)$, with T denoting the type of the sequence elements. An empty sequence is denoted by $\emptyset$ and \cdot represents sequence concatenation. The function $\text{Pref}(s)$ yields the set of all prefixes of a sequence s.

The foundation of the trace semantics is the set of actor names a, b ∈ A and the set of messages m ∈ M that can be communicated between actors. We assign each actor with a specific behavior represented by a class $\mathbf{C} \in \mathbf{CL}$. The function $\text{class}(a)$ gives the class of an actor a. A message m can either be an actor creation new C or a method call $a.mtd(\overline{p})$. mtd denotes some method name and $\overline{p}$ is a list of parameters. A parameter may be a data value d ∈ D or an actor name. The predicate $\text{isMtd}(m)$ checks whether the message m is a method call.

From this foundation, we build the set of events E. An event $e \in E$ represents the occurrence of a message $m = \text{msg}(e)$ being sent by the caller actor $a = \text{callee}(e)$ to the callee actor $b = \text{callee}(e)$. If m is a creation message, b will be the name of the newly created actor while a is its creator. Textually an event e is represented as $a \rightarrow b := \text{new} C$ or $a \rightarrow b.mtd(\overline{p})$ when the message is an actor creation or a method call, respectively. The inclusion of the caller information allows us to distinguish between input and output events with respect to an actor or a group of actors. Eliminating the caller information from an event produces an event content. Taking $L \subseteq A$ to be the set of (local) actors we are considering and, by the open system setting, $F = A - L$ as the set of all (foreign) actors $L$ is interacting with, an event e that appears in the trace can be categorized as follows.

- an input event if $a \in F$ and $b \in L$;
- an output event if $a \in L$ and $b \in F$;
- an internal event if $a, b \in L$;
- an external event if $a, b \in F$.

Only non-external events are of interest here as the environment’s internal behavior is unknown. Method calls expose names to callee actors. As the exposure of actor names is important to decide when an actor can send a message to another actor or pass on names to other actors in the semantics, we define a function $\text{acq}(a \rightarrow b.mtd(\overline{p}))$, short for acquaintance, to extract the finite set of actor names occurring in the parameter list of a method call event. The caller name is transparent to the callee, so it is not part of the acquaintance.

Example 3.1. Consider a server actor s, a client actor c, a worker actor w, some task t and $L = \{s, w\}$. The event $c \rightarrow s.\text{serve}(c, t)$ is an input event for s and an output event for c. The event $s \rightarrow w := \text{new Server}$ is an internal event of L.

A trace $t \in \text{Seq}(E \cup \{\emptyset\})$ is a finite sequence of events that represents a single execution of the entity L it represents. The $\emptyset$ symbol indicates that t is a maximal trace, that is when the environment of L represented by the trace stops sending more input to L, then L stops its activity.

The basic operator on a trace is the projection operator $t$ can be projected to a given a set of actor names $A \subseteq A$, written $t|_A$, where all events, except $\emptyset$, whose neither caller nor callee is not in A are dropped. The function is refined by a caller (or callee) parameter $t|_{A, \text{callee}} (t|_{A, \text{caller}})$ where the callers (callees) of the retained messages are in A. With respect to some local actor set L, a trace is called an input (output) trace when all its events are input (output) events.

Given a set T of traces, the projection $T|_A$ yields the set of traces $T'$ where each trace t in T is projected to A. The set of acquaintance is lifted to the traces. It is straightforward to show that $\text{acq}$ grows monotonically with respect to the length of the trace. The function $\text{cr}(t)$ produces the set of actors that are created in a trace t.
Similar to \( acq \), \( cr \) also grows monotonically. Because these functions are used in conjunction to know which actor has been exposed to a group of actors \( A \) in a trace \( t \), we abbreviate \( acq(t \downarrow_{callers}) \cup cr(t \downarrow_{callers}) \) as \( exposedTo(t, A) \).

The behavior of an actor system can be represented in terms of a set of valid traces. This notion of valid trace set ideally should be derived from the operational semantics of the actor systems, which lies outside of the scope of this paper. Intuitively, a valid trace is a trace that contains no external events, starts with the creation of some actor and allows the environment to make method calls to local actors when they are exposed. Taking into account the open environment setting, we require a valid trace set of a set of local actors to be a set of valid traces that is prefix-closed and allows foreign actors to make a method call to exposed local actors at any time. The prefix-closedness allows us to observe the behavior of a group of actors(s) at any point in time. In addition to this valid trace restriction, we assume a creation message always produces a valid trace set definition. An actor trace of class \( C \) of \( a \) is a valid trace set (Def. 3.1) such that

\[
\begin{align*}
&\exists \forall e \cdot t \in Traces(a) \cdot callee(e) = a \land msg(e) = new C \\
&\exists \forall t \cdot e \in Traces(a) \cdot caller(e) = a \land m = msg(e) \implies \{callee(e) \mid isMtd(m) \} \cup acq(e) \subseteq acq(t \downarrow_{callers}) \cup \{a\} \\
&\exists \forall e \cdot t \in Traces(a) \cdot isMtd(m) \\
&\exists \forall e \cdot t \in Traces(a) \cdot callee(e) = a \land m = msg(e) \implies \{callee(e) \mid isMtd(m) \} \cup acq(e) \subseteq acq(t \downarrow_{callers}) \cup \{a\} \\
&\exists \forall e \cdot t \in Traces(a) \cdot isMtd(m) \\
&\exists \forall e \cdot t \in Traces(a) \cdot callee(e) = a \land m = msg(e) \implies \{callee(e) \mid isMtd(m) \} \cup acq(e) \subseteq acq(t \downarrow_{callers}) \cup \{a\} \\
&\exists \forall e \cdot t \in Traces(a) \cdot isMtd(m)
\end{align*}
\]

A valid trace which either callee or caller of its events is a valid trace which satisfies the properties above is an actor trace.

The definition above closes the exposure requirement from the environment side left open in the valid trace set definition. An actor trace of class \( C \) that ends with \( \checkmark \) indicates that no other input events are sent to the actor and the actor finishes processing all input events. We denote by \( [[C]] = Traces(a) \) the semantics of actors \( a \) of class \( C \).

### 3.2 Actor-Based Components

The next step is to compose these primitive blocks to make a component. First, we define how we compose a group of actors. The basic operation is the plain composition, which takes a set of arbitrary actors. The interaction between these actors is taken as traces whose projection to each actor matches some trace of that actor. Because we deal with a group of actors, the scope of the environment shrinks. To be more precise, when an actor exposes some name to some other actor, it does not mean that the environment can directly use the exposed name, as the other actor may also be part of the group. For the validity to hold, the exposure clause in Def. 3.1 needs to be explicitly enforced.

**Definition 3.3 (Plain trace set).** Let \( L \subseteq A \) be a set of actors and \( F = A - L \). The plain trace set \( Traces(L) \) is the largest possible set such that

\[
\begin{align*}
&\exists \forall t \in Traces(L), a \in L \cdot t |_{\{a\}} \in Traces(a), \text{ and} \\
&\exists \forall e \cdot t \cdot e' \in Traces(L) \cdot caller(e') \in F \land m = msg(e') \implies \{callee(e') \mid isMtd(m) \} \cup acq(e') \subseteq exposedTo(e \cdot t, F) \\
&\exists \forall e \cdot t \cdot e' \in Traces(L) \cdot caller(e') \in F \land m = msg(e') \implies \{callee(e') \mid isMtd(m) \} \cup acq(e') \subseteq exposedTo(e \cdot t, F) \\
&\exists \forall e \cdot t \cdot e' \in Traces(L) \cdot caller(e') \in F \land m = msg(e') \implies \{callee(e') \mid isMtd(m) \} \cup acq(e') \subseteq exposedTo(e \cdot t, F) \\
&\exists \forall e \cdot t \cdot e' \in Traces(L) \cdot caller(e') \in F \land m = msg(e') \implies \{callee(e') \mid isMtd(m) \} \cup acq(e') \subseteq exposedTo(e \cdot t, F)
\end{align*}
\]

It follows that an actor trace set is also plain. Thus this composition does not violate the single actor behavior.

**Lemma 3.1.** Let \( L = \{a\} \). Then \( Traces(L) = Traces(a) \) is plain.

**Proof.** Follows fromDefs. 3.1 and 3.2 \( \square \)

Using plain composition, we can characterize a component as a set of actors that does not create actors outside
Plain composition is not the ideal semantical representation for components because it reveals all internal events. To abstract away from all these internal events, we box the components. This means that all internal events become hidden. This characterization allows using the component without having to care about the internal details and simply focus on what happens on its boundary. In other words, only the interface of the component is of importance. The hiding is done by projecting away events that do not involve foreign actors.

**Definition 3.5 (Boxed component).** Let \( L \) be a component and \( F = A \setminus L \). The boxed component of \( L \), denoted by \([L]\), is the trace set \( \text{Traces}([L]) \) where

\[
\text{Traces}([L]) = \{ t \downarrow_F \mid t \in \text{Traces}(L) \}.
\]

We refer to a trace in \( \text{Traces}([L]) \) as a boxed component trace. Given that the component \( L \) has an initial actor of class \( C \), then \([[[C]]]\) denotes trace set \( \text{Traces}([L]) \). Boxing a component does not affect its validity as shown by the following lemma.

**Lemma 3.3 (Boxed component trace set validity).** Let \( L \) be a component. The trace set \( \text{Traces}([L]) \) is valid.

**Proof (sketch).**
1. Projection does not affect prefix-closedness property.
2. Projection does not add new events into the trace set.
3. The initial creation property remains after projection because the caller is a foreign actor.
4. Internal events does not provide exposure to the environment, thus projecting them away does not affect local actor exposure to foreign actors.
5. Projection does not remove any input events from the trace.

Boxing a single actor component does not change the trace set because no internal events appear in the trace set.

**Lemma 3.4.** Let \( L = \{a\} \) be a component. Then, \( \text{Traces}(L) = \text{Traces}([[L]]) \).

**Proof (sketch).** Because self call events are ruled out, all events in some trace \( t \in \text{Traces}(L) \) are either input or output. Therefore, it will not disappear after projection.

If we know that a trace set \( T \) is a trace set of a boxed component \( L \), we can derive the visible local actors \( L' \subseteq L \) based on the name transfer that happens within the traces. This derivation, denoted by \( \text{ext}(T) \), short for name extraction, is made by collecting the names that are exposed through actor creation, method call parameters and...
call `e` of a method call event. The name extraction is useful for splitting the trace of a boxed component into input and output traces.

**Definition 3.6 (Name extraction).** Let `T` be a (valid) trace set of some boxed component. `L' = ext(T)` is the subset of actors in the component, where `ext(T) = ∪_{t∈T} ext(t)`, `ext([ ]) = ∅`, and

\[
\begin{align*}
\text{ext}(t \cdot e) &= \begin{cases} 
\text{ext}(t) \cup \{\text{callee}(e)\}, & \text{if } \text{msg}(e) = \text{new } C \\
\text{ext}(t) \cup \{\text{caller}(e)\} \cup \text{acq}(e) - (\text{acq}(t) - \text{ext}(t)), & \text{if } \text{isMtd}(\text{msg}(e)) \land \text{callee}(e) \notin \text{ext}(t)
\end{cases}
\end{align*}
\]

The local actor name extraction of `T` is done by examining each trace `t` in `T` and combining the result of each examination. If `t` ends with a creation event `e`, then the callee is part of the local name. By definition of the boxed component, there is exactly one creation event visible in any trace of `T` which is the creation of the initial actor of the component. If `t` ends with a method call and it is directed to some foreign actor, then the callee of this event and all non-foreign actor names in the method call arguments are included.

Hiding all internal events of a component trace is at times too strict, especially when we want to know the interaction between the component’s subcomponents. If we know how the component is structured, we may allow internal events between these entities to appear in the trace set of the component. This way of composing subcomponents and actors into a component is called glass box composition [8, p. 5].

**Definition 3.7 (Glass box composition).** Let `L = L_1 \cup \ldots \cup L_n \cup \{a_1, \ldots, a_m\}` be a component such that `L_1, \ldots, L_n` are components and `a_1, \ldots, a_m` are actor names where `L_1, \ldots, L_n, \{a_1, \ldots, a_m\}` are pairwise disjoint. Let `F = A - L`, `a ∈ \{a_1, \ldots, a_m\}` and `C = \text{class}(a)`. The glass box composition of `L`, denoted `\langle L_1, \ldots, L_n, |a_1, \ldots, |a_m| \rangle`, is the largest trace set `T = \text{Traces}(\langle L_1, \ldots, L_n, |a_1, \ldots, |a_m| \rangle)` where

- \(\forall t \in T \cdot (\forall L_i \cdot t \downarrow_{L_i} \in \text{Traces}(\{L_i\}) \land \forall a_i \cdot t \downarrow_{|a_i|} \in \text{Traces}(a_i))\)
  (all traces can be projected to all elements of `L`),
- \(\forall e \cdot t \in T \cdot \text{callee}(e) = a \land \text{caller}(e) \in F \land \text{msg}(e) = \text{new } C\)  
  (a is the initial actor),
- \(\forall e \cdot t \cdot e' \in T \cdot \text{msg}(e') = \text{new } C \implies \text{callee}(e') \in L \land \text{callee}(e') \in L\)
  (all other creation messages create local actors), and
- \(\forall e \cdot t \cdot e' \in T \cdot \text{callee}(e') \in F \land \text{isMtd}(\text{msg}(e')) \implies \{\text{callee}(e')\} \cup \text{acq}(e') \subseteq \text{acq}(t \downarrow_{\text{callee}}) \cup \text{cr}(e)\)
  (the environment uses only exposed local actors).

Restrictions in terms of creation are applied to the actors and components that are composed, because the resulting composition should be a component. Each actor or component must be created by some local actor except for the initial actor `a_m`. As with the plain trace set definition, traces where name exposure property is not preserved must be excluded in order to maintain validity. Composing components and actors in a glass box manner produces a valid trace set.

**Lemma 3.5 (Glass box trace set validity).**

Let `L_1, \ldots, L_n, a_1, \ldots, a_m` fulfill Def. 3.7

Then, \(\text{Traces}(\langle L_1, \ldots, L_n, |a_1, \ldots, |a_m| \rangle)\) is valid.

**Proof.** Similar to the proof for Lemmas 3.2 and 3.3.

Deriving the black box semantics of a glass box composition is done by projecting the trace set to the foreign actors.

**Lemma 3.6 (Boxing glass box component).** Let component `L = L_1 \cup \ldots \cup L_n \cup \{a_1, \ldots, a_m\}` fulfill Def. 3.7 and `F = A - L`. Then, \(\text{Traces}(\langle L_1, \ldots, L_n, |a_1, \ldots, |a_m| \rangle) |_F = \text{Traces}([L])\).

**Proof.** Follows directly from Def. 3.5.

### 4. Specification Technique

Hoare advocates that to specify the functional behavior of a program is to specify the connection between the input and output of the program [17]. This approach is extended by Broy [7] to deal with components by letting the input and output be streams of events. Here we adopt their approaches to specify the functional behavior of actor components by letting the specification be a set of triples (similar to Hoare triple) where the pre- and postconditions are described using assertions on traces. As in Hoare logic, triples have the form

\[
\{p\} D \{q\}
\]

where `p` and `q` are input and output trace assertions, respectively, and `D` is either `C` or `[C]`. We call `\{p\} C \{q\}` an actor triple, where as `\{p\} C \{q\}` is called a component triple.

The trace assertions are first-order logic formulas that can use a special constant `$` called the trace constant. A trace assertion is checked against some variable assignment and input/output trace, and every occurrence of `$` is replaced by the trace [2]. The input and output traces are obtained from a valid actor trace by filtering the input and output events. Unlike Hoare logic, where the second part of the triple is some implementation, here we only have the name of the entity we represent. Nevertheless, knowing whether the entity is a boxed component or an actor class allows us to link the specification with the correct kind of trace semantics.

Before going into more details about the syntax and semantics, we motivate the specification technique by

---

2 To be exact, `$` is replaced by the event content sequence of the trace, as each actor/component should be unaware who is calling/creating it.
specifying our server example. To distinguish the logical variables appearing in the specification from the program variables, the logical variables will be underlined.

4.1 Specifying the Running Example

In this subsection, we illustrate how the server and worker actor and components described in Sect. 2 behave when a single request comes from a client. To save space, the following abbreviations are employed. The Server class is abbreviated as $S$, Worker as $W$, serve as $sv$, propagateResult as $pr$, response as $resp$, taskSize as $sz$, firstTask as $fst$, restTask as $rst$, compute as $cmp$ and merge as $mrg$.

The following specification of the server class states that when a server is created and a request comes, the server creates a new worker and passes the worker the task and tells it to start propagating the result.

\[
\exists w \cdot S = (\text{this := new } S \cdot \text{this.sv } (\exists t \cdot \sqrt{v}))
\]

For the server case above, the input trace assertion restricts the behavior to cases in which the environment creates the server and sends a single $sv$ message. The server actor processes the input by creating a new worker, passing the task to the newly created worker and starting result propagation before stopping. When the input trace assertion is not satisfied by the input trace, the behavior of the server is unspecified. For example, we do not know what the server does when it receives more than one $sv$ request. As can be seen in the specification, the trace constant is used by comparing it to the sequence of the event contents (pairs of callee and message). By using this comparison technique, we specify exactly what the server does when it receives the exact input that is stated in the input trace assertion. As standard in Hoare logic, any logical variable that appears only in the output trace assertion needs to be existentially quantified.

When we consider the server component as a whole, we would like to see that a request from the client is replied by a response to the client with the computed result. This requirement is represented using the specification below.

\[
\exists w \cdot S = (w := \text{new } W \cdot w.\text{do } (t) \cdot w.\text{pr } (\text{null }, c) \cdot \sqrt{v})
\]

This ends the worker class specification of our example.

If we box the worker class, we obtain a component whose members are exactly the set of workers needed to process a task. Its specification is exactly as that of the worker class, but instead of splitting the tasks into subtasks, the worker group evaluates the whole task and returns the computation result merged with the previous result to the client.

\[
\exists w \cdot S = (w := \text{new } W \cdot w.\text{do } (\text{rest } t) \cdot w.\text{pr } (\text{mrg } (w.\text{cmp } (\text{fst } t)), c) \cdot \sqrt{v})
\]

4.2 Syntax and Semantics

Triples use trace assertions to formulate input and output conditions. A trace assertion is a first-order logic formula in which the special trace constant $S$ can be used. In the input (output) condition, $S$ represents the event content sequence of the input (output) trace. We assume that there are functions and predicates over traces that can be used in trace assertions. For the purposes of this paper, we only need an equality comparison operation, written $S = ec$, that compares the result with a sequence of event contents $ec$, as seen in the previous subsection. Event contents are used instead of events because from an actor’s point of view, the origin of the events is not known unless it is the actor who is initiating them. Thus, the caller of the event does not play a role when we want to specify the behavior of a component. The main idea of the equality comparison is that given an (input or output) trace, there is a mapping of the variables in $ec$ to data and actor names such that stripping this trace of the caller information yields a match to the mapped $ec$.

**Definition 4.1 (Trace assertions).** Let $S$ be a trace constant representing a trace. Trace assertions $p, q$ are defined inductively by the following first-order logic clauses:

- Boolean expressions are assertions ($S$ may be present).
• If $p, q$ are assertions and $x$ is a variable, then $\neg p, p \land q, \exists x : p$ are also assertions.

The other logical operators, e.g., $\lor, \implies$ and $\forall$, are derived in the usual way. Given a trace assertion $p$, the function $\text{free}(p)$ extracts the set of all free variables appearing in $p$.

To define the semantics of a trace assertion, the variables must be assigned to some values. Let $V$ be the set of all variables. A variable assignment $\sigma : V \rightarrow A \cup D$ is a function that maps (some) variables to values.

The semantics of a trace assertion $p$ with respect to a variable assignment $\sigma$ and a trace $t$ is a mapping

$$\llbracket p \rrbracket : (V \rightarrow A \cup D) \times \text{Seq}(E \cup \{\sqrt{\cdot}\}) \rightarrow \{\text{true, false}\}.$$  

We write $p(\sigma, t)$ if $\llbracket p \rrbracket(\sigma, t) = \text{true}$. This mapping of a trace $t$ is similar to the standard first-order logic interpretation based on states (see, e.g., Apt, de Boer and Olderog [4]). Occurrences of $\$ are replaced directly by the event content sequence of $t$. The equality comparison operator $\$ = ec can be formulated in first-order logic by using $\sigma$ to replace all the variables in ec before comparing it with the stripped $t$. As we assume that creation events in the trace always yields a fresh name, this freshness aspect does not need to be handled explicitly by the semantics of the trace assertion.

Substitution of all free occurrences of a variable $x$ by some expression or assertions $r$ in a trace assertion $p$ is denoted by $p[x/r]$. We assume that all variables and all substitutions are correctly typed.

As seen in the example, a specification triple $\{p\} D \{q\}$ consists of the trace assertions $p$ and $q$ and some entity name $D$, which is either some actor class name $C$ or a boxed component with an initial actor of class $C$.

We call $p$ and $q$ input and output trace assertions, respectively. All variables appearing only in $q$ (possibly due to an explicit creation of another actor or an implicit exposure of locally created actors) must be existentially quantified. As convention, the initial actor created is referred to by the variable this. The specification triple $\{p\} D \{q\}$ partially characterizes the semantics of the entity represented by $D$. Partial means that for each trace $t \in \llbracket D \rrbracket$ whose input part satisfies $p$, its output part satisfies also $q$. This specification technique does not give information about the rest of the traces that do not satisfy $p$. Despite the underspecification, the specification triple eliminates traces which satisfy $p$ and do not satisfy $q$.

An actor triple of class $C$ enforces that an actor of class $C$ is created and the environment can only call methods of this actor. The restriction given in the definition above

3In the classical Hoare logic [17], $p$ and $q$ are said to be the precondition and postcondition of the triple, respectively. However, the specification in this paper deals with input and output traces, which may include passing of a (new) actor name in an output event that later appears as the callee of an input event. In this sense, $p$ and $q$ are not pre- and postconditions.

is not enough to ensure that indeed only a single actor is considered local. Consider a trace assertion $q \equiv \text{true}$. A group of actors whose initial actor is the only exposed actor of the group can produce traces that matches the specification. By comparing the specification with a real trace semantics of the actor whose characteristics are described in Def. 3.2 we avoid this problem. Note that we should only use maximal traces to define the semantics of the specification, as a non-maximal trace lacks the information whether the actor has finished the tasks. It is possible that the actor responses with less or more events.

To define the semantics of the actor triple, a trace $t$ needs to be split into input and output traces. The function

$$\text{split}(t, L) = (t_1 \downarrow_{F,\text{callee}} \downarrow_{L,\text{callee}}, t_2 \downarrow_{L,\text{callee}} \downarrow_{F,\text{callee}})$$

does exactly so, where $F = A - L$.

**Definition 4.2 (Actor triple semantics).** Let $[[C]]$ be a trace set satisfying Def. 3.2 and representing the trace semantics of some actor $a$ of class $C$. $[[C]]$ satisfies $\{p\} C \{q\}$, written $\models C \{q\}$, if for all maximal trace $t \in [[[C]]]$ with $\text{split}(t, \{a\}) = (ti, to)$, the following holds:

$$\forall \sigma \bullet p(\sigma, ti) \implies q(\sigma, to).$$

A component triple $\{p\} [C] \{q\}$ states how the component with an initial actor of class $C$ replies to a given input trace. The semantics of a component triple is compared to the appropriate boxed component trace set and has the same form as for the actor triple. Thus the traces need to be split into input and output traces. This splitting can be done using the help of the function $\text{ext}(t)$ from Def. 3.6.

The definition below covers the semantics of boxed component triples.

**Definition 4.3 (Component triple semantics).** Let $[[C]]$ be a trace set satisfying Def. 3.3 that represents the trace semantics of component with an initial actor of class $C$. $[[C]]$ satisfies $\{p\} C \{q\}$, written $\models [p] C \{q\}$, if for all maximal trace $t \in [[[C]]]$ with $\text{split}(t, \text{ext}(t)) = (ti, to)$, then

$$\forall \sigma \bullet p(\sigma, ti) \implies q(\sigma, to).$$

Given triples as defined above, the specification of an actor class or a boxed component is a set of such triples. A trace set representing the actual behavior of the class or the component must satisfy each of the specification triples.

**Definition 4.4 (Specification).** Let $D$ be an actor class $C$ or a boxed class representing a boxed component $[C]$. A specification for $D$ is a set of specification triples

$$S = \{\{p_1\} D \{q_1\}, \ldots, \{p_n\} D \{q_n\}\}.$$  

$[[D]]$ satisfies $S$, written $\models S$, if

$$\forall (\{p\} D \{q\}) \in S \bullet \models \{p\} D \{q\}.$$
In Sect. 3, we provide a proof system for verifying the component specifications. To prove the component specifications, we need to come up with proof rules based on the trace semantics of the component specifications. To be able to prove the component specifications, we can argue for the conclusion.

In Sect. 3, the specifications for server and worker actor classes are given and we assume that they are correct. The specifications for the component counterparts are also given, but left as proof obligations. The server component is built by composing the worker component with the server actor. Thus, the component specification is built by composing as many worker actors as needed to complete the given task. To verify this component, the INDUCTION rule is used.

To focus more on the proof rules and their usage, we use the following abbreviations:

- \( \text{ISrv} \) defined as \((\text{this} \leftarrow \text{new} \ S \cdot (\text{this}.\text{sv}(\xi, t)) \cdot \sqrt{)}\)
- \( \text{OSrv} \) defined as \((\text{w} \leftarrow \text{new} \ W \cdot \text{do}(t) \cdot \text{pr}(\text{null}, \xi) \cdot \sqrt{)}\)
- \( \text{IWrk} \) defined as \((\text{this} \leftarrow \text{new} \ W \cdot \text{this}.\text{do}(t) \cdot \text{this}.\text{pr}(\xi, \xi) \cdot \sqrt{)}\)

The premises are trace assertions or specification triples. A rule allows to prove the conclusion from the premises. A rule with no premise is called an axiom (i.e., the conclusion is assumed to be true). To ensure that we obtain the correct conclusions, each rule must be proved to be sound, meaning that when the premises are assumed, using the semantics of the assertions and the specifications, we can argue for the conclusion.

In context of a proof system, a proof of a component triple is a sequence of proof rule applications. This sequence of applications is represented by a proof tree, where each node contains the specification triples and assertions that hold and the edge is labeled with the proof rules that is used. An assertion holds if for each trace and variable assignment, it is always evaluated to true.

RPSA also includes standard auxiliary rules similar to what is done by Apt, de Boer, and Olderog [4] and Poetzsch-Heffter [24]. The auxiliary rules we need for our example are given in Fig. 7. [INVARIANCE] allows to add conjuncts that do not refer to the trace constant. The predicate \( \text{consFree}(p) \) checks that \( p \) is free of any occurrences of \$. [SUBSTITUTION] allows substitutions of free variables \( x \) to some assertion or expression \( r \) that contains no trace constants. Except for the standard CONSEQUENCE rule and CLASSAXIOM (which should be checked against the implementation), all rules in RPSA are explained in more details along side their application in the running example.

5. Proof System

The specification technique described in the previous section allows us to focus on interesting functional properties of actor-based components and systems. Unlike the standard Hoare logic, where a primitive program statement (i.e., the second element of the triple) holds the basis how the assertions can evolve, we only have the information of an actor class name and its boxed status. In the proof system, we use the actor class specifications as axioms assuming that they are satisfied by the implementation. This assumption allows us to focus on proving the component specifications. To be able to prove the component specifications from the actor class specifications, we need to come up with proof rules based on the trace semantics given in Sect. 3. In this paper, we provide a proof system called Relational Proof System for Actors (RPSA) that can handle daisy chain composition. By daisy chain composition, we mean that an actor that creates another actor, sends messages to this newly created actor, never exposes its own name to the newly created actor nor the name of the newly created actor to other acting forming a chain of one way interaction.

RPSA presented in Fig. 5 (with the helper predicates stated in Fig. 6) contains an axiom and a number of rules. A rule consists of a number of premises and a specification triple as conclusion. The premises are trace assertions or specification triples. A rule allows to prove the conclusion from the premises. A rule with no premise is called an axiom (i.e., the conclusion is assumed to be true). To ensure that we obtain the correct conclusions, each rule must be proved to be sound, meaning that when the premises are assumed, using the semantics of the assertions and the specifications, we can argue for the conclusion.

In context of a proof system, a proof of a component triple is a sequence of proof rule applications. This sequence of applications is represented by a proof tree, where each node contains the specification triples and assertions that hold and the edge is labeled with the proof rules that is used. An assertion holds if for each trace and variable assignment, it is always evaluated to true.

RPSA also includes standard auxiliary rules similar to what is done by Apt, de Boer, and Olderog [4] and Poetzsch-Heffter [24]. The auxiliary rules we need for our example are given in Fig. 7. [INVARIANCE] allows to add conjuncts that do not refer to the trace constant. The predicate \( \text{consFree}(p) \) checks that \( p \) is free of any occurrences of \$. [SUBSTITUTION] allows substitutions of free variables \( x \) to some assertion or expression \( r \) that contains no trace constants. Except for the standard CONSEQUENCE rule and CLASSAXIOM (which should be checked against the implementation), all rules in RPSA are explained in more details along side their application in the running example.
• OWrkP $= (c, \text{resp}(\text{mrg}(c, \text{cmp}(t)))) : \checkmark$
• OWrk $= (w := \text{new} w \cdot w.json(rst(t)) \cdot \text{wp}(\text{mrg}(w, \text{cmp}(\text{fist}(t))), s)) : \checkmark$
• ISrvC $= (\text{this} := \text{new} s \cdot \text{this}.sv(c, t)) : \checkmark$
• OSrvC $= (c, \text{resp}(\text{cmp}(t)) : \checkmark$

The name are picked such that ISrv, for example, represents the input event content equality of the server actor triple, whereas OWrk represents the output event content equality of the worker actor triple in the inductive case. The C suffix indicates the assertion is used in a component triple. Note that the event content equality assertions in both worker actor triples and the worker component triple are the same, i.e., ISrv. In addition, the event content equality assertions for the worker actor triple of the primitive case and the worker component triple is the same, i.e., OWrkP. Let cse, short for content sequence extractor, be a function that extracts the event content sequences from these abbreviations.

Server Component. We start with proving the server component triple. The intention is to compose the server actor with the worker component. The rule that accommodates this composition is BoxedComposition

The BoxedComposition rule defines how an actor of class C can be combined with another actor or boxed component D to create boxed component [C]. For this rule to be applicable, three premises must hold. First, the actor triple C must guarantee that the actor’s name will not be exposed.

\[
\text{noSelfExp}(i, s) \equiv \forall c : c \in \text{Pref}(i) \cdot \text{call}(c) \neq \text{acq}(s)
\]

The predicate noSelfExp, short for no self exposure, takes variable i and the trace constant s representing the input and output traces, respectively. It guarantees a one way interaction, or in other words, ensures diacyl chaining because the current actor is not exposed. To ensure no exposure of an actor is made, the acquaintance of the output trace must not contain that actor. Second, the triple of D must ensure that no new actor is created by the instance of D. The predicate nonCr does exactly that.

\[
\text{nonCr}(s) \equiv \forall e^c : e \in \text{Pref}(s), C^c \in \text{CL} \cdot \text{msg}(e) \neq \text{new } C^c
\]

Third, the output produced by the actor of class C must match the input of the instance of D. In other words, the actor of class C exclusively feeds the instance of D in this particular case. This matching is handled by predicate match.

\[
\text{match}(q, q', D) \equiv q \implies \exists a \in A \cdot \text{firstCreated}(a, s) \land \text{classOf}(a, D) \land q'
\]

The predicate firstCreated checks if the first event is an actor creation and a represents the created actor. The predicate classOf checks if the created actor is of class D. The match predicate relies on the valid trace restriction (see Def. 3.1) where a trace starts with an actor creation. This restriction applies because the evaluation of q’ is done against an input trace, which always starts with an actor creation. For match to hold, the free variables of q and q’ should coincide. Note that because match is only used to link output trace assertion q to input trace assertion q’, there is no need to explicitly check that q represents an output trace assertion.

Neither the server actor triple nor the worker component triple is in the form needed to apply BoxedComposition. Therefore, we need to transform these triples using the Consequence, Invariance, and Substitution rules.

\[
\text{Axiom 1} [\text{ISrv} \land \exists w \cdot \text{OSrv}]
\]

\[
\text{Inv} [\exists w \cdot \text{OSrv} \land \exists s \cdot \text{OSrv}(s)]
\]

By Invariance a logical variable i is introduced to store the input trace. Because i cannot directly refer to s, we extract the event content sequence from the event content comparison ISrv. Then Consequence is used to strengthen the input trace assertion as required. At the same time, the output trace assertion is enriched with noSelfExp.

\[
\text{Sub} [\text{OWrkP}][\text{ISrv} \land \exists w \cdot \text{OSrv} \land \text{noSelfExp}(i, s)]
\]

The worker component triple must be adjusted, so it is ready to receive the input from the server actor. Substituting the variable v with null ensures we can match the output of the server actor to the input of the worker component. From the assumption in Sect. 2, we know that mrg(null, cmp(t)) = cmp(t). Therefore, we can infer OSrvC from OWrkP[\text{v/Null}]. The output trace assertion is enhanced by nonCr to state that the worker component creates no external actors, which holds from the definition of components (Def. 3.4). Now we apply BoxedComposition to obtain that the server component specification holds.

\[
\text{Sub} [\text{ISrv} \land i = s \land \exists w \cdot \text{OSrv} \land \text{noSelfExp}(i, s)]
\]

When a server actor receives a single request, we can match the output of the server actor to the input of the worker component. Thus the last premise of BoxedComposition is handled and we can derive the server component specification.
Worker Component. To prove the worker component triple, we can apply the [INDUCTION] rule. Similar to the server component case, the worker actor triples need to be massaged before the [INDUCTION] rule can be applied.

The [INDUCTION] rule deals if a component that creates as many instances of itself as needed to solve the task it has to do. This rule relies on having a measure expression $m$ depending on the event content sequence. The base and inductive cases are represented by the measure comparison $m = 0$ and $m > 0$, respectively, as mentioned.

If the measure yields zero, the actor on its own must represent the behavior of a component. That is, it does not create any other actor. For the inductive case, we see how the initial actor behaves. If from the actor specification it creates another actor of the same class and passes on a similar input to this new actor with the measure being reduced, this means we could apply the same specification again and again until we end in the base case. The reduction in the measure is captured by $z$, a variable that does not appear in other parts of the corresponding triple. The match predicate enforces this behavior. As in [BOXEDCOMPPOSITION] noSelfExp ensures that no self exposure is done. When these premises are fulfilled, then the component triple holds.

For the worker component, let the function $gt$ get the task parameter from an event content sequence (including $\$); in other words, $gt$ refers to the task parameter of the do method. Thus, we can define the measure as $sz(gt(\$)) - 1$. To improve the presentation, arithmetic manipulation to any boolean expressions is directly applied.

We begin with the worker actor that receives only a primitive task $t$ (i.e., $sz(t) = 1$). In this particular case the goal is to box the worker actor for creating no other actors. [BOXING] captures exactly this intention.

When the worker deals with a non-primitive task, the worker follows the inductive case of the worker actor triple. To achieve the second premise of [INDUCTION] the output trace assertion must include the information that the measure is reduced. In our case, we know that $sz(rst(t)) < sz(t)$, but we cannot extract the right task $t$ information from the trace constant of the output trace assertion. To get around this problem, we utilize the same approach to capture the event content sequence contained in the input trace assertion into a variable. By introducing $sz(gt(cse(IWrk))) = z$ to the output trace assertion using [INVARIANCE] we can weaken the output trace assertion to include the needed information. To include the no self exposure information in the output trace assertion, we follow the same approach to transform the worker component triple. In the proof tree below, we abbreviate $sz(gt(\$)) = z \land sz(gt(\$)) > 1$ as ind.

As the base and inductive cases hold, we can apply the [INDUCTION] rule. Thus, the worker component triple holds.

5.2 Soundness of RPSA

The soundness of RPSA is proved by induction on the depth of the proof trees. This means that each rule applied within the proof trees must be sound and the axiom is applied only when the actor triple is proved in the underlying system. The following lemmas show for all rules that the derived component’s specifications are valid if the premises are valid.

Lemma 5.1 (Soundness of [CONSEQUENCE]). Let $D$ be either an actor class $C$ or a component with initial actor of class $C$. Suppose $\models \{p_1\} D \{q_1\}, p \implies p_1$ and $q_1 \implies q$. Then $\models \{p\} D \{q\}$.

Proof. Follows from the first-order logic semantics. $\square$

The soundness of [CONSEQUENCE] follows from a straightforward manipulation of first-order logic.
Lemma 5.2 (Soundness of Boxing). Let $C$ be an actor class, $p$ and $q$ be trace assertions. Suppose $\vdash \{p\} C \{q\}$ holds. Then, $\vdash \{p\} C \{q\}$. 

Proof. The created actor is acting as a component (i.e., it follows Def. 3.4). \hfill \square

The soundness of the Boxing rule comes from the actor fulfilling the component definition (Def. 3.4) for input traces that fulfills the input trace assertion $p$. This rule immediately becomes unsound when we remove the no actor creation restriction because it falsifies the component definition.

Lemma 5.3 (Soundness of BoxedComposition). Let the following assumptions hold.

A1. $\vdash \{p \land i = \$\} C \{q \land \text{noSelfExp}(i, \$)\}$
A2. $\vdash \{q\} D \{r \land \text{nonCr}()\}$
A3. $\text{match}(q, q', D)$

Then, $\vdash \{p\} C \{r\}$. 

Proof. By cases. Here we consider $D$ to be an actor class $C'$. The proof for $D = [C']$ follows a similar outline.

Suppose $a$ is an actor of class $C$ and $b$ an actor of class $C'$, $t$ is a glass box trace that involves actors $a$ and $b$. Let $\text{split}(t, \{a\}) = (t_{i_{a} \rightarrow t_{o_{a}}} \rightarrow t_{i_{b} \rightarrow t_{o_{b}}})$, $F = A - \{a, b\}$ and $\sigma$ be a variable assignment such that $p(\sigma, t_{i_{a}})$. The goal is that if all assumptions hold, $t_{i_{F}} \in \text{Traces}([\{a, b\}])$, where in this case $\text{Traces}([\{a, b\}])$ represents $\llbracket [C] \rrbracket$.

- If $\lnot (q \land \text{noSelfExp}(i, \$))(\sigma, t_{o_{a}})$, then $t_{i_{\{a\}}} \notin \text{Traces}(a)$ by A1 and Def. 4.2. By Def. 5.7, $t_{i_{F}} \notin \text{Traces}(\{a, b\})$.
- If $(q \land \text{noSelfExp}(i, \$))(\sigma, t_{o_{a}})$, then $t_{i_{\{a\}}} \in \text{Traces}(a)$ by A1 and Def. 4.2. Because of A3, we have $q'(\sigma, t_{i_{a}})$.

- If $\lnot (r \land \text{nonCr}())(\sigma, t_{o_{b}})$, then either $t_{i_{\{b\}}} \notin \text{Traces}(b)$ by A2 and Def. 4.2 or $\lnot \text{noCr}()$. For the former case, by Def. 3.7, $t \notin \text{Traces}(\{a, b\})$. For the latter, A2 is not fulfilled.

- If $(r \land \text{nonCr}())(\sigma, t_{o_{b}})$, then $t_{i_{\{b\}}} \in \text{Traces}(b)$ by A2 and Def. 4.2. This means $t \in \text{Traces}(\{a\})$ and by Lemma 5.6, $t_{i_{F}} \in \text{Traces}(\{a, b\})$.

By Def. 4.3, $\vdash \{p\} C \{r\}$. \hfill \square

The essence of the lemma’s proof is that whenever the initial actor is given some input trace that satisfies the input trace assertion, the actor will produce an output trace to the other subcomponent of $[C]$ such that this subcomponent produces the output trace that is required by the output trace assertion. The matching, no self exposure and non-creational predicates restrict the case such that no other output event is leaked out except for the expected ones.

Lemma 5.4 (Soundness of Induction). Suppose the following assumptions hold.

B1. $\vdash \{p \land m = 0\} C \{q\}$
B2. $\vdash \{p \land m = z \land m > 0 \land i = \$\} C \{p' \land m < z \land \text{noSelfExp}(i, \$)\}$
B3. $\text{match}(p', p, C)$

Then, $\vdash \{p\} C \{q\}$. 

Proof. By induction. Suppose we clone the class $C$ into an unbounded number of classes named $C_0, C_1, \ldots$. Without loss of generality, we reformulate assumptions B2 and B3 as follows:

B2'. $\forall k \in \mathbb{N} \cdot \vdash \{p_k \land m = z \land m > 0 \land i = \$\} C_k$
B3'. $\forall k \in \mathbb{N} \cdot \text{match}(p'_{k+1}, p_k, C_k)$

where $p_k$ and $p'_{k+1}$ are the same as $p$ and $p'$, respectively, except for the created actor in the input and output traces of $C_k$ being of class $C_{k-1}$, respectively. $\mathbb{N}$ represents the set of natural numbers. Moreover, $k$ is always picked exactly to handle the measure $m$.

Suppose $L = \{a_0, a_1, \ldots\}$ are actors of classes $C_0, C_1, \ldots$ respectively (i.e., $a_0, a_1, \ldots$ are all of class $C$) and $F = A - L$. We proceed with the proof by induction on the number of actors that are created. Let $t$ be a valid trace such that there are $k + 1$ actors $a_0, a_1, \ldots, a_k$ created with $a_k$ to be the initial actor. The goal is to prove for each case that if the input trace assertion $p$ is fulfilled, then if output trace assertion $q$ is fulfilled, $t_{i_{F}} \in \llbracket [C] \rrbracket$.

Base case: $k = 0$. Because $a_0$ is the only actor created in $t$, $t$ is already an actor trace of $a_0$. By Lemma 3.4, $t$ is also a boxed trace. Let $\text{split}(t, \{a_0\}) = (t_{i_{a_0}})$, $t_{i_{a_0}}$ satisfies $\{a_0\}$, $t_{i_{F}} \in \llbracket [C] \rrbracket$, otherwise $t_{i_{F}} \notin \llbracket [C] \rrbracket$.

Inductive step: $k = n + 1$. Let $L' = \{a_0, \ldots, a_{n-1}\}$ and $t'$ be a boxed trace of $L'$ with the initial actor $a_{n-1}$ and $\text{split}(t', L') = (t'_{i_{a_{n-1}}}, t'_{i_{a_n}})$. The induction hypothesis is that $L'$ is a component and if $t'$ satisfies $p_{n-1}(\sigma, t'_{i_{a_{n-1}}})$ for some variable assignment $\sigma$, then $q(\sigma, t'_{i_{a_n}})$.

Let $\text{split}(t_{i_{a_n}}) = (t_{i_{a_n}}, t_{o_{a_n}})$ and $\text{split}(t_{i_{L'}}) = (t_{i_{L'}}, t_{o_{L'}})$. Let $\sigma$ be a valid assignment such that $p_{n}(\sigma, t_{i_{a_n}})$.

If $\lnot p'_{n}(\sigma, t_{o_{a_n}})$, then $p_{n+1}$ is violated. Thus, $p'_{n}$ must hold. Furthermore, $a_n$ does not expose itself to actors it creates. From B3' we obtain that $p_{n-1}(\sigma, t_{i_{L'}})$. By the induction hypothesis, $q(\sigma, t_{i_{L'}})$ must hold. Because $a_n$ is not exposed, $t_{o_{L'}}$ is also the output trace of $L$ (that is, the output trace of $L'$ does not contain output events that are directed to $a_n$). Note that $a_n$ is the only actor that can be created by the environment, because $L'$ is boxed and the initial actor $a_{n-1}$ is created by $a_n$. By Def. 3.4, $L$ is a component. Moreover, as $p_n$ is essentially the same as $p$, the input trace of $L$ is the same as $t_{i_{a_n}}$. Therefore, $t_{i_{F}} \in \llbracket [C] \rrbracket$.
By induction principle and Def. [4.3] \( p \models \{ C \} \{ r \}. \)

The soundness of \textbf{Induction} is proved by induction on the number of actors of class \( C \) that are created. Because all actors carry the same characteristics, never expose one actor to the next one, and produce output that is passed as a whole to the next actor (except for the last actor that produces output exclusively to the environment), we can hierarchically box the actors from the innermost to the outermost layer by layer. By employing the boxed second outermost layer as induction hypothesis, the proof of the inductive case is carried out in a similar way to the proof for the soundness of the \textbf{BoxedComposition} rule.

From the lemmas above, we conclude that RPSA is sound.

**Theorem 5.5.** The proof system RPSA is sound.

### 6. Related Work

We consider related work in the areas of semantics, component specification, and logic.

**Semantics.** There are plenty of semantics proposed for actor-based systems (e.g., [2, 9, 16, 28, 30]). The trace semantics used in this paper is inspired by Vasconcelos and Tokoro’s trace semantics [30] and Talcott’s interaction path [28]. Rather than having the behavior of actors evolve depending on what input the actors receive, classes are used to provide more structure to the behavior of the actors. Instead of employing an independence relation or a partial-order relation between events to mimic the buffered message passing communication [2], we use method interaction and the caller information is introduced into the events to allow projection-based composition. This approach avoids the need to come up with and maintain these relations. The traces can be extracted from actor-based programming languages using the guess and merge approach [3].

**Specification.** \( \pi \)-calculus [22] can be used to specify actor systems, but it needs some restrictions on the syntax and introduction of actor identities. Verifying a component specification from the actor class specifications involves bisimulation.

Specification Diagram [26] provides a detailed way to specify how an actor system behaves. To check whether a component specification produces the same behavior as the composition of the specification of its subcomponents one has to perform a non-trivial interaction simulation on the level of the state-based operational semantics. By extending \( \pi \)-calculus, a may testing [13] characterization of Specification Diagram can be obtained [29].

Our specification technique is strongly related to Focus [8]. However, Focus provides no support for messages, name transfer and dynamic creation — necessary features for actor systems.

**Logic.** Several logics have been developed to reason about the functional behavior of actor-based implementations. For the verification process, most of them are based on a state-based semantics and rely on having the actual implementations. For example, Darlington and Guo [11] provides a linear logic description of the state-based semantics of an actor system. Arts and Dam [2], the running example’s source, used extended first-order \( \mu \)-calculus [10] to verify Erlang programs (51) by evaluating the state of the program. The use of temporal logic have been considered by Duarte [15] and Schacht [25].

De Boer [12] presented a Hoare logic for concurrent processes that communicate by message passing through FIFO channels (similar to actors). He described a similar two-tier architecture, where the assertions are based on local and global rules. The local rules deal the local state of a process, whereas the global rules deal with the message passing and creation of new processes. However, they only work for closed systems.

An example of a logic that is based on trace semantics is the work by Soundarajan [27]. Soundarajan proposed a specification technique more general than ours and a proof system than can handle a fixed finite number of processes. A specification of an object of a single class may be represented using Soundarajan’s specification technique:

\[
p[$/t_{input}] \implies q[$/t_{output}]
\]

Nevertheless, our Hoare-like triple is convenient when the interleaving of input and output need not be specified.

Ahrendt and Dylla [3] and Din et al. [14] extended Soundarajan’s work to deal with actor systems. They consider only finite prefix-closed traces, justifying it by having only finite number of actors to consider in the verification process. Din et al. particularly verified whether an implementation of an actor class satisfies its actor triples by transforming the implementation in a simpler sequential language, applying the transformational method proposed by Olderog and Apt [23]. The main difference between this work and the aforementioned work on trace semantics is on the notion of component that hides a group of actors into a single entity. This avoids starting from the class specifications of each actor belonging to a component when verifying a property of the component.

### 7. Conclusion

In this paper, we have presented a logic that supports compositional specification and verification of open concurrent component-based systems in terms of an actor model. The semantics of actors is represented using traces of events, from which we derived the notion of dynamic, hierarchical components. As each event reveals either an action of sending a message from an actor to another or creation of a new actor, our trace semantics provides all information about the observable behavior of the actors.
and components. The Hoare-like specification triples are designed to state the precise, albeit partial, response of an actor or a component to the input it receives. We then proposed a sound and compositional axiomatic proof system that handles components that form a daisy chain with only one-way interaction between their subcomponents. By assuming the actor specifications, the proof system can focus on proving component and system-wide functional properties. An illustration on how the specification technique and the proof system can be applied is given by means of a client-server example.

**Future work.** Several future directions include deriving the trace validity definition from a simple operational semantics of actor systems, incorporating a more general specification technique by adapting the Soundarajan’s approach and allowing abstract state information, and an extension of the proof system to cover more flexible composition schemes. In addition, to have a closer connection to actor programming/modeling languages, such as ABS, we would like to support more complex communication constructs such as futures.

**Acknowledgments**

We thank Mads Dam for several fruitful discussions and for his suggestion to use the problem described in [6] as an example. Yannick Welsch gave feedback to improve the presentation of the paper. This work is partially supported by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models.

**References**


