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

Keywords actors, specification techniques, open system

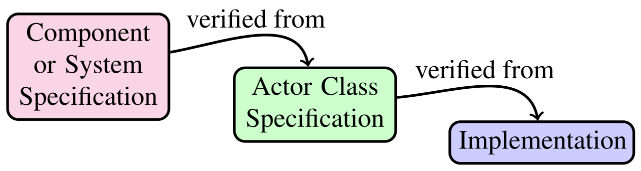

Figure 1. Two-tier verification

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 [18] and ABS [19], 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. [13] 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 trace-based specification technique 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 [17]. 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 [16]. 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 actors 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 [5] 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. Proofs of all lemmas can be found in the supplementary material.

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 [18] and ABS [19], 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 names.

As a running example, we use a variant of the client-server setting given in an industrial Erlang case study by Arts and Dam [5]. 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 passes 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. In the actor setting, 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.

```
interface Client { response(Value); }
interface Server { serve(Client, CompTask); }
interface Worker { do(CompTask);
                   propagateResult(Value, Client); }
actor class AServer implements Server {
  serve(Client c, CompTask t) { // taskSize(t) ≥ 1
    Worker w = new AWorker();
    w.do(t); w.propagateResult(null, c);
} }
```

Figure 2. Interfaces and the actor class `AServer`

Figures 2 and 3 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(c, 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 task 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(p̅)` is executed. This statement, known also as a method call, sends the message `m(p̅)` to the receiver actor `r` where `m` is the method name with a list of parameters `p̅`. 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 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. 2) 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.

```
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 {
    if (nxtWorker == null) {
      c.response(merge(myResult, v));
    } else {
      nxtWorker.propagateResult(merge(myResult, v), c);
} } }
```

Figure 3. Actor class `AWorker`

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. [20, 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:

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

where `compute(t)` computes the result of `t`; `taskSize(t)` yields the 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:

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

A task `t` consists of at least one chunk; if `t` cannot be partitioned, the function `firstTask` returns `t`; 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; 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 [7, 17]. 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 [28] 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 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 *sub*components interact with each other. For this purpose, we provide a glass box view [7, p. 5], given the structure of the component.

3.1 Trace Foundation

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

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

From this foundation, we build the set of *events* $\mathbf{E}$. An event $e \in \mathbf{E}$ represents the occurrence of a message $m = msg(e)$ being sent by the *caller* actor $a = caller(e)$ to the *callee* actor $b = callee(e)$. 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 $a \rightarrow b := \texttt{new}\ C$ or $a \rightarrow b.\texttt{mtd}(\overline{p})$ 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 $L \subseteq \mathbf{A}$ to be the set of (**l**ocal) actors we are considering and, by the open system setting, $F = \mathbf{A} - L$ as the set of all (**f**oreign) actors L is interacting with, an event e that appears in the trace can be categorized as 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 $a, b \in L$; or 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 $acq(a \rightarrow b.\texttt{mtd}(\overline{p}))$, 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.

A *trace* $t \in Seq\langle \mathbf{E} \cup \{\surd\}\rangle$ is a finite sequence of events that represents a single execution of the entity L it represents. The $\surd$ 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 $A \subseteq \mathbf{A}$, written $t{\downarrow}_A$, where all events, except $\surd$, whose neither caller nor callee is not in A are dropped. The function is refined by a caller (or callee) parameter $t{\downarrow}_{A,caller}$ ($t{\downarrow}_{A,callee}$) 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 $T{\downarrow}_A$ 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 follows that acq grows monotonically with respect to the length of the trace. The function $cr(t)$ 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 $acq(t{\downarrow}_{A,callee}) \cup cr(t{\downarrow}_{A,caller})$ 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) actor(s) at any point in time. In addition to this valid trace restriction, we assume a creation message always produces a fresh actor name. Thus, the creation of actors forms a tree indicating which actor is created (directly or indirectly) by which other actor.

Definition 3.1 (**Valid trace set**)**.** Let $L \subseteq \mathbf{A}$ be the set of local actors, $F = \mathbf{A} - L$ and $C \in \mathbf{CL}$. Let also e be an event such that $caller(e) = a$, $callee(e) = b$, $msg(e) = m$. A set of traces T is *valid* if

1. $\forall t \cdot e \in T \bullet t \in T$ (prefix closed);
2. $\forall t \cdot e \in T \bullet a \neq F \vee b \neq F$ (non-external events);
3. $\forall e \cdot t \in T \bullet m = \texttt{new}\ C \wedge a \in F \wedge b \in L$ (initial creation);
4. $\forall t \cdot e \in T \bullet a \in F \implies$
$\{b \mid isMtd(m)\} \cup acq(e) \subseteq exposedTo(t, F) \cup F$
(proper local actor exposure to foreign actors);
5. $\forall t \in T, e \in \mathbf{E} \bullet a \in F \wedge b \in L \wedge isMtd(m) \wedge$
$\{b\} \cup acq(e) \subseteq exposedTo(t, F) \cup F \implies t \cdot e \in T$
(open environment admissibility).

A trace that satisfies requirements 2, 3 and 4 is a *valid trace*.

The primitive building blocks of our systems are actors whose behavior is represented by some class. We assume that a self call does not appear in the trace of an actor. Changes that occur with a self call can be simulated allowing non-deterministic choices of trace continuation, without affecting the information the actor receives from its environment.

Definition 3.2 (**Actor trace set**)**.** Given a class name $C \in \mathbf{CL}$ and an actor a where $class(a) = C$, the *trace set of the actor of class* C is a valid trace set (Def. 3.1) $Traces(a)$ such that

- $\forall e \cdot t \in Traces(a) \bullet callee(e) = a \wedge msg(e) = \texttt{new}\ C$ (starts with a creation of a), and
- $\forall t \cdot e \in Traces(a) \bullet caller(e) = a \wedge m = msg(e) \implies$
$\{callee(e) \mid isMtd(m)\} \cup acq(e) \subseteq acq(t{\downarrow}_{F,callee}) \cup \{a\}$
(proper foreign actor exposure to a).

A valid trace which either callee or caller of its events is the actor a and 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 $\surd$ 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 $L \subseteq \mathbf{A}$ be a set of actors and $F = \mathbf{A} - L$. The *plain* trace set $\mathit{Traces}(L)$ is the largest possible set such that

- $\forall t \in \mathit{Traces}(L), a \in L \bullet t{\downarrow_{\{a\}}} \in \mathit{Traces}(a)$, and
- $\forall e \cdot t \cdot e' \in \mathit{Traces}(L) \bullet \mathit{caller}(e') \in F \wedge m = \mathit{msg}(e') \implies \{\mathit{callee}(e') \mid \mathit{isMtd}(m)\} \cup \mathit{acq}(e') \subseteq \mathit{exposedTo}(e \cdot t, F)$.

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 $\mathit{Traces}(L) = \mathit{Traces}(a)$ is plain.*

Using plain composition, we can characterize a component as a set of actors that does not create actors outside of the set. This dynamic notion of a component is motivated by the hierarchical nature of actor creation, which makes the component independent of actors in the environment with respect to its internal behavior. Note that this does not prevent the component to interact with the environment. We restrict ourselves to components where only one actor is created by the component's environment. This actor is called the *initial actor* of the component. This restriction allows us to refer to a component by the class of the initial actor C. The notion of such a component is formalized as follows.

Definition 3.4 (Component). L is a *component* with an initial actor of class C if

- $\forall e \cdot t \in \mathit{Traces}(L) \bullet \mathit{msg}(e) = \textbf{new}\ C$
  (starts by creating some actor of class C), and
- $\forall e \cdot t \cdot e' \in \mathit{Traces}(L) \bullet \forall C' \in \mathbf{CL} \bullet \mathit{msg}(e') = \textbf{new}\ C' \implies \mathit{callee}(e') \in L$
  (all created actors afterwards are local).

By this definition, an actor that never creates another actor is a component. The presence of $\surd$ in a trace of a component means the component receives no more input from the environment and finishes processing all input events.

Composing the traces of individual actors that form a component retains the validity property of the resulting trace set as shown in the following lemma. The main reason the validity holds is that we define the plain trace set to be the largest set of possible traces.

Lemma 3.2 (Component plain trace set validity). *Let $\mathit{Traces}(L)$ be a plain trace set of a component L. Then, $\mathit{Traces}(L)$ is valid.*

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 $F = \mathbf{A} - L$. The *boxed* component of L, denoted by $[L]$, is the trace set $\mathit{Traces}([L])$ where

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

We refer to a trace in $\mathit{Traces}([L])$ as a boxed component trace. Given that the component L has an initial actor of class C, then $[\![[C]]\!]$ denotes trace set $\mathit{Traces}([L])$. 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 $\mathit{Traces}([L])$ is valid.*

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, $\mathit{Traces}(L) = \mathit{Traces}([L])$.*

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 $\mathit{ext}(T)$, short for name extraction, is made by collecting the names that are exposed through actor creation, method call parameters and callee 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. $L' = \mathit{ext}(T)$ is the subset of actors in the component, where $\mathit{ext}(T) = \bigcup_{t \in T} \mathit{ext}(t)$, $\mathit{ext}([]) = \emptyset$, and

$$\mathit{ext}(t \cdot e) = \begin{cases} \mathit{ext}(t) \cup \{\mathit{callee}(e)\}\ , \text{ if } \mathit{msg}(e) = \textbf{new}\ C \\ \mathit{ext}(t) \cup \{\mathit{caller}(e)\} \cup \mathit{acq}(e) - (\mathit{acq}(t) - \mathit{ext}(t)), \\ \qquad \text{if } \mathit{isMtd}(\mathit{msg}(e)) \wedge \mathit{callee}(e) \notin \mathit{ext}(t) \end{cases}$$

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 caller 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 [7, p. 5].[1]

Definition 3.7 (**Glass box composition**). Let $L = L' \cup \{a\}$ be a component such that L' is a component and $a \notin L'$ is an actor name. Let also $F = \mathbf{A} - L$, $C = class(a)$. The *glass box composition* of L' and a, denoted $\langle L'|a\rangle$, is the largest trace set $T = Traces(\langle L'|a\rangle)$ where

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

Restrictions in terms of creation are applied to the actor and the component that are composed, because the resulting composition should be a component that starts with the initial actor a. 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', a fulfill Def. 3.7. Then $Traces(\langle L'|a\rangle)$ is valid.*

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 $L = L' \cup \{a\}$ fulfill Def. 3.7 and $F = \mathbf{A} - L$. Then, $Traces(\langle L'|a\rangle){\downarrow_F} = Traces([L])$.*

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 [16]. This approach is extended by Broy [6] 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 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.

$$\{\ \$ = (\underline{\texttt{this}} := \texttt{new S} \cdot \underline{\texttt{this}}.\texttt{sv}(\underline{\texttt{c}},\underline{\texttt{t}}) \cdot \surd)\ \}\quad \texttt{S}$$
$$\{\ \exists \underline{\texttt{w}} \bullet \$ = (\underline{\texttt{w}} := \texttt{new W} \cdot \underline{\texttt{w}}.\texttt{do}(\underline{\texttt{t}}) \cdot \underline{\texttt{w}}.\texttt{pR}(\texttt{null},\underline{\texttt{c}}) \cdot \surd)\ \}$$

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.

$$\{\ \$ = (\underline{\texttt{this}} := \texttt{new S} \cdot \underline{\texttt{this}}.\texttt{sv}(\underline{\texttt{c}},\underline{\texttt{t}}) \cdot \surd)\ \}\quad [\texttt{S}]$$
$$\{\ \$ = (\underline{\texttt{c}}.\texttt{resp}(\texttt{cmp}(\underline{\texttt{t}})) \cdot \surd)\ \}$$

The name `[S]` indicates that the triple deals with a boxed

[1] In Def. 3.7, we describe the glass box composition between an actor and a component, sufficient for proving the soundness of the proof system. A generalized definition can be found in the supplementary material.

[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.

server component. That is, the assertions are checked against a boxed component trace with an initial actor of class S.

Specifying the worker class motivates why we may have more than one triple that represents the functional behavior of a worker. More precisely, the worker class is described using two specification triples, each handling the base and inductive cases, respectively. The first specification triple handles the case when the task has exactly one subtask. In this particular case, the worker sends back to the client the result of merging the propagated result value with the computation of the subtask.

$\{ \$ = (\underline{\mathtt{this}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{pR}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \cdot \surd) \wedge \mathtt{sz}(\underline{\mathtt{t}}) = 1 \}$

$\mathtt{W}$

$\{ \$ = (\underline{\mathtt{c}}.\mathtt{resp}(\mathtt{mrg}(\underline{\mathtt{v}}, \mathtt{cmp}(\underline{\mathtt{t}}))) \cdot \surd) \}$

In the second case, the task consists of multiple subtasks. In this case, the worker creates another worker, passes on the rest of the task, then processes the current subtask. When the computation of the current subtask is finished, the worker merges the computation result with the previous result it receives and propagates the merged result to the other worker.

$\{ \$ = (\underline{\mathtt{this}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{pR}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \cdot \surd) \wedge$
$\quad \mathtt{sz}(\underline{\mathtt{t}}) = \underline{\mathtt{n}} \wedge \underline{\mathtt{n}} > 1 \}$

$\mathtt{W}$

$\{ \exists \underline{\mathtt{w}} \bullet \$ = (\underline{\mathtt{w}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathtt{w}}.\mathtt{do}(\mathtt{rst}(\underline{\mathtt{t}})) \cdot$
$\qquad \underline{\mathtt{w}}.\mathtt{pR}(\mathtt{mrg}(\underline{\mathtt{v}}, \mathtt{cmp}(\mathtt{fst}(\underline{\mathtt{t}}))), \underline{\mathtt{c}}) \cdot \surd) \}$

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.

$\{ \$ = (\underline{\mathtt{this}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{pR}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \cdot \surd) \}$

$[\mathtt{W}]$

$\{ \$ = (\underline{\mathtt{c}}.\mathtt{resp}(\mathtt{mrg}(\underline{\mathtt{v}}, \mathtt{cmp}(\underline{\mathtt{t}}))) \cdot \surd) \}$

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 \$ can be used. In the input (output) condition, \$ 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 $\$ = ec$, 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 \$ 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 (\$ may be present).
- If p, q are assertions and $\underline{x}$ is a variable, then $\neg p, p \wedge q$, $\exists \underline{x} : p$ are also assertions.

The other logical operators, e.g., $\vee$, $\implies$ and $\forall$, are derived in the usual way. Given a trace assertion p, the function *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 $\mathbf{V}$ be the set of all variables. A *variable assignment* $\sigma : \mathbf{V} \to \mathbf{A} \cup \mathbf{D}$ is a function that maps (some) variables to values.

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

$$[\![p]\!] : (\mathbf{V} \to \mathbf{A} \cup \mathbf{D}) \times Seq\langle E \cup \{\surd\}\rangle \to \{\mathbf{true}, \mathbf{false}\}\ .$$

We write $p(\sigma, t)$ if $[\![p]\!](\sigma, t) = \mathbf{true}$. 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 σ 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 $\underline{x}$ by some expression or assertions r in a trace assertion p is denoted by $p[\underline{x}/r]$. 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 $\underline{\mathtt{this}}$. The specification triple $\{p\}\ D\ \{q\}$ partially characterizes the trace semantics of the entity represented by D. Partial means that for each trace $t \in [\![D]\!]$ 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 is not enough to ensure that indeed only a single actor is considered local. Consider a trace assertion $q \equiv \mathbf{true}$. 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 $split(t, L) = (t{\downarrow_{F,caller}}{\downarrow_{L,callee}}, t{\downarrow_{L,caller}}{\downarrow_{F,callee}})$ does exactly so, where $F = \mathbf{A} - L$.

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 $\vDash \{p\}\ C\ \{q\}$, if for all maximal traces $t \in [\![C]\!]$ with $split(t, \{a\}) = (ti, to)$, 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 $ext(t)$ 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. 3.5 that represents the trace semantics of component with an initial actor of class C. $[\![[C]]\!]$ satisfies $\{p\}\ [C]\ \{q\}$, written $\vDash \{p\}\ [C]\ \{q\}$, if for all maximal traces $t \in [\![[C]]\!]$ with $split(t, ext(t)) = (ti, to)$,

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

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 $\vDash S$, if

$$\forall (\{p_i\}\ D\ \{q_i\}) \in S \bullet \vDash \{p_i\}\ D\ \{q_i\}.$$

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 actors forming a chain of one way interaction.

ClassAxiom

$$\{p\}\ C\ \{q\}$$

Boxing

$$\frac{\{p\}\ C\ \{q \land nonCr(\$)\}}{\{p\}\ [C]\ \{q\}}$$

BoxedComposition

$$\frac{\begin{array}{c}\{p \land \underline{i} = \$\}\ C\ \{q \land noSelfExp(\underline{i}, \$)\}\\ \{q'\}\ D\ \{r \land nonCr(\$)\} \qquad match(q, q', D)\end{array}}{\begin{array}{c}\{p\}\ [C]\ \{r\}\\ \text{where } \underline{i} \notin free(p) \cup free(q)\end{array}}$$

Consequence

$$\frac{\begin{array}{c}p \implies p_1\\ \{p_1\}\ D\ \{q_1\}\\ q_1 \implies q\end{array}}{\{p\}\ D\ \{q\}}$$

Induction

$$\frac{\begin{array}{c}\{p \land m = 0\}\ [C]\ \{q\} \qquad match(p', p, C)\\ \{p \land m = \underline{z} \land m > 0 \land \underline{i} = \$\}\ C\ \{p' \land m < \underline{z} \land noSelfExp(\underline{i}, \$)\}\end{array}}{\begin{array}{c}\{p\}\ [C]\ \{q\}\\ \text{where } \underline{i} \notin free(p) \cup free(q)\end{array}}$$

Figure 4. Proof rules for *RPSA*

$$noSelfExp(i, \$) \overset{\text{def}}{=} \forall c \cdot ec \in Pref(i) \bullet callee(c) \notin acq(\$)$$

$$nonCr(\$) \overset{\text{def}}{=} \forall ec' \cdot e \in Pref(\$), C' \in \mathbf{CL} \bullet msg(e) \neq \mathtt{new}\ C'$$

$$match(q, q', D) \overset{\text{def}}{=} q \implies \exists a \in \mathbf{A} \bullet firstCreated(a, \$) \land classOf(a, D) \land q'$$

Figure 5. Helper predicates for *RPSA*

Invariance

$$\frac{\{p\}\ D\ \{q\}}{\begin{array}{c}\{p \land r\}\ D\ \{q \land r\}\\ \text{where } consFree(r)\end{array}}$$

Substitution

$$\frac{\{p\}\ D\ \{q\}}{\begin{array}{c}\{p[\underline{x}/r]\}\ D\ \{q[\underline{x}/r]\}\\ \text{where } x \in free(p) \cup free(q)\\ \text{and } consFree(r)\end{array}}$$

Figure 6. Auxiliary rules for *RPSA*

RPSA presented in Fig. 4 (with the helper predicates stated in Fig. 5) 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* (that is, the conclusion is assumed to be true). To ensure that we obtain the correct conclusions, each rule must be proved to be *sound*. Sound means 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 are 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 [22]. The auxiliary rules we need for our example are given in Fig. 6. INVARIANCE allows to add conjuncts that do not refer to the trace constant. The predicate $consFree(p)$ checks that there are no occurrences of \$ in p. SUBSTITUTION allows substitutions of free variables x to some assertion or expression r that does not contain the trace constant. Except for the standard CONSEQUENCE rule and CLASS-AXIOM (which should be checked against the implementation), all rules in *RPSA* are explained in more details along side their application in the running example.

5.1 Verifying Example Component Specifications

In Sect. 4, 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 BOXEDCOMPOSITION is used to verify the server component specification. The worker component is built by composing as many worker actors as needed to complete the given task. To verify this component, INDUCTION is used.

To focus more on the proof rules and their usage, the event content equality assertions are abbreviated as follows.

$$\mathsf{ISrv} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{this}} := \mathbf{new}\ \mathsf{S} \cdot \underline{\mathsf{this}}.\mathsf{sv}(\underline{\mathsf{c}}, \underline{\mathsf{t}}) \cdot \surd)$$
$$\mathsf{OSrv} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{w}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathsf{w}}.\mathsf{do}(\underline{\mathsf{t}}) \cdot \underline{\mathsf{w}}.\mathsf{pR}(\mathbf{null}, \underline{\mathsf{c}}) \cdot \surd)$$
$$\mathsf{IWrk} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{this}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathsf{this}}.\mathsf{do}(\underline{\mathsf{t}}) \cdot \underline{\mathsf{this}}.\mathsf{pR}(\underline{\mathsf{v}}, \underline{\mathsf{c}}) \cdot \surd)$$
$$\mathsf{OWrkP} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{c}}.\mathtt{resp}(\mathtt{mrg}(\underline{\mathsf{v}}, \mathtt{cmp}(\underline{\mathsf{t}}))) \cdot \surd)$$
$$\mathsf{OWrkI} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{w}} := \mathbf{new}\ \mathtt{W} \cdot \underline{\mathsf{w}}.\mathsf{do}(\mathtt{rst}(\underline{\mathsf{t}})) \cdot \underline{\mathsf{w}}.\mathsf{pR}(\mathtt{mrg}(\underline{\mathsf{v}}, \mathtt{cmp}(\mathtt{fst}(\underline{\mathsf{t}}))), \underline{\mathsf{c}}) \cdot \surd)$$
$$\mathsf{ISrvC} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{this}} := \mathbf{new}\ \mathsf{S} \cdot \underline{\mathsf{this}}.\mathsf{sv}(\underline{\mathsf{c}}, \underline{\mathsf{t}}) \cdot \surd)$$
$$\mathsf{OSrvC} \stackrel{\text{def}}{=} \$ = (\underline{\mathsf{c}}.\mathtt{resp}(\mathtt{cmp}(\underline{\mathsf{t}})) \cdot \surd)$$

The name are picked such that ISrv, for example, represents the input event content equality of the server actor triple, where as OWrkI 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., IWrk. 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 *c*ontent *s*equence *e*xtractor, 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.

$$noSelfExp(i, \$) \stackrel{\text{def}}{=} \forall c \cdot ec \in Pref(i) \bullet callee(c) \notin acq(\$)$$

The predicate *noSelfExp*, short for *no self exposure*, takes variable i and the trace constant \$ representing the input and output traces, respectively. It guarantees a one way interaction, or in other words, ensures daisy 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 foreign actor is created by the instance of D. The predicate *nonCr* does exactly that.

$$nonCr(\$) \stackrel{\text{def}}{=} \forall ec' \cdot e \in Pref(\$), C' \in \mathbf{CL} \bullet msg(e) \neq \mathbf{new}\ C'$$

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*.

$$match(q, q', D) \stackrel{\text{def}}{=} q \implies \exists a \in \mathbf{A} \bullet firstCreated(a, \$) \wedge classOf(a, D) \wedge q'$$

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.

$$\textsc{Cns}\ \dfrac{\textsc{Inv}\ \dfrac{\textsc{Axiom}\ \dfrac{}{\{\mathsf{ISrv}\}\ \mathsf{S}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OSrv}\}}}{\{\mathsf{ISrv} \wedge \underline{\mathsf{i}} = cse(\mathsf{ISrv})\}\ \mathsf{S}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OSrv} \wedge \underline{\mathsf{i}} = cse(\mathsf{ISrv})\}} \quad \begin{array}{c} \mathsf{ISrv} \wedge \underline{\mathsf{i}} = \$ \implies \mathsf{ISrv} \wedge \underline{\mathsf{i}} = cse(\mathsf{ISrv}) \\ \exists \underline{\mathsf{w}} \bullet \mathsf{OSrv} \wedge \underline{\mathsf{i}} = cse(\mathsf{ISrv}) \implies \exists \underline{\mathsf{w}} \bullet \mathsf{OSrv} \wedge noSelfExp(\underline{\mathsf{i}}, \$) \end{array}}{\{\mathsf{ISrv} \wedge \underline{\mathsf{i}} = \$\}\ \mathsf{S}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OSrv} \wedge noSelfExp(\underline{\mathsf{i}}, \$)\}}$$

By INVARIANCE, a logical variable $\underline{\mathsf{i}}$ is introduced to store the input trace. Because $\underline{\mathsf{i}}$ cannot directly refer to \$, 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*.

$$\textsc{Cns}\ \dfrac{\textsc{Sub}\ \dfrac{\{\mathsf{IWrk}\}\ [\mathtt{W}]\ \{\mathsf{OWrkP}\}}{\{\mathsf{IWrk}[\underline{\mathsf{v}}/\mathbf{null}]\}\ [\mathtt{W}]\ \{\mathsf{OWrkP}[\underline{\mathsf{v}}/\mathbf{null}]\}} \quad \mathsf{OWrkP}[\underline{\mathsf{v}}/\mathbf{null}] \implies \mathsf{OSrvC} \wedge nonCr(\$)}{\{\mathsf{IWrk}[\underline{\mathsf{v}}/\mathbf{null}]\}\ [\mathtt{W}]\ \{\mathsf{OSrvC} \wedge nonCr(\$)\}}$$

The worker component triple must be adjusted, so it is ready to receive the input from the server actor. Substituting the variable $\underline{\mathsf{v}}$ 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 $\mathtt{mrg}(\mathtt{null}, \mathtt{cmp}(\underline{\mathsf{t}})) = \mathtt{cmp}(\underline{\mathsf{t}})$. Therefore, we can infer OSrvC from $\mathsf{OWrkP}[\underline{\mathsf{v}}/\mathtt{null}]$. 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.

$$\textsc{Cmp}\ \dfrac{\begin{array}{c}\{\mathsf{ISrv} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{S}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OSrv} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\}\\ \{\mathsf{IWrk}[\underline{\mathsf{v}}/\mathtt{null}]\}\ [\mathtt{W}]\ \{\mathsf{OWrkP}[\underline{\mathsf{v}}/\mathtt{null}] \wedge \mathit{nonCr}(\$)\}\\ \mathit{match}(\exists \underline{\mathsf{w}} \bullet \mathsf{OSrv}, \mathsf{IWrk}[\underline{\mathsf{v}}/\mathtt{null}], \mathtt{W})\end{array}}{\{\mathsf{ISrvC}\}\ [\mathtt{S}]\ \{\mathsf{OSrvC}\}}$$

When a server actor receives a single request, we can match the output of the server actor to the input of the worker component. The last premise of BOXEDCOMPOSITION is handled, entailing 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 $\underline{\mathsf{z}}$, a variable that does not appear in other parts of the corresponding triple. The *match* predicate enforces this behavior. As in BOXEDCOMPOSITION, *noSelfExp* ensures that no self exposure is done. When these premises are fulfilled, 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 m as $\mathsf{sz}(\mathit{gt}(\$)) - 1$. 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 $\underline{\mathsf{t}}$ (i.e., $\mathsf{sz}(\underline{\mathsf{t}}) = 1$). In this particular case the goal is to box the worker actor for creating no other actors. BOXING captures exactly this intention.

$$\textsc{Box}\ \dfrac{\textsc{Cns}\ \dfrac{\textsc{Axiom}\ \dfrac{}{\{\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\$)) = 1\}\ \mathtt{W}\ \{\mathsf{OWrkP}\}}\quad \begin{array}{c}\mathsf{OWrkP} \implies \mathsf{OWrkP} \wedge \mathit{nonCr}(\$)\end{array}}{\{\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\$)) = 1\}\ \mathtt{W}\ \{\mathsf{OWrkP} \wedge \mathit{nonCr}(\$)\}}}{\{\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\$)) = 1\}\ [\mathtt{W}]\ \{\mathsf{OWrkP}\}}$$

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 $\mathsf{sz}(\mathtt{rst}(\underline{\mathsf{t}})) < \mathsf{sz}(\underline{\mathsf{t}})$, but we cannot extract the right task $\underline{\mathsf{t}}$ 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 $\mathsf{sz}(\mathit{gt}(\mathit{cse}(\mathsf{IWrk}))) = \underline{\mathsf{z}}$ 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 $\mathsf{sz}(\mathit{gt}(\$)) = \underline{\mathsf{z}} \wedge \mathsf{sz}(\mathit{gt}(\$)) > 1$ as *ind*.

$$\textsc{Cns}\ \dfrac{\textsc{Inv}\ \dfrac{\textsc{Cns}\ \dfrac{\textsc{Inv}\ \dfrac{\textsc{Axiom}\ \dfrac{}{\{\mathsf{IWrk} \wedge \mathit{ind}\}\ \mathtt{W}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI}\}}}{\{\mathsf{IWrk} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk})\}\ \mathtt{W}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk})\}} \quad \begin{array}{c}\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$ \implies\\ \mathsf{IWrk} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk})\\ \exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk}) \implies\\ \exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk}) \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\end{array}}{\begin{array}{c}\{\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{W}\\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk}) \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\}\end{array}}}{\begin{array}{c}\{\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{W}\\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk}) \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$) \wedge \mathsf{sz}(\mathit{gt}(\underline{\mathsf{i}})) = \underline{\mathsf{z}}\}\end{array}} \quad \begin{array}{c}\mathsf{IWrk} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$ \implies \mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$\\ \exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{IWrk}) \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$) \wedge \mathsf{sz}(\mathit{gt}(\underline{\mathsf{i}})) = \underline{\mathsf{z}}\\ \implies \exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \mathsf{sz}(\mathit{gt}(\$)) < \underline{\mathsf{z}} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\end{array}}{\begin{array}{c}\{\mathsf{IWrk} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{W}\\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \mathsf{sz}(\mathit{gt}(\$)) < \underline{\mathsf{z}} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\}\end{array}}$$

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

$$\textsc{Ind}\ \dfrac{\begin{array}{c}\{\mathsf{IWrk} \wedge \mathsf{sz}(\mathit{gt}(\$)) = 1\}\ [\mathtt{W}]\ \{\mathsf{OWrkP}\}\\ \{\mathsf{IWrk} \wedge \mathit{ind} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{W}\\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI} \wedge \mathsf{sz}(\mathit{gt}(\$)) < \underline{\mathsf{z}} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\}\\ \mathit{match}(\exists \underline{\mathsf{w}} \bullet \mathsf{OWrkI}, \mathsf{IWrk}, \mathtt{W})\end{array}}{\{\mathsf{IWrk}\}\ [\mathtt{W}]\ \{\mathsf{OWrkP}\}}$$

As the worker component specification is proved, the server component specification holds. These proofs depend on the *RPSA* being sound, subject of the next subsection.

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* $\vDash \{p_1\}\ D\ \{q_1\}$*,* $p \implies p_1$ *and* $q_1 \implies q$*. Then* $\vDash \{p\}\ D\ \{q\}$*.*

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* $\vDash \{p\}\ C\ \{q \wedge nonCr(\$)\}$ *holds. Then,* $\vDash \{p\}\ [C]\ \{q\}$*.*

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**). *If* $\vDash \{p \wedge \underline{i} = \$\}\ C\ \{q \wedge noSelfExp(\underline{i}, \$)\}$*,* $\vDash \{q'\}\ D\ \{r \wedge nonCr(\$)\}$ *and* $match(q, q', D)$ *hold, then* $\vDash \{p\}\ [C]\ \{r\}$*.*

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**). *If* $\vDash \{p \wedge m = 0\}\ [C]\ \{q\}$*,* $\vDash \{p \wedge m = \underline{z} \wedge m > 0 \wedge \underline{i} = \$\}\ C\ \{p' \wedge m < \underline{z} \wedge noSelfExp(\underline{i}, \$)\}$ *and* $match(p', p, C)$ *hold, then* $\vDash \{p\}\ [C]\ \{q\}$*.*

The soundness of 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 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, 8, 15, 26, 28]). The trace semantics used in this paper is inspired by Vasconcelos and Tokoro's trace semantics [28] and Talcott's interaction path [26]. 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.

Specification. π-calculus [21] can be used to specify actor systems, but it needs some restrictions on the syntax and actor identities. Verifying a component specification from the actor class specifications involves bisimulation.

Specification Diagram [24] provides a very 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 π-calculus, a may testing ([12]) characterization of Specification Diagram can be obtained [27].

Our specification technique is strongly related to FOCUS [7]. However, FOCUS provides no message and dynamic naming support necessary for actor systems.

Logic. Several logics have been developed to reason about the functional behavior of actor-based implementations. Most of them (e.g., [9, 10, 14, 23]) are based on a state-based semantics and rely on having the actual implementations.

De Boer [11] presented a Hoare logic for concurrent processes that uses FIFO channels for communication (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 [25]. 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. [13] extended Soundarajan's work to deal with actor systems. They consider only finite prefix-closed traces, justifying their consideration of having only finite number of actors 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. 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 [5] 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*.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *JFP*, 7(1):1–72, 1997.

[3] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *SCP*, 77(12):1289–1309, 2012.

[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs, 3rd Ed.* Springer, 2009.

[5] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *WCFM*, pages 682–700, 1999.

[6] M. Broy. A logical basis for component-oriented software and systems engineering. *Comput. J.*, 53(10):1758–1782, 2010.

[7] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[8] W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT, Cambridge, MA, 1981.

[9] M. Dam, L.-Å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *COMPOS*, pages 150–185, 1997.

[10] J. Darlington and Y. Guo. Formalising actors in linear logic. In *OOIS*, pages 37–53, 1994.

[11] F. S. de Boer. A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. *TCS*, 274 (1–2):3–41, 2002.

[12] R. De Nicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984.

[13] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *JALP*, 81(3):227–256, 2012.

[14] C. H. C. Duarte. Proof-theoretic foundations for the design of actor systems. *MSCS*, 9(3):227–252, 1999.

[15] C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, Oct. 1969.

[17] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, Aug. 1978.

[18] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *TCS*, 365(1-2):23–66, 2006.

[19] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO 2010*, LNCS, pages 142–164. Springer, 2011.

[20] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.

[21] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–77, 1992.

[22] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Habil. thesis, TU Munich, Jan. 1997.

[23] S. Schacht. Formal reasoning about actor programs using temporal logic. In *LNCS*, pages 445–460. Springer, 2001.

[24] S. F. Smith and C. L. Talcott. Specification diagrams for actor systems. *HOSC*, 15(4):301–348, 2002.

[25] N. Soundarajan. Axiomatic semantics of communicating sequential processes. *ACM TOPLAS*, 6(4):647–662, Oct. 1984.

[26] C. L. Talcott. Composable semantic models for actor theories. *HOSC*, 11(3):281–343, 1998.

[27] P. Thati, C. L. Talcott, and G. Agha. Techniques for executing and reasoning about specification diagrams. In *AMAST*, pages 521–536, 2004.

[28] V. T. Vasconcelos and M. Tokoro. Traces semantics for actor systems. In *Object-Based Concurrent Computing*, LNCS, pages 141–162. Springer, 1991.