Verification of Open Concurrent Object Systems*

Ilham W. Kurnia and Arnd Poetzsch-Heffter

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

Abstract. We present an approach for the specification and verification of open concurrent object systems. A concurrent object system consists of a dynamically changing set of objects that work concurrently and interact with the environment of the system. An object system is called open if the environment of the system is not known. To proof that an open system satisfies a desired property P, we have to show that P holds for all possible environments. We describe a simple calculus for concurrent object systems together with an operational and a trace-based semantics. Then, we present a specification technique that supports looseness and restriction. Looseness allows specification refinement. Restriction allows expressing assumptions of the environment. Finally, we introduce a verification technique to prove that an open system S satisfies a specified property P. The technique supports hierarchical reasoning, i.e., the properties of S's components can be used in the proof of P.

1 Introduction

There is a clear trend that modern software becomes more concurrent and distributed. To add to this development, virtualization is more and more prominent as software is deployed in the cloud. The consequence is that developers should consider the factors such as architecture, processors, network latency and load that are unknown during the development, meaning they have to consider building open software systems.

In this chapter, we look into such systems built using concurrent objects [42], also known as actors [1] or active objects [23], which simplifies the concurrency and distribution model[1]. In particular, we investigate how to verify their functional behavior. Traces are used as the basis of specification and verification as they abstract from the internal representation of the actual implementation [18].

The verification approach follows a two-tier verification as shown in Fig. 1. The two tiers are used by, e.g., Misra and Chandy [28] and Widom et al. [41] to avoid the complex task of directly reasoning about system properties on the implementation level. In the first tier, the class implementation is verified against a specification of the class. This task is not considered in this chapter, but addressed by Din et al. [14] and Ahrendt and Dylla [3]. The second tier, which is the subject of this chapter, focuses on how to use class specifications to reason about system properties. The verification is

* This work is partially supported by the EU project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models*.

[1] In the following we refer to concurrent objects as simply objects.

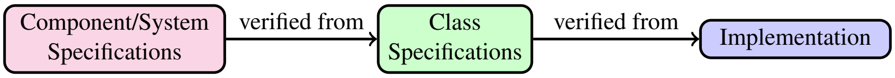

Fig. 1. Two-tier verification

done modularly such that component specifications that have been verified from class specifications can be used to verify specifications of other components.

This chapter follows on the previous work by the authors [22], where the specification and verification approach is presented in detail. The central contribution presented here is the connection between a core operational semantics of concurrent object systems and its trace semantics in the context of open systems. This allows a more refined component definition that is comparable to well-known component models, such as OSGi [39] and COM [25].

Chapter structure. In Sect. 2 we explain the running example used in this chapter. Then we present core operational semantics of concurrent object systems in Sect. 3 and how components and their trace semantics can be extracted from the operational semantics in Sect. 4. The specification and verification technique is presented in Sect. 5 with the help of the running example. Some discussion is provided at the end of each section.

2 Running Example

To have a sufficiently clear background for the following discussion on verification, we informally introduce a core concurrent object language ActJ together with an example for illustrating our approach. ActJ can be thought as a watered-down version of ABS [19], where it focuses on the actor concurrency layer as discussed by Hähnle in his tutorial chapter for ABS (Chap. ??). ActJ features a java-like syntax, interfaces, class-based programming, asynchonous method calls without returns, concurrent objects and typed references.

As a running example, we use a variant of the client-server setting treated in an industrial Erlang case study by Arts and Dam [4]. 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 client name must be passed around, that the last worker can return the task computation result to the client. This example illustrates unbounded object creation.

An implementation of the example in ActJ is provided in Fig. 2. The central class is `Server`, which can receive the call `serve(c, t)`. A `Server` object deals with such a request by processing the computation task `t` and sending back an appropriate response to the client. To enable concurrent execution of tasks, the server delegates the task to

```
interface Client { response(Value); }

class Server {
  serve(Client c, CompTask t) {
    // taskSize(t) ≥ 1
    Worker w = new Worker;
    w.do(t);
    w.propagate(null, c);
  }
}

class Worker {
  Value myResult = null;
  Worker nextWorker = null;

  do(CompTask t) {
    if (taskSize(t) > 1) {
      nextWorker = new Worker;
      nextWorker.do(restTask(t));
    } else {
      nextWorker = null;
    }
    myResult = compute(firstTask(t));
  }
  propagate(Value v, Client c)
    guard myResult != null {
    if (nextWorker == null) {
      c.response(merge(myResult, v));
    } else {
      nextWorker.propagate(
              merge(myResult, v), c);
    }
  }
}
```

Fig. 2. Server-worker implementation in ActJ

a dynamically created 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 `propagate` calls, initiated by the server, the results of the different chunks are collected, merged, and the final result is sent back to the client. We assume that there are implicit interfaces, each for `Server` and `Worker`, where the available methods are listed.

There are two ways objects can interact with each other: creating new objects and calling methods of other objects. Object creation is represented in ActJ by the execution of a statement of the form `new C`. For example, a worker object is created when the server receives a request. Object creation is non-blocking. Though not present in the example, a class description may be enriched with a parameterless `run` method which is executed when an instance of that class is created. Thus an object can behave actively, not just reacting to method calls. The object creation `new C` is regarded as a message.

A method call is produced when a statement of the form $\texttt{r.m}(\overline{\texttt{p}})$ is executed. This statement sends the message $\texttt{m}(\overline{\texttt{p}})$ to the receiver object `r` where `m` is the method name with a list of parameters $\overline{\texttt{p}}$. The parameters can be data values or object identities. Such a method call is non-blocking; execution directly continues with the next statement. Thus, in general, a method call leads to concurrent behavior. For each of the calls it receives, an object has a *body* that describes how it reacts to a method call. For example, an instance of class `Server` reacts to a message `serve(c, t)` as follows: It creates a worker object, sends first a `do` and then a `propagate` call to the worker. We assume that the execution of method bodies always terminate.

It remains to explain what happens when a method call is received. We assume that objects, similar to actors [1], have an unbounded input queue and are input-enabled (cf. [24, p. 257]); i.e., objects can always accept new input. Method calls 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 object has control over the execution of

```
interface Server {
  serve(Client c, CompTask t); }

class Client {
  Server s = null;
```

```
  link(Server server) {
    s = server;
    s.serve(randomTask(), this);
  }
  response(Value v) {}
}
```

Fig. 3. A sample `Client` in ActJ

incoming method calls. Method call selection is (weakly) fair for method calls with guards, meaning that a message whose guard is infinitely often evaluated to `true` will eventually be picked for processing. In Fig. 2, the class `Worker` uses a guard to select a `propagate` call only if the computation result is available.

As well as the concurrency-related aspects, ActJ supports recursive data types and function definitions for handling data (as in functional programming languages). In the 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 task `t`; `taskSize(t)` gives 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. The properties below are also assumed.

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

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`.

One possible way to use the server is by constructing a client whose implementation can be seen in Fig. 3. By using the implicit interfaces of `Server`, the client sends some random task to a server when the client is linked to the server. When it receives a response, it does nothing. When a client and a server are created and the client is linked to the server, we expect that a client makes a request to the server, then the server creates a number of worker to handle the task, and the last worker returns the task computation result back to the client.

3 Core Operational Semantics

To characterize the behavior of a concurrent system, we look into the behavior of its elements, i.e., the objects, and how they interact with each other. Communication is done through asynchronous message passing and each object has its own single thread of computation and a mailbox to capture the incoming messages. We allow an object to multitask (while keeping the single-threadedness). The concept of method return is seen as yet another message send (i.e., futures [6] are not featured). An important feature that we take from the concurrent object and actor models is that objects do not share states. Therefore, if an object needs any information from other objects, or wants to manipulate other objects, this must be done by sending the other objects messages.

Following the design from the modeling languages Creol [20] and ABS [19], the behavior of objects are characterized by classes. This is different to the less structured approach of actors, where each actor changes its behavior after processing a message (see, e.g., [40,2]). This restriction allows us to separately define the behavior of objects without having to provide a single relation that works for all objects. In the following, we use a slightly less faithful model in terms of the communication between objects to capture the behavior of classes. This simplified model is expressive enough for our purpose of showing the connection between operational semantics of concurrent object systems and their trace semantics.

3.1 Classes

We start the description of our formal model by defining the basic sets. Let **O** be the set of all object identities, **M** the set of messages that can be communicated between objects and **D**, disjoint from **O**, the set of data values. We use object identities to represent both the object and its actual identity. The function $class(o)$ gives the class of an object o. A message m can either be an object 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 object identity. The predicate $isMtd(m)$ checks whether the message m is a method call.

The set of *events* **E** is built on the messages. An event $e \in \mathbf{E}$ represents the occurrence of a message $m = msg(e)$ being sent by the *caller* object $o = caller(e)$ to the *callee* object $o' = callee(e)$. If m is a creation message, o' will be the name of the newly created object while o is its creator. Textually an event e is represented as $o \to o'.$**new** C or $o \to o'.mtd(\overline{p})$ when the message is an object creation or a method call, respectively. The inclusion of the caller information allows us to distinguish between input and output events with respect to a single object or a group of objects. To collect the (finite number of) object identities occurring in the parameter list of a method call, we define a function $acq(mtd(\overline{p}))$, short for *acquaintance*.

To describe the behavior of a specific object, only on the callee and message information of an event are needed. Eliminating the caller from an event produces an *event content*. The set of event contents **EC** are derived from the set of events **E** in this manner. We use $\mathsf{e} \in \mathbf{EC}$ (with serif font) to represent its typical element.

The class of an object characterizes how the object behaves. In this chapter, we use transition relations to describe how an object of a certain class can act independently and react to incoming messages.

Definition 1 (Class). *A class* $C \in \mathbf{CL}$ *is a parameterized tuple* $\langle Q_C, q_0^C, \rho_C, \alpha_C, \kappa_C\rangle(\mathtt{this})$ *where*

- Q_C *is the set of states,*
- q_0^C *is the initial state,*
- $\rho_C : \mathbf{M} \times Q \times Q$ *is the message receive relation,*
- $\alpha_C : Q \times \mathbf{EC} \times Q$ *is the action relation,*
- $\kappa_C : Q \to 2^{\mathbf{O}}$ *is the function mapping a state to a set of known objects, and*
- $\mathtt{this}$ *is an object identity given when the object is created,*

such that for each state $q, q' \in Q_C$*, message* $m \in M$ *and event content* $\mathsf{e} \in \mathbf{EC}$*,*

- $\langle m, q, q'\rangle \in \rho_C \implies \mathit{isMtd}(m)$ *(creation of objects is handled separately);*
- $\{\mathtt{this}\} \in \kappa_C(q)$ *(the object always knows its own identity);*
- $\langle m, q, q'\rangle \in \rho_C \implies \kappa_C(q') \subseteq \kappa_C(q) \cup \mathit{acq}(m)$
  (the object knows another object through incoming method calls, or)
- $\langle q, \mathsf{e}, q'\rangle \in \alpha_C \wedge \neg \mathit{isMtd}(\mathit{msg}(\mathsf{e})) \implies \kappa_C(q') \subseteq \kappa_C(q) \cup \mathit{callee}(\mathsf{e})$
  (the object knows another object by creating it);
- $\langle q, \mathsf{e}, q'\rangle \in \alpha_C \wedge \mathit{isMtd}(\mathit{msg}(\mathsf{e})) \implies \{\mathit{callee}(\mathsf{e})\} \cup \mathit{acq}(\mathsf{e}) \subseteq \kappa_C(q)$
  (the callee and parameters of method calls made by the object must be known).

A class C is described by a parameterized quintuple, representing the set of states Q_C an object of that class can have, the initial state an object has when it is created, the transition relations ρ_C and α_C, and a function κ_C describing which other objects an object knows when it is at some state. The single parameter is some object identity represented by the variable $\mathtt{this}$. This variable is assigned value o when an object o of class C is created. In general, Q_C can be an infinite set because an object may have fields that store object references. Furthermore, the object may have an internal mailbox to manage how incoming messages are processed. The transition relation ρ_C states when an object receives a message and the effect of receiving that message on the object's state. ρ_C is given such that any object of this class can only receive messages of certain forms, i.e., the class obeys a certain interface. The transition relation α_C describes what an object does when it reaches a certain state, i.e., it can create another object or send a message to another object. We assume that no self calls occur. Changes that occur with a self call can be simulated by allowing non-deterministic choices of states when the object receives or sends a message.

For a class tuple to follow the real operational semantics of an object-based language such as ActJ, we translate well-known invariants into restrictions that explain the relationship between ρ_C, α_C and κ_C. First, the messages dealt by ρ_C are method calls. The rest of the restrictions tell us how κ_C evolves and is used. The function κ_C can only admit object identities it knows from previous interactions and its own identity, represented by the keyword $\mathtt{this}$. Using this knowledge, we restrict the method calls produced by the object such that both the callee and the parameters of the method calls are known to the object.

Instead of stating $\langle m, q, q'\rangle \in \rho_C$ or $\langle q, \mathsf{e}, q'\rangle \in \alpha_C$, we write $C : (m, q) \to q'$ and $C : q \xrightarrow{\mathsf{e}} q'$, respectively. The function $\mathit{extMtd}(C)$ extracts the method calls that can be received by the class C. In other words, it extracts the implicit interface supported by

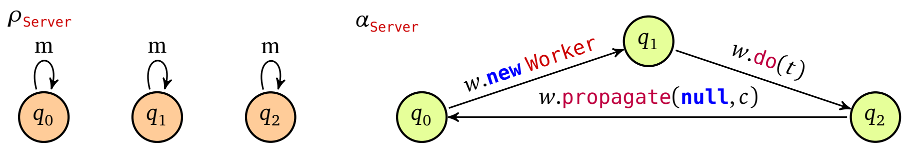

Fig. 4. Graphical representation of the message receive and action relations for Server class where $m = \mathtt{serve}(c,t)$

C. We assume that classes are defined such that their instances are input-enabled. This assumption can be easily defined using *extMtd*.

Example 1. The Server class ican be represented as follows. Each state in $Q_{\mathtt{Server}}$ is partitioned into five parts: internal state $\in \{q_0, q_1, q_2\}$, the client identity c, the worker identity w, the task t and the internal message queue u. We use the sequence data structure $Seq\langle T\rangle$ to represent the message queue, with T denoting the type of the sequence elements (where $T = \mathbf{M}$ in this case). 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. By virtue of infinite number of object identities and the internal message queue, $Q_{\mathtt{Server}}$ is an infinite set.

The initial state is $\langle q_0, \mathbf{null}, \mathbf{null}, \mathbf{null}, []\rangle$. The message receive relation $\rho_{\mathtt{Server}}$ is represented symbolically by $\mathtt{Server} : (\mathtt{serve}(c,t), \langle q_0, \_, \_, \_, u\rangle) \to \langle q_1, c, \_, t, u \cdot \mathtt{serve}(c,t)\rangle$. An underscore represents any value.

The action relation $\alpha_{\mathtt{Server}}$ is represented symbolically as follows.

– $\mathtt{Server} : \langle q_0, \_, \_, \_, \mathtt{serve}(c,t) \cdot u\rangle \xrightarrow{w.\mathbf{new}\ \mathtt{Worker}} \langle q_1, c, w, t, u\rangle$
– $\mathtt{Server} : \langle q_1, c, w, t, u\rangle \xrightarrow{w.\mathtt{do}(t)} \langle q_2, c, w, t, u\rangle$
– $\mathtt{Server} : \langle q_2, c, w, t, u\rangle \xrightarrow{w.\mathtt{propagate}(\mathbf{null}, c)} \langle q_0, c, w, t, u\rangle$

The known object function $\kappa_{\mathtt{Server}}$ is represented symbolically as follows.

– $\kappa(\langle q_0, \_, \_, \_, \_\rangle) = \{\mathbf{this}\}$
– $\kappa(\langle q_1, c, w, \_, \_\rangle) = \{\mathbf{this}, c, w\}$
– $\kappa(\langle q_2, c, w, \_, \_\rangle) = \{\mathbf{this}, c, w\}$

Graphically, $\rho_{\mathtt{Server}}$ and $\alpha_{\mathtt{Server}}$ can be viewed as in Fig. 4, by focusing only on the internal states.

Discussion. The class definition above does not constrain an object to behave strictly as a pure concurrent object or a pure actor with multitasking capability. In particular, it does not enforce that the actions an object takes correspond to processing a single message. This means that the definition does not explicitly restrict the object to have a single-threaded computation, except that there cannot be two messages being sent out in parallel due to the interleaving nature of the transition relation. In this chapter, we consider this issue to be dealt on a lower level of abstraction, and assume that the action relation α_C follows an actor model with multitasking, such as ABS [19].

Despite the input-enabledness assumption, message dropping can be represented. Moreover, the class definition allows (internal) nondeterminism to occur already within

ObjectCreation

$$\frac{C : q \xrightarrow{\mathsf{e}} q' \qquad m = o'.\texttt{new}\ C' \qquad o'\ \mathit{fresh}}{C\ o : q, \mathcal{C} \xrightarrow{o \to o'.\texttt{new}\ C'} C\ o : q', C\ o' : q_0^{C'}, \mathcal{C}}$$

MessageSend

$$\frac{C : q \xrightarrow{\mathsf{e}} q' \qquad \mathsf{e} = o'.mtd(\overline{p}) \qquad \{o'\} \cup acq(mtd(\overline{p})) \subseteq \kappa_C(q) \qquad C' : (mtd(\overline{p}), q_1) \to q_1'}{C\ o : q, C'\ o' : q_1, \mathcal{C} \xrightarrow{o \to o'.mtd(\overline{p})} C\ o : q', C'\ o' : q_1', \mathcal{C}}$$

Fig. 5. Core operational semantics of concurrent object systems

a single object. The main reason is that the structure of the states are not constrained; in particular, the state of an object may not have any message queue (or bag [2]) at all. For the purpose of our example, we would use only class definitions where message queues are part of the states without message dropping.

3.2 Operational Semantics

To describe the interaction between a number of objects, we need run-time configurations that capture the current state of those objects. The current state of each object determines how objects act and react to incoming messages.

Definition 2 (Run-time configuration). *A* run-time configuration *of an object o is a tuple $\langle C, o, q\rangle$ where C is the class of o and $q \in Q_C$ is the run-time state of the object. The configuration of a* group *of object $O \subseteq \mathbf{O}$ is a set of configuration $\mathcal{C}$ where for each object $o \in O$ there is at most one configuration of o in $\mathcal{C}$.*

A run-time configuration of an object o consists of its class C, its identity o and its current state q, with respect to the state description of its class. The identity of the object is used to replace `this` occurring in any part of the class tuple (i.e., o acts as the parameter in the class definition). For the remainder of the chapter, we use the notation $C\ o : q$ to represent a run-time configuration of o.

Example 2. $\langle \texttt{Server}, s, \langle q_0, \texttt{null}, \texttt{null}, \texttt{null}, \texttt{serve}(c_1, t_1) \cdot \texttt{serve}(c_2, t_2)\rangle\rangle$ is a possible configuration for a `Server` object s. This states that s has two `serve` incoming method calls and currently in the initial state to process the next incoming request.

The way the run-time configuration is affected by interaction between objects are shown in Fig. 5, representing the core operational semantics of concurrent object systems. The semantics consists of two rewrite rules that modify the configurations: ObjectCreation and MessageSend. ObjectCreation states that when an object is in a state where, according to its class action relation, it creates a new object, a new object with fresh identity is created with the default configuration. The configuration transition is labeled with the corresponding creation event. MessageSend states that when an object is in a state where according to its class action relation it calls a method on another known object with known acquaintances and it causes the other object when receiving the message to change to another state, then both object changes their state accordingly. The configuration transition is labeled with the corresponding method call event. When there are multiple ways to apply the rewrite rules, one is chosen at random.

We assume that any computation that does not involve calling methods of another object or creating a new object is done in one transition. To model divergence, the class transition relations can go to some final state without any outgoing transition. A straightforward consequence of the operational semantics is that an object always keeps its class designation as shown in the lemma below.

Lemma 1 (Class preservation). *An object never changes its class.*

Proof. Follows from Def. 2 and the operational semantics rules.

The class-based approach is different to the actor model [1], where an actor's behavior is not structured by means of some class. Instead the actor's behavior changes whenever it receives a message or finishes sending out messages in reply to a message. Being able to refer to specific entities with regards to the behavior of objects is useful to later on structure the behavior of a group of objects without needing extra constructs. Therefore, class preservation of objects serves as an important basis to this usage.

Discussion. The operational semantics above only has two rules. They cover the necessary observable operations on a concurrent object model (cf. [37]). The internal computation needed to produce the messages is abstracted away in our model by means of object states. Vasconcelos and Tokoro [40] even reduced the number of kinds of observable operations into one, by just considering the method calls (which in their work is simply called *messages*). Object creation is considered similar to internal computation such that in the single transition rule, the number of objects that have been created implicitly changes. In our case, the object creation needs to be considered separately because of the class information.

The MESSAGESEND rule pairs the action relation of one class to the message receive relation of another class. As a consequence the method call orders between pairs of objects are always the same, which is different to the actor model which retains pure asynchronicity [10,1,2]. Yonezawa, Briot and Shibayama [42] argued, however, that having this guarantee eases describing distributed algorithms.

As each object can concurrently do their internal process at possibly different speed, there is a need for external nondeterminism. This need is supplied by the random application of operational rules. This choice raises the question about the fairness of choosing the objects to which the rules are applied. Because this issue is not prominent for our discussion on specification and verification (unlike the discussion on actors itself [2], for example), we simply assume weak fairness on the operational rule application.

3.3 Translation to Traces

Given a configuration, we can define an execution from this configuration as a sequence of interleaved configurations and configuration transition, where each transition represents an application of the operational semantics rules. We can extract from an execution the trace by taking the label of configuration transitions. If we know the default initial configuration, the traces can be used to abstract from the internal representation [18,9]. This is exactly why we will use them later on as semantic foundation for specifications.

Definition 3 (Execution and trace). *An* execution *is a sequence of interleaved configurations and (possibly empty) events $C_0 \xrightarrow{e_1} C_1 \xrightarrow{e_2} C_2 \cdot \ldots$ which may end with $\surd$ after a configuration. A* trace $t \in Seq\langle \mathbf{E} \cup \{\surd\}\rangle$ *of an execution is the projection of the execution to the events by leaving out the configurations. t ends with $\surd$ if the corresponding execution ends with $\surd$.*

The $\surd$ symbol indicates that an execution has finished (i.e., no more operational rule can be applied). In other words, the execution is *maximal* (cf. [5, p. 96]).

Example 3. Assume that a `Client` object c and a `Server` object s has been created and the message `link`(s) is already sent to c. Using an underscore to represent an unimportant configuration content, the following execution may happen.

$$\texttt{Client}\ c : \_, \texttt{Server}\ s : \langle q_0, \_, \_, \_, [\,]\rangle \xrightarrow{c \to s.\texttt{serve}(t,c)}$$
$$\texttt{Client}\ c : \_, \texttt{Server}\ s : \langle q_0, \_, \_, \_, \texttt{serve}(t,c)\rangle \xrightarrow{s \to w.\texttt{new Worker}}$$
$$\texttt{Client}\ c : \_, \texttt{Server}\ s : \langle q_1, c, w, t, [\,]\rangle, \texttt{Worker}\ w : \_ \xrightarrow{s \to w.\texttt{do}(t)}$$
$$\texttt{Client}\ c : \_, \texttt{Server}\ s : \langle q_2, c, w, t, [\,]\rangle, \texttt{Worker}\ w : \_ \xrightarrow{s \to w.\texttt{propagate}(\texttt{null},c)}$$
$$\texttt{Client}\ c : \_, \texttt{Server}\ s : \langle q_0, \_, \_, \_, [\,]\rangle, \texttt{Worker}\ w : \_$$

From the execution above, we can extract the following trace.

$$c \to s.\texttt{serve}(c,t) \cdot s \to w.\texttt{new Worker} \cdot s \to w.\texttt{do}(t) \cdot s \to w.\texttt{propagate}(\texttt{null}, c)$$

To obtain specific information related to a certain object o from a trace t, we use the projection operator, denoted $t{\downarrow_o}$, stating the projection of t to o where all events where o is neither caller nor callee are removed from t. When necessary, the object parameter can be enriched with *callee* or *caller* to denote that we are focusing on the events where o is the callee or the caller, respectively. This operator is lifted to a set of traces T and a set of objects O in a natural way. Other operators are introduced as needed.

The restriction on the class definition (Def. 1) can be transferred to traces by requiring that an object can only send messages to other objects it has been exposed to. A trace should contain enough information to determine whether an object is exposed to another object. This information are extracted using the functions *acq* and *cr*. The function $cr(b.\texttt{new}\ C)$, short for *created*, extracts the identity of the newly created object from an object creation (i.e., the callee b). These functions are lifted to events and traces.

Definition 4 (Well-formed trace). *Let t be a trace and e a place holder for a method call event $o \to o'.mtd(\overline{p})$. A trace t is* well-formed *if*

$$\forall o \in \mathbf{O}, t' \cdot e \in Pref(t) \bullet \{o'\} \cup acq(e) \subseteq acq(t{\downarrow_{o,callee}}) \cup cr(t{\downarrow_{o,caller}}) \,.$$

The definition above states that for every method call an object makes, it must know the identity of the object it is calling and also the identities of each object present as parameters of the method call. Because we assume there is no self-call, we obtain the following corollary.

Corollary 1. *A non-empty well-formed trace t begins with a creation event.*

The following lemma shows that when we start from a configuration containing an object in its initial state, using the core operational rules and the restrictions given on the class definition, the generated trace(s) is well-formed.

Lemma 2 (Well-formedness preservation). *Let* **CL** *be a set of classes and a singleton* $\mathcal{C} = C\ o : q_0^C$ *is some configuration. Then, by applying the operational rules from Fig. 5, the generated trace is well-formed.*

Proof (sketch). What necessary for the proof is that the functions *acq* and *cr* are monotonic and never introduce alien object identities that do not appear in the events. As this is weaker than the restriction of κ in Def. 1 and the restriction given on MESSAGESEND is the same as in Def. 4, the generated trace is well-formed.

4 Systems and Components

Having an operational semantics for a group of objects is the basis to understand how a system behaves. One natural way to structure the grouping is to use classes to form components. Using the component notion, the system's behavior can be understood by composing the behavior of the components. In this section, we explore the possible ways to structure the group into components and link the components with the notion of open systems.

4.1 Closed Systems

First, we need to know whether we have information to apply the rules of the operational semantics. This means we need to have the necessary class definitions.

Definition 5 (Definition-complete). *Let* $\mathbf{C} \subseteq \mathbf{CL}$ *be a set of classes.* $\mathbf{C}$ *is* definition-complete *if for each method call* $o.mtd(\overline{p})$ *such that* $class(o) = C'$ *and object creation* `new` C' *appearing in* α_C *of any class* $C \in \mathbf{C}$, $C' \in \mathbf{C}$.

A set of classes is definition-complete if for each object present within the interaction, the class definition of that object is available also within the set.

Definition 6 (Closed system). *A* closed system $CS = (\mathbf{C}, C_0)$ *is a definition-complete set of classes* $\mathbf{C}$ *with a distinguished* activator class $C_0 \in \mathbf{C}$.

A closed system is described by having all class definitions needed to know precisely how each object within the system behaves. An activator class points the class of the initial object of the system (similar to starting a Java program, for example, with some initial class containing a `main` method). This initial object should then create other objects necessary for the system to run.

Definition 7 (Closed system trace semantics). *The trace semantics of a closed system* $CS = (\mathbf{C}, C_0)$ *is the prefix-closed trace set* $Traces(CS)$ *where for each* $t \in Traces(CS)$, *there is a matching execution starting from the initial configuration*
$\mathcal{C} = C_0\ o : q_0^{C_0}$.

The trace semantics of a closed system is a trace set which collects all traces that can be made from an initial configuration whose member is the initial configuration of the object of activator class. Using the closed system definition, we can define the trace semantics of a class.

Definition 8 (Class trace semantics). *The trace semantics of a class C is a set of traces $Traces(C)$ where for each trace $t \in Traces(C)$, there is a closed system $CS = (\mathbf{C}, C_0)$ such that $C \in \mathbf{C}$ and there is a trace $t' \in Traces(CS)$ such that $t = t'\downarrow_o$ where $class(o) = C$.*

The trace semantics of a class C is obtained by taking all traces of all closed system that contain C, then projecting all those traces down to objects of class C. We denote this trace semantics as $[\![C]\!]$.

Example 4. A trace of a `Server` class is $c \to s.\texttt{serve}(c,t) \cdot s \to w.\textbf{new}\ \texttt{Worker} \cdot s \to w.\texttt{do}(t) \cdot s \to w.\texttt{propagate}(\textbf{null}, c)$. This is obtained by taking the projection of trace from Ex. 3 to the `Server` object s.

Discussion. The trace semantics can as well directly be defined based on the class definition. However, in a concurrent nondeterministic setting, it is not easy to do. In particular all possible message reception and sending has to be taken into account.

4.2 Open Systems and Components

A closed system deals with a group of objects that are completely executable without any influence from outside, while single classes only deal with the behavior of each of those objects. This shows a gap between single classes and closed systems because of the lack of possibility to know the behavior only for a subgroup of objects of that closed system. To help reason about the behavior of closed systems from the single classes, we create an abstraction that is called components. Components should share the characteristics of closed systems and single classes. Furthermore, they should also allow hiding behavior within the components. In this chapter, we choose to do the abstraction based on object creation.

Definition 9 (Creation-complete). *Let $\mathbf{C} \subseteq \mathbf{CL}$ be a set of classes. $\mathbf{C}$ is* creation-complete *if for each object creation* **new** *D appearing in α_C of any class $C \in \mathbf{C}$, $D \in \mathbf{C}$.*

A set of classes is creation-complete if for each created object, its class is within that set. This means that we may not know the behavior of some objects whose references are passed on within the interaction and, therefore, it fits with the goal of abstracting from the closed systems.

Definition 10 (Component). *A* Component *$\mathbb{C} = (\mathbf{C}, C_0)$ is a creation-complete set of classes $\mathbf{C} \subseteq \mathbf{CL}$ with some activator class $C_0 \in \mathbf{C}$.*

A component is essentially a system that starts with an object of some activator class. In comparison to a closed system, a component may have some parts *open* to be matched with some context. Usually, a set only consisting of a single class is not a component because it may create an object of a different class.

Example 5. The pair (\{`Server`\}, `Server`) is not a component because a server may create a worker, and the class `Worker` is not within the set of classes. The pair (\{`Worker`\},

`Worker`) on the other hand is a component because the only objects a worker may create are of the same class. In addition, we can combine the `Worker` component with the `Server` class to create a new component: ({`Server`, `Worker`}, `Server`). Note that none of these components are closed systems because they are missing the `Client` class.

As seen from the example above, in order to know the behavior of a component, we need to declare the context how the component is used. With this context, we have complete information the interaction between all objects within the component (and also the context).

Definition 11 (Context). *A context of a component* $\mathbf{C}, C_0$ *is* $\mathbb{X} = (\mathbf{C}^x, C_0^x)$ *such that* $\mathbf{C}^x \cup \mathbf{C}$ *is declaration-complete and* $C_0^x \in \mathbf{C}^x$.

A context is a set of classes with some activator class that completes the class definition of a component. A context need not be definition- or creation-complete on its own. However, paired with a matching component, we get a closed system.

Example 6. The pair ({`Client`}, `Client`) for some `Client` class whose instances send a request to a server is a context of the component ({`Server`, `Worker`}, `Server`).

Similar to the closed system and the class trace semantics, the behavior of a component can be defined by taking all traces produced by the combination of all possible contexts with the component. The main issue is to which objects should the resulting traces be projected. This problem is handled by the following function that keeps track of objects transitively created by the initial object of the component.

Definition 12 (Identity extraction). *Let* t *be a trace and* C *is some (activator) class. The* object identity extractor *function* $idx(t, C)$ *is defined as follows.* $idx([], C) = \emptyset$,

$$idx(t \cdot e, C) = \begin{cases} \{callee(e)\}\ ,\ \text{if } msg(e) = \textbf{new}\ C \wedge idx(t, C) = \emptyset \\ idx(t, C) \cup \{callee(e)\}\ ,\ \text{if } msg(e) = \textbf{new}\ C' \wedge caller(e) \in idx(t, C) \\ idx(t, C)\ ,\ \text{otherwise} \end{cases}$$

The function *idx* allows the gathering of objects directly and indirectly created by the initial object of the component. It works by waiting until the initial object of the component is created and then collecting the identities of the created objects afterwards. It is possible that more than one instance of the activator class object is created by some object outside the result of the function. As we are interested only on identifying the behavior of a single instance of the component, the function result is what we needed to obtain the plain trace semantics of the component. *idx* is lifted to a set of traces T by taking the union of the application of *idx* to each trace in T.

Definition 13 (Component plain trace semantics). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component. The* plain trace semantics *of* $\mathbb{C}$ *is a set of traces* $Traces(\mathbb{C})$ *where for each trace* $t \in Traces(\mathbb{C})$*, there is a context* $\mathbb{X} = (\mathbf{C}^x, C_0^x)$ *of* $\mathbb{C}$ *such that in the resulting closed system* $CS = (\mathbf{C}^x \cup \mathbf{C}, C_0^x)$*, there is a trace in* $t' \in Traces(CS)$ *and* $t = t'{\downarrow}_{idx(t', C_0)}$.

We call the definition above as the *plain* trace semantics of a component. The term plain refers to the lack of hiding performed on the internal interaction between objects within the component. Implicit within the definition is the usage of object creation tree to define the instances of components. Discussion on other ways to define the component instances is deferred until the end of this section. The closed system used to obtain the traces is just one way to compose the component with the context. The activator class of the resulting closed system is taken from the context because it is the context which decides when the component is instantiated.

Using this trace semantics, we have enough information to link the operational semantics to an openness property of the trace semantics necessary for proving the soundness of the proof system presented later. The openness property shows that once an object of the component is exposed to the context (i.e., the component's environment), the context can do anything with it, in particular making all possible method calls (with respect to all other exposed objects). We adopt the notation from our previous work [22], where the objects of the component instance are grouped into L (for *l*ocal) and the objects of the context are grouped into F (for *f*oreign).

Property 1 (Open system trace property [22]). Let T be a set of traces of a group of objects L, $F = \mathbf{O} - L$ and $e = o \rightarrow o'.mtd(\overline{p})$ an event such that $o \in F$, $o' \in L$, and $mtd(\overline{p}) \in extMtd(class(o'))$. T is *open* if

$$\forall t \in T, e \in \mathbf{E} \bullet \{o'\} \cup acq(e) \subseteq exposed(t,F) \cup F \implies t \cdot e \in T$$

where $exposed(t,F) = cr(t\downarrow_{F,caller}) \cup acq(t\downarrow_{F,callee})$.

To define the openness property, we assume that we can already distinguish between the objects of the instances and the object of the context. This distinction is straightforward to achieve through hierarchical naming structure of the identities of the created object. To instantiate this property in our trace semantics without loss of generality we assume the same identity of initial object of the component, and take the collection of all objects created by that initial object in all traces. The following lemma shows the connection between the core operational semantics presented in this chapter with the trace semantics, with respect to the openness property.

Lemma 3 (Component trace semantics openness). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component. Given* $L = idx(Traces(\mathbb{C}), C_0)$*,* $Traces(\mathbb{C})$ *is open.*

Proof. Let $t \in Traces(\mathbb{C})$ be achieved by composing $\mathbb{C}$ with some context $\mathbb{X} = (\mathbf{C}^x, C_0^x)$. Let also $o' \in exposed(t,F) \cap L$. Without loss of generality, we assume that

- $\mathbf{C}^x \cap \mathbf{C} = \emptyset$, and
- C_0^x is such that all necessary objects of the context are created before the activator class of the component is created.

We create another context $\mathbb{X}' = (\mathbf{C}^{x\prime}, C_0^{x\prime})$ that is the same as $\mathbb{X}$, except that for every class $C \in \mathbf{C}^x$ we extend it to C' where we pick some new method name mtd that do not appear in any class and add as many states as necessary such that

1. for all $o \in \kappa_{C'}(q), o'' \in F, C' : q \xrightarrow{o''.mtd(o)} q'$
   (the knowledge is shared among all objects of the context),
2. $C' : (mtd(o), q) \rightarrow q'$ *(all objects of the context remains input-enabled)*, and
3. if $o' \in \kappa_{C'}(q), C' : q \xrightarrow{o'.mtd'(\overline{p})} q'$ such that $acq(\overline{p}) \subseteq \kappa_{C'}(q)$ for all $mtd' \in extMtd(class(o'))$ *(all objects of the context can send to any exposed object of the run-time component a method call within the implicit interface of the class of that exposed object).*

Therefore, we can obtain the projected trace t where all objects of the context share the identities of all objects of the context and exposed objects of the component. Because of point 3, once o' is exposed any object of the context $o \in F$ can make a method call to o'. By the input-enabledness assumption, MESSAGESEND can be applied and we get a projected trace $t \cdot o \rightarrow o'.mtd(\overline{p}) \in Traces(\mathbb{C})$. □

The main idea behind the proof is that for each context that produces a trace of that component, we can always produce another context such that the newly exposed object is immediately used by the context in all possible way. Thus, the other context ensures that the open system trace property holds.

As stated earlier, we want to create an abstraction between classes and closed systems. This means that we would like to hide the *internal* behavior that happens between objects within the component. Such view allows bottom-up reuse of the components, for example, when the component is a library or a framework. Moreover, the view is suitable to deal with open systems, in particular systems with a non-software environment (e.g., GUI interaction with a human user – see [31] for more discussion). By using *idx*, hiding is straightforward to define.

Definition 14 (Component trace semantics). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component and* $Traces(\mathbb{C})$ *its plain trace semantics. The* component trace semantics *is a trace set* $Traces([C_0]) = Traces(\mathbb{C}){\downarrow}_{\mathbf{O}-idx(Traces(\mathbb{C}),C_0)}$.

To distinguish the component plain trace semantics from the one where hiding is performed, we use the notation $[C_0]$. This notation, read as boxed C_0, comes from the way we structure our component instances, where the component instance can be thought as a box of objects starting from the initial instance of the activator class C_0. By using the projection to the set of objects of the environment, the above definition reflects only the *observable* behavior which captures the communication with the context. For example, the client is only interested how the server component (as a whole) interacts with it, not how the server does its job. We denote the trace semantics of $\mathbb{C}$ by $[\![[C_0]]\!]$. Because the projection does not change the interaction that happens on the boundary of the component, we have the following corollary.

Corollary 2. $Traces([C_0])$ *is also open.*

Also of interest is that any trace in a component trace semantics contains at most a single creation event. This event represents the creation of the initial object of the activator class. In fact, this event is always the first event of a non-empty trace.

Lemma 4 (Initial creation event). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *and* $Traces([C_0])$ *its component trace semantics. The following properties hold.*

1. $\forall e \cdot t \in \mathit{Traces}([C_0]) \bullet \mathit{msg}(e) = \texttt{new}\ C_0$
(trace begins with creation of an instance of C_0)

2. $\forall e \cdot t \cdot e' \in \mathit{Traces}([C_0]) \bullet \mathit{isMtd}(\mathit{msg}(e'))$
(no other creation event appears in the trace)

Proof. 1. We assume that initially only an object of the context exists. To instantiate the component, the initial object of the activator class must be created at some point. Due to the projection, the first event in any non-empty trace of the component trace semantics is creation of an instance of the activator class.

2. All other objects within the run-time component are created transitively by the initial object of the run-time component. These creation events are abstracted away. □

Discussion. The term *activator class* used in this chapter comes from OSGi [39], where their component is instantiated through `BundleActivator`. The model where the component is instantiated by instantiating a single object of the activator class is not uncommon. For example, it coincides with the object adaptor of CORBA Component Model [30] and the class factory of COM [25]. Should a need arise for having more than one initial object, it can be simulated by our model by having the activator class as a stub that only creates the other objects.

Our component definition does not include the notions of dependency and versioning, because we want to focus on the behavior characterization of the component instances. The dependency issue is implicit within the definitions of the context and definition-completeness. Versioning can be modeled as as components with different classes but the same implicit interfaces.

The main question with defining the behavior of components is how we separate the internal and observable behavior. For this purpose, the notion of *boundary* enclosing the objects of a component at run-time is important. With our component model, we identify three particular ways to group objects into instances of a component or run-time component Using the component definition above, we identify three particular ways to group objects into instances of a component, called *run-time components*: static, programmer-defined and dynamic.

Static run-time component A static run-time component contains all objects in **C**. As such, grouping the objects into the component is trivial to define, by following the class of each object. However, the drawbacks of doing so are numerous. As the run-time component contains all objects in **C**, every object is at the boundary. This means that we cannot hide the internal behavior. Furthermore, the components then cannot share classes, as there is no way to separate the run-time instances of intersecting components. In our example, a static run-time component of the ({`Server`, `Worker`}, `Server`) component includes all server and worker objects. Hence, we cannot focus only on a single server with the workers it creates to represent the run-time view of the component. As a consequence, not only the focus on the activator class is lost, it is also difficult to specify the behavior that the component should have.

Programmer-defined run-time component A programmer-defined run-time component contains all objects in the way how the programmer defines it by specifying at the point of creation to which run-time component the newly created object belongs to. For example, in JCoBox [34], this is done by extending the **new** statement with **in** o to say that newly created object resides in the same run-time component as o. Various type-based ownership approaches can also be used (see, e.g., Banerjee and Naumann's confinement approach [7]). This approach is the most flexible as it provides fine-grained information which objects are at the boundary. However, this leads to additional constructs which makes it more complex to handle.

Dynamic run-time component A dynamic run-time component contains all objects that are created directly or indirectly by the initial object of the activator class. In other words, the run-time component is formed from the object-creation tree, with the initial object of the activator class as the root. This gives a more fine-grained grouping than the static approach, but less specific than the programmer-defined approach. Additionally, we can keep track of which objects are then on the boundary, while keeping track which new objects are included in the run-time component. This enables staying on focus on the behavior at the component boundary. After all, the hidden objects do not appear at the boundary and hence hiding them reduces the communication with the context. Because of its dynamicity, it is less straight forward than the static approach to give up front the exact instances of a component. In this chapter, we follow the dynamic approach of identifying run-time components which allows the use the activator class to represent the component.

5 Specification and Verification

Using the semantics of object classes and run-time component characterization based on the object creation tree, we now specify their functional behavior. Assuming the given specifications of the classes hold[2], we verify the specifications of the components ndeed hold based on the class specifications. We use our specification and verification technique [22] to illustrate how this task can be achieved.

5.1 Specification

An ideal specification technique should have a similar way to specify the behavior of classes and components. Our idea is to generalize the specifications of methods in the sequential case, which relate pre- to poststates, by relating input traces to output traces. To realize this idea, we use a specification format similar to the Hoare triple [17]. That is, the specification is of a triple form $\{p\}\ D\ \{q\}$, where p and q are trace assertions and D is either a class or a component. Informally, this triple means that if an input trace of D satisfies p, the output trace will satisfy q. In the following, we formally define the meaning of each part of this specification.

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 *caller suppressed* input (output) trace. This suppression, which yields an event content sequence, reflects the

[2] As stated in the introduction, one can use the technique by Ahrendt and Dylla [3] or Din et al. [14] to verify whether the actual class implementation satisfies the specification.

lack of knowledge on the receiver side (i.e., the entity we are specifying) who the sender of a message is.

Definition 15 (Trace assertion). *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* $\underline{\mathtt{x}}$ *is a variable, then* $\neg p, p \land q, \exists \underline{\mathtt{x}} : p$ *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 $\mathit{free}(p)$ extracts the set of all free variables appearing in p. For this chapter, we use an equality comparison operator, written $\$ = \langle ec \rangle$, that compares the suppressed trace with a maximal event content sequence *ec*. 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 object identities such that stripping this trace of the caller information yields a match to the mapped *ec*.

To define the semantics of a trace assertion, we substitute all occurrences of the trace constant with the actual trace. As a generic substitution mechanism, we use the notation $p[\underline{\mathtt{x}}/r]$ to denote the substitution of all (free) occurrences of a variable $\underline{\mathtt{x}}$ or the trace constant by some expression or assertion r in a trace assertion p. We assume that all variables and all substitutions are correctly typed. Using first-order logic, we then map the assertion to boolean values $\{\mathbf{true}, \mathbf{false}\}$. Given a trace t, we write $\models_{FOL} p[\$/t]$ if it is mapped to **true**, and $\models_{FOL} p$ if for any trace t, $\models_{FOL} p[\$/t]$. The *FOL* index indicates that the variable assignment is done using the first-order logic semantics.

Example 7. $\$ = \langle \underline{\mathtt{this}}.\mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \rangle$ is a trace assertion stating that the trace starts with a creation of a `Worker` object, stored into the free variable $\underline{\mathtt{this}}$. Then, the worker is sent a task computation request followed by a result propagation. Only this sequence appears in the trace.

Given the semantics of the trace assertions, we can give the semantics of the specification triples $\{p\}\ D\ \{q\}$. The triple characterizes the output trace the instance of class or component D produces when faced with the input trace p. Because of their nature, 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 object or an implicit exposure of locally created objects) should be existentially quantified. As convention, the initial object created is referred to by the variable $\underline{\mathtt{this}}$. The triple 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.

To define the triple semantics, a trace t needs to be split into input and output traces. For that we need the set of objects L that represents entity D at run-time. Then, the function $\mathit{split}(t, L) = (t{\downarrow_{F,caller}}{\downarrow_{L,callee}}, t{\downarrow_{L,caller}}{\downarrow_{F,callee}})$ does exactly so, where $F = \mathbf{A} - L$.

In the case where the entity represented by D is a class C, L is taken to be a singleton object o whose class is C. We call $\{p\}\ C\ \{q\}$ a *class triple*. The *split* function ensures that only the interaction done by o appears in the input and output traces that are being considered in the semantics of the specification triple.

Definition 16 (Class triple semantics). *Let C be a class, $[\![C]\!]$ its semantics, and o an object such that $class(o) = C$. $[\![C]\!]$ satisfies $\{p\}\ C\ \{q\}$, written $\models \{p\}\ C\ \{q\}$, if for all maximal traces $t \in [\![C]\!]$ with $split(t, \{o\}) = (ti, to)$ the following holds.*

$$\models_{FOL} p[\$/ti] \implies q[\$/to]$$

Example 8. 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.
$\{\ \$ = \langle \underline{\texttt{this}} := \textbf{new}\ \texttt{Server} \cdot \underline{\texttt{this}}.\texttt{serve}(\underline{\texttt{c}}, \underline{\texttt{t}})\rangle\ \}$
  `Server`
$\{\ \exists \underline{\texttt{w}} \bullet \$ = \langle \underline{\texttt{w}} := \textbf{new}\ \texttt{Worker} \cdot \underline{\texttt{w}}.\texttt{do}(\underline{\texttt{t}}) \cdot \underline{\texttt{w}}.\texttt{propagate}(\textbf{null}, \underline{\texttt{c}})\rangle\ \}$
From Def. 16 above, the semantics of the specification is a trace set of an object s of class `Server`, where for each maximal trace, the input and output parts are as stated in the specification. Just from the specification, the trace set may include traces where all output events appears before the input events. In this case, the specification allows a worker to be created by the server before the server receives any request. This is as expected because the specification abstracts from the actual behavior of the implementation and for this chapter, the specification need not precisely describe the exact interleaving that happens.

When D is a component represented by its activator class $[C]$, the set of objects L of the run-time component need be extracted from the semantics. From Lemma 4, in any trace of $[\![[C]]\!]$ there is only one creation event of the initial object of the component. Thus, we can define the following function that extracts the set of objects of the run-time component that lies on the boundary of that run-time component.

Definition 17 (Boundary extraction). *Let T be a trace set of some component. $L' = bound(T)$ is the subset of objects of the run-time component, where*
$bound(T) = \bigcup_{t \in T} bound(t)$, $bound([]) = \emptyset$, and

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

The local object 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 run-time component. By definition of the component trace semantics, there is exactly one creation event visible in any trace of T which is the creation of the initial object of the component instance. If t ends with a method call and it is directed to some foreign object, the caller of this event and all exposed objects in the method call arguments are included. Using the boundary extractor function above, the semantics of $\{p\}\ [C]\ \{q\}$, called a *component triple*, is defined as follows.

Definition 18 (Component triple semantics). *Let $[C]$ represent a component, $[\![[C]]\!]$ its semantics and $B = bound([\![[C]]\!])$ the set of objects on the boundary of the component. $[\![[C]]\!]$ satisfies $\{p\}\ [C]\ \{q\}$, written $\models \{p\}\ [C]\ \{q\}$, if for all maximal traces*

$t \in [\![[C]]\!]$ with $split(t,B) = (ti, to)$ the following holds.

$$\vDash_{FOL} p[\$/ti] \implies q[\$/to]$$

Example 9. As a component, the worker replies to the client by merging the partial result passed on to the component with the computation of the remaining task as a whole. This property can be specified as follows.
$\{ \$ = \langle \underline{\mathtt{this}} := \mathtt{new\ Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \rangle \}$
$[\mathtt{Worker}]$
$\{ \$ = \langle \underline{\mathtt{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\underline{\mathtt{t}}))) \rangle \}$
Similar to Ex. 8, the semantics of the specification above only deals with the maximal traces, where the input part are in the order of creating a new `Worker` object, obtaining the request to do a task and then propagate the computation result. When the input part is satisfied, the worker component produces a response back to the client by computing the whole remaining task (following the assumption on `compute` given in Sect. 2) and merging it with the given partial result.

Usually, we want to cover as much as possible the behavior of the entity we are specifying. Thus, its specification is a collection of triples. An implementation satisfies the specification, if its trace semantics satisfies all triples within the specification.

Definition 19 (Specifications). *Let D be a class or a component. 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 $\forall (\{p_i\}\ D\ \{q_i\}) \in S \bullet \vDash \{p_i\}\ D\ \{q_i\}$.*

Example 10. 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.
$\{ \$ = \langle \underline{\mathtt{this}} := \mathtt{new\ Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \rangle \wedge \mathtt{size}(\underline{\mathtt{t}}) = 1 \}$
$\mathtt{Worker}$
$\{ \$ = \langle \underline{\mathtt{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\underline{\mathtt{t}}))) \rangle \}$
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.
$\{ \$ = \langle \underline{\mathtt{this}} := \mathtt{new\ Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \rangle \wedge \mathtt{size}(\underline{\mathtt{t}}) = \underline{\mathtt{n}} \wedge \underline{\mathtt{n}} > 1 \}$
$\mathtt{Worker}$
$\{ \exists \underline{\mathtt{w}} \bullet \$ = \langle \underline{\mathtt{w}} := \mathtt{new\ Worker} \cdot \underline{\mathtt{w}}.\mathtt{do}(\mathtt{restTask}(\underline{\mathtt{t}})) \cdot$
$\underline{\mathtt{w}}.\mathtt{propagate}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\mathtt{firstTask}(\underline{\mathtt{t}}))), \underline{\mathtt{c}}) \rangle \}$

5.2 Verification

The semantics of the specifications includes the open system aspect. And unlike the usual specification where one can refer to the program model, we have only the class or

$$\textsc{ClassSpec}\ \ \{p\}\ C\ \{q\}$$

$$\textsc{Consequence}\ \ \frac{p \implies p_1 \quad \{p_1\}\ D\ \{q_1\} \quad q_1 \implies q}{\{p\}\ D\ \{q\}}$$

$$\textsc{Boxing}\ \ \frac{\{p\}\ C\ \{q \wedge \mathit{nonCr}\}}{\{p\}\ [C]\ \{q\}}$$

$$\textsc{BoxedComposition}\ \ \frac{\{p \wedge \underline{i} = \$\}\ C\ \{q \wedge \mathit{noSelfExp}(\underline{i})\} \quad \{q'\}\ D\ \{r \wedge \mathit{nonCr}\} \quad \mathit{match}(q, q', D)}{\{p\}\ [C]\ \{r\}}$$
$$\text{where } \underline{i} \notin \mathit{free}(p) \cup \mathit{free}(q)$$

$$\textsc{Induction}\ \ \frac{\{p \wedge m = 0\}\ [C]\ \{q\} \quad \mathit{match}(p', p, C) \quad \{p \wedge m = \underline{z} \wedge m > 0 \wedge \underline{i} = \$\}\ C\ \{p' \wedge m < \underline{z} \wedge \mathit{noSelfExp}(\underline{i})\}}{\{p\}\ [C]\ \{q\}}$$
$$\text{where } \underline{i} \notin \mathit{free}(p) \cup \mathit{free}(q)$$

$$\textsc{Invariance}\ \ \frac{\{p\}\ D\ \{q\}}{\{p \wedge r\}\ D\ \{q \wedge r\}} \quad \text{where } \mathit{consFree}(r)$$

$$\textsc{Substitution}\ \ \frac{\{p\}\ D\ \{q\}}{\{p[\underline{x}/r]\}\ D\ \{q[\underline{x}/r]\}}$$
$$\text{where } \underline{x} \in \mathit{free}(p) \cup \mathit{free}(q) \text{ and } \mathit{consFree}(r)$$

Fig. 6. Inference rules for RPSA

component name. Because of these aspects, to use specification of classes and smaller components to verify the behavior of a larger component, we need a corresponding proof system. The proof system we present below is by far incomplete and only handles systems of similar nature to the running example. However, it is sound and illustrates the complete picture of our approach.

The proof system RPSA presented in Fig. 6 consists of a few inference rules. Each premise can be a trace assertion, which is applied to any trace using first-order logic semantics, or a specification triple with the semantics as declared in Sect. 5.1. For this chapter, we only use specification triples as consequences. A *proof* is a tree of inference rule applications. Each node is a (possibly empty) premise and each edge is a rule application which in the following will be labeled with the applied inference rule. The root of the proof is the main goal: a specification triple. The leaves are either a valid trace assertion or an empty premise.

ClassSpec allows a given class triple to be used as an end node of a proof tree branch, if it is a triple that is assumed to be true. That is, the class triple must be proved against the implementation. If the class triple is not part of the assumed class triples, we cannot apply ClassSpec, as it would render the proof system unsound.

The inference rule Consequence is a standard Hoare logic rule, where the input trace assertion can be weakened and the output trace assertion can be strengthened.

Boxing transforms a class triple into a component triple, when the output trace assertion states that no object is created (represented by the predicate *nonCr*) by the instance of that class. This coincides with the component trace semantics definition, because already the class traces that satisfy this specification guarantees no object creation is observed.

The BoxedComposition rule composes a class triple with a subcomponent triple. The idea is that if the triple of class C is restricted enough to guarantee that its instance only create a subcomponent, and the output of the instance as a whole becomes the input of the instance of the subcomponent, then we can infer to the component triple of $[C]$ by taking the input trace assertion of the original class triple and the output trace assertion of the subcomponent triple. In other words, the instance of C encapsulates the subcomponent.

The last rule is INDUCTION. As the name suggests, this inference rule deals with the case when multiple objects of the same class are created to handle the same input with some parameter converges into some base case, as indicated by the measure variable m. In a way, this is similar to making a recursive call inside an object. The main difference is that due to the concurrent nature of the objects, each input parameter may be processed independently by each created object, possibly optimizing the computation.

In addition to the inference rules above, RPSA also includes some standard auxiliary rules: INVARIANCE and SUBSTITUTION. INVARIANCE allows a predicate containing no trace constant to strengthen both input and output trace assertions of a triple. SUBSTITUTION allows a free variable to be substituted to some predicate or expression r which must not contain the trace constant.

Theorem 1 (Soundness [22]). *The proof system in Fig. 6 is sound.*

Using the sound proof system described above, we can show that the server component replies back to a request from a client with the appropriate computation result. In the example below, we illustrate how the proof system is used by verifying the server component triple from server class triple and worker component triple. A complete account how the server component can be verified from the class triples can be found in the authors' previous work [22].

Example 11. A specification of the server component stating that a request from the client is replied by a response to the client with the computed result is as follows.

$\{\ \$ = \langle \underline{\mathsf{this}} := \mathbf{new}\ \mathtt{Server} \cdot \underline{\mathsf{this}}.\mathtt{serve}(\underline{\mathsf{c}}, \underline{\mathsf{t}}) \rangle\ \}$
$[\mathtt{Server}]$
$\{\ \$ = \langle \underline{\mathsf{c}}.\mathtt{response}(\mathtt{compute}(\underline{\mathsf{t}})) \rangle\ \}$

Compared to the server class triple given in Ex. 8, there are two notable differences. First is that the represented element is the component $[\mathtt{Server}]$. Second, the output trace assertion directly deals with the expected output behavior of the component. Therefore, the specification hides how the server achieves the production of the output. The input assertion remains the same.

By applying the inference rules on the worker component (Ex. 9) and server class specifications (Ex. 8), we can infer the specification of the server component. In other words, when the server is implemented in a way described by the worker component and the server class specifications, no matter how the context the server component is used, the server behaves as specified. Due to margin, we abbreviate the following assertions used in the specifications.

– $\mathsf{InSrv} \overset{\text{def}}{=} \mathsf{InSrvC} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{this}} := \mathbf{new}\ \mathtt{Server} \cdot \underline{\mathsf{this}}.\mathtt{serve}(\underline{\mathsf{c}}, \underline{\mathsf{t}}) \rangle$
– $\mathsf{OutSrv} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{w}} := \mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathsf{w}}.\mathtt{do}(\underline{\mathsf{t}}) \cdot \underline{\mathsf{w}}.\mathtt{propagate}(\mathbf{null}, \underline{\mathsf{c}}) \rangle$
– $\mathsf{InWrkC} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{this}} := \mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathsf{this}}.\mathtt{do}(\underline{\mathsf{t}}) \cdot \underline{\mathsf{this}}.\mathtt{propagate}(\underline{\mathsf{v}}, \underline{\mathsf{c}}) \rangle$

– $\mathsf{OutWrkC} \overset{\text{def}}{=} \$ = \langle \underline{\mathtt{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\underline{\mathtt{t}})))\rangle$
– $\mathsf{OutSrvC} \overset{\text{def}}{=} \$ = \langle \underline{\mathtt{c}}.\mathtt{response}(\mathtt{compute}(\underline{\mathtt{t}}))\rangle$

The abbreviations are chosen such that InSrv, for example, represents the input event content equality of the server class triple, whereas $\mathsf{OutWrkC}$ represents the output event content equality of the worker class triple. The C suffix indicates the assertion is used in a component triple. We also introduce the function *cse*, short for *c*ontent *s*equence *e*xtractor, to extract the event content sequences from these abbreviations.

To achieve the inference of the server component specification, we work backwards until the server class and worker component specifications are obtained. The suitable rule for this inference is BOXEDCOMPOSITION. The input to the server component is handled fully by the server object, while the output of the server object is captured completely by the worker component instance. In order to match the output trace assertion of the server class triple, the partial result variable in the input trace assertion of the worker component instance has to be initialized to **null**.

$$\textsc{Cmp}\ \frac{\begin{array}{c}\{\mathsf{InSrvC} \land \underline{\mathsf{i}} = \$\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \land \mathit{noSelfExp}(\underline{\mathsf{i}})\} \\ \{\mathsf{InWrkC}[\underline{\mathsf{v}}/\mathbf{null}]\}\ [\mathtt{Worker}]\ \{\mathsf{OutSrvC} \land \mathit{nonCr}\} \\ \mathit{match}(\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv}, \mathsf{InWrkC}[\underline{\mathsf{v}}/\mathbf{null}], [\mathtt{Worker}])\end{array}}{\{\mathsf{InSrvC}\}\ [\mathtt{Server}]\ \{\mathsf{OutSrvC}\}}$$

As both triples left as proof obligation in the proof tree above are not of the assumed form, they must be transformed. The proof tree below shows how the server class triple above can be obtained from the original specification, after which only CLASSSPEC is needed. We need to store the input trace and transfer it to the output trace assertion of the triple, in order to determine whether a reference to the created server object is not exposed. In this example, the content sequence extractor function is used to get the suppressed input trace that we need. Note that the input trace assertions of the server class and server component triples are the same.

$$\textsc{Cns}\ \frac{\begin{array}{c}\textsc{Inv}\ \dfrac{\textsc{Spec}\ \dfrac{}{\{\mathsf{InSrv}\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv}\}}}{\{\mathsf{InSrv} \land \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv})\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \land \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv})\}} \\ \mathsf{InSrv} \land \underline{\mathsf{i}} = \$ \implies \mathsf{InSrv} \land \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv}) \\ \exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \land \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv}) \implies \exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \land \mathit{noSelfExp}(\underline{\mathsf{i}})\end{array}}{\{\mathsf{InSrv} \land \underline{\mathsf{i}} = \$\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \land \mathit{noSelfExp}(\underline{\mathsf{i}})\}}$$

The transformation for the worker component triple is straight forward to obtain. All occurrences of variable $\underline{\mathsf{v}}$ are substituted by **null** and the *nonCr* predicate is trivially implied by definition of components.

$$\textsc{Cns}\ \frac{\begin{array}{c}\textsc{Sub}\ \dfrac{\{\mathsf{InWrkC}\}\ [\mathtt{Worker}]\ \{\mathsf{OutWrkC}\}}{\{\mathsf{InWrkC}[\underline{\mathsf{v}}/\mathbf{null}]\}\ [\mathtt{Worker}]\ \{\mathsf{OutWrkC}[\underline{\mathsf{v}}/\mathbf{null}]\}} \\ \mathsf{OutWrkC}[\underline{\mathsf{v}}/\mathbf{null}] \implies \mathsf{OutSrvC} \land \mathit{nonCr}\end{array}}{\{\mathsf{InWrkC}[\underline{\mathsf{v}}/\mathbf{null}]\}\ [\mathtt{Worker}]\ \{\mathsf{OutSrvC} \land \mathit{nonCr}\}}$$

Thus, we have completed the inference proof of the server component triple.

5.3 Discussion and Related Work

The specification technique is incomplete, as it does not provide any information on the partial order between each message appearing in the input and the output traces [41]. Thus, the verification technique is also incomplete. Nonetheless, the specification technique can handle every relation R on input and output traces that we can define in the underlying functional specification framework: $\{\$ = ti\}\ D\ \{R(ti)\}$. Particularly, the input trace assertion may depend on the exposure of some objects which appears in the output trace assertion. The technique is also sufficient when interleavings between the input and output need not be specified.

Techniques for verifying and specifying concurrent behavior are subject of many well-known works. Classical minimal models CSP [18], CCS [26] and π-calculus [27] allow specifying and reasoning of interactions among threads. But because of their minimality, they are too abstract for two-tier verification. A possibility to apply these models is on the upper level tier (i.e., verifying component/system specifications from class specifications). However, showing the connection between component/system specifications and class specifications requires a significant adaptation to the setting. For example, Sangiorgi and Walker devoted a chapter in the final part of their book [32] to show how to restrict π-calculus to simulate object-orientation.

Misra and Chandy [28], Soundarajan [36] and Widom et al.[41] proposed proof methods handling network of concurrent processes using traces. Misra and Chandy focused on next-step invariance of the processes using all information present in the trace prior to the appearance of a message. Soundarajan related invariants on process histories to the axiomatic semantics of a parallel programming language. Widom et al. discussed the necessity of having prefix-closed trace semantics and partial ordering between messages of different channels to reach a complete proof system. They deal only with closed systems of fixed finite processes (and channels) and, because of their generality, make no use of the guarantees and restrictions of the concurrent object model.

Several works have focused on concurrent object or actor models. Specification Diagram [35] provides a detailed, graphical 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 ([13]) characterization of Specification Diagram can be obtained [38].

State-based specification and verification techniques have also been developed (e.g., [12,11,16,33]), but they rely on having the actual implementations, bypassing the intermediate tier that we would like to have. De Boer [8] presented a Hoare logic for concurrent processes that communicate by message passing through FIFO channels in the setting of Kahn's deterministic process networks ([21]). 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.

Ahrendt and Dylla [3] and Din et al. [14,15] extended Soundarajan's work to deal with concurrent object systems. They consider only finite prefix-closed traces, justifying it by having only finite number of objects to consider in the verification process. Din et al. particularly verified whether an implementation of a class satisfies its triples by transforming the implementation in a simpler sequential language, applying the transformational method proposed by Olderog and Apt [29]. The main difference to our approach is on the notion of component that hides a group of objects into a single entity. It avoids starting from the class specifications of each object belonging to a component when verifying a property of the component.

6 Conclusion

We have seen in this chapter a formal system that covers a small part of open concurrent object systems. An attempt is made to describe what it means to compose classes (and smaller components) into components and how to use these composition to verify functional properties of the composed entity in an open setting. It turns out that the notion of class composition chosen in this chapter is closely related to popular component framework. While it is clear that components should have interfaces and hide behavior, connecting these concepts to verification is not clear. The approach given here suggests that to obtain a sound verification technique which makes use of components, one needs to be explicit about the class of properties and how they are specified.

The investigation on how to apply the component notion for this verification purpose is far from over. While the connection between the first layer of verification (i.e., from the implementation to the class specification) given in the introduction is dealt by the latest research [3,14,15], its connection with the proposed specification techniques is not yet explored. Furthermore, the presented proof system is useful for one way pipeline communicating components. To extend the proof system into other cases, other common patterns of communication between components need be considered. Two particular extensions of interest are the sound rules to compose more than two components/classes and two-way communication between components.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
3. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. 77(12), 1289–1309 (2012)
4. Arts, T., Dam, M.: Verifying a distributed database lookup manager written in Erlang. In: World Congress on Formal Methods. pp. 682–700 (1999)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
6. Baker, Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. SIGART Bull. pp. 55–59 (August 1977)
7. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. J. ACM 52(6), 894–960 (2005)
8. de Boer, F.S.: A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. Theor. Comput. Sci. 274(1–2), 3–41 (2002)

9. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer-Verlag New York, Secaucus, NJ, USA (2001)
10. Clinger, W.D.: Foundations of Actor Semantics. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (1981)
11. Dam, M., Fredlund, L.Å., Gurov, D.: Toward parametric verification of open distributed systems. In: COMPOS. pp. 150–185 (1997)
12. Darlington, J., Guo, Y.: Formalising actors in linear logic. In: OOIS. pp. 37–53 (1994)
13. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
14. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: Component reasoning for concurrent objects. J. Log. Algebr. Program. 81(3), 227–256 (2012)
15. Din, C.C., Dovland, J., Owe, O.: Compositional reasoning about shared futures. In: SEFM. pp. 94–108 (2012)
16. Duarte, C.H.C.: Proof-theoretic foundations for the design of actor systems. Mathematical Structures in Computer Science 9(3), 227–252 (1999)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (Oct 1969)
18. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (Aug 1978)
19. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: FMCO 2010. pp. 142–164. LNCS, Springer (2011)
20. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365(1-2), 23–66 (2006)
21. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress. pp. 471–475 (1974)
22. Kurnia, I.W., Poetzsch-Heffter, A.: A relational trace logic for simple hierarchical actor-based component systems. In: AGERE! ACM (2012), to appear
23. Lavender, R.G., Schmidt, D.C.: Active object – an object behavioral pattern for concurrent programming. In: Pattern Languages of Programs (1995)
24. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
25. Microsoft: Component Object Model (COM) (Jan 1999), available at `http://www.microsoft.com/com/default.asp`
26. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)
27. Milner, R.: Communicating and Mobile Systems – The π-Calculus. Cambridge University Press (1999)
28. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Software Eng. 7(4), 417–426 (1981)
29. Olderog, E.R., Apt, K.R.: Fairness in parallel programs: the transformational approach. ACM Trans. Program. Lang. Syst. 10(3), 420–455 (Jul 1988)
30. OMG: CORBA component model v4.0 (2006), `http://www.omg.org/spec/CCM/`
31. Poetzsch-Heffter, A., Feller, C., Kurnia, I.W., Welsch, Y.: Model-based compatibility checking of system modifications. In: ISoLA 2012. pp. 97–111. LNCS, Springer (October 2012)
32. Sangiorgi, D., Walker, D.: The Pi-Calculus – A Theory of Mobile Processes. Cambridge University Press (2001)
33. Schacht, S.: Formal reasoning about actor programs using temporal logic. In: Concurrent Object-Oriented Programming and Petri Nets. pp. 445–460 (2001)
34. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: ECOOP 2010. pp. 275–299. LNCS, Springer (2010)

35. Smith, S.F., Talcott, C.L.: Specification diagrams for actor systems. Higher-Order and Symbolic Computation 15(4), 301–348 (2002)
36. Soundarajan, N.: A proof technique for parallel programs. Theoretical Computer Science 31(1–2), 13–29 (1984)
37. Talcott, C.L.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation 11(3), 281–343 (1998)
38. Thati, P., Talcott, C.L., Agha, G.: Techniques for executing and reasoning about specification diagrams. In: AMAST. pp. 521–536 (2004)
39. The OSGi Alliance: OSGi core release 5 (2012), `http://www.osgi.org`
40. Vasconcelos, V.T., Tokoro, M.: Traces semantics for actor systems. In: Object-Based Concurrent Computing. LNCS, vol. 612, pp. 141–162. Springer (1991)
41. Widom, J., Gries, D., Schneider, F.B.: Completeness and incompleteness of trace-based network proof systems. In: POPL. pp. 27–38 (1987)
42. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: OOPSLA. pp. 258–268 (1986)