Verification of Open Concurrent Object Systems*

Ilham W. Kurnia and Arnd Poetzsch-Heffter

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

Abstract. A concurrent object system is a dynamically growing collection of concurrently interacting objects. Such a system is called *open* if the environment of the system is unknown. Proving properties about open systems is challenging because the properties must be shown to hold for all possible environments of the system. *Hierarchical* reasoning, which infers properties of large components from the properties of smaller subcomponents, is a key enabler to manage the reasoning effort.
This chapter presents an approach to hierarchically specify and verify open concurrent object systems. We introduce a core calculus for concurrent object systems. The behavior of such a system is given by a standard operational semantics. To abstract from the internal representation of the objects, we develop an alternative trace-based semantics that captures the behavior in terms of the communication traces between the objects of the system and the environment. The main advantage of the trace-based model is its extendability to components and open systems while remaining faithful to the operational model. The specification approach is also directly based on the traces and supports hierarchical reasoning using the following two central features. *Looseness* allows specification refinement. *Restriction* allows expressing assumptions on the environment. Finally, we provide a Hoare-style proof system that handles the given specifications.

1 Introduction

Modern software systems are becoming more and more concurrent and distributed. The main reasons are that they have to exploit the resources of many-core computer architectures and realize distributed applications. Unfortunately, concurrency increases the complexity of software and is often a source of errors. Whereas testing is a successful technique to improve the quality of sequential software, testing is much less appropriate for concurrent software because of its high degree of nondeterminism. Here, verification techniques come to help (see, e.g., [8, Chap. 1], [49, Chap. 1]).

Object-oriented framework is a recommended choice for building concurrent and distributed systems [31]. There are many approaches how concurrency can be introduced into the object-oriented framework (see, e.g., a survey by [47]). In this chapter, we describe a technique to verify functional properties of *concurrent object* systems [63]. A concurrent object (the intrinsic concurrency model of the modeling language ABS [33] discussed in this volume) is a computational entity whose state is fully encapsulated.

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

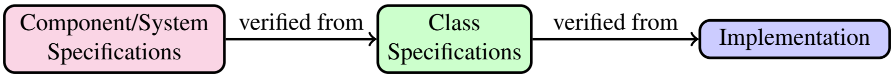


Fig. 1. Two-tier verification

It communicates with other objects exclusively by exchanging asynchronous messages. Within each object, only a single, cooperatively scheduled thread is running at any time. Thus, the behavior of single objects[1] can be verified using sequential techniques. Similar to popular programming languages such as Java and C++ and also ABS, classes are used as to describe the behavior of objects. The combination of encapsulation, concurrency among objects, and sequential execution within objects grants scalability as explained, e.g., in [6,28]. Concurrent objects are similar to actors [3]; the two approaches mainly differ in how behavior changes and interfaces are expressed.

There are already a number of verification techniques for concurrent objects available (see, e.g., [23,11,5,20,22]). The technique presented in this chapter centers around two specific aspects: *openness* and *hierarchical reasoning*. Openness allows the verification of components without knowing the environments in which they are used, while hierarchical reasoning allows verifying larger components from smaller ones in a hierarchical way. The technique follows a two-tier verification approach proposed by, for example, Misra and Chandy [43] and Widom et al. [62], as shown in Fig. 1. In the first tier, the behavior of a class is specified and verified against its implementation. That is, the program code of the class is used to prove that objects of the class have a specified behavior at their interfaces. This issue is *not* treated in this chapter. It is handled, e.g., in [5,20,22]. Here we focus on proving the specification of larger components from the specifications of smaller ones where the smallest components are classes. The verification technique is *modular*, i.e., the specification of smaller components is sufficient to prove larger ones.

Essential to our verification approach is the notion of “component”. Components according to this notion should satisfy at least the following requirements:

– The specification technique for components must be applicable to classes, because they are the base of the hierarchical approach.
– Components should allow the construction of increasingly large systems.
– Components need clear semantical interfaces that can hide internal behavior.

In our approach to components, we follow a basic idea from software component models such as OSGi [60] and COM [40]. A component has an activator class AC. A component instance is created by creating an object X_{AC} of class AC. At run time, the component instance consists of the set $O(X_{AC})$ of objects transitively created by X_{AC}. A communication of $O(X_{AC})$ with the environment happens when an object in $O(X_{AC})$ sends a message to an object outside of $O(X_{AC})$ or vice versa. Of course, the exact formation of a component instance is influenced by how the environment interacts with the component. A component is then by nature an open system.

The specification of a component abstracts from the *internal* messages and objects. It only describes the communication of the component with the environment. Composition is done by developing new components using other components for their implementation. It is similar to procedural programming in that a procedure calls other

[1] Henceforth we refer to concurrent objects simply as objects.

procedures and in that the specification abstracts from procedure-local state and local calls.

To specify the desired properties of components we need to define their semantics. We first introduce an operational semantics of the classes, that is, of the base components, by means of transition systems and core operational rules. From the operational semantics, we derive a trace semantics. That is, we represent an execution of an object system by a trace of observable events [30]. The advantage of dealing only with traces is the abstraction from the actual state representation of the system. The semantics of objects and components is expressed by *trace sets*.

Based on the trace semantics, we develop a specification technique relating input traces to output traces. Input (output) traces represent events an object or a component receives (produces). Distinguishing input and output traces is done based on the trace set. Formally, a specification consists of a finite set of Hoare-like triples [29]. A triple $\{p\}\ D\ \{q\}$ means that if an input trace satisfies p, component D produces output traces satisfying q. The component D represents either the behavior of single objects of class C or the (external) behavior of groups of objects with an initial object of class C. A triple specifies the behavior of a component only for inputs satisfying p. This input condition provides assumptions about how the component is used and help to focus the reasoning.

This chapter follows closely on the authors' previous work [37], which explains the specification and verification approach in detail. We focus here on the connection between an operational semantics of concurrent object systems and its trace semantics in the open system context.

Chapter structure. In Sect. 2 we explain the setting and provide a running example used in this chapter. Section 3 deals with the core operational semantics of concurrent object systems. Section 4 describes how components and their trace semantics can be extracted from the operational semantics. Section 5 presents the specification and verification technique with the help of the running example. We conclude this chapter with some discussion how this approach can be extended. Related work and discussion on the subject of each section are provided at the end of each section.

2 Setting and 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 of as a stripped down version of ABS [33][2]. ABS is an object-oriented modeling language that permits, among others, actor-style synchronous communication. ActJ focuses on the actor concurrency layer of ABS. It features a Java-like syntax, interfaces, class-based programming, asynchronous method calls without returns, concurrent objects, abstract data types with first-order side-effect free functions and typed references. This language is used to describe a variant of an industrial distributed database system [7].

[2] Information about ABS can also be found in the chapter by Reiner Hähnle in this volume [27]. Various design choices of ActJ can be inferred from the design choices of ABS.

2.1 ActJ

Objects in ActJ is described by means of classes. A class is a template how an object is represented and behaves according to the representation. Through fields a class defines what data can be in an object and through methods a class defines how operations can be applied on the object. An object can be created by instantiating the class. A method implementation may be equipped a *guard*, regulating when a method may be executed (explained later). All fields are private to individual instances, providing strict encapsulation on instance level. Object manipulation described exclusively through methods, which are public. Methods contain a list of statements. A statement can be an object creation, an asynchronous method call, a conditional statement, field assignments and a sequential composition of statements. Expressions contained in a statement are pure functions and abstract data types such as lists can be used.

ActJ follows the principle of *programming to interfaces* [25], where the declared types of objects are interface types. Interfaces consists of a number of method signatures, stating the name and parameters of operations that can be applied on the object. To focus on the core behavior of the objects, subtyping is not introduced in ActJ. For the same reason, each class implements exactly one interface. As a simplification, method signatures do not contain have a return type. Hence, computation result that needs to be communicated back is delivered through making method calls. Examples of ActJ programs are given in Figs. 2 and 3, which is explained in the following subsection.

2.2 Running Example

As a running example, we use a variant of the client-server setting treated by Arts and Dam [7]. The server receives requests from the client, where each request contains a query. The server system responds to the requests with the appropriate computation results. To serve each request, the server creates a worker and passes on the query to be computed. A query can be divided into multiple chunks, therefore concurrent processing of a request can be introduced as follows. Before each worker processes the first chunk of the query, it creates another worker to which the rest of the query is passed on. When the computation of the first query chunk is finished, the worker merges the previous result with this computation result and propagates the merged result to the next worker. Eventually all chunks of the query are processed, and the last worker sends back the final result to the client. The client identity must be passed around, so that the last worker can return the query computation result to the client.

An implementation of the server side is given in Fig. 2. It consists of two classes: `Server` and `Worker`. The `Server` class implements the `IServer` interface, whereas `Worker` implements `IWorker`. These classes use the `IClient` interface so they can communicate with the client. A client can send a request to the server by calling the `serve` method. A query is represented by the `Query` data type. The server processes the request by creating a new `Worker` object, passing on the query to this new worker and then initiate the result propagation. The execution of the `serve` method acts as an example how objects can interact with other objects in ActJ.

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

```
interface IClient {
  response(Value);
  link(IServer);
}
interface IServer {
  serve(IClient c, Query y);
}
interface IWorker {
  do(Query y);
  propagate(Value v, IClient c);
}

class Server implements IServer {
  serve(IClient c, Query y) {
    // querySize(y) ≥ 1
    IWorker w = new Worker;
    w.do(y);
    w.propagate(null, c);
  }
}
```

```
class Worker implements IWorker {
  Value myResult = null;
  IWorker nextWorker = null;
  do(Query y) {
    if (querySize(y) > 1) {
      nextWorker = new Worker;
      nextWorker.do(restQuery(y));
    }
    myResult = compute(fstQuery(y));
  }
  propagate(Value v, IClient 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

of a statement of the form `new C`, where `C` is a class name. Object creation is *blocking*, meaning that the execution cannot continue to the next statement before the object is created. A method call is produced when a statement of the form `r.m(p̄)` is executed. This statement sends the message `m(p̄)` to the receiver object `r` where `m` is the method name with a list of parameters p̄. The parameters can be data values or object identities. A method call is *non-blocking*; execution directly continues with the next statement. Thus, in general, a method call leads to concurrent behavior. The asynchronous nature of method calls leads to the possibility that when two method calls are sent one after the other, they may arrive at their respective destination in a different order. For each of the calls an object receives, it has a *body* that describes how it reacts to a method call.

The `Worker` class illustrates how `guard` is used. The class has two methods `do` and `propagate` and two fields `myResult` and `nextWorker`, both initialized to `null`. The `do` method checks (via the function `querySize`) whether the query the worker has to do can be split to further subqueries. If the query is splittable, a new worker is created and the reference of this new worker is stored in the `nextWorker` field. The new worker is then sent the rest of the query (via the function `restQuery`) the current worker is not handling. The current worker then proceeds on computing the result of the first subquery (or the entire query if it is not splittable). The computation is done using the function `compute` and when it is completed, the result is stored in `myResult`.

The `propagate` method states how a worker propagates the result of its computation. The `guard` ensures that the result propagation is carried out *only* when the worker's part of the query computation is finished. If the worker is the last worker handling the query (indicated by the non-existence of the next worker object), a `response` call is sent to the client, whose reference is passed on with the call. This response contains the merged result of the worker's computation of the subquery it handled with the value

```
class Client implements IClient {
  IServer s = null;
  link(IServer server) {
    s = server;
    s.serve(this, randomQuery());
  }
  response(Value v) {}
}
```

Fig. 3. A sample `Client` in ActJ

passed on with the call. Otherwise, the merged result is passed on to the next worker. In short, the series of `propagate` calls, initiated by the server, collect and merge the results of computing the different query chunks and send the clients the response.

It remains to explain what precisely happens when a method call is received. We assume that objects, similar to actors [3], have an unbounded input queue and are input-enabled (cf. [39, p. 257]); i.e., objects can always accept new input. An object is always in one of two modes: *idle* and *active*. Whenever an object is idle, meaning it has just been created or finished executing a method, it selects a method call from its input queue to be executed. If the queue is empty, then the object is idle. In other cases, an object is active. When a method call is selected, it is removed from the queue. During this selection process, the guards of the corresponding method definitions are evaluated. Method definitions that do not have a guard are equivalent to method definitions whose guard is always **`true`**. Method calls are then *selected* from the queue primarily in a First-In-First-Out manner. In other words, the first pending method call in the queue whose guard is evaluated to **`true`** is selected. This selection process guarantees (weak) fairness, meaning that a method call whose guard is continuously evaluated to **`true`** will eventually be picked for processing. The presence of guards gives an object control over the execution of incoming method calls.

We assume appropriate definitions for the pure functions used in the `Worker` class. In addition, the following properties are assumed.

```
querySize(t) ≥ 1
querySize(t) > 1  ⟹  compute(t) = merge(compute(fstQuery(t)),
                                        compute(restQuery(t)))
querySize(t) = 1  ⟹  compute(t) = compute(fstQuery(t))
merge(null, v) = v
```

A query consists of at least one chunk; computing a non-primitive query is the same as merging the result of computing the first query with the computation of the rest of the query; computing a single query chunk is the same as computing the first query of the chunk; merging with **`null`** with some value `v` results in the value `v`.

One possible way to use the server is by constructing a client whose implementation can be seen in Fig. 3. Through the `IServer` interface, the client sends some random query (by means of some built-in method `randomQuery`) to a server when the client is linked to the server. When it receives a response, it does nothing. When a client is created and the client is linked to a server, the client makes a request to the server. Then the server creates a number of worker to handle the query, and the last worker returns the query computation result back to the client.

The running example highlights the features of concurrent objects, namely asynchronous method calls, sequential execution of each method and encapsulation of internal state. Moreover, the example shows unbounded object creation, a non-trivial aspect to handle in verifying functional behavior. The unbounded object creation is caused by the lack of knowledge on the server side with respect to the number of subqueries a query can be split into. Recall that a subquery is handled by a separate worker.

2.3 Discussion

In object-oriented programming, it is common for objects to interact with other objects by field access. In the concurrent setting, however, allowing field access also means that explicit signaling between objects is needed to indicate that an object is being modified. This leads to complex techniques, such as [1,14], to understand what exactly is going on with the objects. Fields in ActJ are *object private*, meaning each object can only see and modify its own fields; even fields of other objects of the same class are inaccessible. Putting the plug on field access allows an object to rely on the stability of the values of its fields during a method execution. More discussion on the mechanisms of concurrent interaction between objects is available in [32].

In general, it is desirable to be able to synchronize the progress between different method call executions (cf. **`await`** construct in ABS). An object is then allowed to *multitask*, meaning that the execution of several messages can be in progress (not just residing in the queue). This concept does not appear in our running example (only partially through the use of **`guard`**). To accommodate this concept, the object idle mode needs to be extended to the state when a method call execution awaits for certain synchronization. Also, objects need to be equipped with a more specific scheduler.

Methods in ActJ do not have return values except the implicit indication that an execution of a method is finished. Returning specific values is emulated by sending a method call. If an object needs any information from other objects or wants to manipulate other objects, it must be done by calling methods of the other objects. A more elegant way to handle returns is to use futures [9] as in ABS.

3 Core Operational Semantics

To characterize the behavior of a concurrent object system, we first look into the behavior of objects and how they interact with each other. Following the design of modeling languages such as Creol [34] and ABS [33], classes provide blueprints how their instances, i.e., the objects, behave. Instead of directly providing the operational semantics based on the code structure, the classes are modeled using parameterized transition systems. This approach allows for a simple run-time configuration model. We use a slightly less faithful model in terms of the communication between objects to capture the behavior of classes. This model is expressive enough for showing the connection between operational semantics of concurrent object systems and their trace semantics. The trace semantics is used as the basis for specification and verification.

3.1 Notation

We first define the notation used in this chapter. Capital letters represent sets. Typical elements of sets and variables are represented by small letters. Constants are written using a typewriter font, usually to indicate the connection with the running example. Exceptions to the notation are clearly noted. A summary of numerous helper predicates and functions used in the semantics description is given in Table 1 as a quick reference.

We use the data structure $\mathit{Seq}\langle\mathbf{T}\rangle$ to represent finite sequences, with $\mathbf{T}$ denoting the type of the sequence elements. An empty sequence is denoted by `[]` and $\cdot$ represents sequence concatenation. The function $\mathit{Pref}(s)$ yields the set of all prefixes of a sequence s. The projection operator $s{\downarrow_T}$ produces the longest subsequence[3] of the sequence s that contains sequence elements in T of type $\mathbf{T}$. The projection operator can be refined by considering the structure of $\mathbf{T}$.

3.2 Classes

We start the description of our formal model by defining the basic sets. Let $\mathbf{O}$ be the set of all object identities, $\mathbf{CL}$ the set of all classes, $\mathbf{M}$ the set of messages that can be communicated between objects and $\mathbf{D}$, disjoint from $\mathbf{O}$, the set of data values. We use object identities to represent both the object and its actual identity. The function $\mathit{class}(o)$ gives the class of an object o. A message m can either be an object creation **`new`** C or a method call $\mathit{mtd}(\overline{p})$, where C is a class, 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 $\mathit{isMtd}(m)$ checks if the message m is a method call. The messages are the observable units of communication exchange between objects.

[3] A sequence s is a *subsequence* of another sequence s' if s can be derived from s' by deleting some elements of s' while preserving the order of the remaining elements [26, p. 4].

Table 1. Helper predicates and functions

Predicate/Function	**Description**
$\mathit{Pref}(s)$	The set of all prefixes of sequence s.
$\mathit{class}(o)$	Returns the class of object o.
$\mathit{isMtd}(m)$	Checks if message m is a method call.
$\mathit{callee}(e)$	Returns the callee of event e.
$\mathit{caller}(e)$	Returns the caller of event e.
$\mathit{msg}(e)$	Returns the message of event e.
$\mathit{acq}(t)$	Returns the accumulated object identities exposed in each method call event in trace t.
$\mathit{cr}(t)$	Returns the set of objects created in trace t.
$\mathit{extMtd}(C)$	Returns the set of method calls that can be received by objects of class C.
$t{\downarrow_O}$	Projects trace t to non-external events of a set of objects O. The operator can also take *callee* or *caller* as an extra parameter.
$\mathit{exposed}(t, O)$	$\mathit{acq}(t{\downarrow_{O,\mathit{callee}}}) \cup \mathit{cr}(t{\downarrow_{O,\mathit{caller}}})$
$\mathit{idx}(t, C)$	Extracts the identities of objects transitively created by the initial object of class C from trace t.
$\mathit{sup}(t)$	Returns the event core sequence of t.
$\mathit{split}(t, L)$	Splits t into input and output traces $(\mathit{ti}, \mathit{to})$ based on local objects L.
$\mathit{bound}(T)$	Extracts the largest visible subset of local objects from a component trace set T.

The set of *events* $\mathbf{E}$ 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 $o_1 = caller(e)$ to the *callee* object $o_2 = callee(e)$. If m is a creation message, o_2 will be the name of the newly created object while o_1 is its creator. Textually an event e is represented as $o_1 \rightarrow o_2.\mathtt{new}\ C$ or $o_1 \rightarrow o_2.mtd(\overline{p})$ when the message is an object creation or a method call, respectively. With respect to some group of objects $O \subseteq \mathbf{O}$, an event $o_1 \rightarrow o_2.m$ can be classified into *internal event*, if $o_1, o_2 \in O$; *external event*, if $o_1, o_2 \notin O$; *input event*, if $o_1 \notin O$ and $o_2 \in O$; and *output event*, if $o_1 \in O$ and $o_2 \notin O$. 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 the callee and message information of an event are needed. Following the design choice of ABS, the caller information is transparent from the callee of a method call. In the same way, an object does not know who creates it, unless this information is passed on explicitly. Therefore, if we focus only on a specific object, the caller information becomes redundant. Eliminating the caller from an event produces an *event core*. The set of event cores $\mathbf{EC}$ are derived from the set of events $\mathbf{E}$ by removing the caller from events in $\mathbf{E}$. The functions *msg* and *callee* applied to event cores are defined as for events.

The class of an object characterizes how the object behaves. One way to model classes is by transition relations. We model classes using two transition relations: one describing what an object does when it receives a message and the other describes what an object does when it reaches a certain state. As the model should reflect common invariants on classes, it also contains a function that tells which objects are known to the class instance. By restricting the model via this function, the class model guarantees, for example, that method calls can only be sent to known objects.

Definition 1 (Class). *A* class C *is a parameterized tuple* $\langle Q, q^0, \rho, \alpha, \kappa\rangle(\mathtt{this})$ *where*

- Q *is the set of states,*
- $q^0 : O \rightarrow Q$ *is the parameterized initial state,*
- $\rho : \mathbf{M} \times Q \times Q$ *is the message receive relation,*
- $\alpha : Q \times \mathbf{EC} \times Q$ *is the action relation, and*
- $\kappa : Q \rightarrow 2^{\mathbf{O}}$ *is the function mapping a state to a set of known objects,*

where for each state $q, q' \in Q$*, message* $m \in M$ *and event core* $\mathsf{e} \in \mathbf{EC}$*, representing* $\langle m, q, q'\rangle \in \rho$ *and* $\langle q, \mathsf{e}, q'\rangle \in \alpha$ *as* $C : (m, q) \rightarrow q'$ *and* $C : q \xrightarrow{\mathsf{e}} q'$*, respectively, and letting* $q \rightarrow^* q'$ *represent transitive closure of* ρ *and* α*, the following holds:*

1. $\kappa(q^0(\mathtt{this})) = \{\mathtt{this}\}$ *(the object initially knows only its own identity);*
2. $q^0(\mathtt{this}) \rightarrow^* q \implies \{\mathtt{this}\} \in \kappa(q)$ *(the object always knows its own identity);*
3. $C : (m, q) \rightarrow q' \implies \kappa(q') \subseteq \kappa(q) \cup acq(m)$
(the object knows another object through incoming method calls)
4. $C : q \xrightarrow{\mathsf{e}} q' \wedge \neg isMtd(msg(\mathsf{e})) \implies \kappa(q') \subseteq \kappa(q) \cup callee(\mathsf{e})$
(or the object knows another object by creating it);
5. $C : (m, q) \rightarrow q' \implies isMtd(m)$
(creation of objects is not part of the class semantics);
6. $C : q \xrightarrow{\mathsf{e}} q' \wedge isMtd(msg(\mathsf{e})) \implies \{callee(\mathsf{e})\} \cup acq(\mathsf{e}) \subseteq \kappa(q) \wedge \kappa(q') \subseteq \kappa(q)$
(the callee and parameters of method calls made by the object must be known);
7. $C : q \xrightarrow{\mathsf{e}} q' \implies callee(\mathsf{e}) \neq \mathtt{this}$ *(no self calls are allowed).*

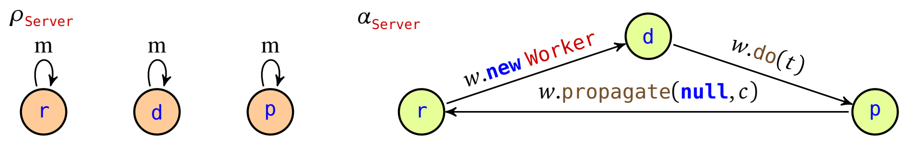


Fig. 4. Graphical representation of the message receive and action relations for `Server` class where $m = \texttt{serve}(c,t)$ and only the internal states are represented

A class $C \in \mathbf{CL}$ is defined as a parameterized quintuple. Q is the set of states an object of class C can have. In general, Q can be an infinite set because an object may have fields that store object references. Furthermore, the object has an internal buffer to manage how incoming messages are processed. $q^0(\texttt{this})$ represents the initial state the object has when it is created. The initial state is parameterized based on the identity of the object given when the class is instantiated. The transition relation ρ describes when the object receives a message and the effect of receiving that message on the object's state. ρ should be given such that any object of this class can only receive messages of certain forms, i.e., the class obeys a certain interface. In other words, any object of this class can only receive messages of certain forms. The transition relation α 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. The function κ describes which other objects an object knows when it is at some state. The single parameter represented by the variable **`this`** is assigned the value o when an object o of class C is created. When necessary, the members of the tuple are indexed with the class name for clarity.

For the class tuple to represent classes expressed by codes written in, e.g., ActJ, we put restrictions that explain the relationship between ρ, α and κ. The first four restrictions state how κ can evolve. Initially an object knows no other object but itself. An object can always refer to its own identity regardless of what kind of transition it previously has. It corresponds directly to the keyword **`this`** in ActJ. As in the programming language, there are two ways how an object o may know another object o', namely through references passed on as parameters and by creating o'. Once the reference o' is available to o, o can decide whether to store this information. As storing this information (through variable assignments) may cause o to forget another reference that is previously stored, the evolution κ is described using the subset relation. The freshness of a newly created object and its assignment of initial state are handled within the operational semantics, as hinted by the fifth restriction. 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. After making a method call, the object may forget some references to other objects. The seventh restriction on no self call acts as a simplification for easier classification of events into input or output events. Changes that occur with a self call can be simulated by making a nondeterministic choice of states as the effect of receiving or sending a message.

We assume that classes are defined such that their instances are input-enabled with respect to some consistent interface. The message part used in ρ and α is assumed to always be well-typed. The function $\mathit{extMtd}(C)$ extracts the interface supported by C. The function returns the set of method call messages that objects of class C can receive.

Example 1. The Server class can be represented as follows. Each state in $Q_{\texttt{Server}}$ is partitioned into five parts: internal state $q_i \in \{\texttt{r}, \texttt{d}, \texttt{p}\}$ representing ready, do, and propagate states, respectively; the client identity c; the worker identity w; the query y; and the internal message queue u. We use the sequence data structure $Seq\langle \mathbf{M} \rangle$ to represent the message queue. By virtue of the internal message queue and the infinite number of object identities, $Q_{\texttt{Server}}$ is an infinite set.

The initial state is $\langle \texttt{r}, \textbf{null}, \textbf{null}, \textbf{null}, [\,] \rangle$. The message receive relation $\rho_{\texttt{Server}}$ is represented symbolically by

$$\texttt{Server} : (\texttt{serve}(c', y'), \langle q_i, c, w, y, u \rangle) \rightarrow \langle q_i, c, w, y, u \cdot \texttt{serve}(c', y') \rangle \ .$$

Formally this representation is translated as the following relation:

$$\{\texttt{Server} : (m, q) \rightarrow q' \mid \exists c', y', q_i, c, w, y, u \bullet m = \texttt{serve}(c', y') \wedge \\ q = \langle q_i, c, w, y, u \rangle \wedge q' = \langle q_i, c, w, y, u \cdot \texttt{serve}(c', y') \rangle \} \ .$$

Stated in words, the server can always receive a request, which is then stored in the internal message queue. Other parts of the server's state remain the same. Applying *extMtd* to Server results in

$$\{\texttt{serve}(c, y) \mid \exists c, y, q, q' \bullet \texttt{Server} : (\texttt{serve}(c, y), q) \rightarrow q' \} \ .$$

This set represents the interface IServer that the Server class is implementing.

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

A1. $\texttt{Server} : \langle \texttt{r}, \_, \_, \_, \texttt{serve}(c, t) \cdot u \rangle \xrightarrow{w.\textbf{new}\ \texttt{Worker}} \langle \texttt{d}, c, w, t, u \rangle$
A2. $\texttt{Server} : \langle \texttt{d}, c, w, t, u \rangle \xrightarrow{w.\texttt{do}(t)} \langle \texttt{p}, c, w, t, u \rangle$
A3. $\texttt{Server} : \langle \texttt{p}, c, w, t, u \rangle \xrightarrow{w.\texttt{propagate}(\textbf{null}, c)} \langle \texttt{r}, c, w, t, u \rangle$

An underscore represents any value and is used when the value of the particular part of the state is of no significance. The first relation states that the server is in the internal state of r where it is ready to process another serve message provided one is present in the internal message queue. As can be seen in Fig. 2, the server then creates a new worker. In this situation, it is important for the next steps that the server remembers the query, the client and worker references. The second and third relations proceed with the execution of the method serve, where d and p act similar to a program counter. All in all, this relation mimics the serve method of class Server given in Fig. 2. Graphically, $\rho_{\texttt{Server}}$ and $\alpha_{\texttt{Server}}$ can be viewed as in Fig. 4, by focusing only on the internal states.

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

K1. $\kappa(\langle \texttt{r}, \_, \_, \_, \_ \rangle) = \{\textbf{this}\}$
K2. $\kappa(\langle \texttt{d}, c, w, \_, \_ \rangle) = \{\textbf{this}, c, w\}$
K3. $\kappa(\langle \texttt{p}, c, w, \_, \_ \rangle) = \{\textbf{this}, c, w\}$

In r, the server forgets specific information about all previous requests that came in. When the server is processing a request, however, it is important to keep track of the client and worker references as explained previously. Thus in d and p, both the client and the worker references are tagged as known.

Discussion. Another way to describe the behavior of the objects is by assigning a behavior instance to each object. The change in behavior of object after receiving or producing a message is defined by a global relation (see, e.g., [61,4]). This approach has the drawback of giving less structure to play with. By having classes as templates, the behavior of objects can be defined without having to provide a global relation that works for all objects. See [13] for more comparison between object-based and class-based approaches on defining object's behavior.

Din et al. [20] introduce 4-event semantics to faithfully capture asynchronicity (i.e., allowing network delay between sending and receiving), similarly to the dual send and receive in CCS [41] and π-calculus [42]. In this semantics, the sending and the actual start of the method calls or creation are split into different event types. This distinction plays a role in the specification part, as it allows specifications to be represented locally at the cost of a more complex well-formedness condition on the generated traces.

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 assume that the action relation α follows an actor model with multitasking, such as ABS [33].

3.3 Operational Rules

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 objects $O \subseteq \mathbf{O}$ is a set of configurations $\mathcal{C}$ 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. $\langle \texttt{Server}, s, \langle \texttt{r}, \texttt{null}, \texttt{null}, \texttt{null}, \texttt{serve}(c_1, y_1) \cdot \texttt{serve}(c_2, y_2)\rangle\rangle$ is a possible configuration for a `Server` object s. This states that s has two incoming `serve` method calls and currently is in the initial state to process the next incoming request.

The way the run-time configuration is affected by interaction between objects is 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. In each of these rules, the transition from one configuration $\mathcal{C}$ to another $\mathcal{C}'$ is labeled by the corresponding event e that is represented

$$\textsc{ObjectCreation}\quad \frac{C_1 : q \xrightarrow{\mathsf{e}} q' \qquad \mathsf{e} = o_2.\mathtt{new}\ C_2 \qquad o_2\ \mathit{fresh}}{\{C_1\ o_1 : q\} \uplus \mathcal{C} \xrightarrow{o_1 \to o_2.\mathtt{new}\ C_2} \{C_1\ o_1 : q', C_2\ o_2 : q^0_{C_2}(o_2)\} \uplus \mathcal{C}}$$

$$\textsc{MessageSend}\quad \frac{C_1 : q_1 \xrightarrow{o_2.mtd(\overline{p})} q_1' \qquad C_2 : (mtd(\overline{p}), q_2) \to q_2'}{\{C_1\ o_1 : q_1, C_2\ o_2 : q_2\} \uplus \mathcal{C} \xrightarrow{o_1 \to o_2.mtd(\overline{p})} \{C_1\ o_1 : q_1', C_2\ o_2 : q_2'\} \uplus \mathcal{C}}$$

Fig. 5. Core operational semantics of concurrent object systems

by the rule. Textually, each transition has the form $\mathcal{C} \xrightarrow{e} \mathcal{C}'$. The disjoint union operator $\uplus$ ensures the consistency of having a single configuration of one object within C. When there are multiple ways to apply the rewrite rules, one is chosen at random.

ObjectCreation is applicable when an object o_1 is in a state q where it can create a new object of some class C_2 according to its class action relation. The result is that a new object o_2 with fresh identity is created and added into the configuration. The state of this new object is the initial state $q^0_{C_2}(o_2)$ of its class description.

MessageSend states the changes to two parties that act as end points of a method call (i.e., the caller and the callee). An object o_1 (the caller) can perform an asynchronous method call $mtd(\overline{p})$ on another object o_2 (the callee) if the following conditions hold. First, the caller is in a state where it can send a message $mtd(\overline{p})$ to the callee. By the condition placed on the class definitions, the caller knows the callee. Second, the callee is able to receive that message. With the input-enabledness assumption in place, this part is always fulfilled. Thus, the resulting states q_1' and q_2' of both the caller and the callee, respectively, can be derived when the method call is made.

A straightforward consequence of the operational semantics is that an object always keeps its class designation as shown by the lemma below.

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

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

Being able to refer to specific entities with regards to the behavior of objects is useful later on to structure the behavior of a group of objects without needing extra constructs. Class preservation of objects serves as an important basis to this usage.

Discussion. The operational semantics above has only two rules. The rules cover the necessary observable operations on a concurrent object model (cf. [58]). The internal computation needed to produce the messages is abstracted away in our model by means of object states. Vasconcelos and Tokoro [61] even reduced the number of kinds of observable operations into one, by just considering the method calls (which in their work are simply called *messages*). Object creation is encoded implicitly within the internal computation. In a single transition, the number of objects that are present in the configuration can change. In our case, the object creation needs to be considered separately to extract the initial state from the class definition.

The MESSAGESEND rule pairs the action relation of one class to the message receive relation of another class. Between pairs of objects o and o', the order of how methods are called by o to o' is the same as the order of the same calls received by o'. The actor model [16,3,4] retains pure asynchronicity, meaning that it allows message overtaking even between pairs of objects. Yonezawa, Briot and Shibayama [63] argued, however, that having this guarantee eases describing distributed algorithms.

As each object can concurrently perform their internal processing at possibly different speeds, there is a need for external nondeterminism. This need is covered 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. As this issue is not prominent for our discussion on specification and verification (unlike in the discussion on actors [4], for example), we assume weak fairness on the operational rule application.

3.4 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 concatenation of the labels of configuration transitions. If we know the default initial configuration, the traces can be used to abstract from the internal representation of the individual objects [30,12]. For this reason, we use traces as semantic foundation for specifications.

The following definition states what we mean by executions and traces. The symbol $\surd$ distinguishes a *maximal* execution (i.e., a finished execution where no more operational rule can be applied [8, p. 96]) from an execution that can still make progress.

Definition 3 (Execution and trace). *An* execution *is a sequence of interleaved configurations and events $\mathcal{C}_0 \xrightarrow{e_1} \mathcal{C}_1 \xrightarrow{e_2} \mathcal{C}_2 \cdot \ldots$ 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. The trace t ends with $\surd$ if the corresponding execution ends with $\surd$.*

Example 3. Let a be some `Main` object which represents the initial object whose task is to set up the server system. Assume as well a `Client` object c has been created. Using an underscore to represent a configuration content of no interest, the following execution describes a possible way how the server is created and how it responds to an incoming request from the client.

$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_\} \xrightarrow{a \to s.\textbf{new}\ \texttt{Server}}$$
$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_, \texttt{Server}\ s:\langle \texttt{r},\_,\_,\_,[\,]\rangle\} \xrightarrow{a \to c.\texttt{link}(s)}$$
$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_, \texttt{Server}\ s:\langle \texttt{r},\_,\_,\_,[\,]\rangle\} \xrightarrow{c \to s.\texttt{serve}(c,y)}$$
$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_, \texttt{Server}\ s:\langle \texttt{r},\_,\_,\_,\texttt{serve}(c,y)\rangle\} \xrightarrow{s \to w.\textbf{new}\ \texttt{Worker}}$$
$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_, \texttt{Server}\ s:\langle \texttt{d},c,w,y,[\,]\rangle, \texttt{Worker}\ w:\_\} \xrightarrow{s \to w.\texttt{do}(y)}$$
$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_, \texttt{Server}\ s:\langle \texttt{p},c,w,y,[\,]\rangle, \texttt{Worker}\ w:\_\} \xrightarrow{s \to w.\texttt{propagate}(\textbf{null},c)}$$
$$\{\texttt{Main}\ a:\_, \texttt{Client}\ c:\_, \texttt{Server}\ s:\langle \texttt{r},\_,\_,\_,[\,]\rangle, \texttt{Worker}\ w:\_\}$$

This execution shows six transitions. The first two transitions complete the setup of the setting, by having the server object created and having it linked to the client. The ObjectCreation and MessageSend rules are respectively applied by assuming that the states of the main and client objects are the state where they are applicable. The third transition states that the client sends a request to the server. The server stores this request in its internal queue. Then the server processes this request in the remaining transitions as described by the server action relation given in Ex. 11. First it creates a new worker and takes out the `serve` message out of the its queue. As an effect of the transition, the new worker reference is stored as well as the client reference and the query from the `serve` message. The internal state also changes to `d`o. Then, it initiates the result propagation to this new worker. The server is then back to the state where it is ready to process another request (although no more pending requests are present). In the last two transitions, we assume that the worker state changes accordingly.

The following trace can be extracted from the execution.

$a \rightarrow s.\mathtt{new}\ \mathtt{Server} \cdot a \rightarrow c.\mathtt{link}(s) \cdot c \rightarrow s.\mathtt{serve}(c,y) \cdot s \rightarrow w.\mathtt{new}\ \mathtt{Worker} \cdot s \rightarrow w.\mathtt{do}(y) \cdot s \rightarrow w.\mathtt{propagate}(\mathtt{null},c)$

This trace is not maximal, because the operational rules can still be applied to the worker object for processing the query.

To obtain specific information related to a certain object o from a trace t, we use the projection operator $t{\downarrow}_o$. This operator states 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 naturally lifted to a trace set T and a set of objects O. Other operators are introduced as needed.

Example 4. Let t be the trace from the previous example. Then if we project t to the server object s, then we obtain the following trace.

$$t{\downarrow}_s = a \rightarrow s.\mathtt{new}\ \mathtt{Server} \cdot c \rightarrow s.\mathtt{serve}(c,y) \cdot s \rightarrow w.\mathtt{new}\ \mathtt{Worker} \cdot s \rightarrow w.\mathtt{do}(y) \cdot s \rightarrow w.\mathtt{propagate}(\mathtt{null},c)$$

An important restriction on the class definition (Def. 1, No. 6.) is that a method call can be made only if the objects in the parameters of the method call are known. This restriction can be transferred to traces by requiring that an object can only send messages to other objects it has been exposed to. A trace contains enough information to determine whether an object is exposed to another object. This information is extracted using the functions *acq* and *cr*. The function $cr(b.\mathtt{new}\ C)$, 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 e be a method call event $o \rightarrow o'.mtd(\overline{p})$. 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. These identities are collected from previous method calls the caller receives (through the acquaintance function) and the objects that have been created directly by the caller (represented by the created function). Other properties of trace well-formedness, such as the freshness of the newly created objects, can be defined in a similar way. We leave their definition to the reader as an exercise[4]. Because there is no self-call and an object only knows its own identity when it is created as per restriction on κ (Def. 1), 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 κ function (Def. 1), the generated trace(s) is well-formed.

Lemma 2 (Well-formedness preservation). *Let* $\mathbf{CL}$ *be a set of classes and the singleton* $\mathcal{C} = \{C\ o : q^0_C(o)\}$ *the initial configuration of object* o *of class* $C \in \mathbf{CL}$*. Then, by applying the operational rules from Fig. 5, the generated trace is well-formed.*

Proof (sketch). Two important factors for the proof are the monotonicity of the functions *acq* and *cr*, and that no object identities absent from the events are introduced by these functions. Because these factors are weaker than the restrictions on κ in Def. 1, the generated trace is well-formed.

Discussion. We have introduced two different semantics for our objects: the operational semantics and the trace semantics. These semantics are chosen because they represent the two common abstraction levels used for reasoning. The operational semantics allows a program to be executed with respect to some configuration. Trace semantics typically exhibits full abstractness quality as shown in various concurrent models, e.g., [35,2,44]. It is attractive as the basis for specifying the functional property of a system, because the specification can independently be given without determining how it is implemented.

Talcott [58] provided more layers of composable semantics of actors to allow local reasoning at different levels of abstractions. Starting with an operational semantics similar to ours (Sect. 3.3), other layers are formed by hiding external events belonging to the environment, focusing only on the partial order between events, and hiding internal events. Her approach relies on retaining possible global timings of each event.

4 Systems and Components

Having a semantics for a group of objects is the basis for understanding how a system behaves. As motivated in Sect. 1, a natural way to structure the grouping is to use classes to form components. A system's behavior can be understood by composing the behavior of its components. In this section, we explore the possible ways to structure groups of objects into components and link them with the notion of open systems.

[4] For example, Din et al. [20] provides such a definition.

4.1 Closed Systems

First, we need to know whether we have enough 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* $C \in \mathbf{C}$*, it holds that* $C' \in \mathbf{C}$.

A set of classes is definition-complete if for each object present within the interaction, the class definition of that object is also available 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 contains all the class definitions necessary for knowing precisely how each object within the system behaves. An activator class is 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 (Trace semantics of closed systems). *The trace semantics of a closed system* $CS = (\mathbf{C}, C_0)$ *is the trace set* $Traces(CS)$ *where for each* $t \in Traces(CS)$*, there is an execution starting from an initial configuration* $\mathcal{C} = \{C_0\ o : q^0_{C_0}(o)\}$ *whose trace is* t.

The trace semantics of a closed system is a trace set containing all traces that can occur from a singleton initial configuration of an object of the activator class. Using the closed system definition, we can define the trace semantics of a class.

Definition 8 (Trace semantics of classes). *The trace semantics of a class* C *is the trace set* $Traces(C)$ *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 systems that contain C, then projecting all those traces down to objects of class C.

Example 5. A trace of the `Server` class is $a \to s.$`new Server` $\cdot\ c \to s.$`serve`$(c,t) \cdot s \to w.$`new Worker` $\cdot\ s \to w.$`do`$(t) \cdot s \to w.$`propagate`$($`null`$, c)$. This is obtained by taking the projection of the trace from Ex. 3 to the `Server` object s.

Discussion. The trace semantics can as well be directly 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. Ahrendt and Dylla [5], for example, apply *guess and merge* approach when synchronizing the execution of different method calls to obtain the trace set.

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. A gap is present between single classes and closed systems when we focus our attention only on the behavior of some particular group of objects of a closed system. When this group expands dynamically as the system continues to execute (e.g., the workers of the server that handle a request in our running example), it is impractical to know how this group as a whole behaves by referring to the behavior of each of the objects in the group. In turn, it is impractical to verify whether the behavior of a closed system follows directly from the class descriptions contained in that system.

To help reason about the behavior of closed systems from single classes, we abstract a collection of single classes into *components*. Components should share the characteristics of closed systems and single classes and allow hiding internal behavior.

In this chapter, we choose an abstraction based on object creation. This abstraction makes it possible to refer to a component by the class of the component's initial object. Its advantage lies in the inference how the component is structured. That is, the structure of a component comes with how the classes behave, instead of needing to state individually which classes and subcomponents are contained in the component.

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

A set of classes is creation-complete if for each created object, its class is within that set. In comparison to definition-complete (Def. 5), creation-complete allows the possibility of having the behavior of some objects unknown. In the context of the interaction of a group of objects, the creation-complete notion allows the behavior of some objects whose references are passed on within the interaction to be left *open*.

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. A set only consisting of a single class is possibly not a component because it may create an object of a different class.

Example 6. The pair $(\{\texttt{Server}\}, \texttt{Server})$ is not a component because a server may create a worker, and the class `Worker` is not within the set of classes. The pair $(\{\texttt{Worker}\}, \texttt{Worker})$ 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: $(\{\texttt{Server}, \texttt{Worker}\}, \texttt{Server})$. Note that none of these components are closed systems because they are missing the `Client` class.

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 definition-complete and* $C_0^x \in \mathbf{C}^x$.

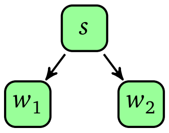
A context is a set of classes with some activator class that completes the class definition of a component. A context may use the same classes as the component, so the component can have certain expectation about the behavior of the context. But since we are following the programming to interfaces principle (see Sect. 2.1), the component generally does not know how objects of the context behave other than they adhere to certain interfaces. A context on its own does not need to be definition-complete or creation-complete. Paired with a matching component, we get a closed system.

Example 7. Let `Main` be a class whose instances create a client and a server and link the server to the client. Then, the pair $(\{\texttt{Client}, \texttt{Main}\}, \texttt{Main})$ is a context of the component $(\{\texttt{Server}, \texttt{Worker}\}, \texttt{Server})$.

Similar to the trace semantics of closed systems and classes, 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. These objects can be derived by following an object creation tree whose root is the initial object of the component. The object creation tree keeps track of objects transitively created by the initial object of the component. Instead of formally defining the tree, we illustrate it using the following example.

Example 8. Consider the server component $(\{\texttt{Server}, \texttt{Worker}\}, \texttt{Server})$ given in Ex. 6. The following trace illustrates the creation of the server object s and its (incomplete) reaction to two requests. All creation events are highlighted.

$a \to s.\texttt{new Server} \cdot a \to c.\texttt{link}(s) \cdot c \to s.\texttt{serve}(c, t_1) \cdot$
$s \to w_1.\texttt{new Worker} \cdot s \to w_1.\texttt{do}(t_1) \cdot s \to w_1.\texttt{propagate}(\texttt{null}, c) \cdot$
$c \to s.\texttt{serve}(c, t_2) \cdot s \to w_2.\texttt{new Worker}$



Each time the server processes a request from the client, it creates a worker. Taking s as the root of the tree, the tree has also nodes containing the workers w_1 and w_2. Graphically, this tree is shown to the right of the trace.

By virtue of the object creation tree, the objects of the component with respect to some well-formed trace can be tracked using the *identity extractor* function.

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.*

$$idx([], C) = \emptyset,$$

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

The function *idx* extracts the identities of objects directly and indirectly created by the initial object of the component. The function takes as arguments a trace t and a class C. It works by determining the initial instance o of C in the trace. This initial instance is the root of the object creation tree. Then the function recursively collects all the objects transitively created by o.

In a trace, there can be more than a single instance of C being created. For defining the behavior of a component, only the first instance of C and other instances transitively created from that first instance are returned by *idx*. This restriction ensures that an object creation tree can be constructed from the result of applying *idx*. We lift *idx* to a trace set T by taking the union of the application of *idx* to each trace in T.

Example 9. Let t be the trace given in Ex. 8. Then, $idx(t, \texttt{Server}) = \{s, w_1, w_2\}$.

Using the object identity extractor function, we are ready to define the first variant of the component trace semantics. We call this variant 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.

Definition 13 (Plain trace semantics of components). *Let $\mathbb{C} = (\mathbf{C}, C_0)$ be a component. The* plain trace semantics *of $\mathbb{C}$ is a trace set $Traces(\mathbb{C})$ 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 $t = t'\downarrow_{idx(t',C_0)}$.*

Implicit within the definition is the usage of object creation tree to build the component instances. Discussion on other ways to define the component instances is deferred to 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.

This trace semantics supplies enough information to link the operational semantics to an openness property of the trace semantics. This property is crucial for proving the soundness of the proof system presented in Sect. 5. 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 [37], where the objects of the component instance are grouped into L (for *l*ocal) and the objects of the context (i.e., objects not part of the component instance) are grouped into F (for *f*oreign). The local objects can be extracted from the plain trace set of a component by applying the identity extraction function. For defining the openness property, the exact content of L is left open.

Definition 14 (Open system trace sets [37]). *Let T be a trace set 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 $exposed(t, F) = cr(t\downarrow_{F,caller}) \cup acq(t\downarrow_{F,callee})$.

We call a trace set *open* with respect to a set of local objects if for each trace in the trace set, a method call event directed to an exposed local object can be constructed using the exposed local objects and the context objects. Appending that method call to the end of the trace results in another trace which is contained in the trace set. The exposed references are derived by taking all local objects created by the context and all local objects whose references are passed on to the context through method calls.

To instantiate this property in our trace semantics we assume without loss of generality that the identity of the initial object of the component is fixed, 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 and the trace semantics, with respect to the openness property.

Lemma 3 (Openness of plain trace semantics of components). *If* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component, then* $\mathit{Traces}(\mathbb{C})$ *is open.*

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

Proof. Let $t \in \mathit{Traces}(\mathbb{C})$ be achieved by composing $\mathbb{C}$ with some context $\mathbb{X} = (\mathbf{C}^x, C_0^x)$. Let also $L = \mathit{idx}(\mathit{Traces}(\mathbb{C}, C_0))$, $F = \mathbf{O} - L$, and $o' \in \mathit{exposed}(t, F) \cap L$. Because of the interface model[5], we can assume without loss of generality 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 create C' where we pick some new method name mtd' that does not appear in any class and add as many states as necessary such that

1. for all $o \in \kappa_{C'}(q), o'' \in F$, $C' : q \xrightarrow{o''.\mathit{mtd}'(o)} q'$
   (the knowledge is shared among all objects of the context),
2. $C' : (\mathit{mtd}'(o), q) \to q'$ *(all objects of the context remain input-enabled)*, and
3. if $o' \in \kappa_{C'}(q)$, then for all $\mathit{mtd} \in \mathit{extMtd}(\mathit{class}(o'))$,
   $C' : q \xrightarrow{o'.\mathit{mtd}(\overline{p})} q'$ such that $\mathit{acq}(\overline{p}) \subseteq \kappa_{C'}(q)$
   (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).

It can be checked that C' follows the class definition (Def. 1) and $\mathbb{X}'$ remains a context (Def. 11). By the operational rules (Sect. 3.3) and Def. 13, 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 $t \cdot o \to o'.\mathit{mtd}(\overline{p}) \in \mathit{Traces}(\mathbb{C})$. □

[5] Not following the programming to interfaces principle brings a problem called *replay* [56]. The replay problem appears when the component exposes some object to an object of the context whose class is part of the component. Because the implementation of the object is fixed, that context object can only use the exposed component object in a specific way. In particular, this context object may only store the exposed component object and openness is not achieved.

As stated earlier, we want to bridge classes and closed systems. This goal is achieved by hiding the *internal* behavior that happens between objects within the component. Such a view allows bottom-up reuse of the component, for example, when it 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 [48] for more discussion). By using *idx*, hiding is straightforward to define.

Definition 15 (Component trace semantics). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component and* $Traces(\mathbb{C})$ *its plain trace semantics, so that* $L = idx(Traces(\mathbb{C}), C_0)$ *is the set of local objects of* $\mathbb{C}$. *The* component trace semantics*, written* $Traces([C_0])$*, is the trace set* $Traces(\mathbb{C})\downarrow_{\mathbf{O}-L}$.

To distinguish the plain trace semantics of components (Def. 13) 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. A component instance can be thought of as a box of objects starting from the initial instance of the activator class C_0. By projecting the traces to the set of objects of the context, the above definition only reflects the *observable* behavior representing the interaction with the context. The component's users only need to focus on the component's observable behavior. For example, the client is only interested in how the server component (as a whole) interacts with it, not in how the server does its job. As the projection does not change the interaction that happens on the boundary of the component, the following corollary holds.

Corollary 2. *If* $(\mathbf{C}, C_0)$ *is a component, then* $Traces([C_0])$ *is 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. $\forall e \cdot t \in Traces([C_0]) \bullet msg(e) = \mathtt{new}\ C_0$
   (trace begins with creation of an instance of C_0*)*
2. $\forall e \cdot t \cdot e' \in Traces([C_0]) \bullet isMtd(msg(e'))$
   (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 projected away. □

Discussion. The term *activator class* used in this chapter comes from OSGi [60]. In OSGi model, a component is instantiated by creating an object of a class that implements the `BundleActivator` interface, an interface each component is required to implement. The model where the component is instantiated by creating an object of the activator class is not uncommon. For example, it coincides with the object adaptor of the CORBA Component Model [46] and the class factory of COM [40]. 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.

The main question with defining the behavior of components is how we separate the internal and observable behavior. For this purpose, the notions of *boundary* enclosing the objects of a component at run-time and encapsulation are important. With our component model, we identify three particular ways to group objects into instances of a component or run-time component. Following the component definition (Def. 10), we identify three particular ways to group objects into instances of a component, called *run-time components*: static, programmer-defined and dynamic. The division between static and dynamic stems from how the objects created during execution are organized within the component instances.

Static run-time component Given a component $(\mathbf{C}, C_0)$, its static run-time representation contains every object of whose class is in $\mathbf{C}$. 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 whose class in $\mathbf{C}$, every object is at the boundary. By having all objects at the boundary, the interactions between these objects are visible. In addition, components 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 $(\{\texttt{Server}, \texttt{Worker}\}, \texttt{Server})$ 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. Not only the focus on the activator class is lost, it is also difficult to specify the desired behavior of the component.

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 CoBox [53], this is done by extending the **`new`** statement with **`in`** *o* to say that the newly created object resides in the same run-time component as *o*. Various other ownership approaches can also be used (see [15] for a survey). This approach is the most flexible as it provides fine-grained information which objects are at the boundary, but also more complex to handle.

The task of defining the exact nature of the run-time components can also be deferred to the specification part. For example, Roth [50, Chap. 4] defines a *Reach* predicate to indicate whether some object o is reachable from o' through field and array accesses (although recall that concurrent objects are not allowed to directly access fields of other objects). While staying on the specification level, this approach offers more precision in stating the current content of some run-time component (i.e., allowing to state that some component may forget a reference).

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 is less specific than the programmer-defined

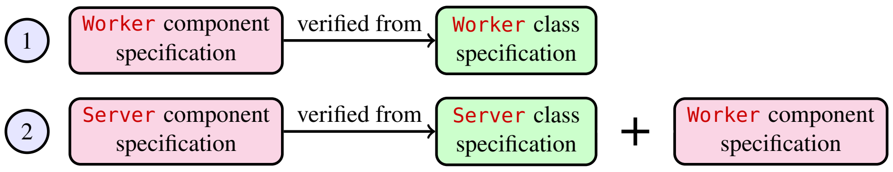

1
Worker component specification
verified from
Worker class specification
2
Server component specification
verified from
Server class specification
+
Worker component specification

Fig. 6. Verification flow of the `Server` component

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 of the activator class to represent the component.

Our notion of context is similar to the notion of program closure in [50, Chap. 2], where it is defined as the minimal set of additional class skeletons needed for a set of classes to compile. The closure is used to create an observer that checks whether method call invocations maintain the program contract. In our setting, it is not necessary for the context to be the minimal set because traces involving external events are filtered out by the projection. That is, the form of this closure is not important.

5 Specification and Verification

Using the semantics of object classes and run-time component characterization based on the object creation tree, we now specify and verify the functional behavior of the components. As stated in the introduction, the main goal is to achieve verification of component specifications from the specifications of the class specifications. Component specifications that have been verified can be used to verify higher level components.

In terms of our running example, this means we give specifications of the functional behavior of `Server` and `Worker` classes and of the `Server` and `Worker` components. In this chapter, we assume the `Server` and `Worker` class specifications to hold, so we can use them directly in our verification effort. To give a complete verification, the `Server` and `Worker` class implementations given in Sect. 2 must be verified against their respective class specifications. As the main goal is to prove the `Server` component property, we may proceed the verification in two steps, as illustrated by Fig. 6. First, we verify the `Worker` component specification from the `Worker` class specification. Then, we use the verified `Worker` component specification together with the `Server` class specification to show that the `Server` component specification holds.

In this section, we describe a specification and verification technique to illustrate how this goal can be achieved. We provide the specifications for the classes and the components, but we only illustrate the second part of the verification effort (i.e., proving the server component specification holds)[6].

[6] Interested readers can attempt the first part. A complete proof is available in [37].

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 [29]. 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 variable* \$ can be used. The trace variable represents the *caller suppressed* trace. This suppression, which yields an event core sequence, reflects the lack of knowledge on the receiver side (i.e., the entity we are specifying) who the sender of a message is. To achieve this suppression, we let the function *sup* transform a trace into caller suppressed trace. Wherever clear, we use caller suppressed trace and normal trace interchangeably.

Definition 16 (Trace assertion). *Let \$ be a trace variable representing a trace.* Trace assertions p, q *are defined inductively by the following first-order logic clauses:*

- *Expressions of boolean type are assertions (\$ may be present).*
- *If* p, q *are assertions and* $\underline{\mathtt{x}}$ *is a variable, then* $\neg p, p \wedge q, \exists \underline{\mathtt{x}} : p$ *are also assertions.*

Other logical operators, e.g., $\vee$, $\Longrightarrow$ and $\forall$, are derived in the usual way. For a trace assertion p, the function *free*(p) returns the set of all free variables appearing in p.

To define the semantics of a trace assertion, we substitute all occurrences of the trace variable 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 variable 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 map the assertion to boolean values $\{\mathbf{true}, \mathbf{false}\}$. Given a trace t, we write $\models_{FOL} p[\$/sup(t)]$ if it is mapped to **true**, and $\models_{FOL} p$ if for any trace t, $\models_{FOL} p[\$/sup(t)]$. The *FOL* index indicates that the variable assignment is done using the first-order logic semantics.

Example 10. $\$ = \langle \underline{\mathtt{this}}.\mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{y}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}})\rangle$ 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 query computation request followed by a result propagation. Only this sequence appears in the trace.

Using trace assertions as the foundation, a specification triple $\{p\}\ D\ \{q\}$ is described as follows. The triple specifies the output trace the instance of class or component D produces when faced with an input trace satisfying 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 is referred to by the variable $\underline{\mathtt{this}}$. The precise trace semantics of the entity represented by D is as follows. For each trace $t \in Traces(D)$ 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. The function $split(t, L) = (t{\downarrow_{F,caller}}{\downarrow_{L,callee}}, t{\downarrow_{L,caller}}{\downarrow_{F,callee}})$ does exactly so, where $F = \mathbf{O} - L$. The first part produces the input trace of t by focusing on events in t where the caller is a foreign object and the callee is the local object. The second part produces the output trace of t in a similar way.

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 in the input and output traces that are being considered in the semantics of the specification triple, only the interaction done by o appears.

Definition 17 (Class triple semantics). *Let C be a class and o an object such that $class(o) = C$. $Traces(C)$ satisfies $\{p\}\ C\ \{q\}$, written $\vDash \{p\}\ C\ \{q\}$, if for all maximal traces $t \in Traces(C)$ with $split(t, \{o\}) = (ti, to)$ the following holds:*

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

Example 11. 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 query and tells it to start propagating the result.
$\{\ \$ = \langle \underline{\texttt{this}} := \textbf{new}\ \texttt{Server} \cdot \underline{\texttt{this}}.\texttt{serve}(\underline{\texttt{c}}, \underline{\texttt{y}})\rangle\ \}$
`Server`
$\{\ \exists \underline{\texttt{w}} \bullet \$ = \langle \underline{\texttt{w}} := \textbf{new}\ \texttt{Worker} \cdot \underline{\texttt{w}}.\texttt{do}(\underline{\texttt{y}}) \cdot \underline{\texttt{w}}.\texttt{propagate}(\textbf{null}, \underline{\texttt{c}})\rangle\ \}$
From Def. 17 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. The trace set may include traces where all output events appear before the input events. In this case, the specification allows a worker to be created by the server before the server receives any request. However, the order of the output events must be as specified by the output trace assertion. This imprecision is as expected because the specification abstracts from the actual behavior of the implementation. In particular, 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 needs to be extracted from the semantics. From Lemma 4, in any trace of $Traces([C])$ 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 18 (Boundary extraction). *Let T be a component trace set. $L' = bound(T)$ is the subset of objects of the run-time component, where*
$bound(T) = \bigcup_{t \in T} bound(t)$, $bound([]) = \emptyset$, and

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

This function works similarly to the *idx* function (Def. 12), but it deals directly with a component trace set T as defined in Def. 15. The local object extraction of T is done by analyzing each trace t in T and combining the result of each analysis. 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. Note that it is necessary to exclude foreign objects that were exposed to the component which is done by $acq(t) - bound(t)$. Using the boundary extractor function above, the semantics of $\{p\}\ [C]\ \{q\}$, called a *component triple*, is defined as follows.

Definition 19 (Component triple semantics). *Let* $[C]$ *represent a component and* $B = bound(Traces([C]))$ *the boundary objects of the component.* $Traces([C])$ *satisfies* $\{p\}\ [C]\ \{q\}$*, written* $\vDash \{p\}\ [C]\ \{q\}$*, if for all maximal traces* $t \in Traces([C])$ *with* $split(t,B) = (ti,to)$ *the following holds:*

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

Example 12. 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 subqueries as a whole. This property can be specified as follows.
$\{ \$ = \langle \underline{\mathtt{this}} := \mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{y}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}},\underline{\mathtt{c}})\rangle \}$
$[\mathtt{Worker}]$
$\{ \$ = \langle \underline{\mathtt{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\underline{\mathtt{y}})))\rangle \}$
Similar to Ex. 11, the semantics of the specification above only deals with the maximal traces. The input trace consists of creating a new `Worker` object, obtaining the request to do a query and then propagating the computation result. When the input part is satisfied, the worker component produces a response back to the client by computing the whole remaining subqueries (following the assumption on `compute` given in Sect. 2.2) and merging it with the given partial result.

Usually, we want to cover as much as possible of 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 20 (Specifications). *Let* D *be a class or a component. A specification for* D *is a set of specification triples* $S = \{\{p_1\}\ D\ \{q_1\},\ldots,\{p_n\}\ D\ \{q_n\}\}$*.* $Traces(D)$ *satisfies* S*, written* $\vDash S$*, if* $\forall(\{p_i\}\ D\ \{q_i\}) \in S \bullet \vDash \{p_i\}\ D\ \{q_i\}$*.*

Example 13. The worker class is described using two specification triples, each handling the base and inductive cases, respectively. The first triple handles the case when the query has exactly one subquery. In this case, the worker sends back to the client the result of merging the propagated result value with the computation of the subquery.
$\{ \$ = \langle \underline{\mathtt{this}} := \mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{y}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}},\underline{\mathtt{c}})\rangle \wedge \mathtt{size}(\underline{\mathtt{y}}) = 1 \}$
$\mathtt{Worker}$
$\{ \$ = \langle \underline{\mathtt{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\underline{\mathtt{y}})))\rangle \}$

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

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

BoxedComposition
$$\frac{\begin{array}{c} \{p \wedge \underline{\mathrm{i}} = \$\}\ C\ \{q \wedge \mathit{noSelfExp}(\underline{\mathrm{i}}, \$)\} \\ \{q'\}\ [C']\ \{r\} \\ \mathit{match}(q, q', C') \end{array}}{\{p\}\ [C]\ \{r\}}$$
where $\underline{\mathrm{i}} \notin \mathit{free}(p) \cup \mathit{free}(q)$

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

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

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

Fig. 7. Inference rules for PSA

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

$\{\ \$ = \langle \underline{\mathtt{this}} := \mathtt{new}\ \mathtt{Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{y}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}})\rangle \wedge \mathtt{size}(\underline{\mathtt{y}}) > 1\ \}$
  $\mathtt{Worker}$
$\{\ \exists \underline{\mathtt{w}} \bullet \$ = \langle \underline{\mathtt{w}} := \mathtt{new}\ \mathtt{Worker} \cdot \underline{\mathtt{w}}.\mathtt{do}(\mathtt{restQuery}(\underline{\mathtt{y}})) \cdot$
$\underline{\mathtt{w}}.\mathtt{propagate}(\mathtt{merge}(\underline{\mathtt{v}}, \mathtt{compute}(\mathtt{fstQuery}(\underline{\mathtt{y}}))), \underline{\mathtt{c}})\rangle\ \}$

5.2 Verification

The semantics of the specifications includes the open system aspect. Unlike the usual specification technique where one can refer to the program model, we have only the class or component name. To handle reasoning between class and component specifications, a special kind of proof system is needed. The proof system should allow one to verify component specifications by transitively inferring them from the class specifications. In this section, we provide a proof system that handles systems of similar nature to the running example to illustrate the complete picture of our approach.

The proof system PSA presented in Fig. 7 has 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. 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 valid trace assertions or assumed class triples.

The rule CONSEQUENCE is a standard Hoare logic rule, where the input trace assertion can be weakened and the output trace assertion can be strengthened. This rule can be applied to a class or a component D.

BOXING transforms a class triple into a component triple, when the output trace assertion states that no object is created by the instance of that class. The output trace assertion uses the predicate *nonCr*, which checks for the lack of creation event in the output trace. This rule is derived from Def. 15 where the only object creation event that can be observed is that of the initial object of the component.

The BOXEDCOMPOSITION rule defines how an object o of class C can be combined with another component instance of initial class C' to create boxed component $[C]$. For this rule to be applicable, three premises must hold.

First, the class triple C must guarantee that the object identity will not be exposed.

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

The predicate *noSelfExp*, short for *no self exposure*, takes variable i and the trace variable \$ representing the input and output traces, respectively. To ensure that no self exposure is made, the acquaintance of the output is checked against the callee of all events in the input trace, alias the object o. It guarantees one way interaction between o and objects of component $[C']$ because the current object is not exposed.

Second, the instance of C creates a component whose initial class is C'. Thus, no foreign object is created by the instance of $[C']$.

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

$$match(q, q', C') \overset{\text{def}}{=} q \implies \exists o \in \mathbf{O} \bullet firstCreated(o, \$) \wedge class(o) = C' \wedge q'$$

The predicate *firstCreated* checks if the first event is an object creation and o represents the created object. The predicate *classOf* checks if the created object is of class C'. The *match* assertion relies on the fact that a class or a component trace starts with an object creation (see Corollary 1, Lemma 4). This restriction applies because the evaluation of q' is done against an input trace, which always starts with an object creation. For *match* to hold, the free variables of q and q' should coincide. 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. Recall that following the semantics of trace assertion *match* is checked against all traces, as it is used as a free standing trace assertion (i.e., not within a triple).

Example 14. Consider the output assertion of server class specification as described in Ex. 11 and the input assertion of the worker component as described in Ex. 12.

- $q = \exists \underline{\mathtt{w}} \bullet \$ = \langle \underline{\mathtt{w}} := \mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathtt{w}}.\mathtt{do}(\underline{\mathtt{y}}) \cdot \underline{\mathtt{w}}.\mathtt{propagate}(\mathbf{null}, \underline{\mathtt{c}})\rangle$
- $q' = \$ = \langle \underline{\mathtt{this}} := \mathbf{new}\ \mathtt{Worker} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{y}}) \cdot \underline{\mathtt{this}}.\mathtt{propagate}(\underline{\mathtt{v}}, \underline{\mathtt{c}})\rangle$

If we instantiate $\underline{\mathtt{v}}$ in q' with **`null`** (i.e., using the worker component to deal with a original request sent by the client), the output assertion q and input assertion q' match. Hence, $\models_{FOL} match(q, q'[\underline{\mathtt{v}}/\mathbf{null}], \mathtt{Worker})$.

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 some input with parameters of method calls coming to those objects converging into a base case, as indicated by the measure variable m. It is similar to making a recursive call inside an object. The difference is that the concurrent nature of the objects causes each call may be processed independently by each created object, possibly optimizing the computation.

In addition to the inference rules above, PSA also includes some standard auxiliary rules: INVARIANCE and SUBSTITUTION. INVARIANCE allows a predicate containing no trace variable 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 variable. As with CONSEQUENCE, these rules also apply for class and component triples.

Any meaningful proof system should be *sound*. A proof system is sound if and only if its inference rules only derive from some premises conclusions which are valid according to the given semantics. In our setting, sound means that each of the rules in PSA derives valid triples according to the corresponding trace semantics. Because of the open setting, proving the soundness of PSA is challenging. The soundness proof comes from our previous work.

Theorem 1 (Soundness [37]). *The proof system in Fig. 7 is sound.*

Using PSA, 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 verify the server component triple from server class triple and worker component triple.

Example 15. 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.
$\{\ \$ = \langle \underline{\mathsf{this}} := \mathtt{\textbf{new}}\ \mathtt{Server} \cdot \underline{\mathsf{this}}.\mathtt{serve}(\underline{\mathsf{c}}, \underline{\mathsf{y}})\rangle\ \}$
$\quad [\mathtt{Server}]$
$\{\ \$ = \langle \underline{\mathsf{c}}.\mathtt{response}(\mathtt{compute}(\underline{\mathsf{y}}))\rangle\ \}$
Compared to the server class triple given in Ex. 11, there are two notable differences. The 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. 12) and server class specifications (Ex. 11), we can infer the specification of the server component. That is, when the server is implemented in a way described by the worker component and the server class specifications, no matter how the server's context looks like, the server behaves as specified. We abbreviate the following assertions used in the specifications.

– $\mathsf{InSrv} = \mathsf{InSrvC} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{this}} := \mathtt{\textbf{new}}\ \mathtt{Server} \cdot \underline{\mathsf{this}}.\mathtt{serve}(\underline{\mathsf{c}}, \underline{\mathsf{y}})\rangle$
– $\mathsf{OutSrv} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{w}} := \mathtt{\textbf{new}}\ \mathtt{Worker} \cdot \underline{\mathsf{w}}.\mathtt{do}(\underline{\mathsf{y}}) \cdot \underline{\mathsf{w}}.\mathtt{propagate}(\mathtt{\textbf{null}}, \underline{\mathsf{c}})\rangle$
– $\mathsf{InWrkC} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{this}} := \mathtt{\textbf{new}}\ \mathtt{Worker} \cdot \underline{\mathsf{this}}.\mathtt{do}(\underline{\mathsf{y}}) \cdot \underline{\mathsf{this}}.\mathtt{propagate}(\underline{\mathsf{v}}, \underline{\mathsf{c}})\rangle$
– $\mathsf{OutWrkC} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathsf{v}}, \mathtt{compute}(\underline{\mathsf{y}})))\rangle$
– $\mathsf{OutSrvC} \overset{\text{def}}{=} \$ = \langle \underline{\mathsf{c}}.\mathtt{response}(\mathtt{compute}(\underline{\mathsf{y}}))\rangle$

The abbreviations are chosen such that InSrv, for example, represents the input event core equality of the server class triple, whereas $\mathsf{OutWrkC}$ represents the output event core 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*ore *s*equence *e*xtractor, to extract the event core 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. 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`**.

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

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. We need to store the input trace and transfer it to the output trace assertion of the triple, to determine whether a reference to the created server object is not exposed. In this example, the core 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.

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

The transformation for the worker component triple is straight forward to obtain. All occurrences of variable $\underline{\mathsf{v}}$ are substituted by **`null`**, which transforms the worker component output trace assertion into the server component output trace assertion. As all parts of the proof tree are closed, the proof of the server component triple is completed.

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

5.3 Discussion and Related Work

The specification technique can handle any triple where the output trace is exclusively determined by the input trace, that is, it fits to a triple $\{\underline{\mathsf{i}} = \$\}\ D\ \{R(\underline{\mathsf{i}}, \$)\}$ where R

forms a trace assertion based on the input and output traces. Particularly, the input trace assertion may depend on the exposure of some objects which appear in the output trace assertion. However, 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 [62]. Stated differently, the technique is sufficient when interleavings between the input and output traces are not important.

Techniques for specifying and verifying concurrent behavior are the subject of many well-known papers. The book by de Roever et al. [49] provides a detailed overview. Here we only focus on those more related to our work.

Several works have focused on concurrent object or actor models. Specification Diagram [54] provides a detailed, graphical way to specify how an actor system behaves. Our specification technique could be encoded into Specification Diagram by utilizing their *choice* operator to go through different kinds of interleaving between input and output events. However, 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 ([42]), a *May testing* ([19]) characterization of Specification Diagram can be obtained [59].

Ahrendt and Dylla [5] and Din et al. [20,22,21] extended Soundarajan's work to deal with concurrent object systems. They considered only finite prefix-closed traces, justifying it by having only finite number of objects to consider in the verification process. The work Din et al. extended previous work [23] where object creation is not treated. In particular, Din et al. 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 [45]. The main difference to our approach is 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.

Other state-based specification and verification techniques for concurrent object systems have also been developed (e.g., [18,17,24,52,11]). However, they rely strongly on having the actual implementations, bypassing the intermediate tier that we would like to have. Apart from [17], they also require the knowledge of the environment.

Misra and Chandy [43], Soundarajan [55] and Widom et al. [62] proposed proof methods handling network of concurrent processes using traces. Misra and Chandy gave a specification technique where the behavior of processes is described by using invariants on the traces. For each trace, a specification states an invariant over the next event that happens afterwards. The semantics of the specification relies on the prefix-closed properties of the trace semantics. Our specification technique differs from theirs by distinguishing the treatment of input and output events. Soundarajan related invariants on process histories of similar style to Misra and Chandy's 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. For this reason, Misra and Chandy's proof system (and also ours) is incomplete. The setting used in these papers 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.

De Boer [10] presented a Hoare logic for concurrent processes that communicate by message passing through FIFO channels in the setting of Kahn's deterministic process networks ([36]). He described a similar two-tier architecture, where the assertions are based on local and global rules. The local rules deal with 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.

Classical models CSP [30], CCS [41] and π-calculus [42] allow specifying interactions among processes. Logics proposed, e.g., in [38] and [57] allow reasoning on CSP and CCS, respectively, while π-calculus models are usually analyzed by means of bisimulation. But because of their minimalistic approach, 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 [51] to show how to restrict π-calculus to simulate object-orientation.

6 Conclusion

We have seen in this chapter a formal system that covers a 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 the composition to verify functional properties of the composed entity in an open setting. The notion of class composition chosen in this chapter is closely related to popular component frameworks. 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 still ongoing. While the connection between the first layer of verification (i.e., from the implementation to the class specification) given in the introduction is dealt with by the latest research [5,20,22], the connection between 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 to be considered. Two particular extensions of interest are the sound rules to compose more than two components/classes and two-way communication between components. And closer to ABS, it is also of interest to investigate patterns involving futures.

Acknowledgment. We thank Yannick Welsch and anonymous reviewers for suggestions to improve the presentation.

References

1. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: Verification for Java's reentrant multithreading concept. In: Nielsen, M., Engberg, U. (eds.) FOSSACS. LNCS, vol. 2303, pp. 5–20. Springer (2002)
2. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. J. Log. Algebr. Program. 78(7), 491–518 (2009)
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
4. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
5. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. 77(12), 1289–1309 (2012)
6. Armstrong, J.: Erlang. Commun. ACM 53, 68–75 (2010)
7. Arts, T., Dam, M.: Verifying a distributed database lookup manager written in Erlang. In: World Congress on Formal Methods. pp. 682–700 (1999)
8. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
9. Baker, Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. SIGART Bull. pp. 55–59 (August 1977)
10. de Boer, F.S.: A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. Theor. Comput. Sci. 274(1–2), 3–41 (2002)
11. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP. LNCS, vol. 4421, pp. 316–330. Springer (2007)
12. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer-Verlag New York, Secaucus, NJ, USA (2001)
13. Cardelli, L.: Class-based vs. object-based languages. PLDI Tutorial (1996)
14. Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: An event-based structural operational semantics of multi-threaded Java. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 157–200. Springer (1999)
15. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-oriented Programming, LNCS, vol. 7850, pp. 15–58. Springer (2013)
16. Clinger, W.D.: Foundations of Actor Semantics. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (1981)
17. Dam, M., Fredlund, L.Å., Gurov, D.: Toward parametric verification of open distributed systems. In: COMPOS. pp. 150–185 (1997)
18. Darlington, J., Guo, Y.: Formalising actors in linear logic. In: OOIS. pp. 37–53 (1994)
19. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
20. 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)
21. Din, C.C., Dovland, J., Owe, O.: An approach to compositional reasoning about concurrent objects and futures. Research Report 415 (2012)
22. Din, C.C., Dovland, J., Owe, O.: Compositional reasoning about shared futures. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM. LNCS, vol. 7504, pp. 94–108. Springer (2012)
23. Dovland, J., Johnsen, E.B., Owe, O.: Verification of concurrent objects with asynchronous method calls. In: SwSTE. pp. 141–150. IEEE Computer Society (2005)

24. Duarte, C.H.C.: Proof-theoretic foundations for the design of actor systems. Mathematical Structures in Computer Science 9(3), 227–252 (1999)
25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
26. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)
27. Hähnle, R.: The Abstract Behavioral Specification language: A tutorial introduction. In: Bonsangue, M., de Boer, F., Giachino, E., Hähnle, R. (eds.) International School on Formal Models for Components and Objects: Post Proceedings. LNCS (2013)
28. Haller, P.: On the integration of the actor model in mainstream technologies: The Scala perspective. pp. 1–6. AGERE! '12, ACM, New York, NY, USA (2012)
29. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (Oct 1969)
30. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (Aug 1978)
31. International Telecommunication Union: Open distributed processing – reference models parts 1–4. Tech. rep., ISO/IEC (1995)
32. Johnsen, E.B., Blanchette, J.C., Kyas, M., Owe, O.: Intra-Object versus Inter-Object: Concurrency and reasoning in Creol. Electr. Notes Theor. Comput. Sci. 243, 89–103 (2009)
33. 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)
34. 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)
35. Jonsson, B.: A fully abstract trace model for dataflow and asynchronous networks. Distributed Computing 7(4), 197–212 (1994)
36. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress. pp. 471–475 (1974)
37. Kurnia, I.W., Poetzsch-Heffter, A.: A relational trace logic for simple hierarchical actor-based component systems. pp. 47–58. AGERE! '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2414639.2414647`
38. Lamport, L., Schneider, F.B.: The "Hoare logic" of CSP, and all that. ACM Trans. Program. Lang. Syst. 6(2), 281–296 (Apr 1984), `http://doi.acm.org/10.1145/2993.357247`
39. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
40. Microsoft: Component Object Model (COM) (Jan 1999), available at `http://www.microsoft.com/com/default.asp`
41. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)
42. Milner, R.: Communicating and Mobile Systems – The π-Calculus. Cambridge University Press (1999)
43. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Software Eng. 7(4), 417–426 (1981)
44. Nain, S., Vardi, M.Y.: Trace semantics is fully abstract. In: LICS. pp. 59–68. IEEE Computer Society (2009)
45. Olderog, E.R., Apt, K.R.: Fairness in parallel programs: the transformational approach. ACM Trans. Program. Lang. Syst. 10(3), 420–455 (Jul 1988)
46. OMG: CORBA component model v4.0 (2006), `http://www.omg.org/spec/CCM/`
47. Philippsen, M.: A survey of concurrent object-oriented languages. Concurrency - Practice and Experience 12(10), 917–980 (2000)
48. 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)

49. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods, Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press (2001)
50. Roth, A.: Specification and Verification of Object-Oriented Software Components. Ph.D. thesis, Universität Karlsruhe (2006)
51. Sangiorgi, D., Walker, D.: The Pi-Calculus – A Theory of Mobile Processes. Cambridge University Press (2001)
52. Schacht, S.: Formal reasoning about actor programs using temporal logic. In: Concurrent Object-Oriented Programming and Petri Nets. pp. 445–460 (2001)
53. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: ECOOP 2010. pp. 275–299. LNCS, Springer (2010)
54. Smith, S.F., Talcott, C.L.: Specification diagrams for actor systems. Higher-Order and Symbolic Computation 15(4), 301–348 (2002)
55. Soundarajan, N.: A proof technique for parallel programs. Theoretical Computer Science 31(1–2), 13–29 (1984)
56. Steffen, M.: Object-Connectivity and Observability for Class-Based, Object-Oriented Languages. Habilitation thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel (Jul 2006), 281 pages
57. Stirling, C.: An introduction to modal and temporal logics for CCS. In: Yonezawa, A., Ito, T. (eds.) Concurrency: Theory, Language, and Architecture, LNCS, vol. 491, pp. 1–20. Springer Berlin Heidelberg (1991)
58. Talcott, C.L.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation 11(3), 281–343 (1998)
59. Thati, P., Talcott, C.L., Agha, G.: Techniques for executing and reasoning about specification diagrams. In: AMAST. pp. 521–536 (2004)
60. The OSGi Alliance: OSGi core release 5 (2012), `http://www.osgi.org`
61. Vasconcelos, V.T., Tokoro, M.: Traces semantics for actor systems. In: Object-Based Concurrent Computing. LNCS, vol. 612, pp. 141–162. Springer (1991)
62. Widom, J., Gries, D., Schneider, F.B.: Completeness and incompleteness of trace-based network proof systems. In: POPL. pp. 27–38 (1987)
63. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: OOPSLA. pp. 258–268 (1986)