# Object-Oriented Specification and Open Distributed Systems

Einar Broch Johnsen[1,2] and Olaf Owe[2]

[1] BISS, FB3, University of Bremen, Germany
[2] Dept. of Informatics, University of Oslo, Norway
{einarj,olaf}@ifi.uio.no

**Abstract** An object-oriented approach to program specification and verification was developed by Ole-Johan Dahl with the long-term ABEL project. Essential here was the idea of reasoning about an object in terms of its observable behavior, where the specification of an object's present behavior is given by means of its past interactions with the environment. In this paper, we review some of the ideas behind this approach and show how they can be fruitfully extended for reasoning about black-box components in open object-oriented distributed systems.

## 1 Introduction

Object-orientation was introduced by Ole-Johan Dahl and Kristen Nygaard with the programming language Simula [13–15,41] in 1966. Since then, object-oriented programming (OOP) has become an increasingly widespread and popular programming paradigm, lately with Java. Also for system specification, many formalisms have adapted ideas from OOP to better organize specifications; for example, Actors [2], Maude [9], Object-Z [46], UML [7], and the $\pi$-calculus [38] all support some object-oriented concepts. The term *object-based* has emerged to describe formalisms that support objects, i.e., that incorporate notions of object identity and encapsulation in the language [40]. To be fully object-oriented, a formalism should also have an inheritance mechanism reminiscent of OOP. We will now explain what we mean by the central object-oriented concepts of object identity, encapsulation, and inheritance in the context of specification notations:

- *Identity.* Objects have explicit identifiers. When communication occurs between named objects, an object knows which objects it addresses with a given communication. Object identifiers can be transmitted from one object to another during such communication. An object's awareness of other objects in its environment can thus increase over time.
- *Encapsulation and information hiding.* An object encapsulates its internal variables (attributes), so these are not directly perceived from outside the object. This has some noteworthy consequences. *Internally*, we gain control of how the object's variables are manipulated. Variables can only be manipulated by operations (methods) that the object offers to its environment, so the state space of an object resembles an abstract data type. *Externally*,

an object appears as a black box that reacts in a (more or less) predictable manner to impulses from its environment. To use an object, knowledge of its implementation is not needed, only of the available methods. (Explicit hiding mechanisms were not present in the first version of Simula.)

- *Inheritance.* A subclass inherits a superclass by adding attributes and modifying or extending its methods. Class inheritance, introduced in Simula, is a powerful structuring mechanism for developing large systems. However, to really be of value, inheritance should not only allow reuse of code, but also of the reasoning done for the superclass [48]. A similar notion of inheritance or reuse at the level of reasoning can be found in behavioral (or predicate) subtyping [35]. In principle, these two notions of inheritance are not directly related, but when class inheritance is restricted to behavioral subtyping, we get *substitutability*, by which we mean that an object of a class $C$ can be replaced by an object of a subclass of $C$ at the level of reasoning.[1]

Restricting class inheritance to ensure behavioral subtyping comes at the expense of free code reuse and may seem too limiting in the eyes of many programmers. Also, combining these notions lead to the so-called inheritance anomalies [37]. In contrast, in the ABEL project [10–12, 16], Dahl takes the approach that reasoning is done at the specification level, and code is shown to implement specifications, for instance by means of type simulation. Hence, in ABEL, a class can simulate a type. Requirement specification of a concurrent object is in terms of its *observable behavior* and may be implemented using (internal) state transitions. The observable behavior of an object up to some point in time is recorded in its communication history (or finite trace), which gives us an abstract view of the object's state, and present behavior can be specified as a function on the history. Traces are well-known from process algebra, for example CSP [25]. However, generator inductive specifications of the permissible traces as suggested by Dahl [10], where fix-points are not needed in the underlying semantics, are different from the process algebraic approach (as explained in Section 2.2). In contrast to approaches based on streams [8, 33], specifications can be expressed by finite traces since the history at any given time is finite.

In this paper, our focus is on formal reasoning and specification of open distributed systems (ODS). These systems are subject to change at runtime, so we consider concurrent objects, and more generally components, that exist in an evolving environment. For instance, new objects can be introduced into the system and old objects can be upgraded or replaced. Objects will often be supplied by third party manufacturers and we cannot generally expect to have knowledge of implementation details concerning objects in the environment. Instead, the behavior of an object can be locally determined by its interaction with other objects in the environment [2], i.e. by its observable behavior. Due to the complexity of ODS, it is often advocated that system descriptions be aspectwise, in so-called viewpoints [26]. In this paper, we address the issue of specifying ODS

[1] An early work on substitutability in the setting of class invariants, pre- and postconditions on methods, and related requirements on method redefinition and external attribute access, is the thesis of Wang [50], supervised by Dahl.

by viewpoints of observable object behavior, based on the tradition in object-oriented specification from Abel.

The paper is structured as follows. In the next section, we give a brief overview of some important principles of object-oriented specification in Abel and suggest extensions towards ODS. Section 3 considers openness within the object-oriented framework. Section 4 introduces object viewpoints and behavioral interfaces in an assumption guarantee specification style [32], inspired by these principles. Section 5 illustrates the use of this formalism by a specification of a software bus, i.e. an open communication infrastructure. Section 6 discusses composition of assumption guarantee specifications in this setting and Section 7 relates this work to other formalisms for specifying ODS and outlines future research issues before we conclude in Section 8.

## 2 Object-Oriented Specification

The Abstraction Building, Experimental Language (Abel) is a long-term research project at the University of Oslo, centered around a student course in formal methods and the development of a theorem prover. Abel is a wide spectrum language, expressing requirement specifications, constructive specifications (models), and classes. The most important sources of ideas for Abel are *object-orientation*, especially the notions of class and subclass, which originated from the work on Simula; *generator induction*, from the work on Larch [23]; and *order-sorted algebras*, from the work on OBJ [22]. Program development in Abel consists of three steps:

- *Applicative level.* Specification in terms of observable behavior uses abstract data types, generator induction, and the local communication history. Subtypes are either syntactical (cf. examples) or predicative. Abel supports partial functions with partial logic [42].
- *Imperative level.* Class implementation is state-based, establishing invariants by means of Hoare logic [12, 24], with the history as a mythical variable.
- *Integration.* The two levels are integrated by means of weak or strong simulation. A type can be simulated by a class.

We will now consider each of the three steps and develop a brief example.

### 2.1 The Applicative Level of Abel

At an applicative level, specifications of concurrent objects are expressed by permissible observable behavior, i.e. by the time sequence of input and output to the program. This fits well with the object-oriented notion of encapsulation; only visible operations are considered at the applicative level and the realization of the object by means of internal data structures and implementation of operations is postponed to the imperative level. An execution can be represented by a sequence of communication events. In the case of non-terminating executions, the sequences are infinite. However, infinite sequences are not easy to reason

about. In order to avoid infinite sequences, specifications are expressed in terms of the finite initial segments of the executions, which express the abstract states of the object. These sequences are commonly referred to as histories [10] or traces [25]. An *invariant* on the history defines a set of traces by the prefix-closure of the set of executions, so history invariants express safety properties in the sense of Alpern and Schneider [3].

Dahl remarks that specifications in a generator inductive style closely resemble programs in an applicative programming language [12]. The values of an abstract data type are completely defined by its set of generator functions (or constructors) in the sense that all inhabitants of the type can be generated by successive applications of the constructors. The definition $f(x_1, \ldots, x_n) == RHS$ is *terminating* and *generator inductive* (TGI) if the right hand side $RHS$ of the equation uses the variables $x_1, \ldots, x_n$, the constructors, case constructs, $f$ itself, and other TGI defined function symbols. In case of direct or indirect recursion, syntactic requirements guarantee termination. Specifications where all expressions are well-defined and only use TGI defined functions, can be evaluated in a convergent term rewrite system. Furthermore, for TGI defined functions, inductive arguments can be used in proofs; to each constructor corresponds one hypothesis in the proof rule. Such proofs can also to a large extent be mechanized by term rewrite systems.

*Finite sequences.* We present an abstract data type specification in the ABEL style for finite sequences parameterized over some type $T$. The type (schema) Seq[$T$] is defined as the union of two subtypes, Eseq, which is the type of the empty sequence over type $T$, and Seq1[$T$], which is the type of non-empty sequences over type $T$.

**type** Seq[$T$] **by** Eseq, Seq1[$T$] ==
**module**
**func** $\varepsilon \quad : \rightarrow$ Eseq
**func** $\hat{} \vdash \hat{} :$ Seq[$T$] $\times\, T \rightarrow$ Seq1[$T$]
**genbas** $\varepsilon$, $\hat{} \vdash \hat{}$
**endmodule**

where ˆ denotes argument positions of functions with mixfix notation. Here, the keyword **genbas** is used to indicate the functions used as a generator basis for the type, so the finite sequences are constructed from two generator functions; $\varepsilon$ generates an empty sequence and $s \vdash x$ generates a non-empty sequence from a sequence $s$ and en element $x$ of type $T$. In ABEL, finite sequences are defined by means of right append (suffix) rather than left append (prefix).

Using TGI definitions, several functions can be constructively defined on the type of finite sequences Seq[$T$]. For instance, we can define left append $\hat{} \dashv \hat{} : T \times \mathrm{Seq}[T] \rightarrow \mathrm{Seq1}[T]$, concatenation $\hat{} \vdash\!\dashv \hat{} : \mathrm{Seq}[T] \times \mathrm{Seq}[T] \rightarrow \mathrm{Seq}[T]$, and length $\sharp\,\hat{} : \mathrm{Seq}[T] \rightarrow \mathrm{Nat}$ by (case-free) equations:

$$x \dashv \varepsilon = \varepsilon \vdash x \qquad s \vdash\!\dashv \varepsilon = s \qquad \sharp\varepsilon = 0$$
$$x \dashv (s \vdash y) = (x \dashv s) \vdash y \qquad s \vdash\!\dashv (t \vdash x) = (s \vdash\!\dashv t) \vdash x \qquad \sharp(s \vdash x) = \sharp s + 1$$

In these function definitions, the free variables in each equation have an implicit universal quantifier, reminiscent of for instance ML, and each line corresponds to a possible generator case.

Many useful functions are only partially defined on Seq$[T]$, but TGI defined and total on the subtype Seq1$[T]$, which has only one generator ($\vdash$). For instance, we can define selector functions: right rest $rr(\hat{})$ : Seq1$[T] \rightarrow$ Seq$[T]$ and left term $lt(\hat{})$ : Seq1$[T] \rightarrow T$ by

$$\begin{array}{ll} rr(\varepsilon \vdash x) = \varepsilon & lt(\varepsilon \vdash x) = x \\ rr((s \vdash y) \vdash x = rr(s \vdash y) \vdash x & lt((s \vdash y) \vdash x) = lt(s \vdash y) \end{array}$$

The type of finite sequences and the functions we have defined above will now be used in an example specification.

*Example 1.* We illustrate the applicative level of the Abel language by the specification of an unbounded buffer object. The buffer receives input by means of a *get* operation and transmits output by a *put* operation. The history of the buffer is a sequence of operation calls, conventionally denoted $\mathcal{H}$, and new method invocations are recorded in the history by suffixing. Hence, the history until present time is always available for reasoning, and present behavior of the buffer is specified in terms of preceding activity. If we represent by a type $Calls$ the *put* and *get* operations ranging over the values of a given type $T$, then $\mathcal{H}$ : Seq$[Calls]$. Given a history sequence, we define a function *cnt* that computes the implicit content of the buffer, *cnt* : Seq$[Calls] \rightarrow$ Seq$[T]$. Thus, the history gives us an abstract view of the state. The invariant of the specification is defined as a predicate on the history, which is updated after every method call. It follows that the history invariant is implicitly both pre- and postcondition of every operation in the interface.

**interface** BufferSpec $[T : Type]$
**begin**
  **opr** *put*(**in** $x : T$)
  **opr** *get*(**out** $x : T$)
 **inv** $I(\mathcal{H})$
**where**
 $I(\varepsilon) = \varepsilon$
 $I(\mathcal{H} \vdash put(x)) = I(\mathcal{H})$
 $I(\mathcal{H} \vdash get(x)) = I(\mathcal{H}) \wedge$ **if** $cnt(\mathcal{H}) \neq \varepsilon$ **then** $x = lt(cnt(\mathcal{H}))$ **else false fi**

 $cnt(\varepsilon) = \varepsilon$
 $cnt(\mathcal{H} \vdash put(x)) = cnt(\mathcal{H}) \vdash x$
 $cnt(\mathcal{H} \vdash get(x)) = rr(cnt(\mathcal{H}))$
**end**

Both the invariant and the auxiliary function *cnt* are defined by terminating generator induction. An invocation $get(x)$ updates the mythical variable: $\mathcal{H} := \mathcal{H} \vdash get(x)$. As new values are added to the right of the content sequence (calculated by $cnt(\mathcal{H})$), the first value is retrieved by the left term function $lt(cnt(\mathcal{H}))$ and the remaining content by the right rest function $rr(cnt(\mathcal{H}))$.

### 2.2 The Imperative Level of ABEL

For the implementation of specifications in program code, a state-based guarded command language is suggested. The invocation of an operation implemented by $G \longrightarrow S$, where $G$ is a guard and $S$ is a program statement, must wait until $G$ holds and results in the execution of $S$. The communication history is available at the implementation level as a "mythical" variable [12], which is implicitly updated with a new event representing the invocation of an operation after evaluation of the operation body. For verifying operations, ABEL relies on Hoare-logic. Given a history invariant, verifying an operation call $op(x)$ consists of establishing the validity of the formula $\{G \wedge I(\mathcal{H})\}\ S\ \{I(\mathcal{H} \vdash op(x))\}$.

*Example 2.* In this example, we propose a program code for the unbounded buffer specified in Example 1. The content of the buffer is stored in an internal variable $cont$ : Seq[$T$]. The program invariant is given by the relationship between the implicit content of the buffer, as extracted from the history, and the actual content which is stored in $cont$.

**class** BufferClass [$T : Type$]
  **implements** BufferSpec [$T$]
**begin**
  **var** $cont$ : Seq[$T$]
    **opr** $put$(**in** $x : T$) $== cont := cont \vdash x$
    **opr** $get$(**out** $x : T$) $== cont \neq \varepsilon \longrightarrow [x := lt(cont); cont := rr(cont)]$
  **inv** $cont = cnt(\mathcal{H})$
**end**

The constructive nature of this form of applicative specifications is close to implementation, as illustrated by this program code for the specification of Example 1. In fact, this class implementation can be derived directly from the specification and the class invariant. (To obtain single transition systems, program statements can be restricted to multiple assignment operations although this is not done here.)

In ABEL, the history is constructed by sequence suffixing, using right append as the constructor in contrast to prefixing by means of left append. The history is always available for reasoning, as an abstract representation of the state, from which we can extract information. Consequently, the correct behavior of the specified object can be determined by a predicate. Choosing the prefix constructor instead would bring us to a recursive setting, similar to process algebras such as CSP. In this case, the history is not available for reasoning and information concerning the state must be passed along in process parameters that reveal parts of the internal structure of the program. An illustrative example here is the formalism CSP-OZ [20], which combines CSP with Object-Z. In this formalism, objects specified in Object-Z are represented as CSP processes by including all attributes as process parameters. ABEL specifications do not need to reveal the internal structure of the program, so they are in this sense more abstract than the corresponding process algebraic specifications. In particular, recursive definition of processes and the resulting fix-point semantics are avoided.

*Type simulation.* In order to show that program code implements the intended specification, ABEL uses type simulation techniques [12]. An abstract type simulates a concrete type by means of an abstraction function. There are many techniques for type simulation. In ABEL, focus has been on both strong simulation, where every function, and thereby every value, on the abstract type is simulated by a function on the concrete type, and on weak simulation, where concrete values at best approximate an abstract value, for instance giving room for capacity constraints at the concrete level. Type approximation corresponds to one possible method of data refinement. There is a rich literature on data refinement; for a recent overview, the reader may consult de Roever and Engelhardt [18]. Although less standard, ABEL's refinement by type approximation reflects a profound concern for the practical applicability of formal methods.

### 2.3 Explicit Object Identities

In this section, we show how the formalism presented above can be extended in order to capture object interaction in systems with many objects. When we consider concurrent objects that communicate in parallel systems, an object may talk to several other objects. In this setting it is therefore not satisfactory to model the object's behavior by its interaction with an implicit environment that consists of a *single entity*, as we did in the previous examples. Instead, we now consider the environment of an object as an unbounded set of other objects. When we specify an object in such a system, we will refer to this set as the object's (communication) environment. For object systems, the history records communication between objects in the form of remote method calls. Therefore, we introduce object identifiers in the events that are recorded in the communication traces, so that every communication event contains the identity of the transmitting object. This lets us express properties that should hold for a single calling object, a particular calling object, or for all calling objects by projections on the history. Also, we can specify how calls from different objects in the environment are interleaved. We illustrate specifications with implicit object identities in communication events by an example, following [17].

*Example 3.* Consider an object controlling write access to some shared data resource. All objects in its environment are allowed to perform write operations on the shared data, but only one object at a time; we want to specify that write access is sequential. Assume that every event has an implicit sender identifier. Now, we can project traces on an object identifier to express projection on the set of all events associated with that identifier (ranging over operations) or on the name of an operation to express projection on the set of events associated with that operation (ranging over object identifiers).

Let $OW$ represent (the completion of) an *open_ write* operation, $W$ a *write* operation, and $CW$ a *close_ write* operation. Let the predicate $t$ **prs** $Reg$ express that a trace $t$ is a prefix of a trace in the regular language $Reg$. The **prs** predicate will be used to express invariant properties on the history. As before, $\mathcal{H}$ denotes the history of communication events involving the current object. We denote by $\mathcal{E}$ the current environment of the object we are specifying.

**interface** SeqWrite $[T : Type]$
**begin**
  **opr** $OW$
  **opr** $W$
  **opr** $CW$
 **inv** $\mathcal{H}$ **prs** $[OW\ W^*\ CW]^* \wedge \forall o \in \mathcal{E} : (\mathcal{H}/o)$ **prs** $[OW\ W^*\ CW]^*$

 **lemma** $\forall \mathcal{H} : 0 \leq \sharp(\mathcal{H}/OW) - \sharp(\mathcal{H}/CW) \leq 1$
**end**

The regular expression ensures that *write* operations $W$ are performed between an *open_write* operation $OW$ and a *close_write* operation $CW$. We want the regular expression to hold for every object in the environment so we quantify over object identifiers. The predicate $\forall o \in \mathcal{E} : (\mathcal{H}/o)$ **prs** $[OW\ W^*\ CW]^*$ quantifies over objects in the environment. By projecting the history on events associated with every calling object, we consider the (pointwize) communication between the objects of the environment and the object of the interface. The predicate above therefore states that every object in the environment adheres to this behavior. The other conjunct $\mathcal{H}$ **prs** $[OW\ W^*\ CW]^*$ states that the full history of the object adheres to this behavior as well. This means that *write* operations only occur when $\sharp(\mathcal{H}/OW) - \sharp(\mathcal{H}/CW) = 1$, for every object in the environment, and the lemma follows. Therefore, at most one object in the environment can perform write operations with the given invariant.

Observe that in this example, the specified object is *passive* because it only receives and never transmits calls. In order to specify objects that are *active* as well, we need to reason about events that are transmitted from the object (output) to different objects in the environment, so communication events need to be equipped with the object identities of both the sender and receiver.

## 3 Object-Orientation and Openness

In open systems software may be changed, interchanged, and updated. This makes program reasoning more difficult as old invariants can be violated by new software, and also static typing can be too restrictive. Although there are applications where this does not seem to be a problem, it is interesting to see how and to what extent strong typing and incremental, textual, and compositional reasoning can be combined with openness. Strong typing alone can ensure essential safety properties, such as providing limited access rights. We will here explore some possibilities for controlled openness within the object-oriented framework.

In order to solve the conflict between unrestricted reuse of code in subclasses, and behavioral subtyping and incremental reasoning control, we suggest to use behavioral interfaces as presented in the previous section to type object variables, and consider multiple inheritance at the interface level as well as at the class level. Interface inheritance is restricted to a form of behavioral subtyping [35], whereas class inheritance may be used freely. Inherited class (re)declarations are

resolved by disjoint union. A class may implement several interfaces, provided that it satisfies the syntactic and semantic requirements stated in the interfaces. An object of class $C$ supports an interface $I$ if the class $C$ implements $I$.

Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface $I$ may be replaced by another object supporting $I$ or a subinterface of $I$.* Subclassing is unrestricted with the consequence that interface implementation claims are not preserved by subclassing. If a class $C$ implements an interface $I$, this property is not always guaranteed by a subclass of $C$, as a method redefinition may violate semantic requirements of $I$. Therefore, implementation claims (as well as class invariants) are not in general inherited.

*Strong typing.* We consider typing where two kinds of variables are declared; an object variable by an interface and an ordinary variable by a data type. Strong typing ensures that for each method invocation $o.m(inputs;\ outputs)$, where $I$ is the declared interface of $o$, the actual object $o$ (if not nil) will support $I$ and the method $m$ will be understood, with argument types "including" the types of the actual ones. Inclusion is defined as the pointwise extension of the subtype and subinterface relation, using co- and contravariance for in- and out-parameters, respectively. Explicit hiding of class attributes and methods is not needed, because typing of object variables is based on interfaces and only methods mentioned in the interface (or its super-interfaces) are visible.

*Modifiability.* An obvious way to provide some openness is to allow addition of new (sub)classes and new (sub)interfaces. In our setting, this mechanism in itself does not violate reasoning control, in the sense that old proven results still hold. Also, additional implementation claims may be stated and proved. However, old objects may not use new interfaces that require new methods.

A natural way to overcome this limitation is through a dynamic class construct, allowing a class to be *replaced* by a subclass. Thus a class $C$ may be modified by adding attributes (with initialization) and methods, redefining methods, as well as extending the inheritance and implements lists. (In order to avoid circular inheritance graphs, $C$ may not inherit from a subclass of $C$.) Unlike addition of a subclass of $C$, all existing objects of class $C$ or a subclass become renewed in this case and support the new interfaces. Reasoning control is maintained when the dynamic class construct is restricted to behavioral subtyping, which can be ensured by verification conditions associated with the class modification [43]. Unrestricted use of the dynamic class construct, which may sometimes be needed, has the impact that objects of class $C$ may violate behavior specified earlier, and constraints on compositions involving objects of class $C$ must be reproved (or weakened).

Notice that as a special case of class modification, one may posteriorly add super-classes to an established class hierarchy. This would be an answer to a major criticism against object-oriented design [21], namely that the class hierarchy severely limits restructuring the system design.

The run-time implementation of dynamic class constructs is non-trivial [36], even typing and virtual binding need special considerations:

- The removal of a method or attribute from a class $C$ violates strong typing, since such a method or attribute may be used in an old subclass of $C$ and by strong typing the method or attribute must exist. This indicates that removal of methods or attributes should not be allowed.
- A modified class $C$ may add an attribute or a method $m$ and thereby an old subclass $D$ inherits $m$. The subclass may also inherit $m$ (with the same parameter types) from another superclass of $D$. An implication is that the virtual binding mechanism must give priority to the old $m$, otherwise objects of class $D$ will behave unpredictably and reasoning control is clearly lost. This can be implemented by a table associated with each class at run-time, updated whenever a superclass is modified.
- The typing and well-formedness of a modification of a class $C$ should not depend on any existing subclass of $C$. Consider the case where a new method $m$ is added to a class $C$. This should be legal even if a subclass $D$ already has a method $m$. The parameter types may well be the same, and this case predicts that we must accept unrestricted method redefinition in a subclass!
- Parameter types may be only slightly changed in class $C$ (say, only for the out-parameters of a method $m$), in which case we must tolerate arbitrary overloading in a subclass. Invocations of $m$ on objects of a subclass $D$ must respect old behavior, otherwise these objects will behave unpredictably and reasoning control is lost. An implication is that the virtual binding mechanism must give priority to local methods (that include the actual parameter types) over inherited ones.

These implications give independent justification for declaring interfaces for object variables, while allowing unrestricted subclassing. In the next section, we consider specification and reasoning with behavioral interfaces more closely.

## 4 Viewpoints to ODS

In ODS, we can represent components by (collections of) objects that run in parallel and communicate asynchronously by means of remote method calls with input and output parameters. Often, such objects are supplied by third-party manufacturers unwilling to reveal the implementation details of their design. Therefore, reasoning about such systems must be done relying on abstract specifications of the system's components. We find specification in terms of observable behavior particularly attractive in this setting and imagine that components come equipped with behavioral interfaces that instruct us on how to use them. Furthermore, as a component may be used for multiple purposes, it can come equipped with multiple specifications. This section presents a formalism for reasoning about object viewpoints in the setting of ODS, extending the formalism of Section 2. Further details about this work can be found in [28, 29, 43].

### 4.1 Semantics

We now propose a formalization of viewpoint specifications for objects communicating asynchronously by means of remote method calls, restricting ourselves

to safety aspects. Let *Objects* and *Mtd* be unbounded sets of object identifiers and method names, respectively, and let *Data* be a set of data values including *Objects*. Denote by List$[T]$ the lists over a type $T$ (and by $[\,]$ the empty list).

**Definition 1 (Communication events).** *A communication event is a tuple* $\langle o_1, o_2, m, t, inputs, outputs\rangle$ *such that* $o_1, o_2 \in Objects$, $o_1 \neq o_2$, $m \in Mtd$, $t \in \{i, c\}$, $inputs, outputs \in \mathrm{List}[\mathrm{Data}]$, *and* $t = i \Rightarrow outputs = [\,]$.

Intuitively, we can think of these events as initiations and completions of calls to a method $m$ provided by an object $o_2$ by another object $o_1$. Initiation events have no output. Communication is asynchronous as other events can be observed in between the initiation and completion of any given call. (We here assume strong typing, so the number and types of input and output parameters are correct by assumption in the events.)

**Definition 2 (Alphabet).** *An alphabet for a set of objects* $\mathcal{O}$ *is a set* $S$ *of communication events such that* $\langle o_1, o_2, m, c, ins, outs\rangle \in S \Rightarrow \langle o_1, o_2, m, i, ins, [\,]\rangle \in S$ *and* $\langle o_1, o_2, \ldots\rangle \in S \Rightarrow (o_1 \in \mathcal{O} \wedge o_2 \notin \mathcal{O}) \vee (o_2 \in \mathcal{O} \wedge o_1 \notin \mathcal{O})$.

At the specification level, the alphabet of an object supporting an interface is statically given by the interface. Denote by $h/S$ and $h \setminus S$ the restrictions of a sequence $h$ to elements of the set $S$ and to the complement of $S$, respectively.

**Definition 3 (Trace set).** *A trace set over an alphabet* $\alpha$ *is a prefix-closed set of sequences* $t \in \mathrm{Seq}[\alpha]$ *such that, for every sequence* $t$ *in the set and every completion event* $\langle o_1, o_2, m, c, inputs, outputs\rangle \in \alpha$,

$$\sharp(t/\langle o_1, o_2, m, i, inputs, [\,]\rangle) \geq \sharp(t/\langle o_1, o_2, m, c, inputs, outputs\rangle.$$

**Definition 4 (Specification).** *A specification* $\Gamma$ *is a triple* $\langle \mathcal{O}, \alpha, \mathcal{T}\rangle$ *where (1)* $\mathcal{O}$ *is a set of object identities,* $\mathcal{O} \subseteq Objects$*, (2)* $\alpha$ *is an infinite alphabet for* $\mathcal{O}$*, and (3)* $\mathcal{T}$ *is a prefix-closed subset of* $\mathrm{Seq}[\alpha]$.

We call $\mathcal{O}$ the object set of the specification $\Gamma$, $\alpha$ the alphabet of $\Gamma$, and $\mathcal{T}$ the trace set of $\Gamma$. In shorthand, these are denoted $\mathcal{O}(\Gamma)$, $\alpha(\Gamma)$, and $\mathcal{T}(\Gamma)$, respectively. For a specification $\Gamma$, we can derive a *communication environment* $\mathcal{E}(\Gamma)$ of objects communicating with the objects of $\Gamma$, from $\mathcal{O}(\Gamma)$ and $\alpha(\Gamma)$. In an ODS setting, we generally think of the communication environment as unbounded. If the object set of a specification $\Gamma$ is a singleton $\{o\}$, we say that $\Gamma$ is an *interface specification* (of $o$). A *component specification* may comprise several objects.

In order to increase readability, we will henceforth represent an initiation event $\langle o_1, o_2, m, i, [i_1, \ldots, i_n], [\,]\rangle$ visually by $o_1 {\rightarrow} o_2.m(i_1, \ldots, i_n)$ and a completion event $\langle o_1, o_2, m, i, [i_1, \ldots, i_n], [v_1, \ldots, v_p]\rangle$ by $o_1 {\leftarrow} o_2.m(i_1, \ldots, i_n; v_1, \ldots, v_p)$.

*Example 4.* Consider the specification SeqWrite from Example 3, which we now reformulate as an interface specification of an object $o$. Let $\mathcal{E} = \{x \in Objects \mid x \neq o\}$ and let *Data* be a set of data values. The specification only considers one

object, so $\mathcal{O}(\text{SeqWrite}) = \{o\}$. The write method $W$ has an input parameter ranging over $Data$. The alphabet of SeqWrite is now

$$\alpha(\text{SeqWrite}) \triangleq \{x{\rightarrow}o.OW(), x{\leftarrow}o.OW(), x{\rightarrow}o.CW(), x{\leftarrow}o.CW() \mid x \in \mathcal{E}\}$$
$$\cup\{x{\rightarrow}o.W(d), x{\leftarrow}o.W(d) \mid x \in \mathcal{E} \wedge d \in Data\}.$$

Controlled write access is obtained by restricting the possible traces of SeqWrite. For this purpose, we use patterns, i.e. regular expressions extended with a binding operator $\bullet$, and extend the **prs** predicate accordingly. Define a pattern $Wcycle$ by

$$[[x{\rightarrow}o.OW()\ x{\leftarrow}o.OW()$$
$$[[x{\rightarrow}o.W(d)\ x{\leftarrow}o.W(d)] \bullet d \in Data]^*$$
$$x{\rightarrow}o.CW()\ x{\leftarrow}o.CW()] \bullet x \in \mathcal{E}].$$

The trace set is now specified by a prefix of the pattern:

$$\mathcal{T}(\text{SeqWrite}) \triangleq \{h : \text{Seq}[\alpha(\text{SeqWrite})] \mid h \ \mathbf{prs}\ Wcycle^*\}.$$

Here, $x$ is bound for each traversal of the loop and this binding operator on calling objects ensures sequential write access. A caller may perform multiple write operations once it has access. Note that a set defined by a predicate $h$ **prs** $R$ is always prefix-closed and that $\mathcal{T}(\text{SeqWrite})$ is a trace set.

*Refinement.* Refinement describes a correct transformation step from specifications to programs, usually by making the specification more deterministic in the sense of model-inclusion. In our setting of partial specifications, a step towards realization of the specification may involve considering additional communication events, suggesting that refinement in our case must be after projection.

**Definition 5 (Refinement).** *A specification $\Gamma'$ refines another specification $\Gamma$, denoted $\Gamma' \sqsubseteq \Gamma$, if (1) $\mathcal{O}(\Gamma) \subseteq \mathcal{O}(\Gamma')$, (2) $\alpha(\Gamma) \subseteq \alpha(\Gamma')$, and (3) $\forall t \in \mathcal{T}(\Gamma') : t/\alpha(\Gamma) \in \mathcal{T}(\Gamma)$.*

Using projection as suggested here, dynamic class extension (Section 3) is well-behaved with respect to refinement: the new extended class refines the old class. When considering liveness, the refinement relation must be extended to exclude additional deadlocks in a refinement step (cf. Section 7).

*Composition.* When two viewpoint specifications are composed, they synchronize on common events. However, as our focus is on the observable behavior of the specifications, internal communication between the objects of the composed specification is hidden.

**Definition 6.** *The internal events of a set $S$ of objects are all communication events between objects of the set, $\mathcal{I}(S) \triangleq \bigcup_{o_1,o_2 \in S}\{\langle o_1, o_2, \ldots\rangle, \langle o_2, o_1, \ldots\rangle\}$.*

We will write $\mathcal{I}(\Gamma)$ instead of $\mathcal{I}(\mathcal{O}(\Gamma))$. As we consider component viewpoints here, events that are internal in one specification may be observable in another. We say that two specifications are *composable* if this is not the case [28, 29].

**Definition 7 (Composition).** *Let $\Gamma$ and $\Delta$ be composable component specifications. Then $\Gamma||\Delta$ is the specification $\langle \mathcal{O}, \alpha, \mathcal{T}\rangle$ where (1) $\mathcal{O} \triangleq \mathcal{O}(\Gamma) \cup \mathcal{O}(\Delta)$, (2) $\alpha \triangleq \alpha(\Gamma) \cup \alpha(\Delta) - \mathcal{I}(\mathcal{O})$, and (3) $\mathcal{T} \triangleq \{h/\alpha \mid h/\alpha(\Gamma) \in \mathcal{T}(\Gamma) \wedge\ h/\alpha(\Delta) \in \mathcal{T}(\Delta)\}$.*

### 4.2 Behavioral Interfaces

Clearly, the specifications of Section 4.1 can be given a syntax à la ABEL. In this section, we consider such a treatment for interface specifications in a generic manner. These specifications are behavioral interfaces; they can be supported by different objects. An interface can be implemented by different classes and a class can implement different interfaces. An interface has the following syntax:

**interface** $F$ [⟨type parameters⟩] (⟨context parameters⟩)
  **inherits** $F_1, F_2, \ldots, F_m$
**begin**
  **with** $G$
    **opr** $m_1(\ldots)$
      ⋮
    **opr** $m_n(\ldots)$
  **asm** <formula on local trace restricted to one calling object>
  **inv** <formula on local trace>
**where** <auxiliary function definitions>
**end**

Interfaces can have both type and context parameters, the latter typically describes the minimal environment representing static links needed by objects that support the interface. An initiation and a completion event is associated with each method declaration (ranging over method parameters). In the interfaces, we use the keyword "*this*" to denote the object supporting the interface and "*caller*" to denote an object in the environment. We shall now briefly consider the remaining parts of the syntax, for technical details and discussion the reader is referred to [29, 43]. The use of interfaces for specification purposes is illustrated by way of examples in Section 5.

*Assumption guarantee predicates.* In ODS, the environment in which an object exists is subject to change and specifications are relative to an assumed behavior of the environment. Hence, we use the assumption guarantee specification style [32], but we adapt it to our setting of observable behavior. Assumptions are the responsibility of the objects of the environment; therefore, assumption predicates consider traces that end with *input* to the current interface and only communication with a single object in the environment. An assumption predicate $A(x, y, h)$ ranges over objects $x$ in the environment, supporting objects $y$, and traces $h$. Let $\mathbf{in}(h, o)$ and $\mathbf{out}(h, o)$ denote functions that return the longest prefix of a trace $h$ ending with an input or output event to an object $o$, respectively. If $A$ is an assumption predicate, define $A^{in}(x, h) \triangleq \forall o \in \mathcal{E} : A(o, x, \mathbf{in}(h, x))$ and $A^{out}(x, h) \triangleq \forall o \in \mathcal{E} : A(o, x, \mathbf{out}(h, x))$. Invariants are the responsibility of the object supporting the interface; they are guaranteed when the assumption holds and consider traces that end with *output* from the current interface. If $I(x, h)$ is an invariant predicate ranging over supporting objects $x$ and traces $h$, define $I^{out}(x, h) \triangleq I(x, \mathbf{out}(h, x)) \wedge A^{out}(x, h)$. The trace set $\mathcal{T}(\Gamma)$ of a specification $\Gamma$ with assumption predicate $A_\Gamma$ and invariant predicate $I_\Gamma$ is the largest prefix-closed subset of $\{h \in \mathrm{Seq}[\alpha(\Gamma)] \mid A_\Gamma^{in}(\mathit{this}, h) \Rightarrow I_\Gamma^{out}(\mathit{this}, h)\}$.

*Inheritance.* Multiple inheritance is allowed for interfaces, but cyclic inheritance graphs are not allowed. If an interface $F$ is declared with an inheritance clause, the alphabets of the super-interfaces are included in the alphabet of $F$ and the traces of $F$ must be in the trace sets of the super-interfaces when restricted to the relevant alphabets. In the subinterfaces, we can declare additional methods and behavioral constraints. An interface will always refine its super-interfaces.

*Mutual dependency.* Because objects are typed by interface, we can specify that only objects of a particular interface (a cointerface) may invoke the methods of the current interface, using the keyword **with**. Furthermore, the current interface knows the methods of the caller visible through the cointerface. This gives strong typing in an asynchronous setting. Semantically, a cointerface declaration changes the alphabet of the current interface as the communication environment is reduced whereas new methods of the caller are added.

## 5 Case Study: the Software Bus

In this section, we illustrate the use of interface specifications to capture viewpoints concerning the dynamic nature of a software bus, a communication platform to which processes may register in order to share data and resources. (This is a stripped version of an actual system used for monitoring nuclear power plants, more details on the software bus and its specification can be found in [31].) We consider a distributed architecture for the software bus, with a *portmapper* and a collection of *data servers*. The general lay-out of the software bus is shown in Figure 1. Processes may connect (and disconnect) to any data server. The task of the portmapper is to manage registration of processes, and communicate information about processes to other processes, via their data servers. Data servers communicate with each other in order to share data processing tasks, on behalf of their processes. These tasks include the creation of variables, the assignment of values to variables, accessing the values of variables, and destroying variables. The software bus system is object-oriented: classes, functions, and variables are treated as software bus objects, i.e. as manipulatable data in the software bus system. An object in the system is identified either by reference or by a name and a reference to its parent. For specification purposes, we identify two types; **Name** for object names and **Ref** for object references. The latter will have subtypes, among them we find **ParentRef** for parent objects and **AppRef** for application processes. We shall now specify a data server interface **SB_Data** for manipulation of data, a data server interface **SB_Connections** for updating information on remote applications, and a portmapper interface **SB_Portmapper**.

### 5.1 Communication between Data Servers

The interface **SB_Data** considers methods for object manipulation between data servers in the **SoftwareBus**. For brevity, we will here only consider two such methods:

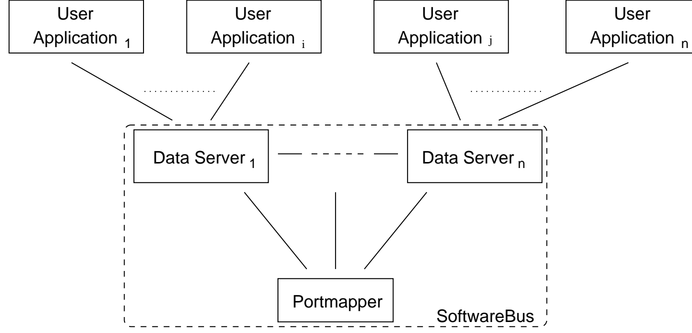


**Figure 1.** Decomposition of **SoftwareBus**. Data servers have an interface **SB_Data** available to other data servers and an interface **SB_Connections** available to the portmapper. The portmapper offers an interface **SB_Portmapper** to data servers.

**opr** id ( **in** name: **Name** , parent_ref: **ParentRef** ; **out** obj_ref: **Ref** )
**opr** del_obj ( **in** obj_ref: **Ref** )

Intuitively, the method *id* obtains the reference to an object at a remote server and *del_obj* deletes an object at a remote server. It is assumed that a data server only attempts to delete an object to which it has obtained a reference (via *id*). If the object is already deleted by another remote server, the method call will not be completed (until the reference has been recreated). We here denote by $S_r$ the set of events that can be associated with a given reference $r$, $S_r = \{x \leftarrow y.id(_,_;r), x \rightarrow y.del_obj(r), x \leftarrow y.del_obj(r) \mid x, y \in Objects\}$, ignoring irrelevant parameters by underscore. The assumption can be formalized as follows:

$$\begin{aligned} A_d(x,y,h) = \forall r \in \mathbf{Ref} :& \\ & h/S_r \ \mathbf{prs}\ [[x \leftarrow y.id(_,_;r)]^+ \\ & \qquad x \rightarrow y.del_obj(r)\ x \leftarrow y.del_obj(r)]^* \end{aligned}$$

The invariant is concerned with output from the current object, in this case completion events to the methods *id* and *del_obj*. The assumption above states that a data server will wait for the completion of a call to the current data server before making new calls to this server. We check if there is a pending call to $del_obj(r)$, using a predicate *pending*:

$$\begin{aligned} pending(mtd,x,y,h) = h/\{x \rightarrow y.mtd(\ldots), x \leftarrow y.mtd(\ldots)\} & \\ \in [x \rightarrow y.mtd(\ldots), x \leftarrow y.mtd(\ldots)]^* x \rightarrow y.mtd(\ldots) & \end{aligned}$$

As we ignore object creation in this example, we assume that an object exists once it has been assigned a reference in a call to *id*. Considering the entire history of a data server (seen through the **SB_Data** interface), we can identify traces after which we believe that a reference is to an existing object:

$$\begin{aligned}
&\neg\, exists(r, \varepsilon)\\
&\neg\, exists(r, h \vdash x \leftarrow y.del_obj(r))\\
&\phantom{\neg\,} exists(r, h \vdash x \leftarrow y.id(_,_;r))\\
&\phantom{\neg\,} exists(r, h \vdash \text{others}) = exists(r, h)
\end{aligned}$$

In this definition, cases are in the considered order and '$h \vdash$ others' handles the remaining cases. The invariant $I_d(h, x)$ expresses that pending calls to $del_obj(r)$ are only completed when the object with reference $r$ is known to exist:

$$\begin{aligned}
&I_d(x, \varepsilon) = \mathbf{true}\\
&I_d(x, h \vdash y \leftarrow x.del_obj(r)) = pending(del_obj, y, x, h) \wedge exists(r, h)\\
&I_d(x, h \vdash \text{others}) = true
\end{aligned}$$

Remark how the case distinction with 'others' allows us to ignore irrelevant events in the above predicates. This way, the predicates can be given in a compact, readable format when only a few events of an alphabet need to be considered. Also, this predicate format facilitates reuse of the predicates in interfaces with extended alphabets, typically in subinterfaces.

We now define the interface **SB_Data** (types for method parameters are as given above).

**interface SB_Data**
**begin**
  **opr** id ( **in** name, parent_ref; **out** obj_ref )
  **opr** del_obj ( **in** obj_ref )
 **asm** $A_d(caller, this, h)$
 **inv** $I_d(this, h)$
**end**

This interface does not consider calls to other servers. The next step is to let **SB_Data** be inherited by a new interface **SB_DataAct**, which includes **SB_DataAct** as a cointerface and $\forall o \in \mathcal{E} : A_d(this, o, h)$ as invariant.

### 5.2 Communication with the Portmapper

In this section, we consider communication between the portmapper and the data servers. First, we specify an interface of the data server, which offers a method *going_down* to portmappers:

**interface SB_Connections**
**begin**
 **with SB_Portmapper**
  **opr** going_down( **in** ref: **AppRef** )
 **asm true**
 **inv true**
**end**

By declaring **SB_Connections** to be a cointerface of **SB_Portmapper**, the interface of the portmapper, the events associated with *going_down* are included in the alphabet of **SB_Portmapper** and we can specify the actual use of the method there. The method will be used to signal that applications in the environment are about to leave the **SoftwareBus**. When an application enters the software bus, its (current) data server will register it with the portmapper, and when it exits, likewise. Furthermore, a data server may contact the portmapper in order to know if (and where) an application is currently registered. The associated methods are

**opr** init ( **in** name: **Name** )
**opr** exit (**in** name: **Name** )
**opr** conn_app ( **in** appl_name: **Name** ; **out** appl_ref: **AppRef** )
**opr** disc_app ( **in** appl_ref: **AppRef** )

Intuitively, *init* signals that an application enters the system, *exit* signals that an application leaves the system, *conn_app* establishes a logical connection to *appl_name*, and *disc_app* disconnects the logical connection to *appl_ref*. Obviously, logical connections should only be disconnected after having been established, which we formalize by the predicate

$$conns(x, y, h) = \forall r \in \mathbf{AppRef} : h/r \ \mathbf{prs} \ [x{\leftarrow}y.conn_app(_; r) \\ x{\rightarrow}y.disc_app(r) \ x{\leftarrow}y.disc_app(r)]^*$$

Furthermore, logical connections and disconnections from an application $x$ may only occur when $x$ is registered with the portmapper $y$.

$$\begin{aligned} &is_reg(x, y, h) = \\ &\quad h/x \ \mathbf{prs} \ x{\rightarrow}y.init(x) \ x{\leftarrow}y.init(x) \\ &\quad\quad [x{\rightarrow}y.conn_app(_) \mid x{\leftarrow}y.conn_app(_; _) \\ &\quad\quad\quad \mid x{\rightarrow}y.disc_app(_) \mid x{\leftarrow}y.disc_app(_)]^* \\ &\quad\quad x{\rightarrow}y.exit(x) \ x{\leftarrow}y.exit(x) \end{aligned}$$

The portmapper assumes that all data servers adhere to this behavior, so define its assumption by the formula

$$A_{pm}(x, y, h) = is_reg(x, y, h) \land conns(x, y, h).$$

The invariant of **SB_Portmapper** considers when output from the portmapper should occur. For this purpose, we determine if an application is currently registered in the system by a predicate *up* on the history:

$$\begin{aligned} &\neg\, up(x, y, \varepsilon) \\ &\ \ up(x, y, h \vdash z{\leftarrow}y.init(x)) \\ &\ \ up(x, y, h \vdash \text{others}) = up(x, y, h) \end{aligned}$$

Similarly, we determine if an application $a_1$ has an established logical connection to another application $a_2$ via the portmapper $p$ after history $h$ by the predicate $conn_up(a_1, a_2, p, h)$:

$$
\begin{aligned}
&\neg\, conn_up(a_1, a_2, p, \varepsilon)\\
&\;\; conn_up(a_1, a_2, p, h \vdash a_1 \leftarrow p.conn_app(a_2))\\
&\neg\, conn_up(a_1, a_2, p, h \vdash a_1 \leftarrow p.disc_app(a_2))\\
&\neg\, conn_up(a_1, a_2, p, h \vdash p \leftarrow a_1.going_down(a_2))\\
&\;\; conn_up(a_1, a_2, p, h \vdash \text{others}) = conn_up(a_1, a_2, p, h)
\end{aligned}
$$

We consider a logical connection closed (or broken) if $a_1$ gets a notification from the portmapper that $going_down(a_2)$. (The events associated with this method come from the cointerface.) We use the abbreviation $notified(n, p, h)$ below for the the predicate $\forall a \in$ **AppRef** : $\neg\, conn_up(a, n, p, h)$ and define the invariant of **SB_Portmapper** as follows:

$$
\begin{aligned}
&I_{pm}(p, \varepsilon)\\
&I_{pm}(p, h \vdash a \leftarrow p.init(n)) =\\
&\qquad \neg\, up(n, _, h) \wedge pending(init(n), p, a, h) \wedge I_{pm}(p, h)\\
&I_{pm}(p, h \vdash p \rightarrow a.going_down(n)) =\\
&\qquad conn_up(a, n, p, h) \wedge pending(exit(n), p, _, h) \wedge I_{pm}(p, h)\\
&I_{pm}(p, h \vdash a \leftarrow p.exit(n)) =\\
&\qquad up(n, _, h) \wedge notified(n, p, h) \wedge pending(exit(n), p, a, h) \wedge I_{pm}(p, h)\\
&I_{pm}(p, h \vdash a \leftarrow p.conn_app(n, r)) =\\
&\qquad \neg\, conn_up(a, n, p, h) \wedge pending(conn_app(n), p, a, h) \wedge I_{pm}(p, h)\\
&I_{pm}(p, h \vdash a \leftarrow p.disc_app(n)) =\\
&\qquad conn_up(a, n, p, h) \wedge pending(disc_app(n), p, a, h) \wedge I_{pm}(p, h)\\
&I_{pm}(p, h \vdash \text{others}) = I_{pm}(p, h)
\end{aligned}
$$

The invariant allows for asynchronous calls to the portmapper, as other events may occur between the initiation and completion of any given call. In particular, the invocation of *exit* explicitly results in calls *from* the portmapper. The **SB_Portmapper** interface is now specified (types for method parameters are as given above).

```
interface SB_Portmapper
begin
    with SB_Connections
        opr init( in n )
        opr exit( in n )
        opr conn_app( in n; out r )
        opr disc_app( in r )
    asm A_pm(caller, this, h)
    inv I_pm(this, h)
end
```

### 5.3 Internal Behavior of the Data Server

In this section, we consider how the two interfaces of the data servers can be combined in order to give a more complete specification of the data server in an interface **SB_DataServer**. In particular, we want to express that a data server

can only make calls to another data server when it has an established logical connection to that data server. For convenience, we inherit auxiliary predicates as well as the semantics through interface inheritance.

This interface considers calls made by the current object, so we will strengthen the invariant of the data server (which we presently perceive as the conjunction of its two interface invariants, restricted to appropriate projections on traces). Define

$$\begin{aligned}
&I_{ds}(x, \varepsilon) \\
&I_{ds}(x, h \vdash x{\to}y.id(_)) = conn_up(x, y, p, h) \wedge I_{ds}(x, h) \\
&I_{ds}(x, h \vdash x{\to}y.del_obj(_)) = conn_up(x, y, p, h) \wedge I_{ds}(x, h) \\
&I_{ds}(x, h \vdash \text{others}) = I_{ds}(x, h)
\end{aligned}$$

The interface **SB_DataServer** can now be specified by

```
interface SB_DataServer
    inherits SB_Connections, SB_Data
begin
    inv I_ds(this, h)
end
```

By definition, any data server with the **SB_DataServer** interface will also support the two super-interfaces **SB_Data** and **SB_Portmapper**. At the semantic level, super-interfaces are always refined by their subinterfaces.

## 6 Composing Assumption Guarantee Specifications

Just as multiple inheritance lets us combine interfaces that are supported by the same objects, we can *compose* specifications where this need not be the case. Definition 7 of Section 4.1 defined composition semantically. In this section we consider a composition rule for specifications made in the assumption guarantee style of interfaces, thus a specification $\Gamma$ is on the form

$$\langle \mathcal{O}(\Gamma), \alpha(\Gamma), \{h \in \mathrm{Seq}[\alpha(\Gamma)] | A_{\Gamma}^{in}(h) \Rightarrow I_{\Gamma}^{out}(h)\}\rangle,$$

where $A_\Gamma$ and $I_\Gamma$ are the assumption and invariant predicates associated with $\Gamma$. The supporting objects are here given by $\mathcal{O}(\Gamma)$, the functions $\mathbf{in}(h)$ and $\mathbf{out}(h)$ return the longest prefix of $h$ that ends with input or output to any object in $\mathcal{O}(\Gamma)$, respectively. Let $\Gamma + \Delta$ denote the *syntactic* composition of two specifications $\Gamma$ and $\Delta$. We want to derive an assumption $A$ and an invariant $I$ that describe the traces of $\Gamma + \Delta$ from the predicates of $\Gamma$ and $\Delta$.

Composition should encapsulate internal communication, so the communication environment $\mathcal{E}(\Gamma + \Delta)$ excludes objects from the object sets $\mathcal{O}(\Gamma)$ and $\mathcal{O}(\Delta)$. Therefore, for the object set, communication environment, alphabet, and internal event set, we follow the semantics (Definition 7) and define $\mathcal{O}(\Gamma + \Delta) \triangleq \mathcal{O}(\Gamma || \Delta)$, $\mathcal{E}(\Gamma + \Delta) \triangleq \mathcal{E}(\Gamma || \Delta)$, $\alpha(\Gamma + \Delta) \triangleq \alpha(\Gamma || \Delta)$, and $\mathcal{I}(\Gamma + \Delta) \triangleq \mathcal{I}(\Gamma || \Delta)$. Let $h \in \mathrm{Seq}[\alpha(\Gamma + \Delta)]$. Now, define an assumption predicate for $\Gamma + \Delta$ by

$$\begin{aligned}
A^{in}(h) &\triangleq A_{\Gamma}^{in}(h/\alpha(\Gamma)) \wedge A_{\Delta}^{in}(h/\alpha(\Delta)) \\
&= \forall o' \in \mathcal{E} : A_{\Gamma}(o, \mathbf{in}(h/o)) \wedge A_{\Delta}(o, \mathbf{in}(h/o)).
\end{aligned}$$

Due to the quantification over objects in the environment, we have that $A_\Gamma(h) \Rightarrow A_\Gamma(h \setminus \mathcal{I}(\Gamma + \Delta))$. However, the assumption $A^{in}(h)$ above is not strong enough to guarantee either of the invariants $I_\Gamma$ and $I_\Delta$, because nothing has been assumed with regard to the internal communication between objects of the two specifications. This leads to the proof conditions (1) and (2) below.

In contrast to the assumption, the invariant does not quantify over the objects of the environment. Therefore, we cannot derive an invariant $I$ directly from the invariants of $\Gamma$ and $\Delta$ by removing internal communication; we need to consider the full alphabet. Let $h \in \mathrm{Seq}[\alpha(\Gamma) \cup \alpha(\Delta)]$. We first define the basic invariant $I$ of the composition by

$$I^{\text{basic}}(h) \triangleq I_\Gamma(\mathbf{out}(h/\alpha(\Gamma))) \wedge I_\Delta(\mathbf{out}(h/\alpha(\Delta))) \wedge A^{out}(h).$$

However, the basic invariant predicate takes internal events into account. It is well-known that hiding corresponds to the introduction of existential quantifiers [1]. For the invariant, we extend the alphabet of $\Gamma + \Delta$ with the hidden internal events, and hide the extension inside an existential quantifier. Without inherited specifications, the derived invariant is

$$I^{out}(h) \triangleq \exists h' \in \mathrm{Seq}[\alpha(\Gamma) \cup \alpha(\Delta)] : h = h' \setminus \mathcal{I}(\Gamma + \Delta) \wedge I^{\text{basic}}(h').$$

*Inheritance.* We will assume that specifications can inherit other specifications like the interfaces of Section 4. Say that a specification $\Gamma$ inherits another specification $\Sigma$. At the semantic level, inheritance is interpreted as refinement: for all traces $h \in \mathcal{T}(\Gamma)$, we have that $h/\alpha(\Sigma) \in \mathcal{T}(\Sigma)$. At the syntactic level, inheritance restricts the set of possible traces $h$ defined by the assumption and invariant predicates by additional conjuncts of the form $A^{in}_\Sigma(h/\alpha(\Sigma)) \Rightarrow I_\Sigma(h/\alpha(\Sigma))$. However, in the composition $\Gamma + \Delta$, these additional conjuncts are only valid for the extended trace, so they must be placed *inside* the existential quantifier of the invariant. (Due to hiding, it is not the case that $\Gamma + \Delta$ directly inherits the super-interfaces of $\Gamma$ and $\Delta$.) Therefore, considering inheritance, we define the invariant of a composition as follows.

**Definition 8.** *For any specification $S$, we denote by $A_S$ and $I_S$ its assumption and invariant predicate, respectively. Consider two specifications $\Gamma$ and $\Delta$ and denote by $\Sigma_1, \ldots, \Sigma_n$ the specifications inherited by either $\Gamma$ or $\Delta$. The invariant of the composition $\Gamma + \Delta$ is defined as*

$$\begin{aligned} I^{out}(h) \triangleq \exists h' \in \mathrm{Seq}[\alpha(\Gamma) \cup \alpha(\Delta)] : h = h' \setminus \mathcal{I}(\Gamma + \Delta) \wedge \\ (\forall i \in \{1, \ldots, n\} : A^{in}_{\Sigma_i}(h'/\alpha(\Sigma_\mathrm{i})) \Rightarrow I^{out}_{\Sigma_i}(h'/\alpha(\Sigma_\mathrm{i}))) \wedge I^{\text{basic}}(h'), \end{aligned}$$

*with the associated proof conditions*

$$\forall h \in \mathrm{Seq}[\alpha(\Gamma)] : (A^{in}_\Gamma(h) \wedge I^{out}_\Gamma(h)) \Rightarrow A^{in}_\Delta(h/\mathcal{I}(\Gamma + \Delta)) \text{ and} \quad (1)$$

$$\forall h \in \mathrm{Seq}[\alpha(\Delta)] : (A^{in}_\Delta(h) \wedge I^{out}_\Delta(h)) \Rightarrow A^{in}_\Gamma(h/\mathcal{I}(\Gamma + \Delta)). \quad (2)$$

In order to maintain reasoning control for $\Gamma + \Delta$, output from $\Gamma$ should not break the assumption of $\Delta$ and vice versa. The proof conditions (1) and (2) ensure that the internal communications of $\Gamma + \Delta$ respect the assumptions $A_\Gamma$ and

$A_\Delta$ of the two component specifications $\Gamma$ and $\Delta$. Circularity in compositional proofs is avoided because assumption predicates concern traces that end with input whereas invariants concern traces that end with output.

With the assumption and invariant predicates derived for $\Gamma + \Delta$, we define the trace set $\mathcal{T}(\Gamma + \Delta)$ as the largest prefix-closed subset of

$$\{h \in \mathrm{Seq}[\alpha(\Gamma + \Delta)] \mid A^{in}(h, o) \Rightarrow I^{out}(h)\}.$$

### 6.1 Soundness of the Composition Rule

In this section, the proof rule for composition is shown to be semantically *sound*, i.e. that any trace in the semantically defined composed specification is included in the trace set obtained through the proof rule: $\mathcal{T}(\Gamma||\Delta) \subseteq \mathcal{T}(\Gamma + \Delta)$. This corresponds to the notion of soundness for regular verification systems, see for instance Apt and Olderog [4]. Here, the soundness proof relies on the distinction between input and output events; when we consider communication between two objects, input to one is output from the other, so we can reason inductively about the communication traces between the two objects. The proof extends a proof made by Dahl and Owe [17] for a somewhat simpler formalism.

*Proof.* Let $\Gamma, \Delta, \Sigma_1, \ldots, \Sigma_n$ (where $n \geq 0$) be (component) specifications such that every $\Sigma_i$ is inherited by either $\Gamma$ or $\Delta$. Assuming that the proof conditions (1) and (2) hold, we now show that $\mathcal{T}(\Gamma||\Delta) \subseteq \mathcal{T}(\Gamma + \Delta)$. Observe that, for any $h \in \mathcal{T}(\Gamma + \Delta)$, if one of $A^{in}_\Gamma(h/\alpha(\Gamma))$ and $A^{in}_\Delta(h/\alpha(\Delta))$ does not hold, neither does $A^{in}(h)$. Now, consider a trace $h \in \mathrm{Seq}[\alpha(\Gamma) \cup \alpha(\Delta)]$ such that $h \setminus \mathcal{I}(\Gamma + \Delta) \in \mathcal{T}(\Gamma + \Delta)$. By assumption, for any such trace $h$, we have that

$$A^{in}_\Gamma(h/\alpha(\Gamma)) \Rightarrow I^{out}_\Gamma(h/\alpha(\Gamma)), \text{ and} \tag{3}$$

$$A^{in}_\Delta(h/\alpha(\Delta)) \Rightarrow I^{out}_\Delta(h/\alpha(\Delta)). \tag{4}$$

We must show that if $A^{in}(h \setminus \mathcal{I}(\Gamma+\Delta))$ holds, then $I(h \setminus \mathcal{I}(\Gamma+\Delta))$ also holds. The proof is by induction over $h$, but we show a somewhat stronger result, namely

$$\begin{aligned} &\forall h : \mathrm{Seq}[\alpha(\Gamma) \cup \alpha(\Delta)] : \\ &\quad \forall i \in \{1, \ldots, n\} : A^{in}_{\Sigma_i}(h/\alpha(\Sigma_\mathrm{i})) \Rightarrow I^{out}_{\Sigma_i}(h/\alpha(\Sigma_\mathrm{i})) \\ &\qquad \wedge I^{out}_\Gamma(h/\alpha(\Gamma)) \wedge I^{out}_\Delta(h/\alpha(\Delta)) \\ &\qquad\quad \wedge A_\Gamma(\mathbf{in}(h/\mathcal{I}(\Gamma + \Delta))) \wedge A_\Delta(\mathbf{in}(h/\mathcal{I}(\Gamma + \Delta))), \end{aligned} \tag{5}$$

from which the invariant follows. Two observations are in order at this point. First, due to quantification over the objects of the communication environment in assumptions, $A^{in}(h \setminus \mathcal{I}(\Gamma + \Delta))$ entails

$$A^{in}_\Gamma(h \setminus \mathcal{I}(\Gamma + \Delta)/\alpha(\Gamma)) \wedge A^{in}_\Delta(h \setminus \mathcal{I}(\Gamma + \Delta)/\alpha(\Delta)). \tag{6}$$

Second, inheritance graphs are acyclic, so we can inductively assume soundness for inherited specifications. Once we have established $A^{in}_\Gamma$ and $A^{in}_\Delta$, we get

$$A^{in}_{\Sigma_i}(h/\alpha(\Sigma_\mathrm{i})) \Rightarrow I^{out}_{\Sigma_i}(h/\alpha(\Sigma_\mathrm{i})) \text{ for } 0 < i \leq n. \tag{7}$$

We now proceed with the proof. As we are dealing with safety properties, the formula (5) holds for the empty trace $\varepsilon$. Next, consider the induction step. We assume that (5) holds for a trace $h$ and show that it holds for $h \vdash m$, where $m \in \alpha(\Gamma) \cup \alpha(\Delta)$. Assume first that $m \in \alpha(\Gamma + \Delta)$, such that $A^{in}(h \vdash m)$. The induction hypothesis gives us

$$A_{\mathit{\Gamma}}(\mathbf{in}(h \vdash m/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))) = A_{\mathit{\Gamma}}(\mathbf{in}(h/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))) \text{ and}$$
$$A_{\mathit{\Delta}}(\mathbf{in}(h \vdash m/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))) = A_{\mathit{\Delta}}(\mathbf{in}(h/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))),$$

so by observation (6), both $A^{in}_{\mathit{\Gamma}}(h \vdash m)$ and $A^{in}_{\mathit{\Delta}}(h \vdash m)$ hold. Then, by observations (3) and (4), the invariants

$$I^{out}_{\mathit{\Gamma}}((h \vdash m)/\alpha(\Gamma)) \text{ and}$$
$$I^{out}_{\mathit{\Delta}}((h \vdash m)/\alpha(\Delta))$$

hold and by (1), (2), and (7), we can conclude that (5) holds for $h \vdash m$. Now, assume that $m \in \mathcal{I}(\mathit{\Gamma} + \mathit{\Delta})$. We have four possibilities. If we consider two object identifiers $o_1$ and $o_2$ such that $o_1 \in \mathcal{O}(\mathit{\Gamma})$ and $o_2 \in \mathcal{O}(\mathit{\Delta})$, these possibilities are $o_1 {\rightarrow} o_2.\mathrm{m}(\ldots)$, $o_1 {\leftarrow} o_2.\mathrm{m}(\ldots)$, $o_2 {\rightarrow} o_1.\mathrm{m}(\ldots)$, and $o_2 {\leftarrow} o_1.\mathrm{m}(\ldots)$. In the first case, $m$ is the initiation of a method m in $o_2$ by $o_1$. But then the event is an output event for $\mathit{\Gamma}$ and hence, the assumption of $\mathit{\Gamma}$ holds by the induction hypothesis:

$$\begin{aligned} A_{\mathit{\Gamma}}(\mathbf{in}((h \vdash m)/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))) &= A_{\mathit{\Gamma}}(\mathbf{in}(h/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}) \vdash m)) \\ &= A_{\mathit{\Gamma}}(\mathbf{in}(h/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))). \end{aligned}$$

Consequently, by observation (6), the assumption $A^{in}_{\mathit{\Gamma}}(h \vdash m)$ of specification $\mathit{\Gamma}$ holds. Therefore, by assumption (3), the invariant $I^{out}_{\mathit{\Gamma}}(h \vdash m)$ of specification $\mathit{\Gamma}$ holds. Now we know that both the assumption and invariant of $\mathit{\Gamma}$ hold, so by proof condition (1), the assumption $A_{\mathit{\Delta}}(\mathbf{in}(h \vdash m/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta})))$ holds. Then, by (6), we get the assumption of $\mathit{\Delta}$ and finally, by observation (4), the invariant $I^{out}_{\mathit{\Delta}}(h \vdash m)$ holds. As we have established the invariants of $\mathit{\Gamma}$ and $\mathit{\Delta}$, observation (7) lets us conclude.

In the next case, $m = o_1 {\leftarrow} o_2.\mathrm{m}(\ldots)$ is an output event from specification $\mathit{\Delta}$. Here, we first consider the assumption $A_{\mathit{\Delta}}$ of $\mathit{\Delta}$, which gives us:

$$A_{\mathit{\Delta}}(\mathbf{in}(h \vdash m/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))) = A_{\mathit{\Delta}}(\mathbf{in}(h/\mathcal{I}(\mathit{\Gamma} + \mathit{\Delta}))),$$

as $m$ disappears by projection. The predicate holds by the induction hypothesis. By similar reasoning to the previous case, we can now establish the invariants of $\mathit{\Gamma}$ and $\mathit{\Delta}$, and (7) gives us the result. The two last cases are similar to these.

## 7 Discussion

The main objective of this paper has been to show how Dahl's notion of object-oriented specification can be extended for reasoning about open distributed systems, as object-orientation is a natural paradigm for ODS [17,26]. In this section,

we draw some lines to related work and suggest some future extensions to the work we have presented in this paper.

Our approach is based on trace descriptions of (aspects of) the observable behavior of objects and components. Traces are well-known from the literature on processes, data flow networks, and modules [8, 25, 33, 39, 45]. These formalisms do not claim to be object-oriented and tend to be based on synchronous communication along channels, fix-point reasoning, and possibly infinite traces, in contrast to ABEL's approach. Object reference passing can be simulated using named channels instead of named objects, for example in the $\pi$-calculus [49], but using explicit object identifiers in the communication events allow a more natural representation. Explicit object identifiers may be found in languages such as Actors [2] and Maude [9]. Both these formalisms also allow asynchronous communication, exchange of object identities, and a large degree of modifiability. However, they are specialized towards system modeling rather than development and reasoning control, lacking for instance refinement notions that capture correctness for system development and modification.

We find specification in terms of observable behavior particularly attractive for reasoning about open distributed systems, where implementation detail need not be available for (client) objects in the environment. On the contrary, such detail can be intentionally hidden, being the intellectual property of some third-party manufacturer. In this respect, our approach is related to coalgebraic formulations of object-orientation such as [27], in which a class specification has assertions that equate sequences of observations on objects of the class. In general these assertions consider the entire history of the object and the conjunction of assertions thus resembles ABEL's history invariant. However, the coalgebraic approach does not seem to allow the kind of dynamic class extensions we have considered here, as the objects are semantically defined in terms of their statically given classes. Of course, state-based approaches may equally well supply a specification of an abstract state that does not directly reflect the implementation of a component. However, refinement becomes complicated when data structures change, in particular for aspects, described using different data structures, and dynamic extensions, captured in our formalism using projection on traces.

The idea of *separation of concerns* in specification seems to have originated with Parnas [44]. Partial specifications are perhaps best known for describing typical case-scenarios in specification notations such as Message Sequence Charts and UML. However, it is unclear how different cases relate to each other through composition and refinement in these notations. The use of interrelated viewpoints is recommended for ODS by the ITU [26] and work on combining viewpoints in this setting has been based on e.g. Object-Z [6, 19] and timed automata [5]. Two major differences between these approaches and ours are, first, that they are state-based whereas we prefer to model objects at an early stage by observations and, second, they are synchronous whereas we find asynchronous communication natural for distributed systems. Viewpoints as presented in this paper resemble aspects of aspect-oriented programming [34], describing aspects by their observable behavior as system services cross-cutting an object grid. Composition in

our formalism corresponds to synchronization of aspects, which suggests a formalism for specification and reasoning about the development of aspect-oriented programs. Further investigation in this context is future work.

Specifications using observable behavior let us describe objects in an abstract way, describing properties by extracting information from the history. Furthermore, we can model object behavior in a constructive *graphical* way with trace patterns. From such graphical specifications, the step to implementation in a state-based guarded command language is straightforward. Much work has been done on developing useful graphical specification notations, for instance with Statecharts, Petri nets, and UML, and on their formalization. Interestingly, there is also work on graphical representations of formal notations, an example being Actor specification diagrams [47]. Our trace patterns try to visualize behavior and could perhaps be expressed graphically in a similar way.

Finally, the formalism as presented here only considers safety specifications. In the context of open asynchronously communicating systems, liveness properties are largely dependent on the environment, which we do not control. A weak form of liveness is to identify *deadlock deterministic* objects, i.e. objects where deadlock is not due to internal non-determinism. For deadlock deterministic objects, we can to some extent reason about liveness properties by means of prefix-closed trace sets, without having to resort to a stronger apparatus including infinite traces, temporal logic, etc. We propose an incremental approach by including exceptions in the reasoning formalism through refinement, and in particular timeouts. Thus, we can reason about liveness properties of our own objects even when the environment is unstable. Initial research in this direction has been done in the context of fault tolerance [30], but more work remains.

## 8 Conclusion

The term "object-oriented specification" was coined by O.-J. Dahl for a specification style where the internal implementation details of objects are encapsulated and behavior is expressed in terms of permissible observable communication. An object's observable communication history represents an abstract view of its state, readily available for reasoning about past and present behavior. Using a (mythical) history variable, the behavior of an object is determined by its communication history up to present time. The approach emphasizes mathematically easy-to-understand concepts such as generator inductive function definitions and finite sequences, avoiding fix-point semantics and infinite traces.

In this paper, we have shown how this approach can be extended in order to reason about open distributed systems. In particular, we consider objects running in parallel, communicating by means of asynchronous remote method calls by which object identifiers can be exchanged. In accordance with the ITU [26], the approach supports partial specification by viewpoints, representing object behavior in behavioral interfaces. Openness appears in the formalism by allowing new (sub)classes, new interfaces for old classes, and a restricted form of dynamic class extension, while maintaining reasoning control.

## Acknowledgments

In developing the ideas for this paper, the authors have benefited from collaboration with the members of the ADAPT-FT project, and in particular with Ole-Johan Dahl and Isabelle Ryl.

## References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
2. G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.
3. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
4. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1991.
5. L. Blair and G. Blair. Composition in multi-paradigm specification techniques. In R. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 401–418. Kluwer Academic Publishers, Feb. 1999.
6. E. Boiten, J. Derrick, H. Bowman, and M. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, Sept. 1999.
7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Mass., 1999.
8. M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer-Verlag, 2001.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
10. O.-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, pages 57–114. Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, Dec. 1977.
11. O.-J. Dahl. Object-oriented specification. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, Series in Computer Systems, pages 561–576. The MIT Press, 1987.
12. O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.
13. O.-J. Dahl. The roots of object orientation: the Simula language. In M. Broy and E. Denert, editors, *Software Pioneers: Contributions to Software Engineering*. Springer-Verlag, June 2002.
14. O.-J. Dahl and K. Nygaard. SIMULA, an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, Sept. 1966.
15. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.
16. O.-J. Dahl and O. Owe. Formal development with ABEL. In S. Prehn and H. Toetenel, editors, *Formal Software Development Methods (VDM'91)*, volume 552 of *Lecture Notes in Computer Science*, pages 320–362. Springer-Verlag, Oct. 1991.

17. O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. Research Report 261, Department of informatics, University of Oslo, Norway, May 1998.
18. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.
19. J. Derrick, H. Bowman, and M. Steen. Viewpoints and objects. In J. P. Bowen and M. G. Hinchey, editors, *The Z Formal Specification Notation, 9th International Conference of Z Users (ZUM'95)*, volume 967 of *Lecture Notes in Computer Science*, pages 449–468. Springer-Verlag, Sept. 1995.
20. C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438. Chapman and Hall, London, 1997.
21. C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 3rd edition, 1998.
22. J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*. Addison-Wesley, 1986.
23. J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
24. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
25. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ., 1985.
26. International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.
27. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 210–231. Springer-Verlag, July 1996.
28. E. B. Johnsen and O. Owe. Composition and refinement for partial object specifications. In *Proc. 16th International Parallel & Distributed Processing Symposium (IPDPS'02), Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'02)*. IEEE Computer Society Press, Apr. 2002.
29. E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer Academic Publishers, Mar. 2002.
30. E. B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental fault-tolerant design in an object-oriented setting. In *Proc. Asian Pacific Conference on Quality Software (APAQS'01)*, pages 223–230. IEEE Computer Society Press, Dec. 2001.
31. E. B. Johnsen, W. Zhang, O. Owe, and D. B. Aredo. Combining graphical and formal development of open distributed systems. In M. Butler, L. Petre, and K. Sere, editors, *Proc. Third International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*, pages 319–338, Turku, Finland, May 2002. Springer-Verlag.
32. C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, UK, June l981.
33. G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proc. IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., Aug. 1974.

34. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.
35. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
36. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, June 2000.
37. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, Cambridge, Mass., 1993.
38. R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus.* Cambridge University Press, May 1999.
39. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.
40. O. Nierstrasz. A survey of object-oriented concepts. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison-Wesley, Reading, Mass., 1989.
41. K. Nygaard and O.-J. Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages.* ACM Press, 1981.
42. O. Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5:208–223, 1993.
43. O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. Research Report 278, Department of informatics, University of Oslo, Norway, Nov. 1999.
44. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
45. D. L. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Department of Computing and Information Science, Queen's University at Kingston, Kingston, Ontario, Canada, Oct. 1989.
46. G. Smith. *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer Academic Publishers, 2000.
47. S. F. Smith and C. Talcott. Modular reasoning for actor specification diagrams. In R. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 401–418. Kluwer Academic Publishers, Feb. 1999.
48. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.
49. D. Walker. Objects in the $\pi$-calculus. *Information and Computation*, 116(2):253–271, Feb. 1995.
50. A. Wang. Generalized types in high-level programming languages. Research Report in Informatics 1, Institute of Mathematics, University of Oslo, Jan. 1974. Cand. Real thesis.