

On the Development of the Active Object Paradigm: A Personal Account

Olaf Owe^[0000-0003-0976-5678]

Department of Informatics, University of Oslo, Norway
email: olaf@ifi.uio.no

Abstract. This paper gives a personal account of research activity performed over decades by the author and his collaborators towards developing expressive and efficient object-oriented language paradigms for distributed systems. The goal is to enable expressive and efficient programs with a mathematically simple semantics, so that programming and verification are as simple as possible. Starting from traditional object-oriented principles, the paper suggests ways to deal with major challenges related to concurrency, communication, inheritance, flexibility of code reuse, and openness. The resulting language can be classified as an active object language, which can be seen as a modernization of object orientation for concurrent systems. The language integrates several aspects of the Actor Model in an object-oriented, imperative setting. The focus of the paper is simplicity of modeling, specification and reasoning. Behavioral subtyping is replaced by the more liberal concept of *interface behavioral subtyping*, and a renewed notion of subclass inheritance allows unrestricted reuse of code and specifications, supported by reasoning control. These benefits are shown useful for subclassing as well as for program evolution. A notion of hierarchy independence limits reverification in the case of multiple inheritance, so that only changed classes and subclasses inheriting from them need to be reverified.

1 Introduction

This paper gives a personal account of the story of going from object-oriented languages to modern active object languages. It gives a summary of research activity over several decades, mainly driven by the author and his collaborators. The story starts with the cooperation with Ole-Johan Dahl and continues with the research development going from more classical object-oriented concepts to an active object language paradigm. The derived active object paradigm is similar to the Actor Model in many aspects, in particular with respect to the concurrency model and asynchronous message passing, but is somewhat different due to the embracement of object-oriented principles, such as the two-way interaction style inherent in the method concept. Active objects are concurrent (like actors) and cooperate by methods call, implemented by asynchronous message passing, while actors communicate solely by asynchronous message passing, see the survey [13]. Neither paradigm allows sharing through remote data access.

For distributed systems, the choice of what should be the unit of concurrency and the interaction and cooperation mechanisms, is essential. The Actor Model [1] is appealing because it has a compositional semantics, unifying interaction and cooperation mechanisms through message passing. It is clearly simpler than the shared variable setting and the general thread-based model where a thread may execute code on several objects, and where non-trivial interference complicates the semantics. For object-oriented systems, the natural unit of interaction is the object [17]. Concurrent objects interacting by remote method calls correspond to actors, allowing two-way interaction by means of remote method calls rather than one-way interaction by messages as in actors.

A problem with interaction by remote methods calls is that it traditionally relies on active waiting for the calling object. Efficient two-way interaction is possible when adding a *suspension* mechanism allowing a caller to handle other processes and method calls while passively waiting for a certain response. This can be realized by an *internal process queue* of uncompleted method activations for each object, combined with cooperative scheduling. Future-free (two-way) interaction simplifies programming and reasoning (see Section 4).

The class and interfaces of an object are traditionally statically defined, in contrast to actors, which are not confined by static class descriptions. Dynamic reconfigurability is essential in the Actor Model [4]. The requirement that each object permanently belongs to a class, imposes constraints on the behavior of an object [2]. It is shown below how to compensate for that by means of a *dynamic class construct*, allowing class updates and changes at runtime. Furthermore, dynamic features include a notion of dynamic subsystem with internal concurrency allowing objects to join and leave dynamically, and a notion of dynamic service discovery, allowing dynamic detection of objects in an environment.

It is well known that behavioral subtyping severely limits the flexibility of subclassing by requiring subclasses to maintain all properties of their superclasses, referred to as the *inheritance dilemma*. The limitations in reusability caused by behavioral subtyping, have made object-oriented subclassing less attractive in applicative and declarative languages, including actors [5] and rewriting logic [64,72]. In order to overcome the inheritance dilemma, each object variable is typed by an interface (rather than a class), called *interface abstraction*, and a renewed notion of inheritance is developed, supporting *interface behavioral subtyping* where a subclass may freely reuse superclass code, and may even violate superclass behavior, as long as the subclass interface is respected. The programmer has full control of what superclass behavior should be inherited and what should not be inherited. Class-wise reasoning is supported (see Section 5). The approach also applies to (a disciplined version of) multiple inheritance, which is crucial in the setting of class updates and program evolution.

This paper summarizes earlier work around active object languages, mainly by the author and his collaborators, presenting a language fragment with the following advantages and features: blocking and non-blocking method interaction, (single and multiple) inheritance, flexible reuse of code and specifications, dynamic update of class code, object and service discovery, subsystem abstraction,

history-based specification and reasoning, class-wise and compositional reasoning, and compliance checking of consent-based privacy aspects. These features are relevant for object-oriented languages as well as active object languages.

In order to accommodate the mentioned features, certain language constructs are added for increased efficiency and expressiveness, including *suspension*, *cointerfaces*, *superclass qualification*, *service discovery*, *superobjects* and *class updates*; and a certain programming discipline is required, such as *interface abstraction*, *history-based invariants*, and maintenance of class invariants upon suspension.

The guiding principle in this work has been ease of verification. The presentation is (mostly) high level, while technical details can be found in the underlying papers. Some simplifications are made here, including the treatment of events. The paper focuses on interaction and inheritance mechanisms, and does not discuss aspects related to self-calls and exception handling. The presentation avoids the use of futures since they complicate reasoning.

Paper outline. To provide a historical basis for the object-oriented philosophy, Section 2 gives an account of main ideas from the initial work on object-oriented languages by Ole-Johan Dahl and Kristen Nygaard. Their work was an inspiration for object-oriented and active object languages. The next sections present the language setting (Section 3), class-wise and compositional history-based reasoning (Section 4), a renewed notion of subclassing and reasoning support (Section 5), constructs for reconfigurability (Section 6), security and privacy (Section 7), and a conclusion (Section 8). An appendix shows notational conventions.

2 The Legacy of Ole-Johan Dahl and Kristen Nygaard

Ole-Johan Dahl and Kristen Nygaard received the Turing Award in 2001 for ideas fundamental to the emergence of object-oriented programming, through their design of the programming languages Simula I and Simula 67. Simula 67 contained classes, inheritance by subclasses, dynamic generation of objects, late (and static) binding, as well as a notion of static objects. Their motivation was system description and the need for a language to model real-world concepts, considering how to reflect real-world scenarios such as service-oriented systems and customers. Simula I was completed in 1964 and Simula 67 in 1967 [18,19]. Early versions of Simula (Simula I) used examples such as airport systems with passengers and clerks as “activities” [50]. According to the original ideas of Simula, an object would in general have its own activity, as well as data and operations. Objects with activity, called *active objects*, were reflecting “independent processes”, and objects without activity (but still with data and procedures) were called *passive* [50]. In Simula 67 these ideas were realized by allowing objects to be co-routines, the natural way at that time to imitate concurrent processes. In today’s world, a natural adaptation would be to let objects be concurrent, and one would obtain a distributed system by a set of objects running in parallel and interacting solely by remote method calls. This is the approach taken in Dahl’s later works [21,17]. Active objects were a key idea of Simula [12], and Simula

was an inspiration for the concurrency model of actors although Simula only supports co-routines and not true concurrency [48].

Simula was an extension of Algol 60, adding object-orientation. As Algol 60 did not have user-defined types, neither did Simula. This was seen as a weakness by Dahl and Nygaard, as well as the lack of protection of local class invariants [17]. Ole-Johan Dahl wrote: “At the Common Base Conference Kristen Nygaard and I proposed to enrich the language with class-like types. The proposal was turned down by the participants responsible for implementations. Today it would seem a better idea to build a programming language around a good general type concept, and subsequently add type-like classes” [15]. Dahl later worked with the author on algebraically defined types and how these can be integrated in an object-oriented setting [20,21,53,16,23].

As a main principle in his work, Dahl would take ease of verification as a measure of the quality of a program, and take simplicity of the verification system as a measure of the quality of a programming language. The book *Verifiable Programming* [16] elaborates on this. Clearly, verifiability does not give an absolute measure of quality since it depends not only on the programming language, but also on the specification principles and the reasoning system, and a more expressive language would in general be more difficult to handle than a simpler one. However, it may be used as a relative measure: If adding or changing a language construct increases verifiability, one may favor that change as long as expressiveness and efficiency are not sacrificed. *Ease of verification* is adopted as a guiding principle for the language discussions of this paper.

In joint work with Dahl [22], a methodology for class specification, implementation, and verification is suggested to remedy shortcomings in the RM-ODP standardization. The suggested language uses active objects for concurrent, mutable units that change over time, communicating by method calls and returns, while an algebraic, applicative sublanguage captures immutable data and abstract data types. A class definition has two parts, the *class specification* and the imperative *class implementation*. This has the advantage that interfaces and classes have the same name space, but gives somewhat less flexibility than separate interfaces. Class specifications (interfaces) are given by identifying the signature of the methods (called operations) which an object o of the class can handle, as well as any assumptions on a caller (if any), and the guarantees of the object, both expressed by predicates over the history of events. An assumption on a caller object c is expressed by the history of events to o initiated by c , and a guarantee is expressed by an invariant expressed over the history of events observed by the object, i.e., the events received by o plus the events initiated by o . Method implementations may be guarded by guards enforcing user requirements and/or enforcing the invariant of the enclosing class. When not satisfied, the former kind of guards leads to errors and the latter kind leads to delayed execution. The latter kind of guard could also be given in the middle of a method, delaying the remaining part of the method execution, meaning that each section of the method can be seen as a critical region, each maintaining the class invariant, and that an object could have a queue of delayed method executions.

A verification system is formalized, including class-wise verification (ensuring that the class implementation maintains the class invariant, and that the class invariant implies the class specification invariant). Compositional assumption/guarantee-style reasoning is formalized and demonstrated. Composition reduces to conjunction of guarantees, using a style of history-based inductive reasoning. The built-in notion of critical regions allows simple sequential-style class reasoning. Inheritance and so-called “heritage anomalies” were addressed.

The language (called OUN for *Oslo University Notation*) was further developed in [58,59], adding dynamic class updates, allowing classes to be modified at runtime, something we felt was essential for long-lived systems. The assumption/guarantee-style reasoning was formalized further in [58,59,35]. OUN was the basis for the *Creol* language [36,38,37], with a number of later extensions, including inheritance semantics [35]. A high level summary is given in [33]. Creol was in turn the basis for the *ABS* language [34], adding object groups and software product lines, but without support for object-oriented inheritance.

Below a core language developed from Creol is outlined, and a program example similar to that in [22] is given in Section 3.2. The example illustrates unrestricted reuse of code and behavior.

3 Language Setting and a Program Example

A core language is used to present and demonstrate main ideas, as well as examples. The active object paradigm presented here is imperative and supports standard object-oriented principles such as inheritance, encapsulation, and dynamic generation of objects. However, the objects are concurrent, long-lived, and can be used in ways beyond that of the standard object-oriented setting, supporting suspension, runtime discovery of objects and services, subsystem abstraction, and program evolution. Method invocations and returns are realized by message passing between the caller and callee in an asynchronous manner. Suspension allows an object to handle a number of processes, being incoming calls, continuations of calls, or self-calls representing self-activity. Non-terminating self-activity can be realized by loops or recursive cycles using suspension or non-blocking self-calls to allow periods with reactive behavior. A class may have an initial method, starting self-activity. This results in a language paradigm which follows the imperative, object-oriented programming paradigm more closely than the actor languages, while embracing the concurrency model of the Actor Model.

The language considered is strongly typed with declaration of interfaces and classes with unique names. An underlying (side-effect-free) applicative data type language allows definition of inductive data types, subtypes, and functions, including lists, sets, and finite mappings. Apart from standard statements such as assignment, different kinds of suspension and call statements are introduced.

Each object variable is typed by an interface stating the externally visible operations on the object and their behavior (*interface abstraction*), as suggested in [22,59,36,52], and also in the Actor Model [3]. This implies that remote access to fields is not possible, since interfaces do not have fields. Thereby sharing

as well as race conditions are avoided. We say that an object *supports* F if the runtime class of the object implements F or a subinterface of F . The type system guarantees that at runtime the object referred to by a variable of declared interface F will support F ; this is called the *interface substitution principle* [55].

Simple semantics is desirable, and even more, reasoning simplicity. In this setting, partial correctness is useful to express axiomatic semantics and to detect reasoning difficulties. As each method is typically small, method termination is often easy to detect. It suffices that each infinite loop or internal call-chain (ignoring one-way calls) has a suspension point. Blocking remote calls may cause deadlock, but deadlock is avoided when any chain of two-way remote calls has at least one suspension point [61].

3.1 Major Language Elements

An *interface* is defined by a number of superinterfaces (describing inheritance at the interface level), a number of method declarations (without body), and an interface invariant expressing visible, abstract object states. Due to interface abstraction, it is not meaningful to specify the state of an object through fields. (Another reason is that an interface may have several class implementations, possibly with different fields.) Furthermore, pre/post-conditions of a method may not be useful for a remote caller since fields used in the condition are not in the name space of the caller. However, pre/post-conditions are useful in order to deal with self-calls and reasoning inside a class, especially for inherited code.

Therefore, the *communication history* of an object represents the state of the object, and an interface invariant is expressed by means of this history. The communication history in a given state of an object o is the sequence of the events reflecting messages sent by o or messages received by o . These events define the *alphabet of the object*, denoted αo , and the history is called the *local history* of o . An interface F is described by an invariant over the communication history of the *self* object, excluding events reflecting self-calls, using the syntax **inv** $I_F(h)$ where h denotes the history of *self* where method completions by *self* are restricted to methods of F . For instance, an invariant of a bank account could say that the account balance is never negative, say **inv** $bal(h) \geq 0$ where the function bal may be defined by induction over the history h . Different interfaces of an object may then describe different usage of the object, for instance a *car* object may have an internet interface and a separate driver interface.

Cointerfaces enable type control of the caller object and enable type-correct call-backs. The method prefix **with** J specifies that the method's caller object must support the interface J , letting *caller* be an implicit parameter of type J . The method may do call-backs to *caller* using methods of J . This gives protection of the usage of the method, since only J objects may call it, and at the same time enables type-correct call-backs. The type system guarantees absence of method-not-understood errors at runtime [43,40]. In the example 3.2, interface Database and class DATABASE require that callers of the interface methods support interface User. Class DATABASE makes use of this cointerface to build its data structure.

A *class* C consists of fields (including class parameters) and a number of method definitions, plus a class invariant. Method definitions have local variables, and a body, given by a statement list ending with a return statement in each path. Formal parameters are read-only, due to reasoning considerations. A class invariant $I_C(h, w)$ is expressed by using the fields w and the local history (h) over the alphabet of the class C , as given by the methods defined in the class, including self-calls. Interfaces of the class are declared by the keyword **implements**, and direct superclasses are declared in a list (using **extends**). Flexible subclassing, not limited by behavioral subtyping, is introduced in Section 5.

We consider objects with their own virtual processor; thus at most one process in an object can be active at a given time. However, method activations may be *suspended* (by **await** statements) to allow other processes to continue, and thereby avoid active waiting. This allows sequential reasoning inside a class, and suspension control is programmed within each method. Thus, neither notification nor signaling from other objects or processes is needed, which simplifies reasoning. The suspension mechanism requires a *process queue* for each object.

Upon suspension, the the remaining part of a method invocation is suspended and placed in the process queue, while the object may continue with another (enabled) process or incoming call instead, while passively waiting for the suspended process to be enabled. Enabledness is controlled by guards, either a boolean condition b on the object state (fields) or the presence of a return value of a call made by the object. Scheduling is non-deterministic (but fair). The conditional suspension **await** b suspends while waiting for b to become true. The suspending call **await** $x := o.m(a)$ initiates the call to m and suspends while passively waiting for the callee o to complete the call. Here a is the parameter list.

Suspension can be used at the beginning of a method to ensure availability of some resources, or in the middle of a method, for instance to wait for the reply of a method call. As an example, a method *openRead* to provide read access to a database could start with **await** $writers = 0; readers := readers + 1$, and a method *openWrite* to provide write access could start with **await** $writers = 0 \wedge readers = 0; writers := writers + 1$, assuming that *readers* and *writers* are fields, and that the class invariant is $writers = 1 \wedge readers = 0 \vee writers = 0$. Readers and writers may use these operations to get access to the database using passive waiting, assuming that a database user calls *write* after *openWrite* and before *closeWrite*, and calls *read* after *openRead* or *openWrite* and before the corresponding closing operation. The protection can be enforced by using cointerfaces to keep track of the permitted objects, rather than using counters.

3.2 Example: Shared Database with Read/Write Protection

A class DATABASE to control read and write access to a database is sketched below, extending a class BASICDATABASE without protection. A cointerface is used to check that write operations of a user object are enclosed by *openWrite* and *closeWrite* calls of the user; and similarly, that read-only access of a user is enclosed by *openRead* and *closeRead* calls, for protection against active writers. The variable *wr* identifies the current writer, if any, and variable *rs* identifies the

current set of readers. Hence, `openWrite` requires `wr=null` and `rs=∅`. A similar example is treated in [22], including versions with reader and writer priority.

Interfaces are given in Fig. 1 and classes in Fig. 2. We assume the types:

```

1 type Data = ... // data, using no() to represent no data
2 type Fail = fail() // to indicate a failure
3 type Data? = Data | Fail // data or a failure

```

where $|$ denotes (disjoint) union of types, such that T is a subtype of $T|T'$.

An invariant is expressed by a predicate referring to the (predefined) history h of type *Hist*, and by functions defined after the keyword **where**. Inductive function definitions consist of several equations, typically one for each generator case, using **other** to match cases not yet dealt with. Generators for lists are *nil* and *append*. The specifications in this example use only one kind of event, namely reply events. The event $self.m(y;v)!c$ reflects that the current object sends a reply message to the caller c , where y are the formal parameters and v is the returned value. This can be abbreviated to $m(y;v)!c$ in local specifications, and to $m(y;v)!$ if c is not needed, and to $m(y)!$ if v is not needed.

The infix syntax $_;_$ is used for sequence append/concatenation, $_+_$ is used to add an element to a set, $_-_-$ is used to remove an element from a set, $_\&_$ denotes conjunction, and $_|_$ denotes disjunction. Type *Void* has only one possible value, *void()*. A mapping M is here a set of pairs, where $M + (k, d)$ adds the pair (k, d) , overwriting any earlier pairs with key k . The lookup function $M[k]$ gives the data value bound to k , if any, otherwise *no()*.

For simplicity, each declaration is preceded by a keyword, such as **var** (for fields, method-local variables, and mathematical variables), **op** (for methods), and **func** (for functions). Keywords are written in boldface, while *//* is used for comments. An **if** statement has a closing **fi**, but not an **if** expression. Variables and method names start with a lower-case letter, interface and type names are capitalized, and class names are written in upper case. For simplicity function definitions use variables as typed in formal parameters, possibly primed, and variables declared after **where**. In interface *Database* and class *DATABASE*, the **with** clause applies to all methods. Note that only the current writer may succeed with `closeWrite` and `write`, which is guaranteed by the use of cointerfaces!

The statement **return** e returns the value of e and terminates the process. It may be omitted for a *Void* method. The statement **return** $o.m(a)$ abbreviates

$x := o.m(a); \text{ return } x$ (where x is a type-correct local variable);

and the statement **return await** $o.m(a)$ is semantically the same as

await $x := o.m(a); \text{ return } x$

except that the process is terminated after initiating the call to m , delegating to o to return the value directly to caller [52].

A method name may be qualified by a superclass name to identify the superclass method. We allow both static and late binding of self-calls, using colon for the former, as in `BASICDATABASE:read(k)`. Function names may also be qualified (by a superclass or superinterface), allowing distinction between `Database:ok(h)` and `BasicDatabase:ok(h)`. Similarly, `A:inv` refers to the invariant of A , letting the syntax $A:\text{inv}$ denote $I_A(h/\alpha A, w')$, where w' are the

```

1 interface BasicDatabase{
2   op Data read(k:Key) // the returned value is val(h,k)
3   op Void write(k:Key,d:Data)
4   inv ok(h) where
5   func ok : Hist -> Bool
6     ok(nil) = true
7     ok(h;read(k;d)!) = d=val(h,k)
8     ok(h;write(k;d)!) = ok(h)
9   func val : Hist * Key -> Data
10    val(nil,k) = no()
11    val((h;read(k;d)!),k') = val(h,k')
12    val((h;write(k;d)!),k') = if k=k' then d else val(h,k')
13  }
14
15 interface Database{
16   with User {
17     op Void openRead()
18     op Void openWrite()
19     op Void closeRead()
20     op Void closeWrite()
21     op Bool write(k:Key,x:Data)
22     op Data? read(k:Key)
23   }
24   inv ok(h) where var b : Bool
25   func val : Hist * Key -> Data
26     val(nil,k) = no()
27     val(h;write(k;d;b)!,k') = if b & k=k' then d else val(h,k')
28     val(h;other,k) = val(h,k)
29   func ok : Hist -> Bool
30     ok(nil) = true
31     ok(h;write(k;d;b)!c) = ok(h) & b=(c=writer(h))
32     ok(h;read(k;d)!c) = ok(h) & d=
33       if c=writer(h) | c in readers(h) then val(h,k) else fail()
34     ok(h;openWrite()!) = ok(h) & writer(h)=null & readers(h)=∅
35     ok(h;openRead()!) = ok(h) & writer(h)=null
36     ok(h;other) = ok(h)
37   func writer : Hist -> User
38     writer(nil) = null
39     writer(h;openWrite()!c) = c
40     writer(h;closeWrite()!c) = if c=writer(h) then null
41                               else writer(h)
42     writer(h;other) = writer(h)
43   func readers : Hist -> Set[User]
44     readers(nil) = ∅
45     readers(h;openRead()!c) = readers(h) + c
46     readers(h;closeRead()!c) = readers(h) - c
47     readers(h;other) = readers(h)
48  }

```

Fig. 1. Types and Interfaces of the Database Example

```

1 class BASICDATABASE implements BasicDatabase {
2 // shared database with unprotected read and write operations
3 var db: Map[Key, Data]; // the database db is a mapping
4 op Data read(k:Key) {return db[k]} // lookup, possibly no()
5 op Void write(k:Key,d:Data) {db:=db+(k,d)} // overwriting
6 inv BasicDatabase:inv & forall k. db[k]=val(h,k)
7 }
8
9 class DATABASE implements Database extends BASICDATABASE {
10 var rs: Set[User] := ∅; // rs denotes set of current readers
11 var wr: User := null; // wr denotes current writer
12 inv (rs=∅ | wr=null) & forall k. db[k]=val(h,k)
13   & rs=readers(h) & wr=writer(h)
14 with User {
15   op Void openRead() {await wr=null;
16     rs:=rs+caller}
17   op Void openWrite() {await wr=null & rs=∅;
18     wr := caller}
19   op Void closeRead() {rs := rs-caller}
20   op Void closeWrite(){if caller=wr then wr:=null fi}
21   op Data? read(k:Key){if caller=wr | caller in rs
22     then return BASICDATABASE:read(k) else return fail() fi}
23   op Bool write(k:Key,x:Data){if caller=wr
24     then return BASICDATABASE:write(k,x) else return false fi}
25   }
26 }

```

Fig. 2. Classes of the Database Example

fields of class *A*. In the example, interface *BasicDatabase* could have used a postcondition for *read* (in line 2) specifying that the returned value is $val(h, k)$. Then the invariant and the *ok* function could have been omitted.

The use of guards allows us to control the scheduling of *Database* operations such that enabled processes correspond to meaningful actions, while non-enabled processes are delayed by means of passive waiting until they become meaningful.

The flexibility of our notion of subclassing (see Section 5) is illustrated in three ways: i) by unrestricted method redefinitions in subclass *DATABASE* (where *read* breaks contravariance, and *write* has an unrelated return type), ii) by overriding the superclass invariant, and iii) by overriding the declared interface of the superclass. The overriding is needed since the invariants of *BASICDATABASE* and *BasicDatabase* are not correct in the subclass. Redefined methods may call superclass methods as long as these calls are type-correct, as done in lines 22 and 24. Interface *Database* does not extend *BasicDatabase* since it does not satisfy the invariant of *BasicDatabase*. Reuse of code is increased by allowing unrestricted method redefinitions. A database is created by **var** *d* : *Database* := **new** *DATABASE*(), using the interface for typing.

3.3 Method Calls with and without Suspension

Consider a remote method call, using the syntax $x := o.m(a)$, where the callee o is an external object supporting a (type-correct) method m , a is the parameter list, and x is the variable receiving the return value. This reflects two-way interaction. In the setting of concurrent objects, the default way of performing such a method call is that the calling object blocks while the callee object (o) is executing the method when free to do so [21]. If the return value is not needed, one may use the syntax $o!m(a)$ to reflect one-way interaction and avoid waiting, called an *asynchronous call*. We distinguish between $o!m(a)$ and $o.m(a)$, where the latter call blocks until the call is completed, but without making use of the return value, whereas the first will continue without blocking (similar to message passing). The asynchronous call mechanism may be generalized to multicast and broadcast, messaging all objects of a given set or interface, respectively [33].

For two-way interaction, one may reduce the waiting time and improve efficiency by adding a list of statements s (a *tail*) allowing the caller to do something *while waiting*, using the syntax $x := o.m(a) \text{ do } s \text{ od}$ where the caller blocks after performing s if the return value has not arrived. A similar efficiency can be achieved with the use of futures [44].

These cases of blocking can be avoided with a suspending call

await $x := o.m(a)$ **do** s **od**

The caller makes the call to o , does s , and suspends if the return value of the call has not arrived. When a return value v arrives, the suspended process becomes enabled and performs $x := v$ when scheduled. The tail is optional and can be used to ensure that an invariant state is reached before suspending. An object making suspending calls is alive and able to respond to incoming calls and to finish suspended processes. For instance, a user of a Database d for storing mutual relationships between user objects could say

await $d.openWrite(); d.write(self, caller); d.write(caller, self); d.closeWrite()$

The process suspends while waiting for write-access, and the asynchronous call $d!closeWrite$ gives no waiting.

Note that the **do-od** construct allows a process to wait for replies from several outstanding calls to objects in the environment. For instance, the code **await** $x := o.m(a)$ **do** **await** $x' := o'.m'(a')$ **od**; $rest$ generates two calls before suspending, and when both return messages have arrived, the object continues with executing the rest of the process (captured by $rest$). Similarly, the code $x := o.m(a)$ **do** $x' := o'.m'(a')$ **od**; $rest$ generates two calls before waiting for both results. As shown in [44], the expressiveness of this style of future-free programming is superior to that of programming with explicit futures (i.e., declared future variables). A detailed comparison of different interaction paradigms with and without futures, their expressiveness and their operational and axiomatic semantics is given in [44]. Further improvements in efficiency can be obtained by allowing statements of the form $\{s_1 || s_2 || \dots\}$ where the statement lists s_i are executed in any order, but such that when one statement list suspends, another

may continue [37]. (Statement lists may run in parallel when there are no read/write conflicts on fields.) Further discussions on drawbacks and benefits of the future mechanism, as well as possible improvements, are given in [30,31].

A modeling framework for different kinds of communication media is given in [39], including wireless networks and radio communication, as well as loosely and tightly connected (FIFO) objects. As considered in Section 6.3, one may communicate in an unknown environment, using the syntax $x := \mathbf{acquire} F$ to find an object supporting interface F .

4 Class-wise and Compositional Reasoning with Histories

Section 4.2 describes compositional reasoning of (sub)systems, building on a simplified notion of events, given in Section 4.1.

In order to express partial correctness, we use Hoare-style triples of the form $\{P\} s \{Q\}$ where s denotes a statement(list), and P and Q are state assertions, with the meaning that if s terminates normally and the pre-state of s satisfies the precondition P , the postcondition Q must hold in the post-state of s . In particular, since there is no remote field access, assignment satisfies the classical partial correctness axiom of sequential languages, i.e., $\{Q_e^x\} x := e \{Q\}$, expressing that a precondition is derived from the postcondition by replacing each occurrence of the variable x by the expression e (renaming bound variables if needed).

For the example, maintenance of $rs = \emptyset \vee wr = \text{null}$ for `openRead` requires $\{rs = \emptyset \vee wr = \text{null}\} \mathbf{await} wr = \text{null}; rs := rs + \text{caller} \{rs = \emptyset \vee wr = \text{null}\}$ which reduces to $\{(rs = \emptyset \vee wr = \text{null}) \wedge wr = \text{null}\} rs := rs + \text{caller} \{rs = \emptyset \vee wr = \text{null}\}$ by the following Hoare-style axiom for reasoning about conditional suspension:

$$\{I_C(h, w)\} \mathbf{await} b \{I_C(h, w) \wedge b\}$$

This gives the verification condition $wr = \text{null} \Rightarrow wr = \text{null}$, which is trivial. Both method `openRead` and `openWrite` maintain $rs = \emptyset \vee wr = \text{null}$, but not without the `await` statements.

4.1 Events, Histories, and Local Reasoning

For the setting of two-way method interaction implemented by message passing, one may in general consider four kinds of communication events. For a method call $o.m(a)$ made by c with parameter list a , these events would be i) the initiation of the call made by c , ii) the reception of the call by o , iii) the reply of the call made by o , and iv) the reception of the reply by c .

However, for specification purposes, the reply events are in general more interesting than the call events [22]. For a call $o.m(a)$ made by an object c , we therefore consider the following two kinds of events in our histories:

- $o.m(a;v)!c$, reflecting that o sends to c the reply to the call $o.m(a)$

- $o.m(a; v)?c$, reflecting that c receives from o the reply to the call $o.m(a)$

The former is in the alphabet of o (the callee) and the latter in the alphabet of c (the caller). This means that distinct objects have disjoint alphabets, which makes compositional reasoning possible. The alphabet of a class or interface contains events of the form $self.m(a; v)!c$ and $o.m(a; v)?self$, which may be abbreviated to $m(a; v)!c$ and $o.m(a; v)?$, respectively. Wellformedness of the global history of a system ensures that every occurrence of $o.m(a; v)?c$ must be preceded by an occurrence of $o.m(a; v)!c$. This states that a message must be sent before it can be received, and that values received are those that are sent.

Moreover, each of these events can be connected to the code: $m(y; v)!caller$ can be seen as a side-effect of the return statement of m where y is the list of formal parameters; and $o.m(a; v)?$ can be seen as a side-effect of the completion of a call $o.m(a)$ where v is the returned value. More specifically, for partial correctness reasoning, the statement **return** e can be understood as the assignment

$$h := (h; self.m(y; e)!caller)$$

where y is the list of formal parameters. Furthermore, a remote blocking call $x := o.m(a)$ **do** s **od** can be understood as the statement sequence

$$s; x' := some; h := (h; self.m(a; x')?o); x := x'$$

where *some* represents an underspecified value, used here for reasoning purposes. This value is then determined by means of compositional reasoning considering a subsystem containing both the caller and callee object. Wellformedness ensures that the received value in the reply event of the caller, must be equal to the returned value of the corresponding return event of the callee. The auxiliary variable x' is needed to avoid binding any occurrences of x in a .

By means of a non-executable **assume** statement used for the purpose of partial correctness, conditional suspension **await** b can be understood as

$$w := some; h := (h; some); \text{assume } I_C(h, w) \wedge b$$

where **assume** has the axiom $\{P\} \text{assume } b \{P \wedge b\}$. The assumption ensures that the invariant and b hold after suspension. A suspending call with a tail, **await** $x := o.m(a)$ **do** s **od**, can be understood as the sequence

$$s; w := some; x' := some; h := (h; some; o.m(a; x')?); x := x'; \text{assume } I_C(h, w)$$

Thus, the new values of w, x, h satisfy the invariant.

From this understanding, one may derive reasoning rules for each of the constructs, including the simple rule for conditional suspension above and the following rule for return statements

$$\{I_C(h; m(y; e)!caller, w)\} \text{return } e \{I_C(h, w)\}$$

where y is the list of formal parameters and e is the expression returned.

For a blocking remote call, say $x := o.m(a)$, one can derive the reasoning rule

$$\{\forall x'. Q_{x'h;o.m(a;x')?}^x\} x := o.m(a) \{Q\}$$

Here the quantifier reflects that x is locally unknown, and x' is needed to avoid binding occurrences of x in e . In the precondition, x and x' refer to the value of x before and after the call, respectively. For a suspending call, the rule is

$$\{I_C(h, w) \wedge \forall w, x', h'. Q_{x'h;h';o.m(a;x')}^{x,h}\} \textbf{await } x := o.m(a) \{I_C(h, w) \wedge Q\}$$

Here the quantification of w and h' reflects that fields may have changed and that the history has grown, but such that the receive event is the last after the call, see [52] for further details.

Reasoning about calls, return, and suspension can therefore be done by sequential-style Hoare logic, including the side-effects on the history, and using universal quantification in the precondition of statements modeled with use of *some*. Blocking self-calls are implemented as for sequential programs (by means of a runtime stack) and do not contribute to events recorded in the history. This makes reasoning about blocking self-calls different from that of blocking remote calls. The reasoning rule for blocking self-calls is omitted. Section 5 will discuss reasoning about inherited code.

Note that two identical calls by the same caller may return different results since the calls could be scheduled at different times when the callee has different states, and this may differ with different executions. Reasoning with simplified events as in our approach, entails that one does not see which reply belongs to which call. This corresponds to the non-determinism inherent in asynchronous communication. In contrast, explicit futures allow identification of results with calls, but different executions may result in different results, so one may control the identification but not the resulting values. Thus, explicit futures are not helpful in reasoning, and also complicate history-based specifications in that more variables are needed. Explicit futures introduce an extra level of complexity in compositional reasoning since one would need to match call-events with return events by means of future identities, and these identities are unknown at reasoning time, giving rise to additional quantification [25,26]. Global reasoning is simpler with future-free events as in our approach than with explicit futures.

4.2 Compositional Reasoning about Systems

Compositional reasoning about concurrent objects is difficult with pre/post-conditions since the caller and callee have different name spaces. For remote calls, fields used in the conditions of a callee are not in the name space of the caller, even the histories have different alphabets, and the histories may have grown before the call is executed. At the interface level, one should not talk about fields since different class implementations of the interface may use different fields.

Reasoning by history invariants does not have these problems. Compositional reasoning about concurrent systems can be derived from the principle of Neelam Soundarajan [63], which uses a *compatibility predicate* relating the global and local histories. This can be exploited in our setting as follows: At the beginning of the execution of a method, and after suspension, one may assume that the class invariant holds. In return one must verify that it is reestablished at method end

and upon suspension. When interacting with another object, one may assume that the object will satisfy its interface invariant. This gives rise a reasoning principle for systems and subsystems of concurrent objects, deriving an invariant for a (sub)system S , say $I_S(h)$ where h is the history of S , i.e., the history of all objects in S . The local history of an object o in S will then be $h/\alpha o$. This use of disjoint alphabets simplifies the notion of compatibility of [63].

We may derive an invariant for S from the invariants of the objects in S : The invariant for the (sub)system is the conjunction of the interface invariants for each object in the (sub)system, applied to the local history of the object, together with a clause $wf(h)$ stating that the the total history is wellformed:

$$I_S(h) = wf(h) \wedge \bigwedge_{(o:F) \in S} I_{o:F}(h/\alpha o)$$

where $I_{o:F}(h)$ is the invariant of interface F where all explicit and implicit occurrences of *self* are replaced by o . Note that projection by the relevant alphabet captures the connection between the history of S and the histories of the individual objects. Wellformedness expresses that a message can only be received after it is sent, and that values received are as sent.

From $I_S(h)$ one can obtain an external specification of S (for the communication with its environment) by considering the external history of S . The external history of S , $external_S(h)$, is defined inductively by ignoring internal events inside S and replacing internal objects by S . The external invariant of S is then $I_S(external_S(h))$. A user-defined external invariant for S , say $J_S(h)$, is proved if it follows from I_S , i.e., $I_S(external_S(h)) \Rightarrow J_S(h)$. Section 6.3 will look at a notion of dynamic subsystems where the set of internal objects and their interfaces can be derived from the history.

The compositional reasoning rule can be generalized to the assumption/guarantee reasoning style, which allows each interface invariant to rely on assumptions on the environment. Assumptions typically restrict input events. The assumptions can be discharged by induction of the history, showing that each output event of an object satisfies the assumptions of the receiver object in addition to its own guarantee, assuming that earlier events satisfy all assumptions and guarantees. Contract specifications for subsystems can then be handled. This style of reasoning has been explored in [22,58,59,35].

The compositional reasoning rule can also be generalized to synchronous events, allowing objects to have overlapping alphabets. For instance, the event reflecting an **acquire** statement could be a synchronous event between the caller and the chosen object. Synchronous events of two or more objects, would appear in their local histories and also in the composed history. Their correspondence is captured by the projection operator letting all objects agree on the synchronous events. This could be used to capture generation of objects and superobjects, and also for explicit synchronization statements, say of multi-object movements.

A more high level and abstract way of providing synchronization is by means of restrictions on the system history. These restrictions must then be enforced by the runtime system and need not be verified. With respect to the compositional

reasoning rule, such implicit synchronization restrictions may be embedded in the wellformedness predicate. These techniques are similar to reasoning techniques of the actor framework [70,6], and their support for coordination [3].

5 Reasoning in Presence of Unrestricted Sublassing

The traditional way of reasoning about classes is based on the use of classes for type control. An object variable typed by a class C may at runtime refer to an object of a subclass of C . The natural way of reasoning is then to ensure that all subclasses of C satisfy the behavior of C [47], as done in Java reasoning [46]. Thus, all future subclasses of C must respect the specifications given in C , something which puts behavioral restrictions on all subclasses of C , unless the specifications given in C are weak. Neither is desirable, especially in an open-ended class hierarchy where the purpose of future subclasses is not known a priori.

In contrast, our setting builds on the *interface substitution principle*: An object variable typed by an interface F may at runtime refer to an object supporting F , and classes may not be used for typing. The natural way of reasoning is then to ensure that all subinterfaces of F satisfy the behavior of F . One may even let classes redefine behavior. In this setting there are no restrictions on code and specification reuse! Each class is verified separately, ensuring that each object of that class satisfies its interfaces. No requirements are imposed on subclasses. Subclasses may or may not reuse behavior.

In order to verify a class C , one must verify that all specifications given in C are satisfied, that inherited methods (exported by an interface) also satisfy the invariant of C , and that the use of inherited code agrees with the invariant of C . For instance, reasoning about suspension in inherited code must use the invariant of C . The advantage is that late binding can be resolved at verification time, and that the class specification of C can be strong since it only applies to C objects, and need not be satisfied for subclasses [55].

Independence of code reuse from reuse of specification gives flexible reuse. But since a class C may violate behavior in a superclass, reasoning results from a superclass cannot in general be reused in C . Thus, reasoning results from the superclass must in general be redone or re-validated in the context of C . This means that we should be more specific about the context of a reasoning result. The verification of a class does not depend on subclasses, but it will depend on superclasses from which it inherits code or behavioral specifications.

5.1 Unrestricted Reuse of Code and Behavior

In order to combine behavioral subtyping with unrestricted reuse of code and behavior, a non-standard notion of subclassing is used. Methods may be overridden as usual, and also *superclass behavior may be overridden*: Class invariants may be overridden simply by stating a new invariant, possibly reusing the invariant of a superclass as a conjunct. If no invariant is given, those of the superclasses

are inherited. Likewise, interfaces of superclasses may be overridden simply by stating a new **implements** list; and if no new list is given, those of the superclasses are inherited. Inheritance of pre/post-conditions of methods is handled similarly. In this way the programmer has full control of what superclass behavior should be inherited and what should not be inherited. A class may therefore violate the class invariant of a superclass it extends, and a method may violate the specification given in a superclass. This approach, called *behavioral interface subtyping* [55], supports the *interface substitution principle*, and is more liberal than lazy behavioral subtyping [27,28]. The need for this is demonstrated in the example in Fig. 2 where class DATABASE violates the interface invariant of its superclass.

A redefinition of a method from an interface must obey contravariance, but a redefinition of a method from a superclass need not! For instance, the redefinition of *read* in DATABASE has a smaller input type (due to the with-clause) and larger result type, and the redefinition of *write* has a different return type than in BASICDATABASE.

A class must satisfy its invariant initially and at suspension and at method end (and similarly for pre/post-conditions of methods), and must satisfy the invariants of interfaces it implements. Further details can be found in [52,51]. An interface invariant is satisfied by C if it implies the interface invariant for the class history when restricted to the alphabet of the interface, i.e.,

$$I_C(h, w) \Rightarrow I_F(h/\alpha F)$$

where $_/_$ is the projection operator, reducing a history by a set of events, giving the subsequence where events outside the set are ignored.

5.2 Multiple Inheritance and Healthiness

Multiple inheritance is useful during evolution, allowing an old class to be updated by adding a new superclass and a new interface. However, the combination of multiple inheritance and late binding may cause confusion when several classes define the same methods. To avoid confusion, we insist on a *healthy binding* strategy. The late binding of a self-call occurring in a method of class B should bind to a class that is strongly related to B , otherwise it may bind to a class that is not intended as a subclass of B and cause an unexpected binding. Two classes A and B are said to be *strongly related*, denoted $A \lesseqgtr B$, if $A \leq B$ or $B \leq A$, where $A \leq B$ is the transitive and reflexive subclass relationship. Thus, two classes are strongly related if they are the same or if one is a direct or indirect superclass of the other. A late bound call must not bind to a superclass of the actual class that is not a subclass of the class of the call.

The healthy binding strategy suggested in [55] is considering the class of the executing object (the actual class), the superclass where a given self-call occurs (the class of the call), and the class defining the called method detected at runtime (the class of the method). Healthy binding requires that the class of the method is strongly related to the actual class and to the class of the call.

We allow static and late bound self-calls, and use superclass names to qualify a method name when needed, letting the syntax $A:m$ denote static binding of the m defined or inherited in A , and letting $A.m$ denote late binding of the m defined or inherited in A . (Thus, dot-notation is used both for remote calls, $o.m$, and for qualified self-calls, $A.m$.) In class A , the syntax $A:m$ and $A.m$ may be abbreviated to $:m$ and $.m$, respectively. Fields may be qualified as in $C:x$. For late binding, it is required that $A.m$ is bound to a class that is strongly related to both A , the class of the call, and the actual class (not a subclass of the actual class). This principle is called *healthy binding*.

In case of multiple inheritance, binding can be based on the order given in the superclass list, using the first superclass if that gives a type-correct binding, similar to how mixins work [32]. During evolution, new superclasses are added at the end of the superclass list. This way, old bindings of $A.m$ are not affected by new superclasses of A . For both static and late binding, it is required that the static typing detects at least one candidate for the call. This guarantees that binding will succeed at runtime.

The binding of static self-calls can be resolved statically, $A:m$ is bound to the first class found with a type-correct method definition, searching through the superclasses of A , denoted $bind(A, m)$. The binding of late bound calls cannot in general be resolved statically, since the same method call may be bound differently depending on the actual class. However, in the setting of interface abstraction, it suffices to verify each class C implementing an interface F , letting the verification of C be done under the assumption that C is the actual class at runtime. This means that we know the runtime class at verification time. Thus, late binding can be resolved at verification time, for each choice of actual class.

However, this requires reverification of inherited code when earlier verification results have a different actual class. In particular, the treatment of suspension and self-calls in inherited code depends on the actual class C , as detailed below.

5.3 Reasoning about Inherited Code

In order to reason about inherited code in a class C , we will therefore use reasoning statements of the form

$$\vdash_{C,B} \{P\} s \{Q\}$$

expressing that $\{P\} s \{Q\}$ is satisfied in the context of the actual class C with B as the class enclosing s . Binding of late bound self-calls in s is then done by:

- $bind(C, B, A.m) = D$ if $C \leq D \wedge D \leq A \wedge D \leq B$, taking the first such D with a *suitable* definition of m , searching from C .

A definition is “suitable” if it is a contravariant redefinition of the method defined/inherited in B . In contrast, remote calls $o.m$ are late bound and typed against the definition of the method in the interface of o .

For instance, the axiom for conditional suspension in 4 is now reformulated as $\vdash_{C,B} \{I_C(h, w)\} \mathbf{await} b \{I_C(h, w) \wedge b\}$, using the invariant of C , and not the

one of B . For non-inherited code in C , we use $\vdash_{C,C} \{P\} s \{Q\}$. Verification of a pre/post-condition pair (P, Q) for a method m inherited by C from class B is done by

$$\vdash_{C,B} \{P\} \text{body}_{B,m} \{Q\}$$

where $\text{body}_{B,m}$ denotes the body of m in class B . For self-calls $A:n$ and $A.n$ in this body, verification of a pre/post condition pair (P, Q) is then handled by

$$\vdash_{C,D} \{P\} \text{body}_{D,n} \{Q\}$$

where D is $\text{bind}(A, n)$ or $\text{bind}(C, B, A.n)$, respectively. If the method is exported through an interface of C , we must verify

$$\vdash_{C,B} \{I_C(h, w)\} \text{body}_{B,n} \{I_C(h, w)\}$$

and in particular, for methods m defined in C itself and exported by an interface

$$\vdash_{C,C} \{I_C(h, w)\} \text{body}_{C,m} \{I_C(h, w)\}$$

In addition, we must verify any pre/post-conditions imported or explicitly stated in C , and these may then be used in the proofs of maintenance of the invariant. Often $\vdash_{B,B} \{P\} s \{Q\}$ implies $\vdash_{C,B} \{P\} s \{Q\}$, for instance when s does not contain suspension nor local calls, and then reverification of s is not needed in the context of C . In this case, one may conclude $\vdash_{C,B} \{P \wedge L\} s \{Q \wedge L\}$ where L refers to fields of C that are not fields of B . For the example in Section 3.2, this is the case when C is `DATABASE` and B is `BASICDATABASE`. It follows that the redefined `read` and `write` methods maintain the invariant of `DATABASE`.

Finally, one must prove that class C satisfies the interface invariant of each implemented interface, either directly from the invariant (as explained in Section 3.1) or indirectly by reconsidering each method implementation. For a method m implemented by $s; \text{return } e$, where e is an expression, and an interface invariant $ok(h)$ with the equation $ok(h; m(y; x)!c) = ok(h) \wedge R(h, y, x, c)$, one must prove (in context C):

$$\{I_C(h, w)\} s \{I_C(h; m(y; e)!caller, w)\} \text{ and } \{I_C(h, w)\} s \{R(h, y, e, caller)\}.$$

For `openWrite` the latter reduces to $\{I_{\text{DATABASE}}(h, db, rs, wr) \wedge wr = \text{null} \wedge rs = \emptyset\} wr := caller \{writer(h) = \text{null} \wedge readers(h) = \emptyset\}$, which is trivial using the `DATABASE` invariant and $ok(h; \text{openWrite}())!$. Proving the interface specification for the other methods is also straight forward since the example is simple.

6 Constructs Supporting Reconfigurability

This section discusses three ways of supporting reconfigurability and openness (in addition to dynamic generation of objects): Program evolution, dynamic reconfigurability, and a notion of dynamic subsystems and service discovery.

6.1 Program Evolution

Programs need to be maintained for several reasons, for instance to extend the services of a class or subsystem, to accommodate new user requirements, to

restructure the class hierarchy, to fix bugs, privacy, or security weaknesses in the code, or to improve specifications and provide better guarantees. This will involve changes in code or specifications, often both. One should be able to change a class in the middle of a class hierarchy, not only at the end. This means that classes below a changed class can be affected if they reuse code that is changed. This is useful for refactoring, by reordering the inheritance hierarchy.

Flexibility is desirable and essential in this respect, in the sense that one should be able to make changes to the code of a system as needed, without any restrictions. One should also be able to make changes to specifications; and there should be reasoning support such that reverification is simple.

We sketch an approach [55] that supports full flexibility, in the sense that one may change a class in whatever way desired (not violating type correctness). The approach supports *hierarchy independence* and *modification independence*, meaning that changes in one class C will only affect the class itself and subclasses. This means that superclasses of C need not be reverified, and that reverification of subclasses can be done in a cycle-free manner. Moreover, if C needs to be changed, this can be done independently of the changes of subclasses.

However, program evolution requires reverification, each modified class and each subclass must be reverified. Making a new version of a class C can be compared to making a subclass of C and then replacing C by that subclass. Our notion of subclassing and inheritance allows flexibility with respect to reuse of code and behavior. Reverification of C is then similar to what has been explained above for inheritance. In order to make indirect superclass dependencies more direct, the use of $B:m$ and $B.m$ should require that B itself defines method m .

The support for flexibility is described below: Program evolution can be done by adding new classes, subclasses, interfaces, and subinterfaces, and by changing old classes. In particular, one may provide more functionality, and old functionality may be reduced. The functionality of a class may be extended by adding methods and making them visible to the environment by adding support for new interfaces. Methods may be added by defining new method definitions in a class, or in a superclass, or by adding a superclass that provides new methods. The only restriction is that one may not modify interfaces. However, an interface F may be removed when all occurrences of F are replaced by an updated version of the interface, provided implements clauses are reverified.

Variables, methods, interfaces, and classes may be removed as long as type correctness is maintained. One may remove an interface F from the implements clause of a class C provided the typing of generated objects of class C remains correct. And one may upgrade a class C by adding a new interface F and adding a new superclass C' , letting C' implement the new methods in F . As mentioned, a redefined method need not respect the typing of superclass definitions, but any given class may only define one method with the same name. This flexibility allows superclasses to be added later without checking the typing of subclasses.

In this setting, multiple inheritance can be useful assuming a healthy binding strategy to avoid confusion and unexpected bindings of late bound self-calls when

several classes define the called methods. The proof of hierarchy independence in [55] builds on healthiness.

6.2 Dynamic Reconfigurability

In the setting of long-lived systems, there is a need to renew or update the software at runtime. An approach to do this is by means of a dynamic class upgrade mechanism, allowing one or more classes to be changed at runtime and allowing addition of new classes and interfaces [58,59,42]. The runtime system defined in [42] allows such upgrades by special messages. After updating a class the existing objects of the class can be updated independently and at different times, for instance when they are idle. The approach entails that one does not need to stop the system, instead the updates are done gradually, and one does not need to stop an object that is being updated. The objects may even run some processes with an old version of the code and some with the new. The runtime system uses a simple version control system to make the objects aware of which version they are running and when they need to update. Safe type checking can be adapted to this setting [71]. The dynamic class update mechanism is somewhat similar to program evolution except that the updates are done gradually at runtime. Much of the same verification techniques can be used; however, the presence of new and old code in the process queue is a complication, which can be overcome when old and new invariants are maintained. Otherwise, an object must be upgraded when the process queue is empty, and after establishing the new invariant. A similar notion of runtime upgrades was independently developed for the Erlang language [7,66], and new versions of code may coexist with old versions. Moreover, Erlang supports major ideas of the active object paradigm, but in a functional language setting. Thus, invariant-based verification as developed here is not supported.

6.3 A Notion of Dynamic Subsystems

Following [14,40], we allow dynamic generation of “superobjects”. A *superobject* is a subsystem that can be referenced, and can be seen as a normal object from the outside, but is implemented by a set of concurrent objects on the inside, in contrast to the notion of object groups (cogs) in the *ABS* language [34]. The set of objects inside a superobject can change dynamically. Objects may dynamically join and leave a superobject, possibly changing the interfaces of the superobject.

A generated superobject is initially empty, but objects may join or leave the superobject, and at the same time enrich the interface of the superobject. Let O be a superobject, then the statement **join** O **as** F can be made by an object o supporting F to join O and make interface F available from outside the superobject. From the outside one may then use O as a normal F object, say by making calls to it, and such a call is handled by one of the objects inside the superobject exporting F (or a subinterface of F). If there are several such objects, one of them is chosen non-deterministically to receive the call request (letting an idle F object, if any, take the call). Furthermore, an object in O may

leave O , or withdraw some of its services to the environment by the statement **leave** O **as** F . Typing implications for the environment are avoided when other objects in O still support F . Join and leave events are recorded in the history.

In this dynamic setting, there is a need to discover objects and services, and we add a construct for service discovery. The statement $x := \mathbf{acquire} F \mathbf{in} O$ detects some object o in O that supports F and assigns it to x . The choice of o is non-deterministic, but idle objects are preferred. Discovery of services outside a group may also be very useful, using the syntax $x := \mathbf{acquire} F$. In addition, one may use broadcast to reach all objects of a given interface, by an asynchronous call to the interface, say *Waiter!fire()* **in** *restaurant*, using the example below.

As an example, a *restaurant* may be defined as a superobject with a number of internal *Waiter* objects, and an internal superobject *kitchen*, with personnel working in the kitchen. The virtual *kitchen* object exports a method *makeDish* with *Waiter* as cointerface. A customer coming from the outside could acquire a *Waiter* object ($myWaiter := \mathbf{acquire} \textit{Waiter} \mathbf{in} \textit{restaurant}$), and then order dishes from that waiter, say **await** $myWaiter.makeOrder(\dots)$ (or $myWaiter!makeOrder(\dots)$ if the result is not needed), which in turn makes a call to the kitchen, say **await** $kitchen.makeDish(\dots)$. This means that the waiter is not blocked and gets notified when the dish is ready, while the person in the kitchen taking the order will be busy making the dish.

7 Other Aspects and Considerations

Security. Security aspects for active objects include information flow analysis through a secrecy-type system to capture inter-object communication [57,62], defining local and global non-interference for active objects in a non-deterministic runtime environment. Static analysis through a secrecy-type system and a trace analysis system capture inter-object and network communication, respectively. Successful static analysis guarantees *interaction non-interference* [62].

Static detection of call flooding is reported in [56], which can be exploited for static detection of distributed denial of service (DDoS) attacks [29]. The detection is hard to do dynamically without access to the system program code.

A notion of *security wrappers* prevents leakage of secure information from concurrent objects. A service provider may use security wrappers to protect its services in an insecure environment. The run-time system automatically adds wrappers to insecure components [45]. Security and safety wrappers can be used in several settings [60], but are here used for communicating by means of futures since the origin of a future is implicit, which causes security challenges. Cointerfaces facilitate improvements of both security and privacy aspects [62,69].

Privacy. For active object languages language-based approaches to privacy have been studied in several ways, focusing on consent-based privacy inspired by GDPR. Language extensions are made to be able to talk about concepts such as data subjects, entities, purposes, and access rights. Privacy policies are then formulated by means of these concepts, and a notion of *privacy compliance* is defined. Certain privacy violations can be found by static checking, while runtime checking is required for full compliance, and for runtime changes in the policies.

In particular, [69] defines a static system that guarantees a static notion of privacy compliance, while [68] defines a runtime system ensuring compliance, where the compliance checks are made in the kernel of the operating system, which means that it could be used for other languages. At runtime, data subjects can make consent changes and request to see all personal data stored about them. An approach where personal data may refer to several data subjects is given in [9], guaranteeing compliance by means of a runtime system. In [8] the runtime checks are generated through static analysis and transformed to checks at the start of a method activation. This means that activations not satisfying the checks will stay on the process queue until they are satisfied. This has the advantages of avoiding program executions that “get stuck” due to runtime errors resulting from privacy checks. The approach also avoids redoing the same checks.

Privacy compliance essentially checks each data access involving personal information, ensuring that the data subject has given consent to the responsible entity to exercise the access in question for the given purpose. However, in both static and runtime analysis there are challenging questions, such as: how to detect the purpose, how to detect the data subject, and how to detect who is responsible for the processing, and how responsibility is transferred to data processors. The above approaches differ in these respects, using a static or dynamic way of solving a question, either automatically or by explicit help from the programmer.

Smart Contracts. As smart contracts cannot be changed, support for modeling, specification, and reasoning is essential. An active object language specialized for smart contracts, developed in [54], gives protection against tampering and gives added trust. It provides mechanisms for safety, security, and privacy, with class-wise verification. It is shown that history-based specification and reasoning are useful for smart contracts. The history is part of the executable language (read-only) and abstracts information normally found on blockchain. The approach is more high level than Solidity and offers less need for roll-backs.

Implementations and Tool Support. There exist operational semantics for the different versions of the language paradigm, and executable interpreters built from the operational semantics using the Maude modeling language [67,72]. This includes versions with time [11,41] and probabilities [10]. The basic active object model can be supported on top of Java by means of a Java package [49].

There exists maintained support, including compilers and a number of tools, for the ABS language [65], which has the same core concurrency model with suspension, but without support for cointerfaces and object-oriented inheritance. However, ABS supports many other features, including exception handling, futures, real-time, software product lines, digital twins, and cloud programming. This paper does not convey the contributions of the ABS language. The present language fragment has a simpler reasoning system than ABS and its notion of delta-oriented specifications, see for instance [24] using symbolic assumptions.

Possible Impact. The benefits of simple class-wise reasoning as above could be adopted in a real programming language with active objects, by allowing user-defined data types and functions, cointerfaces, and suspension, while insisting on interface abstraction and history-based invariants for classes and interfaces.

Support for flexible inheritance and reasoning requires free reuse/redefinition of superclass invariants and declared interfaces, requiring healthy binding of methods in the case of multiple inheritance. Subsystem abstraction and privacy compliance checking can then be added. Popular object-oriented languages such as Java, JML, and Ruby do not enforce interface abstraction.

8 Conclusion

The paper looks at how imperative, object-oriented languages can be devised to support distributed systems by means of active objects running in parallel, combining active and reactive behavior, and exchanging information by method interaction, implemented by asynchronous message passing. The focus is on simplicity of modeling, specification, and reasoning. For this reason, the kernel language is future-free and without remote field access. The paper gives a personal account based on contributions of the author.

Efficient two-way method interaction is met by allowing suspension, and in particular, the combination of method calls with suspension enables passive waiting for method replies. The caller then has the freedom to choose the most suitable interaction style, choosing passive or active waiting, regardless of the method definition. The *cointerface mechanism* is useful for security and privacy, in addition to improved typing control, even for call-backs.

The considered programming language fragment has a compositional semantics, allowing compositional reasoning based on event histories and invariants over these, exploiting that the objects have disjoint alphabets for asynchronous events. Class-wise verification is done by sequential style partial correctness reasoning, incorporating the effects on the local event history. Compositional reasoning is done by conjunction of the local invariants of composed objects, with the addition of a requirement expressing wellformedness of the composed history.

Based on *interface abstraction*, a flexible form of inheritance supports unrestricted reuse of code and specifications. Interfaces and invariants of a superclass can be overridden as needed. Program evolution based on this notion offers full flexibility and reasoning control, in the sense that code and behavior may be changed as desired, and reverification is supported, limiting the verification conditions to changed classes and subclasses inheriting from these. *Healthy binding* is required for multiple inheritance. The flexibility and reasoning control of inheritance and program evolution extend state-of-the-art.

An approach to *subsystem abstraction* allows an object to represent a subsystem consisting of concurrent objects. This is useful in system design to reduce the complexity of large systems, and provides openness by allowing the subsystem to be dynamic, both with respect to the interfaces offered by the subsystem and with respect to its internal object structure. Service discovery is supported.

In order to deal with dynamic reconfiguration, dynamic class upgrades allow a class to be revised, changed, or upgraded during runtime. This mechanism is supported by an operational semantics that allows the updates to be implemented object-wise so that different objects of the same class are updated independently and at different times, with an underlying notion of version control.

The language fragment outlined here is in many ways inspired by the Actor Model and has a similar concurrency model, using active objects in a traditional, imperative setting with Hoare-style verification support and history-based object invariants. Rather than using futures and replacements as in the Actor model, suspension is used to avoid undesired blocking.

Acknowledgments. I am indebted to Ole-Johan Dahl for decades of close collaboration, and to Kristen Nygaard, Neelam Soundararajan, José Meseguer, Carl Hewitt, and Gul Agha for inspiration. I thank previous PhD students, colleagues, and coauthors for cooperation; and I thank Peter Ölveczky for valuable comments and encouragement!

References

1. G. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
2. G. Agha. An overview of actor languages. *SIGPLAN Not.*, 21(10):58–67, 1986.
3. G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.
4. G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *LNCS*, pages 19–41. Springer, 1985.
5. G. Agha and P. Thati. An algebraic theory of actors and its application to a simple object-based language. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, pages 26–57. Springer, 2004.
6. G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 135–153. Springer US, 1997.
7. J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.
8. C. P. Baramashetru, P. Giannini, S. L. T. Tarifa, and O. Owe. A type system for data privacy compliance in active object languages. *Art Sci. Eng. Program.*, 2025.
9. C. P. Baramashetru, S. L. T. Tarifa, and O. Owe. Integrating data privacy compliance in active object languages. In F. S. de Boer, F. Damiani, R. Hähnle, E. B. Johnsen, and E. Kamburjan, editors, *Active Object Languages: Current Research Trends*, volume 14360 of *LNCS*, pages 263–288. Springer, 2024.
10. L. Bentea and O. Owe. A probabilistic framework for object-oriented modeling and analysis of distributed systems. In B. Beckert, F. Damiani, and D. Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *LNCS*, pages 105–122. Springer, 2011.
11. J. Björk, E. B. Johnsen, O. Owe, and R. Schlatte. Lightweight time modeling in timed Creol. In P. C. Ölveczky, editor, *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems*, volume 36 of *EPTCS*, pages 67–81, 2010.
12. A. P. Black. Object-oriented programming: Some history, and challenges for the next fifty years. *Inf. Comput.*, 231:3–20, 2013.
13. F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5), Oct. 2017.

14. D. Clarke, E. B. Johnsen, and O. Owe. Concurrent objects à la carte. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *LNCS*, pages 185–206. Springer, 2010.
15. O.-J. Dahl. Value types and object classes. *ASU Newsletter*, 20(1):8–20, 1992. <https://www.mn.uio.no/ifi/english/about/ole-johan-dahl/bibliography/value-types-and-object-classes.pdf>.
16. O.-J. Dahl. *Verifiable programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.
17. O.-J. Dahl. The birth of object orientation: The SIMULA languages. In *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 15–25. Springer, 2004.
18. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center Report S-2, 1968, 1968. (Revised version: Norwegian Computing Center Report 743, 1984).
19. O.-J. Dahl and K. Nygaard. SIMULA – an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.
20. O.-J. Dahl and O. Owe. A presentation of the specification and verification project ABEL. *ACM SIGSOFT Softw. Eng. Notes*, 10(4):28–32, 1985.
21. O.-J. Dahl and O. Owe. Formal development with ABEL. In S. Prehn and W. J. Toetenel, editors, *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe*, volume 552 of *LNCS*, pages 320–362. Springer, 1991.
22. O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. Technical Report 261, University of Oslo, Dept. of Informatics, 1998. https://www.nb.no/items/URN:NBN:no-nb_digibok_2009042000005?page=19.
23. O.-J. Dahl, O. Owe, and T. J. Bastiansen. Subtyping and constructive specification. *Nord. J. Comput.*, 5(1):19–49, 1998.
24. F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A transformational proof system for delta-oriented programming. In E. S. de Almeida, C. Schwanninger, and D. Benavides, editors, *16th International Software Product Line Conference, SPLC '12, Volume 2*, pages 53–60. ACM, 2012.
25. C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebraic Methods Program.*, 83(5-6):360–383, 2014.
26. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
27. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915–941, 2011.
28. J. Dovland, E. B. Johnsen, O. Owe, and I. C. Yu. A proof system for adaptable class hierarchies. *Journal of Logical and Algebraic Methods in Programming*, 84(1):37–53, 2015.
29. E. Fazeldehkordi, O. Owe, and T. Ramezanifarkhani. A language-based approach to prevent DDoS attacks in distributed financial agent systems. In *Computer Security - ESORICS 2019 International Workshops, Revised Selected Papers*, volume 11981 of *LNCS*, pages 258–277. Springer, 2019.
30. K. Fernandez-Reyes, D. Clarke, E. Castegren, and H. Vo. Forward to a promising future. In G. D. M. Serugendo and M. Loreti, editors, *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018*, volume 10852 of *LNCS*, pages 162–180. Springer, 2018.

31. K. Fernandez-Reyes, D. Clarke, L. Henrio, E. B. Johnsen, and T. Wrigstad. Godot: All the Benefits of Implicit and Explicit Futures. In A. F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
32. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 171–183, New York, NY, USA, 1998. ACM.
33. E. B. Johnsen, J. C. Blanchette, M. Kyas, and O. Owe. Intra-object versus inter-object: Concurrency and reasoning in Creol. In J. Zhao and V. Stolz, editors, *Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software*, volume 243 of *Electronic Notes in Theoretical Computer Science*, pages 89–103. Elsevier, 2008.
34. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *LNCIS*, pages 142–164. Springer, 2010.
35. E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In *Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCIS*, pages 137–164. Springer, 2004.
36. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Softw. Syst. Model.*, 6(1):39–58, 2007.
37. E. B. Johnsen, O. Owe, and M. Arnestad. Combining active and reactive behavior in concurrent objects. In *Proc. Norsk Informatikkonferanse (NIK'03)*. Tapir, 2003.
38. E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. *Electronic Notes in Theoretical Computer Science*, 117:375–392, 2005. Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004).
39. E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas. An object-oriented component model for heterogeneous nets. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects*, volume 5382 of *LNCIS*, pages 257–279. Springer, 2007.
40. E. B. Johnsen, O. Owe, D. Clarke, and J. Bjørk. A formal model of service-oriented dynamic object groups. *Sci. Comput. Program.*, 115-116:3–22, 2016.
41. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. T. Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software*, volume 6528 of *LNCIS*, pages 46–60. Springer, 2010.
42. E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 3535 of *LNCIS*, pages 15–30. Springer, 2005.
43. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1-2):23–66, 2006.
44. F. Karami, O. Owe, and T. Ramezanifarkhani. An evaluation of interaction paradigms for active objects. *J. Log. Algebraic Methods Program.*, 103:154–183, 2019.
45. F. Karami, O. Owe, and G. Schneider. Information-flow control by means of security wrappers for active object languages with futures. In M. Asplund and S. Nadjm-Tehrani, editors, *Secure IT Systems*, volume 12556 of *LNCIS*, pages 74–91. Springer, 2020.

46. G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.*, 37(4):13:1–13:88, 2015.
47. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
48. O. Madsen. Building safe concurrency abstractions. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *LNCS*, pages 66–104. Springer, 2014.
49. C. McDowell and O. Owe. Towards a light-weight approach for concurrent active objects in Java. Technical report, University of California, Santa Cruz (UCSC), 2015. <https://users.soe.ucsc.edu/~charlie/pubs/creolJava2C.pdf>.
50. K. Nygaard and O.-J. Dahl. The development of the SIMULA languages. *ACM SIGPLAN Notices*, 13(8):245–272, 1978.
51. O. Owe. Reasoning about inheritance and unrestricted reuse in object-oriented concurrent systems. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM*, volume 9681 of *LNCS*, pages 210–225. Springer, 2016.
52. O. Owe. Verifiable programming of object-oriented and distributed systems. In L. Petre and E. Sekerinski, editors, *From Action Systems to Distributed Systems - The Refinement Approach*, pages 61–79. Chapman and Hall/CRC, 2016.
53. O. Owe and O.-J. Dahl. Generator induction in order sorted algebras. *Formal Aspects Comput.*, 3(1):2–20, 1991.
54. O. Owe and E. Fazeldehkordi. A lightweight approach to smart contracts supporting safety, security, and privacy. *J. Log. Algebraic Methods Program.*, 127:100772, 2022.
55. O. Owe, E. Fazeldehkordi, and J. Lin. A framework for flexible program evolution and verification of distributed systems. In S. Hammoudi, L. F. Pires, and B. Selic, editors, *Model-Driven Engineering and Software Development, Revised Selected Papers*, volume 1161 of *Communications in Computer and Information Science*, pages 320–349. Springer, 2019.
56. O. Owe and C. McDowell. On detecting over-eager concurrency in asynchronously communicating concurrent object systems. *J. Log. Algebraic Methods Program.*, 90:158–175, 2017.
57. O. Owe and T. Ramezanifarkhani. Confidentiality of interactions in concurrent object-oriented systems. In J. García-Alfaro, G. Navarro-Arribas, H. Hartenstein, and J. Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 10436 of *LNCS*, pages 19–34. Springer, 2017.
58. O. Owe and I. Ryl. OUN: A formalism for open, object-oriented, distributed systems. In B. Victor and W. Yi, editors, *NWPT’99: The 11th Nordic Workshop on Programming Theory*, 1999. Full version: <https://www.mn.uio.no/ifi/english/people/emeriti/olaf/papers/oun-aug99.pdf>.
59. O. Owe and I. Ryl. On combining reasoning control with distribution and openness in object oriented systems. In M. Haverlaen and Owe, editors, *NWPT’00: The 12th Nordic Workshop on Programming Theory*, 2000. Full version: <https://www.mn.uio.no/ifi/english/people/emeriti/olaf/papers/ounrr278.pdf>.
60. O. Owe and G. Schneider. Wrap your objects safely. *Electronic Notes in Theoretical Computer Science*, 253(1):127–143, 2009. Proceedings of the Sixth International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2009).

61. O. Owe and I. C. Yu. Deadlock detection of active objects with synchronous and asynchronous method calls. In *Proceedings of 27th Norsk Informatikkonferanse, NIK 2014*. Bibsys Open Journal Systems, Norway, 2014.
62. T. Ramezanifarkhani, O. Owe, and S. Tokas. A secrecy-preserving language for distributed and object-oriented systems. *J. Log. Algebraic Methods Program.*, 99:1–25, 2018.
63. N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):647–662, 1984.
64. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. In *From Object-Oriented to Formal Methods: Essays in Memory of Ole-Johan Dahl*, pages 334–375. Springer, 2004.
65. The ABS Home Page. The ABS language. <https://abs-models.org/>.
66. The Erlang Home Page. The Erlang language. <https://www.erlang.org/>.
67. The Maude Home Page. The Maude system. https://maude.cs.illinois.edu/wiki/The_Maude_System.
68. S. Tokas and O. Owe. A formal framework for consent management. In A. Gotsman and A. Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 12136 of *LNCs*, pages 169–186. Springer, 2020.
69. S. Tokas, O. Owe, and T. Ramezanifarkhani. Static checking of GDPR-related privacy compliance for object-oriented distributed systems. *J. Log. Algebraic Methods Program.*, 125:100733, 2022.
70. A. Yonezawa and C. Hewitt. Modelling distributed systems. In R. Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 370–376. William Kaufmann, 1977.
71. I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in Creol. In R. Gorrieri and H. Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *LNCs*, pages 202–217. Springer, 2006.
72. P. C. Ölveczky. *Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude*. Undergraduate Topics in Computer Science. Springer, 2017.

A Notational Conventions for Meta-Variables

Upper-Case Letters

A, B, C, D	class
F, J	interface
I	invariant
L	local assertion
O	superobject
P, Q, R	state predicate
S	(sub)system
T	type

Lower-Case Letters

a	actual parameter list
b	boolean condition
c	(caller) object
e	expression
h	history
m, n	method
o	object
s	statement (list)
v	value
x	variable
y	list of formal parameters
w	list of fields