UNIVERSITY OF OSLO
Department of Informatics

OUN: A Formalism for Open, Object Oriented, Distributed Systems

Olaf Owe and
Isabelle Ryl

Research report 270

August 1999

UNIVERSITAS OSLOENSIS • MDCCCXI •

OUN:
A Formalism for Open, Object Oriented, Distributed Systems

Olaf Owe and Isabelle Ryl

August 1999

Department of Informatics
University of Oslo
{olaf,isabelle}@ifi.uio.no

Abstract

OUN is a notation developed within the frame of the ADAPT-FT project. The goal of this project is to provide a platform supporting the formal development of object oriented open distributed systems. The purpose of this platform is to support formal reasoning and, in contrast to most formal methods, deal with the main aspects of open distributed systems such as mobility, flexibility, and reflection. In the platform the notation will be used combined with UML, but is not dedicated to it.

The ultimate purpose of the platform is to enable development of systems satisfying given safety requirements. Safety reasoning is difficult unless the constructs of the specification and design language are chosen carefully. We therefore introduce our own language, OUN, for system specification and design. Openness is taken into account by allowing dynamic addition of (sub)interfaces and (sub)classes, as well as dynamic extension of classes. And as usual in object oriented languages, objects may be created dynamically and their identities may be communicated. To facilitate reasoning, we insist on static typing and static correctness proofs. This leads to some language restrictions, compared to Corba and Java RMI.

OUN is a trace-based notation which allows us to specify contracts, interfaces and classes using high level object oriented concepts like multiple inheritance of interfaces and classes, and allowing implementation of several interfaces by a class. Interface specifications characterize observable behavior of one object, contracts restrict the behavior of a subsystem with two or more objects, while class specifications characterize local attributes as well as internal and external behavior. The notation will take account of synchronous as well as asynchronous communications.

It is essential that the notation is practically useful, and not only reserved for specialists with strong mathematical background: The basic concepts are easy-to-understand, the formulas are expressed in first order logic and the syntax is easy-to-use.

1 Introduction

We are aiming at a formalism which allows us to specify and develop object-oriented open distributed systems in a way that enables safety and liveness reasoning about them. As much as possible, we want to include high level object-oriented programming language concepts supporting openness, but restricted such that reasoning is manageable. In particular, generation of proof obligations at run-time has to be avoided. Thus, system reasoning control will be based on static typing and static proofs, and the generation of verification conditions will be based on static analysis of pieces of program or specification text.

We allow a form of partial compilation, such that pieces of code can be compiled at different times and added to a running system, without restarting or stopping it. The compilation units may build on each other in a natural way. In particular, a compilation unit may add functionality to an old class, by means of so-called "class extension". Each partial compilation will generate a set of verification conditions to be proved. The language ensures that already proven verification conditions do not need to be reproved when adding new compilation units, not even when dynamically extending a class with new methods, but sometimes new verification conditions are needed.

Neither Java nor Corba are suitable for static reasoning. We therefore reconsider object-oriented language constructs supporting both openness and reasoning control. Even though the main focus of this work is at the specification level, we choose to sketch a high level object-oriented design language supporting our goals. In particular we present an imperative class concept allowing dynamic extension of methods and support of new interfaces. This enables us to design systems where old objects may communicate with newer objects through new interfaces (as well as old super-interfaces). The language is chosen so that dynamic extension of classes can be implemented efficiently and without stopping or interrupting the running system.

Due to static typing, software errors such as "method not understood" and "illegal parameter types" may not appear at run time. (However, hardware errors might be reflected by network errors, and objects not responding.) The formalism ensures that proved invariants can not be violated, say, by dynamic extensions of classes or addition of new interfaces or classes.

Objects are considered to have internal activity, running in parallel. An object may support a number of interfaces, and this number may increase dynamically. The notation offers a large degree of *flexibility*: several communication paradigms can be used, operations can be redefined without severe syntactic or semantic restrictions, unrestricted overloading of operators is allowed, multiple inheritance is allowed. Finally, certain kinds of *reflection* is considered: for example one may ask an object if it supports a given interface.

We suggest below a high level object-oriented language concept, supporting the above aims, but where openness is restricted by the above aims, as well as by implementation issues. We will here limit ourselves to consider safety properties (corresponding to safety properties in [1] except deadlock freedom), stated by invariants and a kind of rely-guarantee specifications similar to that introduced in [8]. This allows us to use a rather simple semantics, based on trace sets. A forthcoming paper will deal with liveness and deadlocks.

1.1 Fundamentals of the notation

At the most abstract level, the specification language allows us to specify objects, interfaces, and contracts. At the design level, the programming language allows us to define classes. In both cases, requirement specifications consist of invariants and assumptions about the behavior of the environment, given as formulas on the history of the (sub)system. The purpose of this paper is to present the main aspects of the notation and briefly indicate their semantics. The OUN notation leans on a standard trace semantics, similar to that of CSP (without refusals and divergences) [7], except that objects have identity.

Object variables and object parameters are typed by interfaces, even at the design level. This simplifies reasoning, since an interface is used to describe observable behavior of a certain aspect (role) of an object. However, at run time a given object will belong to one class, which is fixed but may be extended dynamically by adding operations and thereby implementing additional interfaces.

A class is said to *implement* an interface if all operations in the interface are implemented in the class (with the same parameter lists, except for formal parameter names) and if the requirement specification of the class implies that of the interface after appropriate projection of the history (see below). When the class of the object implements several interfaces we say that the object *supports* these interfaces.

In addition there is an underlying language for defining data types, including predefined types such as integer, boolean, character and text. This sub-language is not discussed further; however, in the examples we will use that of [5].

1.2 Openness: Dynamic extension of programs

For a traditional sequential system it is usual to consider the program as a single piece of program text (perhaps with an external library). An open distributed system may be the result of several program pieces written at different times, and perhaps at different locations. One way of achieving openness is to allow incremental addition of code without restarting or stopping the overall system. These program units may then depend on each other in non-trivial ways.

We consider a partial order of compilation units, reflecting their availability in time. A unit may then depend on another one if and only if the latter is less than the former in the ordering. For instance units developed on different locations, at more or less the same time, may not depend on each other and are unordered. Compilation units developed at the same location are assumed to be totally ordered.

The kinds of system additions that might be allowed depend on the programming language (and operating system). In particular we are interested in concepts based on the principles of object orientation. We will not limit ourselves to a particular existing language (even though our notions are inspired by Java and Corba). And as already stated we will limit ourselves to language constructs that enable us to stay within the framework of static typing and avoiding runtime tests creating proof obligations.

We will consider the following kinds of dynamic program extensions: addition of new classes and subclasses, addition of new interfaces and sub-interfaces, addition of new "implements" claims, addition of new operations to a class, strengthening the invariant of a class by an additional conjunct. And of course, a compilation unit may create new objects (of new and old classes), as may the execution of an operation. These issues are described a bit more below.

Dynamic aspects

Our notion of openness is closely connected to that of object orientation:

1. New objects may be created dynamically, and their identities may be submitted as operation parameters, allowing old objects to be aware of newer ones, and vice versa.

2. New (sub-)classes and new (sub-)interfaces may be added to a running system without recompiling the whole system. Thus a running system may be extended by new classes and interfaces, as well as new subclasses and new sub-interfaces, without stopping the running system.

   Notice that the combination of 1 and 2 allows old objects to talk to new objects of new classes through old super-interfaces.

3. Additional "implements"-claims (C *implements* I) may be stated in a compilation unit when needed. In order to establish the fact that a given class implements a given interface, one needs a syntactic check as well as a semantic check which in general generates proof obligations. Such a claim may be stated outside the class or interface (say, at the outmost level of a compilation unit), allowing such relationships to be established incrementally, even when the compilation units defining the class C and the interface I are unordered!

   For instance, this allows relating an old class to a newer interface, or an old interface to a newer class.

4. A class may be extended by adding one or several new operations, and the invariant may be strengthened, see above. Assuming that the run-time system will implement such an extension by loading the code for the operations (in the appropriate place), and then making references to them from the object representing the class (say from a table of operations local to the class), no restart of the run-time system is needed.

5. Operations added to a class are inherited by any (old or new) subclass. Since classes can not be modified from other locations, and since the compilation units on one location form a total order, it will not be possible to (directly or indirectly) modify classes in conflicting ways. As will become clear, this kind of class extension would be difficult if subclasses were to inherit invariants.

The combination of the above allows old and new objects to communicate trough new interfaces!

The dynamical additions and extensions are allowed in such a way that they do not change anything to properties already proved. So the system is incremental, the security of old parts of the system can not be violated by any new part (provided the generated verification conditions are proved).

2 Basics

2.1 Histories

Objects are not static parts of a system, they evolve through interaction with the environment. One may think of the current "state" of an object of a system as resulting from its past interactions with the environment by way of operation calls. Accordingly, the basic concept of OUN will be the notion of finite communication history. The local communication history of an object provides an abstract view of its "state".

A communication history is a sequence of events, where each event is atomic (relative to the current abstraction level) and is related to an operation call. There are two kinds of events: initiation that denotes the initiation of the call of an operation, and termination that denotes the completion of the call of an operation (and return of out-parameters to the calling object). Thus, events recorded in the history represents initiation or termination of operations including associated parameter values and the names of the initiator and the receiver of the operation. They are on the form:

$$\begin{array}{rl} \text{operation initiation:} & o_1 \rightarrow o_2.m(i_1, \ldots, i_j) \\ \text{operation termination:} & o_1 \leftarrow o_2.m(i_1, \ldots, i_j; r_1, \ldots, r_k) \end{array}$$

where m is the name of the called operation and where o_1 and o_2 are respectively the initiator and the receiver of the call. For an initiation event, $i_1, \ldots, i_j$ denotes the **in**-parameter values whereas $i_1, \ldots, i_j; r_1, \ldots, r_k$ denotes the **in**-parameter and **out**-parameter values for a termination event. The record of **in**-parameter values for a termination event may seem to be redundant, however, this form of events often provides an easy way of writing specifications, ignoring initiations. There is an underlying assumption on histories ensuring that for each occurrence of a completion there is a corresponding initiation occurring earlier in the history.

Events may also be seen through the viewpoint of a particular object. For an object o, events may be considered as being inputs or outputs:

$$\begin{array}{rl} o \text{ inputs:} & x \rightarrow o.m(i_1, \ldots, i_j), o \leftarrow x.m(i_1, \ldots, i_j; r_1, \ldots, r_k); \\ o \text{ outputs:} & o \rightarrow x.m(i_1, \ldots, i_j), x \leftarrow o.m(i_1, \ldots, i_j; r_1, \ldots, r_k). \end{array}$$

Note that we assume that, at the most abstract level, objects are connected by an idealized net. However, for each object we represent "external and internal" queues by means of the information in the histories. Imperfect channels may be modeled by separate objects.

2.2 Communication

We consider asynchronous operation calls with return. The syntactic definition of operations is on the following form:

$$\textbf{opr}\ \mathit{my\_operation}\ (\textbf{in}\ p_1 : T_1, \ldots, p_i : T_i; \textbf{out}\ p_{i+1} : T_{i+1}, \ldots, p_j : T_j).$$

Operations may have **in-** and **out**-parameters, typed by data-types or by interfaces. The keyword **in** is default and may be omitted. In particular, object identifiers can be transmitted as parameters. Notice that the identities of the initiator and the receiver of a call are implicit parameters, which can be exploited, both for specification and implementation purposes.

As an operation has an implicit return, the calling object receives an indication of the termination of the operation along with the values of any out-parameters. As operations are asynchronous, and as waiting is allowed in the implementation of the operations, both the calling and the called object can be involved in other observable activity between the initiation and the termination of the operation. Thus, the call of an operation is recorded in the traces on the form:

$$\ldots o_1 \rightarrow o_2.m(\ldots) \ldots o_1 \leftarrow o_2.m(\ldots;\ldots) \ldots$$

Note that the definition of operations is flexible and allows us to consider pure asynchronous communications – when the caller does not wait for the completion indication – as well as pure synchronous communications – when the caller waits for the termination indication.

2.3 Notational conventions

In the following, $\mathcal{H}$ will denote communication histories. The alphabet of $\mathcal{H}$ is the set of externally observable events i.e. events $o_1 \rightarrow o_2.m(\bar{p})$ and $o_1 \leftarrow o_2.m(\bar{p})$ where $o_1 \neq o_2$, m is an operation of some interface implemented by o_2, and $\bar{p}$ denotes the parameters values.

The projection of a finite history h onto a set of events, Set, denoted h/Set, is defined inductively by:

$$\begin{array}{lcl} \varepsilon/Set & = & \varepsilon \\ (h \vdash x)/Set & = & \left\{ \begin{array}{ll} (h/Set) \vdash x & \text{, if } x \in Set \\ (h/Set) & \text{, otherwise} \end{array} \right. \end{array}$$

where ε denotes the empty sequence and $\vdash$ denotes the right append.[1] We denote by $h \backslash Set$ the projection onto the complementary of the set Set. In the following, we will use the following abbreviations:

- the projection of the history onto a set of methods

  $$\mathcal{H}/\{m_1,\ldots,m_n\} \equiv \mathcal{H}/\{o_1 \leftrightarrow o_2.m(\bar{p}) \mid \leftrightarrow \in \{\rightarrow,\leftarrow\} \wedge m \in \{m_1,\ldots,m_n\}\}$$

  where $\bar{p}$ denotes a vector of parameters values,

[1]Right append plays an important role in OUN object specification, for instance consider an invariant $OK(\mathcal{H})$ where the predicate OK is defined inductively on the form

$$\begin{array}{lcl} OK(\varepsilon) & = & true \\ OK(h \vdash x) & = & OK(h) \wedge \textit{<x is acceptable in state h>} \end{array}$$

letting h serve as an abstract representation of the state. Such a specification has a form suitable for refinement into an imperative class implementation, as follows: In general only certain aspects of h are relevant in the last right hand side, these may be formulated by appropriate functions of h (also defined by induction). When refining into a class these functions determine the actual local variables needed.

In contrast the use of left-append would be suitable for describing an object as a finite state machine.

- the projection of the history onto an object o, *the local history of* o,

$$\mathcal{H}/o \equiv \mathcal{H}/\{o_1 \leftrightarrow o_2.m(\bar{p}) \mid \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge (o_1 = o \vee o_2 = o)\}$$

  where $\bar{p}$ denotes a vector of parameters values,

- the projection of the history onto an object o seen through an interface F, denoted by $\mathcal{H}/o : F$, which will be defined later.

We denote by P^y_x the formula obtained by substituting x for every y in the formula P.

When writing specifications, we will use the keyword **me** to denote the current object.

3 OUN-SL

The specification language allows us to specify interfaces and contracts. Classes are not present at this level, objects are seen trough interfaces (roles). Thus, objects having the same role (i.e. offering the same interface) can be used at the same places without considering classes. Object variables and parameters are typed by interfaces such that we always have static control of the allowed operation calls. We allow objects of an interface F to be used anywhere an object of any super-interface of F can be used. This substitutability requires a rigorous definition of interface inheritance.

An interface contains the syntactic definitions of operations and semantic requirement specifications. A requirement specification is a form of assumption-guarantee specification; it may contain an invariant, which is the guarantee, and an assumption about the behavior of the environment, and it may introduce a number of auxiliary functions needed for specification purposes. An interface may not depend on any class, in the sense that object parameters of an operation are typed by means of interfaces, and not by classes. We let all interfaces have a common super-interface, called **any**, which is the empty interface.

3.1 Interfaces

Using histories, two basic specification concepts are offered, the invariant for asserting properties which each object offering an interface must satisfy, and assumptions for stating minimal context requirements. An interface specification is of the general form:

```
interface $F$ [<type_parameters>](<parameters>)
  inherits $F_1, F_2, \ldots, F_m$
begin
  with x: $G$
    opr m$_1$(...)
    ...
    opr m$_i$(...)
  asm <formula_on_$\mathcal{H}$_and_any_external>
  inv <formula_on_$\mathcal{H}$>
end
```

where $F, F_1, F_2, \ldots, F_m$ and G are interfaces. The (optional) **inherits**-clause allows multiple inheritance. The defined interface F may have some parameters (between square brackets) which can be data-types or interfaces. The additional list of parameters (between ordinary brackets) is a list of values (typed by data types) and object parameters (typed by interfaces) which describes the minimal environment that an object of this interface must know at the point of creation. The part of the environment known by an object evolves during its life, due to information about the calling objects and object identifiers transmitted as parameters.

The **with** clause defines the (minimal) interface of an object calling any of the operations m_j ($j \in 1..i$), allowing an object of interface F to communicate with calling objects through this interface. Thus the **with** clause above states that only objects of interface G may talk to objects of interface F through the listed operations. Even though only one **with** clause is introduced in each interface, an interface may have several **with** clauses by way of inheritance. The interfaces that appear in the **with** clause of F and its super-interfaces are said to be *associated* with F.

We write m **in** F to denote that an operation m is defined either in F or in any super-interface of F. An operation m such that m **in** F^G, is an operation defined either in F or in any super-interface of F in a **with** G' clause where G' is G or any super-interface of G.

The projection of the history onto an object o seen through an interface F is defined accordingly, i.e. $\mathcal{H}/o : F$ is the projection of $\mathcal{H}$ onto the set of calls of operations of F received by o and the set of calls of operations of the associated interfaces of F initiated by o. Let $G_1, G_2, \ldots, G_n$ denote these associated interfaces. Then, $\mathcal{H}/o : F$ is defined by:

$$\begin{aligned}\mathcal{H}/o : F \equiv \mathcal{H}/\{o_1 \leftrightarrow o_2.m(\ldots) \mid\ & \leftrightarrow \in \{\rightarrow, \leftarrow\} \text{ and} \\ & ((o_2 = o \wedge m \textbf{ in } F) \\ & \vee (o_1 = o \wedge \exists i \bullet m \textbf{ in } {G_i}^F))\}.\end{aligned}$$

The assumption describes the behavior which external objects have to respect when communicating with an object of interface F, in order for the invariant of the interface to be guaranteed.

Remark. Some auxiliary functions may be needed for specification purposes. They may be defined after the invariant on the form:

func <*name*> == ...

Similarly, data types may be defined as well.

3.2 Basic semantics

The control of the communications of an object is distributed between its assumption – for inputs – and its invariant – for outputs. We denotes respectively by $\textbf{in}(\mathcal{H})$ and $\textbf{out}(\mathcal{H})$ the longest left prefix of a trace $\mathcal{H}$ ending by an input event and by an output event.

For an interface F defined without using inheritance, we have:

- the assumption[2] **asm** $A(\mathcal{H})$, where A is a formula, states that for any object o of interface F, and for any external object x, the formula

  $$A^{in}(\mathcal{H}) \equiv (\forall x \neq o \bullet [A_o^{\mathbf{me}}](\mathbf{in}(\mathcal{H}/o : F/x)))$$

  holds.

- the invariant[3] **inv** $I(\mathcal{H})$, where I is a formula, states that for any object o of this interface, the formula

  $$I^{out}(\mathcal{H}) \equiv [I_o^{\mathbf{me}}](\mathbf{out}(\mathcal{H}/o : F)),$$

  holds. Thus, the invariant restricts the communication history local to the current object.

- The trace set of an object o of interface F is defined by the assumption and the invariant of F:

  $$\mathcal{T}_{o:F} \equiv \{\mathcal{H}/o : F \mid A^{in}(\mathcal{H}) \Rightarrow (A^{out}(\mathcal{H}) \wedge I^{out}(\mathcal{H}))\},$$

  where

  $$A^{out}(\mathcal{H}) \equiv (\forall x \neq o \bullet [A_o^{\mathbf{me}}](\mathbf{out}(\mathcal{H}/o : F/x))).$$

  The control of assumption for traces ending by an output event ensures that an object will not break its own assumption.

3.3 Refinement

For objects with the same alphabet, we define refinement simply by subset on trace sets which is a standard definition of refinement, for example as in the notion of behavioral refinement in FOCUS [3, 4] or in CSP [7]. For objects with different alphabets, one must consider a way of relating the alphabets, for instance by an abstraction function.

In the special case of interface refinement, a (super) alphabet may be extended by additional operations in a sub-interface. We suggest the use of projection when relating the trace of a sub-object to that of a super-interface following [6]. An interface F' is said to refine an interface F, if the alphabet of each object o of interface F is included in the alphabet of o seen through interface F', and if the trace set of each object o of interface F' is included in the one of o seen through interface F, that is to say:

$$(\forall H \in \mathcal{T}_{o:F'} \bullet H/o : F \in \mathcal{T}_{o:F}).$$

In the literature most refinement notions related to sub-classing [9, 10] ensure that an object of a subclass will not give surprises when used as an object of a superclass ("plug in" compatibility). This concept of refinement linked to "plug in" is not peculiar to object-oriented technology, it can be found in several well-known specification languages: for example, the operation refinement in Z [13] and the trace refinement of action systems in [2] follow this scheme. Our notion of refinement of interfaces does not ensure this directly, only after the relevant projection. On the other hand if an object offering an interface gets inputs

[2]When not present, the assumption is supposed to be **true**.
[3]When not present, the invariant is supposed to be **true**.

beyond what is syntactically defined in a super-interface, we still require that the object behaves as a super-interface object after projection; a requirement which is usually not included in "plug in" compatibility. Our notion is suitable for multiple inheritance and for describing abstract role-behavior. Even if it does not provide "plug in" compatibility, our language is strongly typed, and no call can be made which results in an operation not understood. Thus "plugging in" a sub-interface object, in our setting, may not create any (new) system failures and may not brake any invariants inherited from super-interfaces. Our form of "plug in" compatibility is strong enough to provide semantic control, and weak enough to allow the kind of redefinitions often used with virtual binding.

In the interface refinement, there is, once again, a need of an abstraction function to relate interfaces (and also systems) with different alphabets. However, the exact definition of these mechanisms is outside the scope of this paper; and the complete semantics, definition of refinement and parallel composition are treated in another paper. Our definition suffices to deal with interface inheritance and interface implementation. Moreover, we only consider safety aspects, deadlock control is not discussed (and would require a more complex semantic treatment).

3.4 Multiple Inheritance

Multiple inheritance of interfaces may be used when defining a new interface. The new interface may add operations as well as an invariant and an assumption.

Objects of a given interface can be used as objects of any of its super-interfaces. So, it is essential that objects of an interface F may play the role of an object of any super-interface of F without any additional control. Thus, the interface F may have its own assumption and invariant but its trace set is defined with respect to trace sets of super-interfaces.

The trace set of an object of interface F that inherits F_1, ..., F_n and whose assumption and invariant are respectively A and I, is defined by:

$$\begin{aligned}\mathcal{T}_{o:F} \equiv \{\mathcal{H}/o : F \mid\ & \mathcal{H}/o : F_1 \in \mathcal{T}_{o:F_1} \wedge \ldots \wedge \mathcal{H}/o : F_n \in \mathcal{T}_{o:F_n} \wedge \\ & A^{in}(\mathcal{H}) \Rightarrow (A^{out}(\mathcal{H}) \wedge I^{out}(\mathcal{H}))\}.\end{aligned}$$

Then, it is obvious that interface inheritance is a particular case of interface refinement: F refines each of its super-interfaces.

We have to deal with possible name conflicts in multiple inheritance. When an interface F inherits several interfaces $F_1, \ldots, F_n$, the same operation name m can appear in several of them. If the different versions of m have different signatures, the problem is solved by overloading (**with** clauses are considered to be part of the signature). In the other case, the operations are considered the same (if this leads to an inconsistence of specification requirements then the problem has to be solved by explicit renaming). An interface also inherits the parameters of the super-interfaces. When an interface inherits two formal parameters with the same name, they must have the same data-type (or a subtype of it) or be both data-types or be both interfaces.

3.5 Contracts

Contracts gives a kind of "glass-box" specifications used to restrict the interaction between two or more objects. A contract is expressed in the form:

```
contract C
begin
    with $o_1 : F_1, \ldots, o_n : F_n$
    inv $<$ *formula_on_$\mathcal{H}$_and_$o_1 \ldots o_n$* $>$
end
```

expressing that for all (distinct) objects $o_1, \ldots, o_n$ of the given interfaces, F_1, $\ldots$, F_n, their "common" communication history must satisfy the given formula P, i.e. the formula

$$P(\mathcal{H}/\{o_i \leftrightarrow o_j.m(\ldots) \mid \quad i \neq j \wedge \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge m \textbf{ in } F_j^{F_i}\}).$$

must hold.

Contracts may be used at the beginning of the specification to capture the global properties of the system. Contracts have to be respected by objects of the involved interfaces as well as specification requirements of the interfaces. Contracts may be split into interface specification requirements during refinement.

4 OUN-SL examples

In examples it is often convenient to restrict (a projection of) the local history to be a prefix of a set of traces, given by a regular expression. Regular expressions are written in standard form, using blank for juxtaposition, a star for repetition and a bar for alternatives. The keyword **prs** denotes "prefix of regular expression". In regular expressions $\leftrightarrow .m$ denotes the juxtaposition of the two events $\rightarrow .m$ and $\leftarrow .m$.

4.1 Virtual meeting

We consider a very simple virtual meeting. Participants have to register in order to be admitted to the meeting, then they can listen to other participants. Only one participant is allowed to speak at a given instant. Participants can request to speak and then the manager makes them speak one at the time. We are interested in the pattern (enter (listen | req_speak listen* speak)* leave)*) where participants use "enter" to register, "req_speak" to request for speaking and "leave" to quit the meeting, a meeting manager uses "speak" to get the speech of a participant, and where everybody can use "listen" the say something to someone else.

```
interface Meeting-Manager
begin
    with x: Participant
        opr enter()
        opr req_speak()
        opr leave()
    asm ($\mathcal{H}\backslash$ {listen}) prs (↔.enter (↔.req_speak   ↔.speak)* ↔.leave)*
    inv  ($\mathcal{H}/$ {↔.speak, →.listen} prs
                [me↔x.speak(t) →.listen(t,x)* | t : string, x : Participant]*)
        ∧ (forall x : Participant : $\mathcal{H}$/x prs (↔.enter (↔.listen | ↔.req_speak
                ↔.listen*   ↔.speak)* ↔.leave)*)
end
```

With the assumption, we suppose that external objects will initiate speak requests between a registration and leave. Moreover, as all the initiation events are followed by the corresponding termination event, we require that all the calls will be synchronous and that a participant waits after each speak request, until he allowed to speak. The invariant ensures that a meeting manager will only call listen and speak operations of participants which have asked to participate to the meeting and that it will only call operation speak after a speak request for the same participant.

```
interface Listener
begin
    with x: any
        opr listen(t : string, from : any)
end

interface Participant
    inherits Listener
begin
    with x: Meeting-Manager
        opr speak(out t: string)
    asm  $\mathcal{H}$\ {listen} prs (↔.enter (↔.req_speak   ↔.speak)* ↔.leave)*
    inv  forall x : Meeting-Manager : ($\mathcal{H}$\ {listen}/x) prs
                                (↔.enter (↔.req_speak   ↔.speak)* ↔.leave)*
end
```

The interface “Listener” offers one operation which can be used by all the objects of the system. There is no invariant and no assumption in this interface, so all the traces are accepted.

In the interface “Participant” we do not assume anything about the behavior of meeting managers, but the invariant of the interface ensures that participants will initiate speak request between a registration and and leave, and will make synchronous call to operations of “Meeting-Manager”, and will wait for a call to speak after each speak request.

4.2 Readers and Writers

We will now consider the well-known example of objects controlling read and write access to some data to illustrate the concepts of the notation.

The first interface we propose controls the read access. Concurrent read operations are allowed, so there is no restriction concerning the use of this interface.

```
interface R [T: Data-Type]
begin
    with x: any
        opr read(out d : T)
end
```

The second interface allows us to control the write access. Write operations are to be exclusive (this is ensured by the invariant) if each calling object encloses its write operations by open and close operations (as required by the assumption).

```
interface W [T: Data-Type]
begin
    with x: any
        opr open_write()
        opr write(d : T)
        opr close_write()
    asm  $\mathcal{H}$ prs ($\leftrightarrow$.open_write  $\leftrightarrow$.write$^*$  $\leftrightarrow$.close_write)$^*$
    inv   $\mathcal{H}/\leftarrow$ prs ($\leftarrow$.open_write  $\leftarrow$.write$^*$  $\leftarrow$.close_write)$^*$
end
```

Now, we can easily define an interface for read/write access control using the two previous ones. Read and write access have to be exclusive too, so read access will have to be enclose by open and close operations as write access. Nevertheless, concurrent read accesses are still allowed.

```
interface RW [T: Data-Type]
    inherits W
begin
    with x: any
        opr open_read()
        opr read(out d : T)
        opr close_read()
    asm  $\mathcal{H}$ prs ($\leftrightarrow$.open_write  $\leftrightarrow$.write$^*$  $\leftrightarrow$.close_write
                 | $\leftrightarrow$.open_read  $\leftrightarrow$.read$^*$  $\leftrightarrow$.close_read)$^*$
    inv      #($\mathcal{H}/$ $\leftarrow$.open_read) - #($\mathcal{H}/$ $\leftarrow$.close_read) = 0
          $\vee$ #($\mathcal{H}/$ $\leftarrow$.open_write) - #($\mathcal{H}/$ $\leftarrow$.close_write) = 0
end
```

The invariant of interface “RW” ensures that read and write access are exclusive and, since the assumption of “RW” implies the assumption of “W”, the exclusivity of write access is controlled by the inherited invariant.

We can also remark that this example illustrates how to use synchronous communications in OUN: as we require each operation initiation to be followed by the corresponding termination, there is no observable activity involving the two objects on these interfaces between these two events. The caller has to wait until it receives the termination event.

5 OUN-DL

OUN-DL is the design language of OUN. This level is constituting a first step of refinement where a class concept is introduced. Most of the concepts we introduce to deal with openness and mobility will affect this level by way of constructs like class extension. Object parameters and variables are typed by interfaces as before. Thus, we keep static control of authorized operation calls even though we allow class inheritance without any restriction. Notice that, as we use interfaces as types, and as objects can only communicate through interfaces, the most important semantic check concerns the implementation claims of classes.

5.1 Classes

A class contains the definition of some typed variables (the attributes), the implementation of operations and, like in the case of interfaces, an invariant and some assumptions. A class definition has the syntax:

```
class $C$ [<type_parameters>](<parameters>)
    implements $F_1, F_2, \ldots, F_m$
    inherits $C_1, C_2, \ldots, C_n$
begin
    var $v_1 : V_1$
         $v_2 : V_2$
     ...
        init
           <imperative-code>
    with $x_1$: $G_1$
        opr $\mathrm{m}_1(\ldots)$ == <imperative-code>
        ...
        opr $\mathrm{m}_i(\ldots)$ == <imperative-code>
        asm <formula_on_$\mathcal{H}$_and_$x_1$>
    ...
    with $x_k$: $G_k$
        opr $\mathrm{m}_j(\ldots)$ == <imperative-code>
        ...
        opr $\mathrm{m}_l(\ldots)$ == <imperative-code>
        asm <formula_on_$\mathcal{H}$_and_$x_k$>
    inv <formula_on_$\mathcal{H}$>
end
```

where $F_1, F_2, \ldots, F_m$ and $G_1, G_2, \ldots, G_k$ are interfaces and $C_1, C_2, \ldots, C_n$ are classes. A class may have parameters (between square brackets) which are data-types or interfaces. The additional list of parameters (between ordinary brackets) is the list of parameters (typed by data-types or by interfaces) that must by given at the point of creation of an object of this class. A class may implement several interfaces ($F_1, F_2, \ldots, F_m$) and inherit several classes ($C_1, C_2, \ldots, C_n$). A class may also have attributes which are typed by data-types, interfaces. The **init** part contains some initialization statements executed at the creation of a object, they allow for example to give initial values to attributes and to make some initial calls.

As for interfaces, a **with** clause states that only objects of the interface mentioned in the clause may talk to objects of class C through the operations listed in this clause. The difference is that a class may have several **with** clauses.The description of the behaviors of operations, the <*imperative-code*>, will be given as guarded commands on the form <*guard* $\rightarrow$ *statements*>.

As classes do not inherit assumptions and invariants, there may be an assumption in each **with** clause, constraining the behaviors of objects of the interface mentioned in this **with** clause and an invariant for the class.

The projection of the history onto an object o of class C, denoted by $\mathcal{H}/o : C$ is the projection of $\mathcal{H}$ onto the set of calls of operations of o and the set of calls, initiated by o, of operations of the interfaces that appear in the **with** clauses of C and in the **with** clauses of its superclasses. Let $G_1, G_2, \ldots, G_n$ denote these interfaces. Then $\mathcal{H}/o : C$ is defined by:

$$\begin{array}{ll} \mathcal{H}/o : C \equiv \mathcal{H}/\{o_1 \leftrightarrow o_2.m \mid & \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge \\ & ((o_2 = o \wedge m \textbf{ in } C) \\ & \vee (o_1 = o \wedge m \textbf{ in } G_1 \cup \ldots \cup G_n))\}. \end{array}$$

- An invariant[4] **inv** $I(\mathcal{H})$ of a class C, where I is a formula, states that for any object o of class C the formula

  $$\bar{I}^{out}(\mathcal{H}) \equiv [I_o^{\mathbf{me}}](\mathbf{out}(\mathcal{H}/o : C, \bar{h}, \bar{v})),$$

  holds, where $\bar{v}$ refers to local variables and where $\bar{h}$ refers to some class histories that we will discuss later in subsection 5.5.

- An assumption[5] **asm** $A(\mathcal{H})$ of a class C given for an interface G, where A is a formula, states that for any object o of class C, and for any external object x of interface G,

  $$\bar{A}^{in}(\mathcal{H}) \equiv (\forall x \neq o \textbf{ of } G \bullet [A_o^{\mathbf{me}}](\mathbf{in}(\mathcal{H}/o : C/x : G))),$$

  where x **of** G_i means that x implements G_i or one of its sub-interfaces, holds.

  Note that the assumption of a class may not refer to local variables.

- The trace set of an object o of class C is defined by the assumptions and the invariant of C. Let $A_1, \ldots, A_k$ denote the assumptions of C given in the **with** clauses associated with $G_1, \ldots, G_k$, respectively. The trace set of o is:

  $$\begin{array}{ll} \mathcal{T}_{o:C} \equiv \{\mathcal{H}/o : C \mid & [\forall 1 \leq i \leq k \bullet \bar{A_i}^{in}(\mathcal{H}))] \\ & \Rightarrow [(\forall 1 \leq i \leq k \bullet \bar{A_i}^{out}(\mathcal{H}))) \wedge \bar{I}^{out}(\mathcal{H}))]\}, \end{array}$$

  where

  $$\bar{A}^{out}(\mathcal{H}) \equiv (\forall x \neq o \textbf{ of } G \bullet [A_o^{\mathbf{me}}](\mathbf{out}(\mathcal{H}/o : C/x : G))).$$

Remark. Some notations are referring to classes, $\mathcal{H}/o : C$ for example. These notations are used for abbreviated writing. In a compilation unit, their meanings are understood "at this time" (i.e. the time of compilation without considering possible future extensions of the class). The verification conditions are generated based on this meaning.

[4]When not present, the invariant is supposed to be **true**.
[5]When not present, the assumption is supposed to be **true**.

5.2 Classes Implementing Interfaces

A class may implement several interfaces. A class implementing an interface must have operations with exactly the same signatures (or larger in-types and smaller out-types) as in the interface (explicit renaming may be considered), thus the object parameters of these operations must be typed by interfaces in the class as well.

An actual parameter matches a formal object parameter if the type of the actual parameter is the same as that of the formal parameter, or a sub-interface of it. This ensures that an actual class of an object parameter may be any class implementing the interface of the corresponding formal parameter, and that all operation calls prescribed on formal parameters can be realized.

A class C implementing an interface F must satisfy the specification requirements of the interface, this can be expressed in terms of trace sets:

$$\forall o : C \bullet (\forall H \in \mathcal{T}_{o:C} \bullet H/o : F \in \mathcal{T}_{o:F}).$$

Thus, implementation of interfaces can be seen as ordinary refinement.

5.3 Class Inheritance

Multiple inheritance of classes may be used to define new classes.

A subclass may add attributes, add operations, and redefine operations. A subclass inherits all attributes, and inherits those operations which are not redefined. A subclass need not respect the requirement specification of the superclass (a similar point of view is discussed and motivated in [12]). Thus, operations may be redefined in a subclass without any syntactic or semantic restrictions! A subclass does not inherit the requirement specifications of the superclass, and it does not inherit their “implements” claims. Thus the list of interfaces that a subclass implements has to be restated (and proved) again for the subclass.

Notice that even when a subclass has no redefinition with respect to a superclass, the requirement specification of the latter need not be satisfied by the subclass, since added operations may make changes to the inherited attributes violating the invariant of the superclass.

Subclasses may redefine operations. In order to allow the most flexible use of redefinition, the binding mechanism follows the following scheme, similar to that of Java: a call of an operation m which is redefined refers to the redefined version when the (static) types of the parameters match (*i.e. is the same or a smaller in-type*) those of the redefined operation. Otherwise, the call refers to the operation as defined in the superclass.

We have to deal with the possible name conflicts in multiple inheritance. When a class C inherits several classes, the same operation name m can appear in several of them. If the different versions of m have different signatures, the problem is solved by overloading. In the other case, operation conflicts are solved by disjoint union, i.e. one must use the superclass names to distinguish the different versions. Inherited local variables (attributes) with the same name must have the same type, and are considered to be the same variable. A class inherits parameters in the same way.

5.4 Dynamic Extension

We also consider the possible dynamic evolution of a class. Using class extension, some operations and some interfaces can be added dynamically to a class while maintaining the original name. A class extension has the syntax:

```
class extension $C$
    implements $F_1, F_2, \ldots, F_m$
begin
    with $x_1$: $G_1$
        opr $\mathrm{m}_1(\ldots)$ == <imperative-code>
        ...
        opr $\mathrm{m}_i(\ldots)$ == <imperative-code>
        asm <formula_on_$\mathcal{H}$_and_$x_1$>
    ...
    with $x_k$: $G_k$
        opr $\mathrm{m}_j(\ldots)$ == <imperative-code>
        ...
        opr $\mathrm{m}_l(\ldots)$ == <imperative-code>
        asm <formula_on_$\mathcal{H}$_and_$x_k$>
    inv <formula_on_$\mathcal{H}$>
end
```

The addition of new operations create a need to update the invariant and the assumptions. From another point of view, relations of existing objects of the class with the environment must not be damaged by the extension, so the specification requirement of the extended class have to respect the specification requirements of the old class.

Let us consider a class C which we denote by C_{old} in the formulas and an extension of C. Let I denote the new invariant given in the extended class, and $A_1, \ldots, A_n$ denote the assumptions given in the **with** clauses of the extended class, associated with interfaces $G_1, \ldots, G_n$, respectively. Then, the trace set of a object o of class C, after extension, is defined by[6]:

$$\begin{aligned}\mathcal{T}_{o:C} \equiv \{\mathcal{H}/o : C \mid \quad & \mathcal{H}/o : C_{old} \in \mathcal{T}_{o:C_{old}} \wedge \\ & [\forall 1 \leq i \leq k \bullet \bar{A}_i^{\,in}(\mathcal{H}))] \\ & \Rightarrow [(\forall 1 \leq i \leq k \bullet \bar{A}_i^{\,out}(\mathcal{H}))) \wedge \bar{I}^{out}(\mathcal{H}))]\},\end{aligned}$$

The extension of a class leads to immediate extension of all of its subclasses. This implicit extension of subclasses may create name conflicts, the name of a new operation may already be used in a subclass. If the two versions of the operations have different profiles, the problem is solved by overloading, otherwise, the operation of the subclass is considered to be a redefinition of the operation of the superclass.

[6]Note that the alphabet of each interface F implemented by C_{old} is included in the alphabet of C_{old} which is included in the alphabet of C. So, according to the definition of $\mathcal{T}_{o:C}$, it is obvious that each interface implemented by C_{old} is still implemented by C.

5.5 Dynamic generation of objects

The dynamic generation of an object of class C is realized using the construct:

$$\textbf{new}\ C(\ldots).$$

We can for example consider that an object x has an attribute $o : F$ and that the class C implements the interface F. Then we may have a statement $o := \mathit{new}\ C(\ldots)$ executed by x. At the reasoning level (i.e. in the history) this creation is reflected by the event

$$x \to y.\mathit{new}(C, \ldots)$$

where y is the created object. This event is not considered to be observable neither by x nor by y as it is not a call to an operation of the class C. However, creation events are part of the life of a class. At run-time, each class is supposed to be represented by a "class-object" which is not visible by users. Object creations are recorded in the history of this "class-object", the class history of a class C is denoted by $C.\mathcal{H}$. The class histories may be used in the invariants of classes.

For reasoning purposes, all generated objects have a unique identity linked to the state of history at their instant of creation.

5.6 Interface Testing

We say that an object o supports an interface F if the class of o implements F. In our formalism, an object may support a number of interfaces. As classes may be dynamically extended, implementing more and more interfaces, and as old classes may implement new interfaces, we do not have complete static control over which interfaces are supported by which objects. There will be a need to test whether an object o supports an interface F (i.e. to check whether F appears in the implementation claims of the class of o); for this we have introduced the notation $o : F?$ as a boolean expression in the programming language. In addition we may wish to ask for an object offering a given interface; this may be done in a specialized if-construct, say

$$\textbf{if any}\ x : F?\ \textbf{then}\ < \mathit{may\ use\ x\ here} >\ \textbf{else}\ < \mathit{no\ such\ x\ exists} >\ \textbf{fi}$$

which may occur in the implementation of an operation. When desirable, also the location of x may be retrieved.

5.7 Distribution

In a distributed system we talk about a number of *locations*, identified by unique names. A program unit belongs to a particular location. By naming a location, a program unit on one location may access interfaces, and objects located on another location. We do not allow classes to be shared between locations (since they may be changed dynamically), however, one may copy them from one location to another. When an interface on one location has the same name as one on another location, we use the location name to uniquely identify the two interfaces. No location name is needed when talking about entities on the current location.

5.8 Partial compilations of the system

As already said, one way we choose to achieve openness is to allow incremental addition of code without stopping the whole system. This means that some new parts will be compiled separately and added to the running system. The constructs we have proposed in our notation aim at allowing modifications of the system without re-proving what has already been stated. Proofs generated by addition of code will only be local and will only involve new elements of the system: anything which has been stated must never have to be checked again. Partial compilations can occur in several cases:

1. new interfaces or sub-interfaces are added.

2. new classes or sub-classes and possibly new objects are added.

3. some classes are extended.

In cases 1 and 2, we just add some new elements to the running system. As long as we only consider properties of new elements, the new part is quite "independent". The interesting point of such an addition is the interaction between old and new parts of the system. Old objects can only talk to new objects through old interfaces. Because we have allowed interface inheritance in such a way that an object of an interface can always play the role of an object of any super-interface, the additions cannot disturb the running system.

In the third case, classes may be extended. An extension not only adds new elements to the system but *modify* the capabilities of existing objects. As soon as the new methods (or the whole class depending on the system) are compiled, and the class extended, the old version of the class is not visible anymore. This means that all the objects of this class, old ones as well as new ones, correspond to the new definition of the class. The significant point is that implementation relationships cannot be suppressed. Thus, old objects used to talk with an object of class C through an interface I will not be affected by the extension of the class C and will not observe the extension unless they also use new interfaces.

At the semantic level, calls to new methods of an extended object can be observed in the local histories of objects as well as objects of new classes or new interfaces. However, as objects communicate through interfaces, the only thing that has to be checked is that the implementation claims are correct: the new proof obligations only concern the new classes or new interfaces.

6 OUN-DL examples

6.1 Virtual meeting

In this first example we present a class implementing the interface Meeting-Manager seen in subsection 4.1. The underlying language provides sets as types, with operators to add an object to the set (+), to remove an object from the set (-), and to test whether an object belongs to the set (**in**). Moreover, the design language offers the possibility to use broadcast by sending a message to a set of objects.

```
class Manager
    implements Meeting-Manager
begin
    var s : set[Participant]
        t : TEXT
begin
    with x: Participant
        opr enter() == if x in s then ... else s := s +x endif
        opr open_speak() == x.speak(t); s.listen(t)
        opr leave() == s := s -x
    asm  ($\mathcal{H}$\ {listen}) prs (↔.enter (↔.req_speak   ↔.speak)* ↔.leave)*
    inv   ($\mathcal{H}$/ {↔.speak, →.listen} prs
              [me↔x.speak(t) →.listen(t,x)* | t : string, x : Participant]*)
        ∧ (forall x : Participant : $\mathcal{H}$/x prs (↔.enter (↔.listen |
            ↔.req_speak  ↔.listen*   ↔.speak)* ↔.leave)*)
end
```

Since the assumption and the invariant of the class Manager are the same as those of Meeting-Manager, it is obvious that Manager implements Meeting-Manager.

```
class CParticipant
    implements Participant
begin
    with x: any
        opr listen(t : string, from : any) == ...
    with x: Meeting-Manager
        opr speak(out t: TEXT) == ...
        asm $\mathcal{H}$\ {listen} prs (↔.enter (↔.req_speak   ↔.speak)* ↔.leave)*
    inv   forall x : Meeting-Manager : ($\mathcal{H}$\ {listen}/x) prs
                                  (↔.enter (↔.req_speak   ↔.speak)* ↔.leave)*
end
```

Participants may talk privately with other participants of the meeting using the operation "listen" since this operation is not dedicated to objects of interface "Meeting-Manager". As the registration of participants is made by the meeting manager, a participant only knows another participant if the latter has already spoken *i.e.* if its identity has been transmitted as parameter of the operation "listen".

6.2 Readers and Writers

We will now present some classes implementing interfaces of the subsection 4.2. The first class implements the interface R and controls the read access to a shared data.

```
class Read-Control [T: Data-Type]
    implements R [T]
begin
    var shared_data : T
    with x: any
```

```
        **opr** read(**out** d : T) == d := shared_data
**end**
```

The second class implements the interface W and controls the write access to a shared data. The invariant and the assumption of the class are exactly the same as the ones of the interface.

```
class Write-Control [T: Data-Type]
    implements W [T]
begin
    var shared_data : T,
        flag : bool := true
    with x: any
        opr open_write() == flag $\rightarrow$ flag := false
        opr write(d : T) == shared_data := d
        opr close_write() == flag := true
        asm  $\mathcal{H}$ prs ($\leftrightarrow$.open_write  $\leftrightarrow$.write$^*$  $\leftrightarrow$.close_write)$^*$
    inv  $\mathcal{H}/\leftarrow$ prs ($\leftarrow$.open_write  $\leftarrow$.write$^*$  $\leftarrow$.close_write)$^*$
end
```

The third class implements read and write access control. We can reuse the code of the read operation for example, but not the invariant and assumption of the super-classes.

```
class Read-Write-control [T: Data-Type]
    implements W [T], RW [T]
    inherits Read-Control, Write-Control
begin
    var nb_readers : int := 0
    with x: any
        opr open_read() == (flag $\vee$ nb_readers $\neq$ 0) $\rightarrow$ flag := false;
                           nb_readers := nb_readers +1
        opr close_read() == nb_readers := nb_readers -1;
                           if nb_readers = 0 then flag := true endif
    asm $\mathcal{H}$ prs ($\leftrightarrow$.open_write  $\leftrightarrow$.write$^*$  $\leftrightarrow$.close_write
                | $\leftrightarrow$.open_read  $\leftrightarrow$.read$^*$  $\leftrightarrow$.close_read)$^*$
    inv     (#($\mathcal{H}/$ $\leftarrow$.open_read) - #($\mathcal{H}/$ $\leftarrow$.close_read) = 0
            $\vee$   #($\mathcal{H}/$ $\leftarrow$.open_write) - #($\mathcal{H}/$ $\leftarrow$.close_write) = 0)
        $\wedge$   (#($\mathcal{H}/$ $\leftarrow$.open_write) - #($\mathcal{H}/$ $\leftarrow$.close_write) = 0
            $\vee$ #($\mathcal{H}/$ $\leftarrow$.open_write) - #($\mathcal{H}/$ $\leftarrow$.close_write) = 1)
end
```

This class could also implement the interface "R" which has no assumption and no invariant.

7 A larger example

7.1 A bank account

A bank offers customers the possibility to have bank accounts. Account owners must have a personal number and are allowed to put money into accounts, take money out, and consult the balance, whereas only bank clerks are allowed to create accounts.

We have three interfaces, "Account_owner", "Account" and "Bank_clerk" and three classes implementing these interfaces. We will point out some interesting aspects.

```
interface Account_owner
begin
    with x: any
        opr personal_number(out pn : int)
end
```

```
interface Account
begin
    with x: Account_owner
        opr in (amount : int)
        opr out(amount : int; out ok : bool)
        opr balance(out b : int, ok : bool))
    inv (h ⊢ x←me.out(a,true) head $\mathcal{H}$) ⇒ (bal(h/{←.in,←.out}) ≥ a)
        func  bal(s:sequence):int
        def   bal(empty) == 0,
              bal(h ⊢ ←.in(a)) == bal(h) + a,
              bal(h ⊢ ←.out(a, ok)) == if ok then bal(h) - a
                                                  else bal(h) endif
end
```

The invariant of the interface "Account" ensures that the balance of the account will always be positive. We see here an example of an auxiliary function "bal" used for specification purpose.

```
interface Bank_clerk
begin
    with x: Account_owner
        opr open_account(out a : account)
end
```

```
class CAccount_owner (n : Natural)
    implements Account_owner
begin
    var number : int := n
    with x: any
        opr personal_number(out pn : int) == pn := number
end
```

```
class CBank_clerk
    implements Bank_clerk
begin
```

```
    with x: Account_owner
        opr open_account(out a : Account) == a:= new CAccount (x)
end
```

In the class "CBank_clerk", we have an example of object creation. A new object of class "CAccount" is created using the **new** construct with parameter values. Here, "x" denotes the caller of the operation "open_account". Notice that the type of the parameter "a" is "Account" since object parameters are typed by interfaces.

```
class CAccount (p : Account_owner)
    implements Account
begin
    var  bal : int := 0,
         owner : Account_owner := p
    with x: Account_owner
        opr in (amount : int) == bal := bal + amount
        opr out(amount : int; out ok : bool) ==
             if owner = x ∧ bal ≥ amount
             then bal := bal - amount; ok := true
             else ok := false endif
        opr balance(out b : int, ok : bool) ==
             if owner = x then b := bal; ok := true
             else ok := false fi
    inv  (h ⊢ x←me.out(a,true) head ℋ) ⇒
                    (Account.bal(h/{←.in,←.out}) ≥ a)
end
```

In this last class, the parameter "p" is the formal parameter corresponding to the parameter in the **new** construct. We can also notice that we use dot notation, as in "Account.bal", to access auxiliary functions defined in other classes or interfaces.

7.2 Savings accounts

Now imagine that a bank is growing and wants to offer its customers the possibility to have a savings-account. We add two new interfaces and a class to the system:

```
interface Savings_account
    inherits Account
begin
    with x: Account_owner
        opr rate(out r : int)
end


interface Savings_account_Management
begin
```

```
    **with** x: Bank
        **opr** interest()
        **opr** change_rate(new_rate : **int**)
**end**


**class** CSavings_account
    **implements** Savings_account,Savings_account_Management, CAccount
    **inherits** CAccount
**begin**
    **var** account_rate : **int** := 5
    **with** x: Account_owner
        **opr** rate(**out** r : **int**) == r := account_rate
    **with** x: Bank
        **opr** interest() == bal := bal + bal * account_rate / 100
        **opr** change_rate(new_rate : **int**) == account_rate := new_rate
    **inv** CAccount.inv
**end**
```

The class "CSavings_account" inherits from "CAccount" attributes and operations but not implementation claims and specification requirements. As we just want to make a new class by adding some new attributes and operations without changing the meaning of old operations, we also want to keep the same invariant. This can be done using the dot notation "CAccount.inv"; be careful, this notation is just used as a shorthand and must be replaced by the exact definition of the invariant of "CAccount" at compilation time of "CSavings_account". The invariant of the class "CSavings_account" will not be affected by possible extension of "CAccount".

7.3 Transfer

Now we have a running system with bank accounts and savings accounts. Sometimes, people having both kinds of accounts want to transfer money from one to the other. So, it would be desirable to add a transfer operation to the accounts. We will add a new interface to the system and we will extend the superclass "CAccount":

```
**interface** Transfer
    **inherits** Account
**begin**
    **with** x: Account_owner
        **opr** trans(amount : **int**; to : Account; **out** ok : **bool**)
    **inv**  (h ⊢ x←**me**.trans(a,to,**true**) **head** $\mathcal{H}$) ⇒
                                        (bal2(h/{←.in,←.out,←.trans }) ≥ a)
        **func**  bal2(s: **sequence**):**int**
        **def**   bal2(**empty**) == 0,
              bal2(h ⊢ ←.in(a)) == bal2(h) + a,
              bal2(h ⊢ ←.out(a, ok)) == **if** ok **then** bal2(h) - a
                                                     **else** bal2(h) **endif**,
              bal2(h ⊢ ←.trans(a, ok)) == **if** ok **then** bal2(h) - a
```

```
                                    else bal2(h) endif
end

class extension CAccount
    implements Tranfer
begin
    with x: Account_owner
        opr trans(amount : int; to : Account; out ok : bool) ==
            if owner = x ∧ bal ≥ amount
            then bal := bal - amount; to.in(amount); ok := true
            else ok := false endif
    inv  (( h ⊢ x←me.trans(a,to,true)
         ∨ h ⊢ x←me.out(a,true)) head 𝓗) ⇒
             (Transfer.bal2(h/{←.in,←.out,←.trans }) ≥ a)
end
```

We extend the class "CAccount" by addition of an operation. The extended class implements the new interface "Transfer" and its total invariant is the conjunction of the one given in the extension and the old one.

The extension of "CAccount" is transmitted to its subclasses by way of the inheritance mechanisms: operations are inherited but not implementation claims and specification requirements so we have to add them. Now, if we want to use the transfer operation for a savings account too, we have to prove that it implements the "Transfer" interface. For instance, we can extend the class "CSavings_account".

```
class extention CSavings_account
    implements Tranfer
begin
    inv  CAccount.inv
end
```

As already said, "CAccount.inv" refers to the invariant of "CAccount" at compilation time of the unit we are writing so, it now refers to the invariant of the extended class.

7.4 Account with credit_line

As a final example, we now consider that the bank also wants to offer accounts with a credit line. Such an account works like a savings account except that the balance may be negative, and then the interest is subtracted.

We will here redefine operations, giving them a different meaning than before.

```
interface Account_with_credit
    inherits Savings_account, transfer
begin
end
```

```
interface Credit_line_management
    inherits Savings_account_management
begin
end


class CAccount_with_credit (c : int, r : int)
    implements Account_with_credit, Credit_line_management
    inherits CSavings_account
begin
    var credit : Natural
    init
        credit := c; account_rate := r
    with x: Account_owner
        opr trans(amount : int; to : Account; out ok : bool) ==
             if owner = x ∧ bal ≥ amount - credit
             then bal := bal - amount; to.in(amount); ok := true
             else ok := false endif
        opr out(amount : int; out ok : bool) ==
             if owner = x ∧ bal ≥ amount - credit
             then bal := bal - amount; ok := true
             else ok := false endif
        opr interest() == bal := bal - bal * account_rate / 100
    inv   ((h ⊢ x←me.trans(a,to,true) ∨ h ⊢ x←me.out(a,true)) head ℋ)
          ⇒(bal2(h/{←.in,←.out,←.trans }) + credit ≥ a)
end
```

The class “CAccount_with_credit” inherits “CSavings_account” and it redefines three operations. Notice the use of initialization statements inside the class (following the style of Simula).

8 Conclusions and future work

We have introduced a notation based on general principles known from formal methods, combined with all essential object-oriented concepts, and with more flexibility and more dynamic considerations than existing formal methods. We insist on static typing, which means that the software will be reliable in the sense that type errors will not occur, and operations calls to remote objects will always be syntactically correct.

It has been essential to be able to combine static typing with a minimum of dynamic behavior: Objects, interfaces and classes may be added dynamically, and old and new objects may communicate by means of new interfaces (as well as old ones). This is inspired by Java’s concept of byte code and virtual machine.

Object variables and object parameters are typed by interfaces (rather than classes) which not only helps reasoning, but makes software more reusable, more abstract (disallowing write-access to remote variables) and more understandable, and is essential in order to allow and control dynamic behavior. Thus sub-interfaces are inheriting semantical constraints (after projection) while subclasses do not.

At the class level, we allow unrestricted redefinition of operations, possibly violating inherited invariants. This opens up for flexible reuse of code [12] and gives the same notion of subclassing as for instance in Java. Since reasoning and typing are based on interfaces, and since sub-interfaces must respect interface refinement, already proven verification conditions cannot be violated by adding subclasses and redefining operations. (Notice that a class invariant in itself does not create a verification condition, and a subclass violating it does not violate any verification conditions.)

We include a form of dynamic class extension, which enables us to extend a class dynamically, respecting inherited invariants. Together with dynamic creation of objects and addition of interfaces, this allows non-trivial dynamic behavior. The class extension mechanism may be seen as a controlled version of capabilities in Corba and Java RMI, but staying inside the framework of static typing.

As the class extension mechanism has no other syntactic restrictions than disallowing redefinitions, a consequence is that redefinition must be semantically unrestricted, and operations must be overloaded (since a subclass and an independent extension of a superclass may both define operations with the same name, either with different parameters, or the same parameters but different semantics). Thus the object oriented concepts of our notation are both orthogonal (i.e. can be used with great flexibility) and are well integrated. Reasoning control is achieved by the generation of verification conditions for each program unit, while the language provide a guarantee that already proven verification conditions are not violated.

This paper is focusing on safety properties only (excluding full treatment of deadlocks) in order that the semantics is a simple as possible. Future works will focus on other deadlock and liveness properties. To deal with these kinds of properties, we need a stronger semantics which will increase the complexity of reasoning.

Future works will also include extension of the language in order to deal with essential constructs like time-outs and exceptions. Both time-out and exceptional termination may be denoted in the traces by special, completion-like events. A method call that raises an exception has no "normal" termination, but causes a message return to the calling object indicating the exceptional termination (including its kind). This message can be recorded in the trace by a special event matching the initiation like a normal termination. Thus, we keep the coherence of the trace by coupling initiation and termination-like events. This principle may also by applied to time-outs by adding another kind of events, indicating "termination by time-out". However, the problem is a little bit more complicated than in the case of exceptions, since we have to reason about time. There is no notion of real time in OUN, but we can introduce an approximate notion of time based on reasoning about guarded commands.

Besides these works for extending OUN, some efforts are engaged to develop an OUN Toolkit based on the PVS [11] Toolkit (see [14]).

Acknowledgment

We are grateful for feedback from discussions with the members of the ADAPT-FT project group. In particular, Ole-Johan Dahl, Einar Broch Johnsen, and

Ketil Stølen have provided valuable detailed advice concerning the manuscript.

References

[1] Alpern, B., and Schneider, F. B. Defining liveness. *Information Processing Letters 21*, 4 (Oct. 1985), 181–185.

[2] Back, R. J. R., and Von Wright, J. Trace refinement of action systems. In *Proc. 5th International Conference on Concurrency Theory (CONCUR'94)* (Uppsala, Sweden, 1994), B. Jonsson and J. Parrow, Eds., vol. 836 of *Lecture Notes in Computer Science*, Springer, pp. 367–384.

[3] Broy, M. Compositional refinement of interactive systems. *Journal of the ACM 44*, 6 (Nov. 1997), 850–891.

[4] Broy, M., and Stølen, K. *FOCUS on System Development.* Book Manuscript, 1998.

[5] Dahl, O.-J. *Verifiable Programming.* International Series in Computer Science. Prentice-Hall, New York, N.Y., 1992.

[6] Dahl, O.-J., and Owe, O. Formal methods and the RM-ODP. Tech. Rep. 261, Department of Informatics, University of Oslo, 1998.

[7] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.

[8] Jones, C. B. *Developments Methods for Computer Programms – Including a Notion of Interference.* PhD thesis, University of Oxford, 1981.

[9] Lano, K., and Haughton, H. Reasoning and refinement in object-oriented specification languages. In *Proc. European Conference on Object-Oriented Programming (ECOOP'92)* (Utrecht, The Netherlands, 1992), O. L. Madsen, Ed., vol. 615 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 78–97.

[10] Mikhajlova, A., and Sekerinski, E. Class refinement and interface refinement in object-oriented programs. In *Proc. 4th International Symposium of Formal Methods Europe (FME'97): Industrial Applications and Strengthened Foundations of Formal Methods* (1997), J. Fitzgerald, C. B. Jones, and P. Lucas, Eds., vol. 1313 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 82–101.

[11] Owre, S., Rushby, J., Shankar, N., and von Henke, F. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering 21*, 2 (1995), 107–125.

[12] Soundarajan, N., and Fridella, S. Inheritance: From code reuse to reasoning reuse. In *Proc. 5th Conference on Software Reuse (ICSR5)* (1998), P. Devanbu and J. Poulin, Eds., IEEE Computer Society Press, pp. 206–215.

[13] Spivey, J. M. *The Z Notation: a Reference Manual.* Prentice Hall, 1989.

[14] Traoré, I. The UML specification of the Integrator. Tech. rep., OECD Halden Reactor Project, 1999.