UNIVERSITY OF OSLO
Department of Informatics

A notation for combining formal reasoning, object orientation and openness.

Olaf Owe and
Isabelle Ryl

Research report 278

November 1999

A notation for combining formal reasoning, object orientation and openness.

Olaf Owe* Isabelle Ryl†

November 1999

* Department of Informatics, University of Oslo, Norway.
† L.I.F.L., University of Lille, France.
e-mail:olaf@ifi.uio.no, ryl@lifl.fr

Abstract

Safety reasoning about systems is a difficult task unless the specification language used to describe the system is chosen carefully. In case of object oriented open distributed systems, the fact that most existing formal methods are not allowing the kind of openness and object orientation that are proposed by Corba, Java RMI, ..., increases the difficulty. In this paper, we propose a formalism for specification and design including, as much as possible, high level object oriented programming language concepts, but restricted such that reasoning is statically manageable.

The proposed notation leans on trace semantics and allows us to define interfaces, classes and contracts using classical concepts of object oriented programming languages such as multiple inheritance (for interfaces) and single inheritance (for classes). Dynamical aspects are taken into account by allowing dynamical addition of objects, (sub)interfaces, and (sub)classes to a running system as well as dynamical extension of classes.

1 Introduction

The use of object-oriented programming languages and open distributed systems has grown considerably in the latest years – in parallel to renewed interest among developers for formal methods. But, nowadays, most existing formal methods are not allowing the kind of openness and object orientation that are desired by programmers. This includes the work on process calculi (initiated by CSP or CCS), temporal logic, UNITY, Actors, TLA, Z, VDM, FOCUS, TROLL and so on. A more detailed assessment and comparison is given in [23]. Maude [17] is one of few object oriented formalisms with great flexibility, for instance the class of an object may change from one state to the other, not even restricted by a subclass relationship. However, reasoning is rather limited, even safety reasoning is not (yet) available.

We are aiming at a practically useful formalism, OUN, 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, the reasoning control will be based on static (strong) typing and static proofs, and the generation of verification conditions will be based on static analysis of pieces of program or specification text. The purpose of this paper is to introduce a formal notation that fulfills the requirements, to present its main aspects, and to discuss the combination of formal reasoning, object orientation and openness. We present here the first version of the notation for which we limit ourselves to consider safety properties (corresponding to safety properties in [1] except deadlock freedom).

Efforts have been made to make the notation as clear as possible to programmers: the syntax is close to that of programming languages, formulas are expressed in first order logic and some concepts are inspired by UML which is recognized as a central standard in object-oriented modeling. For example, interaction diagrams of UML can easily by translated into specification requirements in OUN.

2 Fundamental concepts

We suggest to separate the description of a system into two levels. At the most abstract level, the specification language – OUN-SL – allows us to specify interfaces and contracts. At the design level, the language – OUN-DL – allows us to specify classes and to generate objects. This separation between abstract and concrete specification of classes gives us a large degree of freedom in introducing high-level object-oriented concepts at the design level:

- objects are considered to have internal activity, running in parallel, thereby supporting distribution;
- an object may support a number of interfaces, and this number may increase dynamically;
- multiple inheritance of interfaces and single inheritance of class may be used to define new interfaces and new classes;
- new (sub)interfaces and (sub)classes may be added to a running system;
- classes may be dynamically extended by addition of methods.

Requirement specifications of both classes and interfaces consist of invariants and assumptions about the behavior of the environment, given be means of first order formulas on the trace (finite communication history) of the (sub)system. We use a kind of rely-guarantee specification [12], but with an interpretation specialized for communicating systems. The OUN notation leans on standard trace semantics, similar to that of CSP (without refusals and divergences) [11], except that objects have identity.

We consider 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, provided any new verification conditions are proved. For instance, when dynamically extending a class with new methods, it must be proved that state changes caused by the new methods do not violate the old invariant.

Neither Java nor Corba are suitable for static reasoning. We therefore reconsider object oriented language constructs supporting both openness and reasoning control. This means that we cannot support the same kind of openness as Corba or Java RMI. For instance in these languages a program unit may invoke methods that were not known when the unit was written, thus only dynamic information of any semantic specification of such a method will be available, and any related verification conditions can only be generated at the time of the invocation. This is not consistent with textual generation of proof obligations.

The notation offers a large degree of *flexibility*: several communication paradigms can be used, methods can be redefined without syntactic or semantic restrictions, unrestricted overloading of methods is allowed, and multiple inheritance of interfaces is allowed. Finally, certain kinds of *reflection* are considered: for example one may ask an object if it supports a given interface. And interactive users may ask an object which interfaces it supports and call any method in one of these. (Interactive users are not subject to verification conditions.)

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 exceptions or objects not responding.)

2.1 Inheritance versus behavioral subtyping

The notion of behavioral subtyping is useful to describe inheritance (but also to relate classes which are not in a subclass relation). A class B is said to be a *behavioral subtype* of A, if a B-object will behave as an A-object, consistent with the specification of possible A behaviors in some sense. Many formalisms supporting inheritance are based on the notion of (weak or strong) behavioral subtyping [18, 16, 15, 2, 8], i.e. it is required that a subclass B is a behavioral subtype of a superclass A. This means that where you expect an A-object, but get a B-object, no surprises will result, in some sense. This is essential in order to avoid reverifying the client code when based on a superclass.

In most object oriented programming languages, it is indeed the case that a subclass object may be used as a superclass object. In the presence of dynamic (virtual) binding, static information about invoked methods are not available. Therefore behavioral subtyping is the natural way to treat subclassing in presence of a formalism for reasoning. Superclass requirement specifications are then inherited. However, this puts severe semantic restrictions on the subclass mechanism, added methods must maintain the inherited class invariant (which limits the updates of inherited attributes) and redefined methods must obey the old specification. This imposes a significant restriction to the kind of subclassing traditionally allowed in object oriented languages (from Simula to Java). Even though there are different notions of behavioral subtyping, stronger and weaker ones, it is obvious that many useful subclass definitions will be semantically illegal.

On one hand we have seen that subclassing respecting behavior is needed, in order to be able to reason, and on the other hand we have seen that this kind

of subclassing does not go well together with the ideas of code reuse, which are essential in object oriented programming.

A partial solution to this, suggested by several authors [13, 8, 2, 21], is to distinguish between an abstract and a concrete specification of a class. Then object reasoning is governed by the abstract specifications, and behavioral subtyping is only required between the relevant abstract specifications. A concrete specifications is used to prove that a class satisfies the corresponding abstract specification. An abstract specification could for instance be disallowed to refer to state variables. However, if there is exactly one abstract specification for each class, code reuse may still be difficult in that a subclass may break the inherited abstract specification.

As a better solution, we suggest the use of interfaces as follows:

- An interface contains syntactic declaration of methods (somewhat like Java) together with semantic (rely-guarantee) requirements, specifying observable behavior, but does not contain code nor attribute declarations (i.e. variables).
- A class may implement many interfaces. The implements relationships can be stated incrementally, by special "implements-clauses", resulting in verification conditions (see more below).
- There are independent inheritance hierarchies for interfaces and for classes. Thus a subclass need not implement an interface even if a superclass does!
- Multiple inheritance of interfaces and single inheritance of classes are allowed. The later is motivated by simplicity and efficiency, and is essential when the target implementation language only offers single inheritance of classes.
- Finally we suggest that all object variables and object parameters are typed by interfaces, even at the design level! Classes can then be used to declare subclasses and to create new objects.

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 may be extended dynamically by adding operations and thereby possibly implementing additional interfaces.

An object variable x declared of interface F must at run time refer to a class implementing F; thus the assignment $x := e$ is legal if the static type of the expression e is subinterface of F, or is a **new** construct of the form

$$\textbf{new}\ C(l)$$

where C is a class implementing F, and l a list of (type correct) actual parameters. The object may get knowledge of other objects through the list of actual parameters (l) and through actual parameters of methods calls made to the object throughout its life.

A class is said to *correctly implement* an interface if all operations in the interface are implemented in the class and if the requirement specification of the class implies that of the interface, after appropriate projection of the trace (see below). When the class of an object implements several interfaces we say

that the object *supports* these interfaces. For every stated implements-clause between a class and an interface, there must be a proof that the class implements the interface correctly.

The combination of interface and class mechanisms as described above does not seem to exist in any other object oriented language offering requirement specification.

2.2 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 subinterfaces, addition of new "implements" claims, and addition of new operations to a class. And of course, a compilation unit may create new objects (of new and old classes), as may the execution of an operation.

This allows old and new objects to communicate trough new interfaces!

The dynamic 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).

We intend to use a proof system like PVS; then a proof can be represented and stored efficiently as a list of proof commands, which may be rechecked when, say, program code from one location is installed at another location.

3 Basics

3.1 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.

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.

3.2 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, o_1 and o_2 are respectively the initiator and the receiver of the call, and $i_1, \ldots, i_j$ and $r_1, \ldots, r_k$ respectively denote the **in**- and **out**-parameter values. 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.

3.3 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 $\mathcal{H}$ onto a set of events, Set, denoted h/Set, is defined inductively by:

$$\begin{array}{lcl} \varepsilon/Set & = & \varepsilon \\ (\mathcal{H} \vdash x)/Set & = & \begin{cases} (\mathcal{H}/Set) \vdash x & \text{, if } x \in Set \\ (\mathcal{H}/Set) & \text{, otherwise} \end{cases} \end{array}$$

where ε denotes the empty sequence and $\vdash$ denotes the right append.

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,

- 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_x^y the formula obtained by substituting x for every y in the formula P.

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

4 OUN Specification level

4.1 Interface definition

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, 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.

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 $\mathrm{m}_1(\ldots)$
    ...
    opr $\mathrm{m}_i(\ldots)$
  asm <*formula_on_*$\mathcal{H}$*_and_any_external_object_x*>
  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 ordinary brackets) which can be data types or interfaces. The additional list of parameters (between square 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 (or subinterfaces of 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 **with** clauses provide a strong control of callers. First, access to operations is controlled and reserved for some types of objects. Second, an object has a good knowledge of its callers (in terms of minimal interfaces they offer) and may initiate some response itself as the signature of operations of its callers are known. This mechanism may seem rather restrictive, however the interface **any** may always be used to open access to any object.

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

$$\textbf{func} <name> == \ldots$$

Similarly, data type may be defined as well. There is an underlying language for defining data types, including predefined types such as integer, boolean, character and text. This sublanguage is not discussed further; however, in the examples we will use that of [8].

4.2 Basic interface semantics

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, denoted by $\mathcal{H}/o : F$, 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 interfaces that appear in **with** clauses of F (or its super-interfaces) or of the interfaces of the objects parameters of F initiated by o. Let $G_1, G_2, \ldots, G_n$ denote the interfaces that appear in **with** clauses of F and $x_1 : J_1, \ldots, x_k : J_k$ denote the object parameters of the interface F. Thus, the alphabet $o : F$ is defined by

$$\begin{array}{rll} \alpha(o:F) \equiv & \{o' \leftrightarrow o.m(\ldots) \mid \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge m \textbf{ in } F\} \\ \cup & \{o \leftrightarrow o'.m(\ldots) \mid \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge \exists i \bullet m \textbf{ in } {G_i}^F\} \\ \cup & \{o \leftrightarrow x_i.m(\ldots) \mid m \textbf{ in } J_i^F\}. \end{array}$$

We are now able to give a precise definition of assumption and invariant of an interface.

The communication control of a object is distributed between its assumption – for inputs – and its invariant – for outputs. We denote by respectively $\textbf{in}(\mathcal{H})$ and $\textbf{out}(\mathcal{H})$, the longest left prefix of a trace $\mathcal{H}$ ending by an input event and by an output event. An assumption and invariant given in an interface specification have the following meaning:

- the assumption[1] **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

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

  holds.

- the invariant[2] **inv** $I(\mathcal{H})$, where I is a formula, states that for any object o of this interface, the formula

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

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

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

For an interface F defined without using inheritance, the basic trace set of an object $o : F$ is defined by

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

where

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

So, the basic trace set of an object $o : F$ is the set of local traces of o seen through interfaces F characterized by "if the environment satisfies the assumption of the interface then the object o satisfies the invariant of the interface".[3] The assumption control for traces ending by an output event ensures that an object will not break its own assumption.

Example 1 Let us 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 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

Remarks. The operator **prs** denotes "prefix of regular expression", and is used frequently in the examples to describe a communication pattern. We use sets of regular expressions (with — for choice and * for repetition). In a regular expression, the abbreviation $\leftrightarrow.m$ denotes the juxtaposition of the initiation and the termination of the operation m. For convenience, identifiers of objects are not written when irrelevant, and $\mathcal{H}/\leftarrow$ denotes the projection of $\mathcal{H}$ onto all termination events.

[3] In addition, in order to obtain a prefix closed trace set, we exclude all traces having a prefix outside the set.

4.3 Contracts

Beside specification requirements of a single interface, the specification language also offers the possibility to define contracts. Contracts give a kind of "glass-box" specifications, they allow us to describe interactions between two or more objects. Contracts are suitable for expressing specification requirements at an early stage, especially when there are objects mutually depending on each other. The main properties of the interactions between objects of a system may be described by a number of contracts, each involving relevant aspects of object interaction. A contract is expressed within 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 C. The common communication history of these objects is the global history of the system projected onto the set of events involving two of them, it is a sequence of events of the set:

$$Com(o_1 : F_1, \ldots, o_n : F_n)$$
$$=$$
$$\{o_i \leftrightarrow o_j.m(\ldots) \mid i \neq j \wedge \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge m \textbf{ in } F_j^{F_i}\}.$$

Thus contracts express more global properties of the system than invariants and assumptions of interfaces. Contracts have to be satisfied by objects offering the involved interfaces as well as specification requirements of the interfaces themselves so, contracts give additional constraints on the trace sets of objects.

Thus, the trace set $\mathcal{T}_{o:F}$ of an object o of interface F is the subset of its basic trace set $\mathcal{B}_{o:F}$ (defined in the previous subsection) whose traces $\mathcal{H}$ satisfy, for each contract C in which F is involved, the formula:

$$\forall o_1 : F_1, \ldots, o_n : F_n \bullet$$
$$\exists H' \bullet C(H') \wedge H'/o : F = H/Com(o : F, o_1 : F_1, \ldots, o_n : F_n),$$

where $F, F_1, \ldots F_n$ are the interfaces involved in the contract C and where H' is a sequence of events of $Com(o : F, o_1 : F_1, \ldots, o_n : F_n)$.

Contracts may be used at the beginning of the specification to capture the global properties of the system and be split during refinement into assumptions and invariants distributed on several interfaces. The split of contract is not obvious when there are more than two objects and can not be generated mechanically, in general, by projection onto events involving one particular object (this is a well known problem of formal languages theory, see [6] for example). However, in a number of cases, including those involving two objects, the split is very easy to realize. In a realistic situation, when speaking about protocols for instance, the sets of histories defined by the contracts can by described using projections onto each of the involved objects so, the constraints of the contracts can be split in parts of the specifications requirements of the interfaces.

4.4 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 [4, 5] or in CSP [11]. 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 subinterface. We suggest the use of projection when relating the trace of an object of an interface to that of a super-interface following [9]. 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 subclassing [14, 19] 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 [22] and the trace refinement of action systems in [3] 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 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 subinterface 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).

4.5 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 basic trace set of an object of interface F that inherits $F_1, \ldots, F_n$ and whose assumption and invariant are respectively A and I, is defined by:

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

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 inconsistency 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.

Example 2 Now, we can easily define an interface for read/write access control using R and W defined in Example 1. 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 access are still allowed.

interface RW [T: **Data-Type**]
  inherits W, R
begin
  with x: **any**
    opr open_read()
    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 function # denotes the number of events of a sequence. 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 design level

OUN-DL is the design language of OUN. This level constitutes 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'
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' is a class. 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 one class (C'). 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 operation calls received by o and the set of calls, initiated by o, of operations of the interfaces that appear in the **with** clauses of C (or in the **with** clauses of the superclasses of C), and the set of calls to object parameters of C. Let $G_1, G_2, \ldots, G_n$ denote the interfaces of the **with** clauses and $x_1 : J_1, \ldots, x_k : J_k$ denote the object parameters of C. Then, the alphabet $o : C$ is defined by:

$$\begin{array}{lll} o : C \equiv & & \{o_1 \leftrightarrow o_2.m \mid \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge o_2 = o \wedge m \textbf{ in } C\} \\ & \cup & \{o_1 \leftrightarrow o_2.m \mid \leftrightarrow \in \{\rightarrow, \leftarrow\} \wedge o_1 = o \wedge m \textbf{ in } G_1 \cup \ldots \cup G_n\} \\ & \cup & \{o \rightarrow x_i.m(\ldots) \mid m \textbf{ in } J_i\}. \end{array}$$

- The 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

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

  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,

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

  where x **of** G_i means that x implements G_i or one of its subinterfaces, 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:

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

  where

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

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

We can then extend the definition of refinement introduced for interface to classes in a natural way. A class C is said to refine an interface or a class X if and only if for all object o of class C:

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

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.

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 subinterface 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

Single inheritance of classes may be used to define new classes. The use of single inheritance of classes is motivated by the need of simplicity. Moreover, our final goal is to obtain a safe specification of a system that can be implemented in any "popular" programming language, for example Java. So, the use of multiple inheritance of classes in the design of the system would imply a modification of the structure of the system between design and implementation and then cause extra work and break of the model.

A subclass may add attributes, add operations, redefine attributes and redefine operations. A subclass inherits those attributes and 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 [21]). 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.

Subclasses may also redefine attributes, letting the new ones shade for the old ones (as in Java and as for nested Algol60 blocks). But both attributes of the superclass and redefined attributes exist in the implemented data structure. Both shaded methods and shaded attributes may be accessed using an explicit superclass qualification.

Because of this class inheritance definition, free reuse of code, redefinitions and everything that is traditionally appreciated by programmers are allowed. This freedom at the class level is obtained thanks to the typing by interfaces. In this way, we have a complete separation between specification inheritance and code inheritance and we avoid classical problems discussed in the literature as "inheritance anomalies" (see for example [7]).

5.4 Dynamical Extension

Within the context of distributed systems, the recent notion of dynamical extension of classes has to be considered. This concept offers a programmer the possibility of modifying a class and all the already existing objects of this class during the execution of a system. This idea corresponds to the approach of prototype-based languages (see [10] for example) and may be useful in several situations. Let us consider for example some commercial web-site. New functionalities may be added to the server by an administrator without stopping it, that is to say without any trouble for customers.

As already said, we want to include, as much as possible, high level object oriented concepts in OUN. So, OUN offers class extension but, once again, the extension has to be restricted not to violate the safety of the existing system.

Using class extension, some operations and some interfaces can be added dynamically to a OUN 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 and a candidate extension of C that we denote, in the formula by C_{old} and C_{new}, respectively. Then, the extension is safe if and only if the extended class refines the new one. So, it is necessary to prove that for all objects o of the extended class:

$$\forall \mathcal{H} \in \mathcal{T}_{o:C_{new}} \bullet \mathcal{H}/o : C_{old} \in \mathcal{T}_{o:C_{old}}.$$

Since a class extension is realized while objects of this class are involved in some interactions with their environment, it is necessary that these objects still offer at least the same interfaces after than before the extension. By definition, old implementation claims have not to be restated. 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, for each object o of class C, we have:

$$\forall \mathcal{H} \in o : C_{new} \bullet \mathcal{H}/o : F = \mathcal{H}/o : C_{old}/o : F$$

Moreover, by definition of class extension, $\mathcal{H}/o : C_{old}$ belongs to $\mathcal{T}_{o:C_{old}}$ and, since C_{old} implements F, $\mathcal{H}/o : F = \mathcal{H}/o : C_{old}/o : F$ belongs to $\mathcal{T}_{o:F}$, so C implements F.

Clearly new implementation claims have to be proved. Note that because of the class extension mechanisms we do not have complete static control over which interfaces are supported by which objects. So, there is 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

if any $x : F?$ **then** $<$ *may use x here* $>$ **else** $<$ *no such x exists* $>$ **endif**

which may occur in the implementation of an operation.

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.

5.5 Dynamic generation of objects

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

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 $o := new\ C(\ldots)$ executed by x. At the reasoning level (i.e. in the history) this creation is reflected by the event

$$x \rightarrow y.new(C, \ldots)$$

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 to 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 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.

5.7 Example

Let us consider a simple distributed system of a shop: Customers may buy items that carry an electronic tag. A tag is a passive object, it needs a "power on" operation from a tag reader to become active. Then, if it is alive, it gives its price to the reader that has woken it up. When a tag receives a "sleep" order, it stops to communicate and cannot be activated any more.

We will consider the following simplified protocol: When a customer goes to the check-out, tags of the items he wishes to buy communicate with the cashier and the cashier passivates the tags. When a tag comes in front of a security door, the security system of the shop tries to activate the tag, and in case of response, detects a theft and calls some kind of police.

We will consider the interfaces **Tag**, **Door**, **Check-out**, and **Police** to model the system and use contracts to describe the global properties we need.

The invariant of the interface **Tag** ensures that a tag responds to an "on" operation by sending a "give_price" operation, provided it has not yet received any "sleep" order. The operators $\in$ and **head** have to be read as "is an event occurring in" and "is the first event of", respectively.

interface Tag
begin
  with r: Tag_Reader

```
        opr on()
        opr sleep()
    inv H/→ prs
        (r→this.on this→r.give_price | r: Tag_Reader)* sleep (on | sleep)*
end
```

The invariant of this interface says that the tag responds to "on" from a reader by "give_price" to this reader until it receives a "sleep" then it stops to respond.

The interface **Trolley** is used to modelise the fact that the customer is moving in the shop and sometimes arrives in front of a tag reader, the police may just receive calls and the next interface only specifies the operation that any tag reader must implement to receive responses of tags.

```
interface Police
begin
    with d: Door
        opr call()
end
```

```
interface Trolley
begin
    with c: Customer
        opr show(t : Tag)
end
```

```
interface Tag_Reader
    inherits Trolley
begin
    with t: Tag
        opr give_price(price : nat)
end
```

The check-out is a specialized tag reader which switches off tags as soon as it has received their prices. The Door is a specialized tag reader, similar to the check-out except that it calls the police as soon as it has received the price of a tag instead of switching it off .

```
interface Door(station : Police)
inherits Tag_Reader
begin
    inv H= h ⊢ t→this.give_price(x) ⊣ h' ⇒
                        (this→station.call() head h') ∨ (h' = emptyseq)
end
```

```
interface Check-out
    inherits Tag_Reader
begin
    inv H= h ⊢ t→this.give_price(x) ⊣ h' ⇒
                            (this→t.sleep() head h') ∨ (h' = emptyseq)
end
```

We first write a contract saying that as soon as a tag has given its price to a check-out (*i.e.* the corresponding item has been paid), a security system will not call the police.

```
contract No-theft-no-police
begin
    with t: Tag, c: Check-out, d: Door(p)
    inv (H= h ⊢ t→c.give_price(x) ⊣ h') ⇒ (¬ (d→p.call() ∈ h'))
end
```

Then, a second contract ensures that an item that has not been paid and is coming to the door will be detected and that the police will be called.

contract Theft-detection
begin
  with t: Tag, c: Check-out, d: Door(p)
  inv ($\mathcal{H}$= h $\vdash$ d$\rightarrow$t.on $\dashv$ h' $\wedge$ $\neg$ t$\rightarrow$c.give_price(x) $\in$ h) $\Rightarrow$
    (d$\rightarrow$p.call() $\in$ h') $\vee$ (h' = **emptyseq**)
end

This ends the abstract specification of our system. It is possible to check that the invariants given for each interface imply the contracts. Now we can come to the design level. We design three classes corresponding to tags, check-outs and doors. The method codes are very simple, for example, a check-out only register the sum of the prices of tags it has communicated with. We sometimes use the same invariant in a class implementing an interface as in the interface (especially in small examples); then we use the notation $< name >$.**inv** as a shorthand.

class CTag (price : **nat**)
  implements Tag
begin
  var alive : **bool**:= true;
  with r: Tag_Reader
    opr on () == **if** alive **then** r.give_price(price); **endif**
    opr sleep () == alive := false;
  inv Tag.**inv**
end

class CCheck-out
  implements Check-out
begin
  var sum : **nat**:= 0;
  with t: Tag
    opr give_price(price : **nat**) == sum = sum+ price; t.sleep();
  with c: Customer
    opr show(t : Tag) == t.on();
  inv Check-out.**inv**
end

class CDoor(station : Police)
  implements Door, Tag_Reader
begin
  with t: Tag
    opr give_price (price : **nat**) == station.call();
  with c: Customer
    opr show(t : Tag) == t.on();
  inv Door.**inv**
end

Then, we have to prove (exercise for the reader) that each class invariant is true after initialization and is maintained by the operations. Obviously, implementation claims are correct.

Now we imagine a small extension of this system considering a customer who pays for what he has bought. We need to extend the class **CCheck-out** by the addition of a new interface (which is also a new interface in the system) dedicated to customers and a new method giving the price to pay.

```
interface Payment
begin
    with c: Customer
        opr total(out amount : nat)
end

   class extension CCheck-out
      implements Payment
   begin
      with c: Customer
         opr total(out amount : nat) == amount := sum; sum := 0;
      inv Check-out.inv
   end
```

It is trivial to prove that the old invariant is maintained by this new method, so the new version of the class is obviously a refinement of the old.

6 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 subinterfaces 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 [21] and gives the same notion of subclassing as for instance in Java. Since reasoning and typing are based on interfaces, and since subinterfaces 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 in subclasses, 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 extension 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 [20] Toolkit (see [24]).

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] America, P. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop (1990)*, vol. 489 of *LNCS*. Springer-Verlag, 1991, pp. 60–90.

[3] Back, R. J. R., and Von Wright, J. Trace refinement of action systems. In *Proc. CONCUR'94* (Uppsala, Sweden, 1994), vol. 836 of *LNCS*, Springer-Verlag, pp. 367–384.

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

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

[6] Cori, R., and Métivier, Y. Recognizable subsets of some partially Abelian monoids. *Theoretical Computer Science 35*, 2-3 (1985), 179–189.

[7] Crnogorac, L., Rao, A. S., and Ramamohanarao, K. Inheritance anomaly — A formal treatment. In *Proc. FMOODS'97* (Canterbury, UK, 1997), vol. 2, Chapman & Hall, pp. 319–334.

[8] Dahl, O.-J. *Verifiable Programming*. Prentice-Hall, 1992.

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

[10] Dony, C., Malenfant, J., and Cointe, P. Prototype-based languages: From a taxonomy to constructive proposals and their validation. *ACM SIGPLAN Notices 27*, 10 (1992).

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

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

[13] Jones, C. B. *Systematic Software Development Using VDM*, second ed. Prentice Hall, 1990.

[14] Lano, K., and Haughton, H. Reasoning and refinement in object-oriented specification languages. In *Proc. ECOOP'92* (Utrecht, The Netherlands, 1992), vol. 615 of *LNCS*, Springer-Verlag, pp. 78–97.

[15] Liskov, B., and Wing, J. M. A new definition of the subtype relation. In *Proc. ECOOP'93 /* (Kaiserslautern, Germany, 1993), no. 707 in LNCS, Springer-Verlag, pp. 118–141.

[16] Liskov, B., and Wing, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 1 (1994), 1811–1841.

[17] Meseguer, J. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science 96*, 1 (1992), 73–155.

[18] Meyer, B. *Object-Oriented Software Construction*, second ed. Prentice-Hall, 1997.

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

[20] 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.

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

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

[23] Stølen, K. A comparison of eleven specification languages. Tech. Rep. HWR-523, OECD Halden Reactor Project, Norway, 1998.

[24] Traoré, I. The UML specification of the Integrator. Tech. Rep. 275, Dept. of Informatics, Univ. of Oslo, 1999.