Formal Methods and the RM-ODP

Ole-Johan Dahl and Olaf Owe
Department of Informatics, University of Oslo
May 1998

Abstract

The RM-ODP documents are criticized for unsatisfactory definition of basic concepts such as object, class, and subclass, as well as values, types, and subtypes. Whereas elements of an operational semantics of ODP systems is provided, and the applicability of certain well known specification languages is discussed, no methodology for class specification, implementation development, and verification is given. We propose a set of recommendations to remedy these shortcomings. Nontrivial examples are provided.

1 The Basic Concepts

This report contains a few reflections on the contents of the document ISO/IEC JTC1/SC21/WG7, titled "Reference Model of Open Distributed Processing" [7], shorthand RM-ODP, dated 1995-6-7.

The purpose of the RM-ODP is to provide a standardized framework for the description, development, and implementation of Open Distributed Systems. The following paragraph occurs in an early section on "objectives and motivations":

> The development of a framework for system specification and the corresponding infrastructure components is the general goal of ODP standardization. The RM-ODP provides the framework and enables ODP standards to be developed specifying components that are mutually consistent and can be combined to build infrastructures matched to user requirements. Complying with ODP architectural principles and conforming to ODP standards in the construction of distributed systems will result in open distributed systems.

The RM-ODP defines a division of ODP system specifications into several *viewpoints*:

- the *enterprise viewpoint*, which is concerned with the business activities of the specified system;
- the *information viewpoint*, which is concerned with the information that needs to be stored and processed in the system;
- the *computational viewpoint*, which is concerned with the description of the system as a set of *objects* that interact at *interfaces* – enabling system distribution;
- the *engineering viewpoint*, which is concerned with the mechanisms supporting system distribution;

- the *technology viewpoint*, which is concerned with the details of the components from which the distributed system is constructed.

Each viewpoint is supposed to be an "abstraction" that yields a specification of the whole system. It is emphasized that different viewpoint specifications must be mutually consistent, but no advice is offered about how to verify such consistency.

Some general concepts are identified, presumably to be used for specification under any viewpoint. The ODP framework is object-oriented, thus the most basic concepts are said to be those of *object* and *action*:

> ... All things of interest are modelled as objects. Anything of interest that can happen is an action. ...

Also:

> objects can be of an arbitrary granularity (e.g. they can be as large as the telephone network, or as small as an integer [variable?]).

Objects interact through interfaces ranging from asynchronous message passing to autonomous data transfer. An object may have several interfaces, possibly associated with separate "roles". Abstraction is identified as an important principle which enables better modulation. There is no further explanation, and no relationship to the different viewpoints. It is said that an object contains "information", and that its "state" may be changed.

Standard ODP system organization in terms of objects, clusters, capsules and nodes is described. There are guidelines for the implementation of access transparency across physical and administrative borders, in terms of objects performing detailed tasks informally described.

The RM-ODP is divided into 4 parts:

1. Overview
2. Foundations
3. Architecture
4. Architectural Semantics Amendment

In the following section we give some more detailed comments to RM-ODP, in particular parts 1 and 4. In a final section we propose a method for object specification, (abstract) implementation, and verification.

2 Formal Methods and ODP

2.1 RM-ODP Part 1

The object concept may be sufficiently well explained for purposes of concrete programming in a language containing some suitable class concept, but not by far for purposes of formal methods, such as abstraction and refinement techniques. Concepts of class and type are introduced, as well as subclasses and subtypes. Both are said to be defined by sets of predicates, such as "being red". Although (sub-)types and (sub-)classes are said to go "hand in hand", the relationship between these pairs of concepts is not made clear at all. These concepts, as explained in the Part 1, are thus fairly insubstantial. A concept of template is mentioned, somehow describing a class of objects, called a template-class. Template-classes may correspond to class constructs available in some programming languages.

According to the Part 1 classes and types are in general defined by "predicates". One example of such a predicate is provided: "X_is_red", where X is an object, describes X as belonging to the class of "red objects".

Suppose we want to convey the information that X is monochrome. Clearly, the falsity of X_is_blue does not follow logically from X being *red*; in fact, the object might be multicoloured. It would be necessary to add one predicate of the form $X\_is\_not\_C$ for every expressible colour C other than *red*. Thus, a complete specification of the monochromicity of X is only possible if a value type identifying all distinct discernible colours is provided, for instance an enumeration type $COLOUR = \{red, blue, green, \ldots\}$. If so, universal quantification can be used:

$$\forall C : COLOUR \bullet C \neq red \Rightarrow X\_is\_not\_C$$

A more practical approach, however, would be to introduce a function, say Col, applicable to (the state of) X:

$$Col : S \longrightarrow COLOUR$$

giving the colour of X, where S is the state space of X. Then the predicate $Col(X) = red$ would convey the information that X is red. Since values of type $COLOUR$ are by definition distinct, the applicability of Col shows that X is monochrome. Notice that if S is the state space of X, the above function profile expresses the applicability of Col to X. That could be a more important property of X than its actual colour.

The existence of "meaningless" expressions must be considered, such as $Col(Y)$, if the Col function does not apply to Y. It may be necessary to work with 3-valued logic, unless applicability is decidable by textual typing of expressions.

In our opinion the RM-ODP idea of restricting the *entire* universe is a mistake, for reasons of constructivity, as well as by ignoring the applicability or non-applicability of operators. Orthogonal subtype or subclass hierarchies can rather be realized through multiple heritage.

In RM-ODP (Part 3) the use of higher order functions is advocated. This means that function spaces will be regarded as types. For subtypes of function spaces the standard convention of contra-variance on domains is assumed, but that does not agree with the general principle of subtypes as subsets (by cardinality reasoning, assuming function equality is not affected by function values for arguments outside the domains.)

The ODP concept of *object interfaces* corresponds to the traditional notion of signature. There is, however, a need for semantic information in addition to the syntactic one. The notion of *role* corresponds to interface subset. The RM-ODP requires that different roles of an object be considered separately. This is impossible in general as far as semantic considerations are concerned. For instance, in the readers/writers example further below, it would be natural to consider a reading role and a writing role. However, the external invariant given can not be split according to the roles.

2.2 Part 4

In part 4, operational semantics for ODP is outlined, in a formalism close to rewriting logic. The transition rules may be seen as a restricted version of the Maude language [11], and may even be formulated within simple Maude (with the addition of specialized messages, reflecting announcement requests, and interrogations requests and responses.) This means that messages are handled locally by objects in an asynchronous manner, and that shared variable interaction is not explicitly provided. Even though the rules apparently are unconditional, we would imagine that conditional rules could be required.

With respect to openness, we observe that an object may not loose interfaces, but may create new ones. Naturally, objects may also be created and deleted, and references to interfaces may be communicated through messages.

Part 4 also discusses how different aspects (viewpoints) of ODP can be "formalized" in different languages such as LOTOS, Z, SDL, and ESTELLE [8, 12, 10, 9]: Only certain aspects of the the enterprise viewpoint language can be reflected by LOTUS; for instance, one may not express obligation, prohibition or temporal constraints. Z may well reflect the more static parts of the information viewpoint language, but is limited with respect to the dynamic parts. SDL and ESTELLE, used as a high level programming languages, may reflect most aspects of the computational viewpoint language, but essential specification issues (such as invariants, refinement notions) are not handled.

Part 4 does not give guidelines on how formal methods can be used to write specifications. The use of invariants, pre- and post-conditions, and restrictions on traces, is mentioned in earlier parts of RM-ODP, but is not discussed in Part 4. Thus, Part 4 lacks methodology on how to write such specifications and develop them into (abstract) programs.

3 Recommendations

It is necessary to consider the fundamental distinction and relationship between on the one hand information, represented by data, called *values* in the sequel, and objects on the other hand:

- A value is a spaceless, timeless and immutable mathematical concept, such as the number 5 or a piece of text. A value type is a set of values sharing properties and (other) applicable operators.

- An object is a representation in space of some value, its state, which may be subject to change over time. Its current state is a value, and its state space a value type. The possible properties and behaviours of an object are defined by its state space, as well as interfaces, operators, and own actions. Objects sharing these characteristics are said to belong to the same class.

Thus, a type is defined by identifying a value set and an associated set of functions. In a programming language the time- and spacelessness of values are realised by leaving it to the language implementation to manage the representation of unnamed values occurring during a computation process. Class objects on the other hand are generated explicitly at run time, together with pointers (essentially memory addresses) for the purpose of object referencing. Also changes of object state are caused by explicit actions.

These differences are crucial in specification: For an object it is meaningful, and useful, to talk about its *history*, representing its interactions with the environment up to a given time. A value, on the other hand, cannot be said to have a history. This means that class specification is essentially different from type specification, as demonstrated further below.

We consider two useful ways of defining subtypes of given value types and similarly subclasses of given classes:

- by restricting the value type (state space) by predicates (object invariants),
- by providing more applicable operators (thereby possibly extending the state space).

There will be a need for incomplete ("loose") specifications. It is, however, usually practical to insist on complete *signatures* specifying type names and function profiles, as well as interfaces (signatures) for objects. Profiles will specify applicability of functions and object interactions. For an open distributed system it is desirable that the signature, including the name space, be modifiable dynamically, at least extendible. The concept of name space is briefly mentioned in Part 1 (section 7.3.1). It is curious, however, that the problems concerned with the dynamic name space management are not discussed anywhere in the RM-ODP.

For object signatures it will facilitate openness to relax the requirement of complete profiles for object operators. Notice that in old languages such as Algol 60, although strongly typed in most respects, there is no requirement for parameter specification of formal procedures. The same is true in Simula 67 for "virtual" (i.e. dynamically bound) object operators [2].

For the complete specification of unrestricted value types inductive techniques are useful, possibly including specialized constructs for such concepts as enumeration types, Cartesian products, and disjoint unions. For the reasoning about open systems higher order functions may be useful and perhaps necessary, even if they will play no explicit role at the level of concrete object implementations.

In some ODP inspired systems interfaces are classified as being types. Unfortunately that does not agree with the fact that historic information is sometimes an essential part of the interface semantics (as type values do not have a history). Rather than introducing another language category it may be better to take a semantically specified interface to be an abstract object (or class) specification. That may necessitate orthogonal subclassing through multiple inheritance.

Important aspects of subclasses are the reuse of code (in the form of superclasses) and the ability of objects of a subclass to "masquerade" as objects of any superclass. For the purpose of the application of formal methods the latter is considered more important. It is then necessary to observe severe restrictions in the use of "redefinition" of operators of subclasses and dynamic ("virtual") operator binding. Thus, the operators of a subclass must satisfy the specifications, including profiles, of those defined for a superclass. Redefinition may nevertheless be useful in order to take advantage of a restricted state space, and may be *required* for updating operators in order not to violate a strengthened object invariant. Analogous restrictions should apply to the redefinition of operators in subtypes.

For redefinable operators it may be useful to provide incomplete specifications. Even so, "partial" correctness may be the appropriate concept for a redefined operator which has to trap results outside a restricted subtype. Example: arithmetic operations giving values of the type "bounded integer" (seen as a subtype of "integer"). Alternatively, operators may have associated preconditions, leading to proof obligations with use, or to explicit testing. In the latter case a negative result could lead to abortion or, in a concurrent setting, possibly to waiting as in "guarded commands".

For the abstract specification of objects it is in general necessary to consider historic information, such as finite or infinite sequences of parameterized "events" representing interactions between an object and its environment. An object specification in terms of such sequences is fully abstract in the sense that only information visible outside the object is considered, namely its signature and operator invocations including transmitted parameter values.

From this viewpoint subclass specification may consist in enriching historic event sequences by new event types and/or restricting the set of allowed histories. Conversely, a superclass may be obtained by the projection of histories onto a reduced set of event types and/or by removing restrictions on allowed histories. These are special cases of the more general concepts of refinement and abstraction, respectively.

The concept of "heritage anomalies" discussed in the literature, [1], is primarily concerned with the reuse of code. From the point of view of formal methods the satisfaction of specifications is more important. As already mentioned, it may be necessary to rewrite operators in a subclass, e.g by testing for an additional precondition in order to satisfy a strengthened invariant and/or to deal with additional state variables. If, however, the superclass version of the operator is easily callable within the rewritten one, as for instance in JAVA, the result is that old code is nevertheless likely to be reused.

It would seem that formal methods based on principles sketched above can be combined with the use of such concrete object structures as those advocated in the later chapters of the RM-ODP. However, the treatment of a dynamically varying system signature as a prerequisite to openness remains a difficulty. A simplification would result if new objects were restricted to belong to subclasses of, say, a given "environment" superclass.

4 Examples

In the following we suggest a specification formalism which is an alternative to those presented in Part 4 of RM-OPD, and which is more oriented towards practical specification, rather than operational semantics. In particular we show how to specify interfaces, in our view abstract classes, and how to deal with subclasses and redefinitions. These specifications are then developed into abstract designs in a high-level imperative style. We are here limiting the formalism to deal with safety requirements, and we are demonstrating the formalism by some examples.

Example 1

The following examples are drawn from the literature, [1]. We specify a hierarchy of four kinds of message buffer, $B1$ - $B4$, in terms of invariants on historic sequences, $\mathcal{H}$, of "event records":

$$\mathcal{H} \colon \mathit{Seq}(T_1 \cup \ldots \cup T_n)$$

where each T_i is an event record type, $T_i = \mathit{Op}_i(a_1{:}A_1, \ldots, a_{m_i}{:}A_{m_i}),\ i \in \{1..n\}$. Each event record represents an invocation of one of the operators declared for the class, with associated parameter values.

We use an ad hoc syntax. The term "invariant" signifies that the given predicate on $\mathcal{H}$ also applies to any prefix of $\mathcal{H}$. All class attributes, such as class parameters, operators, and invariants, are inherited. In a subclass the invariant may be strengthened by providing an additional conjunct, and the operator list may be extended. The following operators on finite sequences are used [5]:

$\#\_ \colon \mathit{Seq}(T) \longrightarrow \mathit{Nat}$	sequence length
$\_/\_ \colon \mathit{Seq}(T) \times U \longrightarrow \mathit{Seq}(U)$	projection on event subset
$\_ \le \_ \colon \mathit{Seq}(T) \times \mathit{Seq}(T) \longrightarrow \mathit{Bool}$	prefix relation
$\_.a \colon \mathit{Seq}(T) \longrightarrow \mathit{Seq}(A)$	attribute selector lifted to sequences
$\_\mathbf{prs}\_ \colon \mathit{Seq}(T) \times$ <reg T expr> $\longrightarrow \mathit{Bool}$	prefix of regular sequence

```
class $B1(Msg\colon \mathbf{type}) ==$
spec opr $put(x\colon Msg),\ get(y\colon Msg)$
    invariant $(\mathcal{H}/get).y \le (\mathcal{H}/put).x$
endspec
```

```
subclass $B2(N\colon Pos)$ of $B1 ==$
spec invariant $\#(\mathcal{H}/put) - \#(\mathcal{H}/get) \leq N$
endspec

subclass $B3$ of $B2 ==$
spec opr $get2(y1, y2\colon Msg)$ seen as $get(y1), get(y2)$
endspec

subclass $B4$ of $B3 ==$
spec opr $lock,\ unlock$
      invariant $\mathcal{H}$ prs $[[put|get]^*, lock, unlock]^*$
endspec
```

Notes:

- The **seen as** construct is useful for splitting one atomic action into several smaller ones. It implicitly defines an abstraction function, and invariants are referring to the abstract history. Thus, the $B3$ history is of the type $Seq(put \cup get)$, where each $get2$ activation is recorded as two get events, consecutive in $\mathcal{H}$ by definition.
- In the above hierarchy the history of each subclass does satisfy the invariant specified for any of its superclasses. However, a more flexible and robust inheritance concept is obtained if the variable $\mathcal{H}$ occurring in a class C means $\mathcal{H}'/O_C$, where $\mathcal{H}'$ is the actual history of a C-object, possibly belonging to a subclass of C, and O_C is the set of C-operators (including inherited ones). That would for instance be necessary in $B4$ if there is a subclass of $B4$ introducing more operators.
- This notion of satisfaction is stronger than the one used in [1], where histories involving new operators are not restricted by requirements of a superclass.

We sketch below abstract implementations of the specified classes. Operators are represented as guarded commands, in which an unsatisfied guard represents a waiting period, and each command is a critical region with respect to the class object in question. The invariant of a class object relates the local variables to the object history.

The following additional sequence operators are used in the implementations:

$\varepsilon\colon \longrightarrow Seq$	empty sequence
$\_ \dashv \_\colon T \times Seq(T) \longrightarrow Seq(T)$	append term left
$\_ \vdash \_\colon Seq(T) \times T \longrightarrow Seq(T)$	append term right
$\_ \vdash\!\dashv \_\colon Seq(T) \times Seq(T) \longrightarrow Seq(T)$	concatenate sequences
$\_\mathbf{ew}\_\colon Seq(T) \times T \longrightarrow Bool$	sequence **e**nds **w**ith term

```
class $B1(Msg\colon \mathbf{type}) ==$
impl var $Bf : Seq(Msg) = \varepsilon;$
        invariant $(\mathcal{H}/get).y \vdash\!\dashv Bf = (\mathcal{H}/put).x$
        opr $put(x\colon Msg) == true \to Bf := Bf \vdash x$
              $get(y\colon Msg) == Bf \neq \varepsilon \to (y \dashv Bf) := Bf$
endimpl

subclass $B2(N\colon Pos)$ of $B1 ==$
impl invariant $\#Bf \leq N$
        opr $put(x\colon Msg) == \#Bf \neq N \to$ super $put(x)$
endimpl

subclass $B3$ of $B2 ==$
impl opr $get2(y1, y2\colon Msg) == \#Bf \geq 2 \to get(y1); get(y2)$
endimpl

subclass $B4$ of $B3 ==$
impl var $locked\colon Bool = false$
        invariant $locked \Leftrightarrow \mathcal{H}$ ew $lock$
        opr $put(x\colon Msg) == \neg locked \to$ super $put(x)$
              $get(y\colon Msg) == \neg locked \to$ super $get(y)$
              $get2(y1, y2\colon Msg) == \neg locked \to$ super $get2(y1, y2)$
              $lock == \neg locked \to locked := true$
              $unlock == locked \to locked := false$
endimpl
```

Notes:

- In a subclass where an operator is redefined, the keyword **super** gives access to the operator as defined in the superclass. The notation $(y \dashv B) := B'$, where y and B are variables, is used as a shorthand for assigning the first term of B' to y and the rest of B' to B.
- The observable history $\mathcal{H}$ of operator invocation is obtained by extending the history appropriately at the beginning (after the guard) of each external operator invocation, but making the abstractions given in the specification part of the involved classes, as explained above. Thus, $\mathcal{H}$ is viewed as an implicit local variable whose sequence of event records is extended behind the scenes [3, 4]. The history of the caller is extended similarly. Note that history variables are fictitious; they may not be used in executable code.
- A waiting period caused by a guard is on behalf of the calling object, in the sense that the caller has to wait. It is unable to receive calls while waiting. The current object, on the other hand, will react to outside calls. Thus, if the buffer is empty at the time when a *get* operation in $B1$ is issued by an object p, a *put* operation can be executed in the current $B1$ object while p is waiting, which will enable the *get* to go through.

- Whenever the body of an operation contains calls for local operations, it is required that the associated guards are satisfied, so that the whole body can be executed as a single critical region. Such calls are regarded as implementation details not to be recorded in the history (except as indicated by a **seen as** clause in the specification). For instance, the user is obliged to prove that the guards associated with the *get* operations occurring the body of $get2$ both hold.

- If the body of an operation *op* contains a call for an external one, the latter is on behalf of the current object, whose history will be extended accordingly. Any waiting involved will delay the caller of *op* as well.

Let I_e and I_i be the externally specified invariant and the internal one for a class, respectively. The correctness of the implementation is assured (in the partial sense) by proving $I_e \wedge I_i$ for the latter. (In the case of $B1$ a proof of I_i is sufficient, because I_e is then implied). If all changes to inherited variables are effected through calls for inherited operators, the inherited conjuncts of the internal invariant are necessarily satisfied, and only the local part need be verified.

Example 2

The following hierarchy describes objects controlling read/write access to given stored data. Concurrent read operations are allowed, whereas write operations must be exclusive. In each user process read/write operations must be dynamically enclosed by open/close operations. This fact is expressed as a separate part of the historic invariant, since it is a requirement to user processes.

Each event is understood to have an attribute, $p\!:\!Pid$, identifying the calling process, indicated by subscripting. The subscript is omitted if irrelevant. It is understood that a stated requirement applies to each individual user process; thus, **req** $P(\mathcal{H})$ requires the invariance of $\forall p\!:\!Pid \bullet P(\mathcal{H}/p)$. Inside a given class specification Pid denotes the set of external objects, excluding the "current object". We use angular brackets to denote sequences, curly ones to denote sets, and square brackets are used as meta-parentheses in regular expressions.

class $RW1$ $==$
spec opr OR, R, CR (open-read, read, close-read)
OW, W, CW (open-write, write, close-write)
req $\mathcal{H}$ **prs** $[[OR\,R^*\,CR]|[OW\,[R|W]^*\,CW]]^*$
def $NR(\mathcal{H}) == \#(\mathcal{H}/OR) - \#(\mathcal{H}/CR)$ (fact: $NR(\mathcal{H}) \geq 0$)
$NW(\mathcal{H}) == \#(\mathcal{H}/OW) - \#(\mathcal{H}/CW)$ (fact: $NW(\mathcal{H}) \geq 0$)
inv $(NR(\mathcal{H}) = 0 \vee NW(\mathcal{H}) = 0) \wedge NW(\mathcal{H}) \leq 1$
endspec

A disadvantage of an $RW1$ controller is that readers can monopolize access, which is unfair to writers. As a remedy we introduce in a subclass an operation for requesting writing, RW, which inhibits OR's.

subclass $RW2$ **of** $RW1 ==$
spec opr RW (request writing)
  req $\neg\langle RW, OR\rangle$ **in** $\mathcal{H}$
  inv $\forall p{:}\,Pid \bullet \neg\langle RW_p, OR\rangle$ **in** $\mathcal{H}/\{RW_p, OW_p, OR\}$
endspec

Here q **in** r states that the sequence q occurs consecutively in r. Notice that a call for RW will establish a writer priority which lasts until that same process executes a subsequent OW, even if other writers intervene. The added user requirement serves to prevent deadlocks. It follows from the invariant that write requests from distinct user processes must be "honoured" individually, whereas additional requests from one process are ignored.

Writer processes can now monopolize access, which is unfair to readers. In order to obtain fairness for readers we introduce an operation for requesting reading, RR, which inhibits write requests.

subclass $RW3$ **of** $RW2 ==$
spec opr RR (request reading)
  req $\mathcal{H}$ **prs** $[[RR^{?}\,OR\,R^{*}\,CR]|[RW^{?}\,OW\,[R|W]^{*}\,CW]]^{*}$
  inv $\forall p{:}\,Pid \bullet \neg\langle RR_p, \{RR, RW\}\rangle$ **in** $\mathcal{H}/\{RR, OR_p, RW\}$
endspec

The invariant shows that requests of both kinds may involve waiting. Therefore they are required to occur outside read/write regions in each user process. Fairness for readers, as well as writers has now been established, for the following reasons, assuming that RR and RW operations are treated fairly, and that the same is true for OR and OW operations:

1. Reader priority can be established by issuing a RR operation. The priority lasts until that same process executes a subsequent OR. That is allowed to happen when all outstanding write requests have been honoured (but in fair competition with OR/OW-operations from other processes).

2. RR operations cannot be used to monopolize reading, because at most one read request is allowed at any time.

If it were known that the RR caller would not become engaged in interactions with other objects between the RR and OR operations, a more liberal invariant would be sufficient to guarantee fairness:

inv $\forall p{:}\,Pid \bullet \neg\langle RR_p, RW\rangle$ **in** $\mathcal{H}/\{RR_p, OR_p, RW\}$

It is therefore useful to introduce a specification mechanism, say

RR **leads to** OR

to mean that the two events will be immediately successive in the history of the caller, but not necessarily in that of the current object. Thus, the OR operation will be immediate for any RR occurring during read mode, which means that

RW operations will not be blocked when reading is taking place. If, however, a *RR* operation occurs during writing, the corresponding *OR* should be delayed as seen from the current object.

We show abstract implementations. A guarded command has the general format $G_1 \longrightarrow G_2 \rightarrow C$, where the guards G_1, G_2 serve to enforce respectively the user requirement and the invariant of the class specification. (Empty guards are omitted.) If an event occurs in a state not satisfying G_1 for the caller, program abortion results, whereas invalid G_2 results in waiting as usual. Notice that the truth of G_1 cannot be affected by actions of objects other than the caller p, provided that it adequately implements a test on $\mathcal{H}/p$ (see below).

```
class $RW1$ ==
impl var $rd, wr : Set\{Pid\} = \emptyset, \emptyset$
     inv $\forall p{:}Pid \bullet (p \in rd \Leftrightarrow \mathcal{H}/p$ ew $\{OR, R\}) \wedge$
         $\forall p{:}Pid \bullet (p \in wr \Leftrightarrow \mathcal{H}/p$ ew $\{OW, W\}) \wedge$
         $rd \cap wr = \emptyset \wedge \#wr \leq 1$
     opr $OR_p == p \notin rd \cup wr \longrightarrow wr = \emptyset \rightarrow rd.add(p)$
         $CR_p == p \in rd \longrightarrow rd.remove(p)$
         $OW_p == p \notin rd \cup wr \longrightarrow rd = wr = \emptyset \rightarrow wr.add(p)$
         $CW_p == p \in wr \longrightarrow wr.remove(p)$
         $R_p == p \in rd \cup wr \longrightarrow \langle read \rangle$
         $W_p == p \in wr \longrightarrow \langle write \rangle$
endimpl

subclass $RW2$ of $RW1$ ==
impl var $wp : Set\{Pid\} = \emptyset$                    (writer priority)
     inv $\forall p{:}Pid \bullet p \in wp \Leftrightarrow \mathcal{H}/p/\{RW, OW\}$ ew $RW$
     opr $RW_p == wp.add(p)$
         $OR_p == p \notin wp \longrightarrow wp = \emptyset \rightarrow$ super $OR$
         $OW_p ==$ super $OW$; $wp.remove(p)$
endimpl

subclass $RW3$ of $RW2$ ==
impl var $rp : Set\{Pid\} = \emptyset$                    (reader priority)
     inv $\forall p{:}Pid \bullet (\in rp \Leftrightarrow \mathcal{H}/\{RR_p, OR\}$ ew $RR) \wedge \#rp \leq 1$
     opr $RR_p == p \notin rp \cup wp \cup rd \cup wr \longrightarrow rp = \emptyset \rightarrow rp.add(p)$
         $RW_p == p \notin rp \cup rd \cup wr \longrightarrow rp = \emptyset \rightarrow$ super $RW$
         $OR ==$ super $OR$; $rp.remove(p)$
endimpl
```

A natural way to implement a specification such as *RR* **leads to** *OR* would be to include a call for the latter in the body of the former, say thus:

$RR_p == p \notin rp \cup wp \cup rd \cup wr \longrightarrow rp = \emptyset \rightarrow rp.add(p)$; **call** *OR*

where the keyword **call** indicates that a separate event should be included in the history, and that waiting results if that (second) guard is not satisfied. The *OR* operator itself could be made inaccessible directly.

The introduction of the "**seen as**" (and "**call**") mechanisms represents an essential increase in expressivity. For instance, it becomes possible to distinguish between events of "initiation" and "termination" of operations (as e.g. in [6]).

In our example the operations *RR* and *OR* may be seen as the initiation and termination, respectively, of a compound "priority read" operation. We are therefore able to give the following external quasi liveness assertion:

$$\#\mathcal{H}/OR > \#\mathcal{H}/CR \Rightarrow \neg\mathcal{H}\,\mathbf{ew}\,RR$$

In states of quiescence the following slightly stronger assertion holds:

$$\#\mathcal{H}/OW = \#\mathcal{H}/CW \Rightarrow \neg\mathcal{H}\,\mathbf{ew}\,RR$$

As an indication of how openness could be facilitated in the case of example 2, the class $RW1$ might be reformulated thus:

```
class $RW1(M: Memory) ==$
impl ⟨as before⟩
        $R_p == p \in rd \cup wr \longrightarrow M.read$
        $W_p == p \in wr \longrightarrow M.write$
endimpl
where:
class $Memory ==$
spec opr $read, write$
endspec
```

An actual parameter to a *RW* object should be an object belonging to a subclass of *Memory*, whose operators would have parameters appropriate for the kind of device in question. It remains to implement either the replacement of a given memory object parameter by another one at run time, and/or to generate and incorporate new *RW* and *Memory* objects.

5 Verification

All specifications above (except the last) show passive objects not creating any calls to the environment. In the general case we consider objects both receiving and creating calls. Whereas the external invariant of such an object restricts the history of both kinds of events, the user requirement puts assumptions on the history of calls initiated by each external object.

Thus, a user requirement $R(\mathcal{H})$ of an object o is now understood as:

$$\forall p : Pid_o \bullet R(\mathcal{H}/p\rightarrow)$$

denoted $[R]_o(\mathcal{H})$, where Pid_o is the set of objects external to o, and $p\rightarrow$ denotes the set of events which p may initiate. (The index o is omitted when considering a fixed object.) The notation $\rightarrow p$ denotes the set of events which p may receive, $p \rightarrow p'$ denotes $p \rightarrow \cap \rightarrow p'$, and p used as a set denotes all observable events involving p, i.e. $(p\rightarrow \cup \rightarrow p) - (p\rightarrow p)$.

We assume here that a user requirement R is historically monotone, i.e. $R(\varepsilon)$ holds and $h' \leq h \wedge R(h) \Rightarrow R(h')$. An internal invariant is required to hold outside critical regions of operators. An external one need only be "weakly" monotone, in the sense that it is required to hold only for quiescent object states.

Semantically, an object o specified by the pair $(R(\mathcal{H}), I(\mathcal{H}))$, where $R(\mathcal{H})$ is the user requirement and $I(\mathcal{H})$ the invariant, has a trace set given by

$$\{h : Seq(o) \mid [R]_o(h) \Rightarrow I(h)\}$$

Thus, an invariant $I(\mathcal{H})$ of a class specification may be strengthened or weakened by the user requirement $R(\mathcal{H})$ as follows, $[R](\mathcal{H}) \wedge I(\mathcal{H})$ and $[R](\mathcal{H}) \Rightarrow I(\mathcal{H})$, respectively. All three versions of the class specification are semantically equal.

An object is said to *satisfy* a specification if its trace set (properly restricted) is a subset of that of the specification, ignoring traces where user requirements are broken. Similarly, a specification $(R1(\mathcal{H}), I1(\mathcal{H}))$, is *refined* by another specification, $(R2(\mathcal{H}), I2(\mathcal{H}))$, if its trace set is a superset of that of the latter (properly restricted), ignoring traces breaking $R1$; thus it suffices to prove $R1(h) \Rightarrow R2(\mathcal{A}(h))$ and $I2(\mathcal{A}(h)) \Rightarrow I1(h)$ where $\mathcal{A}$ is the abstraction function.

A class implementation is validated by proving the invariance of the external and internal invariants, when assuming the user requirement. I.e. for each operator implementation

$$op_p == G' \longrightarrow G \rightarrow C$$

we must prove the Hoare triple

$$\{[R](\mathcal{H}) \wedge I_e(\mathcal{H}) \wedge I_i(\mathcal{H}) \wedge G\}\ \mathcal{H} := \mathcal{H} \vdash op_p\, ;\ C\ \{[R](\mathcal{H}) \Rightarrow I_e(\mathcal{H}) \wedge I_i(\mathcal{H})\}$$

where I_e and I_i are the external and internal invariants, respectively. (Any calls in C on external objects should also give rise to updates of $\mathcal{H}$.) If C potentially consists of more than one critical region, a separate proof must be provided for each of them.

The guards G' in the implementations shown above are *adequate* in the sense that the user requirements are correctly checked; a corresponding error results if and only if they are violated. The sufficiency and necessity of each guard G' is demonstrated by proving:

$$[R](\mathcal{H}) \wedge I_e(\mathcal{H}) \wedge I_i(\mathcal{H}) \wedge G' \Rightarrow R(\mathcal{H} \vdash op_p/p\rightarrow)$$

respectively:

$$([R](\mathcal{H}) \wedge I_e(\mathcal{H}) \wedge I_i(\mathcal{H}) \wedge R(\mathcal{H} \vdash op_p/p\rightarrow)) \Rightarrow G'$$

All reasoning about fairness remains informal.

Parallel Composition

The (total) trace set of a composition is the set of traces h such that h projected to the event set of each component is a possible trace of that component. The observable trace set of the composition is the total trace set with all internal events ignored.

Assume that the object $A1$ satisfies the specification pair $(R1(\mathcal{H}), I1(\mathcal{H}))$ and that $A2$ satisfies $(R2(\mathcal{H}), I2(\mathcal{H}))$. The parallel composition of the two objects satisfies the specification

$$\begin{array}{ll} \textbf{req} & R1(\mathcal{H}/\rightarrow A1) \wedge R2(\mathcal{H}/\rightarrow A2) \\ \textbf{inv} & \exists h : \mathit{Seq}(A1 \cup A2) \bullet I1(h/A1) \wedge I2(h/A2) \wedge \mathcal{H} = h \backslash (A1 \cap A2) \end{array}$$

for $\mathcal{H} : \mathit{Seq}((A1 \cup A2) - (A1 \cap A2))$, where $h \backslash (A1 \cap A2)$ is h without internal events (calls between $A1$ and $A2$), provided the following two requirements hold:

$$\forall \mathcal{H} : \mathit{Seq}(A1) \bullet [R1]_{A1}(\mathcal{H}) \wedge I1(\mathcal{H}) \Rightarrow R2(\mathcal{H}/A1 \rightarrow A2) \qquad (1)$$

$$\forall \mathcal{H} : \mathit{Seq}(A2) \bullet [R2]_{A2}(\mathcal{H}) \wedge I2(\mathcal{H}) \Rightarrow R1(\mathcal{H}/A2 \rightarrow A1) \qquad (2)$$

(We assume here that the event sets of $A1$ and $A2$ are disjoint; this is obvious if identification of the called object is recorded in each event.)

Proof: From validity of $A1$ and $A2$ we have

$$\forall h : \mathit{Seq}(A1) \bullet (\forall p : \mathit{Pid}_{A1,A2} \bullet R1(h/p \rightarrow A1)) \wedge R1(h/A2 \rightarrow A1) \Rightarrow I1(h/A1)$$

$$\forall h : \mathit{Seq}(A2) \bullet (\forall p : \mathit{Pid}_{A1,A2} \bullet R2(h/p \rightarrow A2)) \wedge R2(h/A1 \rightarrow A2) \Rightarrow I2(h/A2)$$

where $\mathit{Pid}_{A1,A2}$ denotes the set of objects external to both $A1$ and $A2$. We prove

$$\forall h : \mathit{Seq}(A1 \cup A2) \bullet (\forall p : \mathit{Pid}_{A1,A2} \bullet R1(h/p \rightarrow A1) \wedge R2(h/p \rightarrow A2))$$

$$\Rightarrow I1(h/A1) \wedge I2(h/A2) \wedge R1(h/A2 \rightarrow A1) \wedge R2(h/A1 \rightarrow A2)$$

by induction on h. For an empty trace the proof is trivial, since no user requirement can be false on an empty trace (and by validity of $A1$ and $A2$).

Consider a trace $h \vdash x$. Assume the requirements to external objects hold, i.e. $\forall p : \mathit{Pid} \bullet R1(h \vdash x/p \rightarrow A1) \wedge R2(h \vdash x/p \rightarrow A2)$. By monotonicity of user requirements and the induction hypothesis, we have

$$I1(h/A1) \wedge I2(h/A2) \wedge R1(h/A2 \rightarrow A1) \wedge R2(h/A1 \rightarrow A2)$$

In the case of x involving an external object, the proof is trivial, as
$(h \vdash x/A2 \rightarrow A1) = (h/A2 \rightarrow A1)$ and $(h \vdash x/A1 \rightarrow A2) = (h/A1 \rightarrow A2)$.

For an internal event x, we have that either $(h \vdash x/A2 \rightarrow A1) = (h/A2 \rightarrow A1)$ or $(h \vdash x/A1 \rightarrow A2) = (h/A1 \rightarrow A2)$, say the former. Then $R1(h \vdash x/A2 \rightarrow A1)$ holds. By validity of $A1$ we have $I1(h \vdash x/A1)$, and then $R2(h \vdash x/A1 \rightarrow A2)$ holds by requirement (1) above. Finally, validity of $A2$ gives $I2(h \vdash x/A2)$. □

In the proof it was essential that invariants restrict the total interleaving of all relevant events, whereas user requirements restrict the sequence of calls initiated by each external object. Thus, one given event may only violate the user requirement of one object, and one given object may not initiate a call violating its own user requirement.

If it would be desirable that the specification of an object puts user assumptions on (say out-parameters of) calls initiated by itself, it would be sufficient to require that

$$R1(\mathcal{H}/A2 \rightarrow A1) \wedge R2(\mathcal{H}/A1 \rightarrow A2) \Rightarrow R1(\mathcal{H} \vdash x/A2 \rightarrow A1) \vee R2(\mathcal{H} \vdash x/A1 \rightarrow A2)$$

for any $\mathcal{H}$ and x consisting of events in $A1 \cup A2$, in addition to the requirements above.

Notice that historical monotonicity of invariants was not needed in the proof. Even though all invariants given here are monotonic, one could imagine non-monotonic invariants, letting an invariant restrict the states outside critical regions only. (For instance, a **seen as** clause might result in a non-monotonic invariant.)

In the above composition the identity of the two components is not hidden in the observable events. Hiding could be formalized by introducing an abstraction on the total history (h). However, if the operation sets for the two objects are not disjoint, operation renaming could be used to avoid naming ambiguities.

Acknowledgements

We are grateful for feedback from discussions with the members of the ADAPT.FT project group. In addition Anne Salvesen has provided detailed advice concerning the manuscript.

References

[1] L. Crnogorac, A.S. Rao and K. Ramamohanarao: "Inheritance Anomaly – A formal Treatment". In *Proceedings of FMOODS'97*, pages 319-334. Chapman & Hall, 1997.

[2] O.-J. Dahl, B. Myhrhaug, K. Nygaard: "Simula 67 Common Base Language", Norwegian Computing Center, 1968, revised 1970.

[3] O.-J. Dahl: "Can Program Proving be Made Practical?" In *Les Fondements de la Programmation*, M. Amirchahy and D. Néel, (Eds.), INRIA, 1977.

[4] O.-J. Dahl, O. Owe: "Formal Development with ABEL." In S. Prehn, W.J. Toetenel (Eds.): *VDM'91, Formal Software Development Methods, LNCS 552*, Springer 1991, pp. 320-362.

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

[6] O.-J. Dahl: "Monitors Revisited". In A.W. Roscoe (Ed.): *A Classical Mind, Essays in Honour of C.A.R. Hoare*, Prentice Hall, 1994, pages 93–104.

[7] ISO-IEC JTC1/SC21/WG7: "The Reference Model of Open Distributed Processing".
http://www-cs.open.ac.uk/~m_newton/odissey/RMODP.html

[8] ISO-IEC 8807: "LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour". Geneva, 1988.

[9] ISO-IEC 9074: "ESTELLE – A Formal Description Technique Based on an Extended State Transition Model". Geneva, 1989.

[10] ITU Recommendation Z.100-CCITT: Specification and Description Language (SDL). 1993.

[11] J. Meseguer: "Conditional Rewriting Logic as a Unified Model of Concurrency". *Theoretical Computer Science*, 96, pages 73-155, 1992.

[12] J.M. Spivey: *The Z notation: A reference manual*, Prentice Hall, 1989.