Enriching OCL using observational mu-calculus*

Julian Bradfield, Juliana Küster Filipe, and Perdita Stevens

Laboratory for Foundations of Computer Science
University of Edinburgh
JCMB, King's Buildings
Edinburgh EH9 3JZ
United Kingdom
{jcb, jkf, Perdita.Stevens}@dcs.ed.ac.uk

Abstract. The Object Constraint Language is a textual specification language which forms part of the Unified Modelling Language[8]. Its principal uses are specifying constraints such as well-formedness conditions (e.g. in the definition of UML itself) and specifying *contracts* between parts of a system being modelled in UML. Focussing on the latter, we propose a systematic way to extend OCL with temporal constructs in order to express richer contracts. Our approach is based on observational mu-calculus, a two-level temporal logic in which temporal features at the higher level interact cleanly with a domain specific logic at the lower level. Using OCL as the lower level logic, we achieve much improved expressiveness in a modular way. We present a unified view of invariants and pre/post conditions, and we show how the framework can be used to permit the specification of liveness properties.

1 Introduction

In *contract-based design*, the designer not only identifies the parts that the system should have but also specifies explicitly the contracts that those parts should obey. The contract for a part specifies what the developer of that part (who may be someone doing more detailed design, or someone programming) must ensure; it simultaneously specifies what clients of that part may assume. The use of contracts thus provides a process by which dependencies between parts of the system may be made explicit and managed. Commonly used examples of contracts are class invariants, and pre- and post-condition pairs. Contracts today are usually written in natural language or code.

One of the aims of the Object Constraint Language, OCL, is to provide the designer who is modelling a system in the Unified Modelling Language, UML with a language for expressing such contracts which is at the same time formal (and so, unambiguous and possibly open to verification and analysis) and easy to use. There are well-known problems with OCL1.x, and the language is currently undergoing a careful revision. In part, this paper is motivated by a

* Work reported here was supported by the EPSRC grant GR/R16891, and EPSRC Advanced Fellowships held by Bradfield and Stevens.

desire to disambiguate certain aspects of the OCL language and its use, such as when exactly a class invariant is required to hold. More ambitiously, we aim to provide an expressive framework to allow designers to write contracts which have a temporal character: that is, contracts which specify certain aspects of the ongoing behaviour of a part of a system under particular dynamic interaction conditions.

To achieve this we need to extend OCL. It is a challenge to do this in a clean, understandable way which is amenable both to practical use and, eventually, to verification. Temporal logics such as the modal mu-calculus are well-suited to describing dynamic properties such as deadlock, liveness, fairness, etc., but they *only* provide means of specifying such properties; they do not provide means for expressing static properties, such as those currently expressible in OCL. In practice we believe it will be essential to be able to write constraints which flexibly combine both aspects. Here are some simple examples of things we would like to express; later we will show how they are expressed in our proposed logic. Each example is in the context of an object o of class C:

1. after o receives message $m(p, r)$ it will eventually send message $n(r)$ to object p, unless p and r are the same object;
2. each time o receives message $t()$ it will return a positive integer which is larger than the one it returned in response to the previous instance of message $t()$.

In order to combine the dynamic power of the mu-calculus with the static expressiveness of OCL, we propose to use the two-level logic *observational mu-calculus*, with standard OCL as the instantiation of the lower level logic. The resulting logic, $\mathcal{O}\mu$(OCL), is extremely powerful, whilst being designed with verification in mind. Using it directly does, however, require an understanding of temporal logic with fixpoints which it would be unrealistic to expect most developers to be interested in acquiring. We suggest that it is useful to design "templates" of standard usage, with their own developer-friendly syntax, which are then translated into $\mathcal{O}\mu$(OCL). In this paper we define and translate one such template for specifying liveness constraints. Moreover, we show that the existing OCL contract-types, invariants and pre- and post-conditions, can also be regarded as such templates and translated into $\mathcal{O}\mu$(OCL).

This paper is structured as follows. The next section describes the state of the art in OCL extensions. Section 3 introduces an example illustrating some requirements which cannot currently be captured adequately by OCL, but which can be described using a new *after/eventually* template described informally in Section 4. Section 5 describes the logic, $\mathcal{O}\mu$(OCL), which forms the framework we use, and shows how it is an instantiation of a two-level logic $\mathcal{O}\mu$. Section 6 demonstrates how OCL templates are translated into $\mathcal{O}\mu$(OCL). The paper finishes with some conclusions and discussion on future work.

2 State of the art in OCL extensions

In this section we describe various proposals to extend OCL in order to allow it to express contracts involving dynamic constraints. There are also many proposals for non-OCL-based contract notations, but for reasons of space we do not discuss them here, given that we are specifically interested in capitalising on the existing acceptance of OCL.

[9] introduces a temporal extension of OCL in order to describe safety and liveness constraints of reactive and distributed systems. It adds some basic temporal logic operators to the grammar of OCL, such as `until` and `always`. However, there are problems: the main one is that OCL only allows query operations in expressions, whereas general operations are used in the paper, and needed by many of the desired properties. Another approach [6] extends OCL in order to express constraints involving changes of state, and introduces operators `initially` and `eventually`, applied to properties. It also changes the meaning of `@pre`; in standard OCL this is used only in postconditions, to refer to values before the invocation of the operation being specified (c.f. VDM's hook). In this approach it can be used in invariants as well, and means "the value of a property in every pre-state of an operation". Given that this means *any* operation, it is not clear how to interpret this so as to make `@pre` useful for anything but specifying that certain elements of an object's state never change. Both of these approaches are interesting, but in neither case are the new features defined precisely.

A dynamic extension in [7] adds so-called action clauses to OCL. The motivation is to be able to express dynamic constraints involving events, signals or the invocation of operations. An action clause can appear in an operation specification or with an invariant of a classifier. In the first case, it allows us to express that when an operation is executed it triggers the execution of other operations. In the second case, it specifies, for instance, that when certain conditions hold an object has to send events to given objects. Action clauses express some interesting dynamic properties, but their semantics is not entirely understood. The paper has influenced the OCL2.0 proposal [1].

Catalysis is a methodology for component-based software design which uses a textual constraint language similar to OCL but slightly more expressive [5]. For instance, it is possible to refer to operations in a postcondition. This means that we can specify that the effect of an operation is to invoke another operation. Furthermore, such an operation invocation can be either synchronous (the invoked operation is written within brackets `[[operation(...)]]`) or asynchronous (it is further prefixed with `sent`).

Finally, a different approach is taken in [4] which defines the logic BOTL and its transition system semantics. BOTL does not extend OCL by temporal operators itself. Rather what it does is translate a part of OCL v 1.1 into an object-based version of CTL. This means that temporal extensions of OCL in the sense of CTL could be translated into BOTL as well, but such extensions are not provided. The motivation for BOTL is model checking of existing OCL constraints. A strength of the work is that its concepts are clearly and precisely defined. In particular, it provides a semantic model which we shall reuse with minor modifications.

3 Example

In this section we will demonstrate various contracts which can, and cannot, be expressed using OCL. We use a variant of the (in)famous dining philosophers example. Consider the fragment of a class diagram in Figure 1.

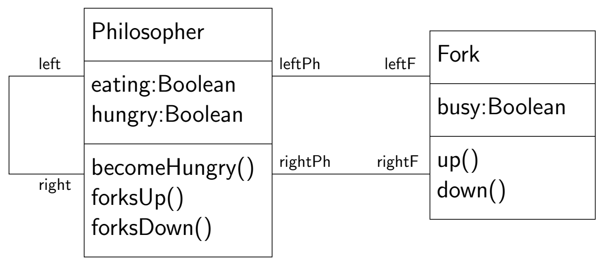

Fig. 1. Class diagram for dining philosophers.

Using operations in the interface, clients of class Philosopher (e.g. instances of other classes in the design, not shown in this fragment) can instruct a philosopher to become hungry, or to pick up or put down both forks. Notice that a philosopher may pick up either fork separately using the interface of class Fork, but it is not possible for an outside client to instruct the philosopher to do so. Associations between Philosopher and Fork link a philosopher with her right and left forks. The association between Philosopher and itself records the philosopher's place at the table; she will have someone on her left and on her right.

The UML class diagram given above does not give us a full description of our intended society of philosophers sharing forks around a table; there are many ways in which developers working in accord with this diagram could make decisions which we might regard as unfortunate. For example, there is nothing yet to forbid an implementation which deadlocks, or which starves certain philosophers, or even one which violates the familiar rules of the problem such as the arrangement of philosophers and forks. To reduce this scope for error, we can add further requirements as OCL constraints. We now give a few examples of OCL constraints, before concentrating on further requirements which we cannot currently express in OCL.

First, we write a class invariant on Philosopher recording that a philosopher's right and left neighbours are different from herself; this ensures that there are at least two Philosophers!

context `Philosopher` **inv**:
  `self.left <> self` **and** `self.right <> self`

Furthermore, adjacent philosophers share a fork:

context `Philosopher` **inv**:
  `self.leftF = self.left.rightF`

Therefore we want the implementation to prevent adjacent philosophers from being able to eat at the same time:

```
context Philosopher inv:
   self.eating=false or self.left.eating=false
```

The following OCL constraint describes the pre- and postconditions of the operation up() of class fork.

```
context Fork::up()
   pre: self.busy=false
   post: self.busy=true
```

This list of contracts is by no means exhaustive: we could continue adding OCL constraints to our model. However, many requirements are inexpressible in OCL. We will provide developer-friendly syntax for certain liveness conditions, including the following paradigmatic examples[1]:

(1) If a philosopher is hungry then she will eventually be eating.
(2) If a philosopher is instructed to pick up the forks (i.e. receives message forksUp()) she will eventually pick up her left fork (i.e. send message up() to the appropriate fork).

To make our new syntax precise, we will also introduce the logic $\mathcal{O}\mu$(OCL) in which a much larger class of temporal constraints can be expressed.

4 A new template

In this section we demonstrate the use of our new *after/eventually* (AE) template. It is a template in the sense that it is a standard framework in which to express a particular liveness property, parameterised on certain OCL expressions. The template and its OCL contents are then translated together into our logic $\mathcal{O}\mu$(OCL). Thus the developer using the template does not have to understand the $\mathcal{O}\mu$(OCL) logic; at the same time, the wholesale embedding of OCL into $\mathcal{O}\mu$(OCL) allows the translation to be very simple, and should help tools to provide understandable feedback based on verification of $\mathcal{O}\mu$(OCL) formulae. (A common problem with verification based on translations of user-friendly languages into less user-friendly underlying syntax is that it is difficult for a tool to give feedback in terms which are understandable to the user; by avoiding translating OCL itself we hope to minimise such problems.) If an expert developer requires the full flexibility of the language – or perhaps some intermediate sublanguage, such as a CTL-like syntax for a subset of $\mathcal{O}\mu$(OCL)– there is of course no obstacle to a tool providing it.

Thus, the template presented here is merely one example of a possible use of the extra temporal power provided by the use of $\mathcal{O}\mu$, which is explained in the next section; in Section 6 we give the translation.

[1] of *requirements*: meeting them may depend on more than one component's design

```
context Classifier:
    after: oclExpression
    eventually: oclExpression
```

Like an invariant or a pre/post-condition pair, an AE template is written in the context of a type, typically a classifier such as a class or a component from a UML model. As there, "self" may be used to refer to the instance of this type to which the contract is being applied.

The **after:** clause expresses some trigger for the contract: once the condition it expresses becomes true, the contract specifies a guarantee that the condition expressed in the **eventually:** clause will eventually become true. Notice that there is, in this particular example template, no intention that the **eventually:** clause should be required to become true immediately; the subtlety of this template is precisely that it is able to talk about consequences at a distance.

The oclExpressions must be either of OCL type Boolean, or may have special forms which extend OCL1.x, but which we expect to be standard in OCL2.0. The **eventually:** clause is permitted to contain an expression like self.left.leftF.up(), specifying that self sends message up() to object self.left.leftF. This is an OCL action expression, as described in the OCL2.0 proposal [1]. Similarly, the **after:** clause may simply name an operation of the Classifier in whose context the template is being used, possible also naming its arguments. For example, **after:** `forksUp()` is to be read as "after self *receives* the message forksUp()"; in UML terms it represents an event, not an action. We call such an expression an OCL event expression: [1] does not currently include these, but as they go naturally with action expressions we believe that the final proposal will do so. ([1] will certainly be revised before OCL2.0 is finalised, as there are still significant semantic ambiguities in the current draft; indeed, we are committed to contribute to that revision.) These considerations are purely syntactic: the AE template is translated as a whole into $\mathcal{O}\mu$(OCL), where the notions are clearly distinguished. In this paper we have chosen to keep our example template very simple: the rationale for allowing OCL event expressions in **after:** but not in **eventually:** and vice versa for OCL action expressions is the philosophy that the developer of a part cannot reasonably be expected to control the events which eventually happen to the part, only the actions eventually done by the part. Technically, however, any use of events, actions and boolean conditions can be translated.

We can now record the example contracts from Section 3. (1) becomes:

```
context Philosopher:
    after: self.hungry=true
    eventually: self.eating=true
```

In this case both clauses are Boolean constraints, already legal in OCL1.x. Notice that we have not specified any limit on the length of time a philosopher can be hungry and not eating, merely that it is not infinite. (2) becomes:

context `Philosopher:`
  after: `forksUp()`
  eventually: `self.leftF.up()`

Again, we are deliberately not stating that `self.leftF.up()` is an immediate reaction to the receipt of message `forksUp()`. The specifier may reasonably choose to use the AE template here in order not to tie the developer down too tightly, and to avoid the problems inherent in specifying what is meant by "immediate", even informally. (Is internal computation permitted between the trigger and its consequence? At least this must be allowed in practice. But what about the sending of a message to self? The sending of a message to another object? And so on.) One natural possibility in a synchronous system where the trigger is the receipt of a message is to specify that the consequence must have happened before the message is replied to; this too is easy to express in $\mathcal{O}\mu$(OCL).

Note the use of an OCL event expression in the **after:** clause and of an OCL action expression in the **eventually:** clause. We distinguish them here by context; if OCL2.0 follows our expectations and standardises both, it may choose a different syntax which of course we could adopt.

We have only specified that the left fork will be picked up. As there is no objection to several contracts applying to the same classifier, we could also apply the corresponding contract for the right fork; both would apply and both forks would be picked up. Alternatively, we could have permitted the contracts to be combined, getting

context `Philosopher:`
  after: `forksUp()`
  eventually: `self.leftF.up() and self.rightF.up()`

that is, combining two OCL action expressions with the OCL `and`. However, the implications of forming complex expressions using OCL action expressions have not been considered in the OCL2.0 proposal, and it may be considered undesirable to allow this at all. Because of this uncertainty, we choose not to permit such combinations for now; at the same time, the probability of wanting greater power of combination later is a reason for defining OCL event expressions rather than allowing events to be discussed only implicitly, as they are at present in pre/post conditions. (We could alternatively have defined a second template, perhaps *pre/eventually*, to be written in the context of a particular operation; this would be similar to what is done in [5] for what they call asynchronous action invocations in postconditions. Then we would not have needed to invent OCL event expressions to express (2). Pragmatically, we prefer the solution presented here, but either could equally well be translated to $\mathcal{O}\mu$(OCL).)

5 Observational mu-calculus

The modal mu-calculus has a long history of being used to describe properties of systems whose behaviour evolves over time. It is a very simple, concise language, which at the same time has sufficient expressiveness to capture a wide range of useful properties. For this reason it has often been used as the target of translations from more user-friendly languages; the specifier writes in some suitable syntax, which is then translated into mu-calculus and verified there. It is a logic defined on labelled transition systems, so that the semantics of a formula is a set of states of the system. As well as the boolean operators, the mu-calculus incorporates two features, modalities and fixpoints, each of which appears in two (dual) forms. The *modalities* provide the means of exploring a labelled transition system locally; they are $\langle a \rangle \Phi$ (from the current state, there is a transition labelled a which leads to a state where Φ holds) and $[a]\Phi$ (from the current state, any transition labelled a leads to a state where Φ holds). Adding these modalities gives Hennessy–Milner logic, which can express simple "finite" properties, but cannot say things about behaviour which continues for unboundedly many steps. The *fixpoints* ($\mu X.\Phi(X)$ and $\nu X.\Phi(X)$) add this power. For a detailed discussion of their meaning and power we refer the reader to references such as [3]; here it suffices to say that they allow the expression of safety properties (something bad is guaranteed not to happen), liveness properties (something good is guaranteed to happen) and indeed a wide class of fairness properties (e.g. provided this thing keeps happening, so will that thing).

In specifying contracts within UML models, these kinds of properties are among those that we need. We also, however, need smooth ways to combine the kinds of properties we can talk about in OCL with the kinds of properties we can talk about in mu-calculus. Notice that we do not simply want the union of these classes of properties: greater power is needed to express requirements such as that a service provider will always respond to a service request by making a request of a third party, where the parameters of that new request are related to those of the original request in some way which can be specified in OCL. The examples given in Section 1 are also of this kind. Semantically, what we say in OCL needs to be entwined with what we say in mu-calculus. However, from the point of view of the specifier, we need to maintain a clear separation.

5.1 The calculus $\mathcal{O}\mu$(OCL)

Observational mu-calculus ($\mathcal{O}\mu$) is an extension of the modal mu-calculus with some first-order features. It is a two level logic: intuitively, the upper level is the modal mu-calculus, and the lower level can be instantiated with any appropriate domain specific logic, such as OCL—thus we write $\mathcal{O}\mu$(OCL) for $\mathcal{O}\mu$ with OCL as its domain logic. In order to provide a clean separation of domain-dependent logical expressions from purely temporal properties, we confine domain-dependent expressions (for example, OCL expressions) to appear only inside modalities and as 'atomic' formulae of the mu-calculus. To provide expressive power, $\mathcal{O}\mu$ works on transition systems where the transitions are labelled with structured data, and the (domain-dependent) choice of what form the transitions have determines what properties of the system can be referred to in specifications. One can see this both as an instantiation of Milner's thesis that observation is fundamental, and as an instantiation of the thesis that systems should only be accessible through defined interfaces.

The link between the two levels is provided by 'cells'; these are first-order variables, in the sense of hybrid logics, but we call them 'cells' to distinguish them from the fixpoint variables. In the upper level, a cell x appears only implicitly, as a first-order variable passed through the fixpoint operators; in the lower level, a 'cell' is seen as a normal first-order variable of some appropriate model-dependent datatype – for example, an OCL variable.

An $\mathcal{O}\mu$ formula 'observes' a system by evaluating a box or diamond modality, which may import data into the cells. The existential modality has the form: $\langle l, C, \phi \rangle \Phi$, where: C specifies a set of *mutable* cells; l is an *action expression*, which is intuitively an expression that is pattern-matched against the structured transition label and thereby assigns values to the mutable cells mentioned in l; and ϕ is a constraint in the domain-specific logic. The constraint ϕ may refer to cells c, meaning their value after the transition is taken and matched against l, and also, in VDM style, to hooked cells $\overleftharpoon{c}$, meaning their value before the transition is taken (hooked cells are syntactic sugar, but useful). Intuitively, a state satisfies the formula if there is a matching transition satisfying the constraints after which Φ is true; the formal semantics is slightly richer.

The formal syntax of $\mathcal{O}\mu$ is:

$$\Phi = \psi \bigm| \mathbf{T} \bigm| \mathbf{F} \bigm| X \bigm| \Phi \vee \Phi \bigm| \Phi \wedge \Phi \bigm| \langle l, C, \phi \rangle \Phi \bigm| [l, C, \phi] \Phi \bigm| \nu X.\, \Phi \bigm| \mu X.\, \Phi$$

where ψ is a formula of the lower-level logic (not mentioning hooked cells).

Since the 'cells' are first-order variables which are implicitly passed through fixpoint variables, the meaning of a formula with no free fixpoint variables is a function from cell valuations to sets of states, rather than just a set of states; we write $A \models_{V,\rho} \Phi$ to mean that given a variable valuation V and cell valuation ρ, the state A satisfies Φ.

The formal semantics of the diamond modality is then: $A \models_{V,\rho} \langle l, C, \phi \rangle \Phi$ iff

- $\exists \rho'$ a new cell valuation differing from ρ only in the values of cells in C ($c \notin C \Rightarrow \rho'(c) = \rho(c)$), and
- $\exists a, A'$ such that $A \xrightarrow{a} A'$ and a matches l in context ρ' and $\rho, \rho' \models \phi$ and $A' \models_{V,\rho'} \Phi$

The box operator is dual, and the fixpoint operators have the usual meaning, but taken over states parametrised by cell valuations rather than over states. Formal details may be found in [2].

To define $\mathcal{O}\mu$ in the specific domain of UML models, we need (a) to define the logic used in constraints ϕ, (b) to define the transition system, (c) to define the action expressions and how they match transition labels.

The low level logic OCL The low level logic used in the constraints ϕ of the modalities is essentially OCL[2], with some re-interpretation to link it to the higher level logic, as follows.

The $\mathcal{O}\mu$ cells provide the link between the two levels, so OCL variables appear as cells of $\mathcal{O}\mu$(OCL). By convention we will always have an $\mathcal{O}\mu$(OCL) cell *self*, which gives the value of the special OCL keyword `self`, and thereby provides a starting point for navigation in the OCL expression. It is then possible, by allowing the value of *self* to change, to write properties whose context changes over time. Similarly, the OCL keyword `result` is tied to an $\mathcal{O}\mu$(OCL) cell *result*.

The reader familiar with [4] will wonder about the OCL `@pre` construct, which requires a quite complicated translation in that work. It is possible in $\mathcal{O}\mu$(OCL) to do the same translation, which essentially involves storing the precondition-time values of `@pre`'ed expressions in auxiliary variables and referring to these at postcondition time, but it is not necessary. Because our lower level logic *is* essentially OCL, and will be evaluated by an OCL-based model-checker, we can leave `@pre` alone, and extend its interpretation thus: an `@pre` expression in a constraint is evaluated at the origin state of the transition taken in the modality. Hence in particular when the modality action expression refers to a complete method invocation (defined below), `@pre` has its usual OCL meaning. This makes `@pre` the lower-level analogue of the $\mathcal{O}\mu$ hook, and so we shall write hook and `@pre` interchangeably.

For typographic reasons we shall use normal logical notations (e.g. $\wedge$) instead of always using OCL keywords (e.g. `and`).

The transition system The definition of a transition system amounts to a formal semantics for UML, which of course is a major research topic in the community. We will therefore adopt for this paper an abstract high-level semantics, a variation on that of [4]. The state of a system comprises at least the attribute and link values of all the objects in the system, together with the set of currently active method calls, and a unique identifier for each active method call. If the model is sufficiently defined, the state may also include other UML state information.

The transitions represent changes in state as the system evolves, and provide the interface to the specification formulae.

Message passing or method invocation is modelled by communication transitions which may represent a complete invocation, or just the sending of or reply to a message. Each message instance is assigned an identifier, and this is exported in the transition label, along with the source and target of the message. (The source and target are implicit in the state, but we can only observe what is exported in the transition labels.) Communication transitions may, then, be:

[2] A minor complication is that OCL is a three-valued logic. We insist that OCL expressions used in $\mathcal{O}\mu$(OCL) not evaluate to undefined: if they do so, the whole $\mathcal{O}\mu$(OCL) formula is meaningless. This is not too restrictive, especially since undefinedness does not always propagate in OCL.

– send(*selector*, *source*, *target*, *id*, *arguments*), representing the sending of a message, or
– return(*selector*, *source*, *target*, *id*, *return_value*), representing the reply to a message (where *source* means the sender of the original message)
– $val = \text{call}(selector, source, target, id, arguments)$ representing the invocation of a method returning *val*, seen as a single *complete transition.*

The complete transitions deserve further comment. In a fully detailed model (such as verification approaches normally assume) they are derived transitions: a complete transition is generated automatically for each possible sequence of transitions starting and ending with corresponding send and return transitions. However, in a verification approach suitable for tool-supported use during ongoing design, it would not be practical to assume that a fully detailed transition system can be generated. A tool might then work with complete transitions without ever representing the underlying transitions from which they would be derived. This might happen for two reasons. First, the user might not have specified the detailed information; the tool would work with any information the user had specified, for example, with pre/post conditions for the operation. (Note that nondeterminism is inherent in working with designs, and also that as the transition systems concerned are infinite or very large, any practical tool would have to be using some form of symbolic representation of the transition system anyway.) Second, the information might be present in the model but not required for the particular task; this is a form of "on the fly" transition system generation.

Finally, we have transitions labelled 'internal', representing some internal computation (including object creation); this may be refined to a more detailed label according to requirements.

It is important to note that we are *not* addressing here the issue of how the transition system is derived from the UML model, but taking it as given. The derivation of a transition system from a model depends on, among other things, the level of detail at which the model is given. Since designs are not concrete systems, and may well be infinite-state, the appropriate representation of transition systems will usually be symbolic; appropriate verification techniques include infinite-state symbolic model-checking, constraint-solving, theorem-proving or abstract games.

Action expressions Now we define the *action expressions.* In the following, O is a navigation expression, m is a selector, and $v, d, s, t, a_1, \ldots, a_n$ are expressions of appropriate types, perhaps including cells. The notation $\hat{a_1}$ (etc.) means the value of a_1 in the current cell valuation, i.e. the value to which a_1 evaluates when current cell values are substituted for cells wherever they occur.

– $O.m_d(a_1, \ldots, a_n)$ matches transitions $\text{send}(m, \hat{self}, \hat{O}, \hat{d}, \hat{a_1}, \ldots, \hat{a_n})$, and is the sending of message $m(a_1, \ldots, a_n)$ with identifier d by the current object to the object O.
– $m_{d,s}(a_1, \ldots, a_n)$ matches transitions $\text{send}(m, \hat{s}, \hat{self}, \hat{d}, \hat{a_1}, \ldots, \hat{a_n})$, and is the receipt of message $m(a_1, \ldots, a_n)$ by the current object from the object s.

- $\overleftarrow{m}_{d,t}(v)$ matches transitions return$(m, \hat{self}, \hat{t}, \hat{d}, \hat{v})$, and is the receipt of a reply to a message m.
- $O.\overleftarrow{m}_d(v)$ matches transitions return$(m, \hat{O}, \hat{self}, \hat{d}, \hat{v})$, and is the sending to O of a reply to a message m with return value v.
- $v = O.m_d(a_1, \ldots, a_n)$ matches $\hat{v} = \text{call}(m, \hat{self}, \hat{O}, \hat{d}, \hat{a_1}, \ldots, \hat{a_n})$, a complete transition from the viewpoint of the caller. (Note that v is in principle an expression; in practice, it will usually be just a mutable cell.)
- $v = m_{d,s}(a_1, \ldots, a_n)$ matches $\hat{v} = \text{call}(m, \hat{s}, \hat{self}, \hat{d}, \hat{a_1}, \ldots, \hat{a_n})$, a complete transition from the viewpoint of the callee.
- τ matches internal computation.

If subscripts or arguments of an action expression are omitted, they are wild cards, and always match; and we write $\star$ as a wild-card *selector*. Thus the action expression $\star$ matches any message receipt action, and $O.\star$ matches any message sent to O. In addition, we adopt the usual practice in modal mu-calculi of allowing sets of action expressions, so that $\{e_1, e_2\}$ matches anything matching e_1 or e_2, and we allow the use of $-$ to mean the complement of a set, so that the action expression $-e$ matches any transition *except* one matched by e, and in particular $-$ matches any action whatsoever. (This notation can be a little confusing: note that according to these rules, $-\star$ matches any action except a message receipt action.)

6 Applying observational mu-calculus

In this section we show how the liveness template we informally presented is expressed in $\mathcal{O}\mu$(OCL). As a further demonstration of expressiveness, we also show how to express the standard invariants and pre- and post-conditions for operations. Designers can continue to use the familiar syntax, whilst we translate them into the underlying observational mu-calculus representation. In the context of a UML verification tool, this is a foundation for a single coherent verification task.

We begin with translating invariants as this leads to slightly simpler $\mathcal{O}\mu$(OCL) formulae. An invariant in OCL has the form

context `TypeName` **inv**:
  `P`

where P is an OCL constraint, to be evaluated with reference to a particular instance of TypeName which may be referred to as "self" in P.

A recurring question is "exactly when is an invariant supposed to hold?" It is clear that the answer cannot be literally "always"; for example, in the case of a constraint that self should always be linked to exactly one Foo, this would make it impossible ever to replace one instance of Foo by another. A common compromise [10, 4] for use with single-threaded synchronous systems is "whenever no method of self is executing". Adopting this, we translate the above invariant into the $\mathcal{O}\mu$(OCL) formula:

$$\nu Z.\,(P \wedge [-\star, \varnothing, \mathbf{T}]Z)$$

which can be read as "P is true now and remains so unless we look inside a method call". The "whenever no method of self is executing" condition appears as the action expression $-\star$; the formula correctly asserts invariance because any method call on self is represented by a complete transition (as well as possibly by complex sequences of transitions). If such a formula is to be verified of a design (as opposed to asserted as part of a specification) the transition system obtained from the UML model will of course have to include detailed specification of the behaviour resulting from a message invocation; in this case, the complete transitions are added to the transition system to represent any sequence of transitions which starts with self receiving a message and ends with self replying to that message.

Given a transition system, the $\mathcal{O}\mu$(OCL) formula is unambiguous. Notice also that although we have used the "whenever no method of self is executing" interpretation of invariant here, any reasonable variant can also be encoded in observational mu-calculus.

Next we consider operation pre- and post-conditions.

context `TypeName::m(par1:Type1,...,parn:Typen):ReturnType`
  pre: `P`
  post:`Q`

Parameters can be used in `P` and `Q`, and `result` in `Q`. This is expressed as

$$\nu Z.\,([\mathit{result} = m(\mathit{par1}, \ldots, \mathit{parn}), C_1, \neg\overleftarrow{P} \vee \neg Q]\mathbf{F} \wedge [-, \varnothing, \mathbf{T}]Z)$$

where C_1 is $\{\mathit{par1}, \ldots, \mathit{parn}, \mathit{result}\}$. This can be read as 'it is always impossible to do an m action such that either P fails before it or Q fails after it'. Note the constraint $\neg\overleftarrow{P} \vee \neg Q$: this is because an OCL precondition is a guard as well as a Hoare precondition. Note also that we have assumed that pre-/postconditions come in pairs: OCL in fact allows pre- and postconditions to be stated independently, for which the natural meaning is to conjoin all the stated preconditions and conjoin all the stated postconditions, and then interpret as above.

The concise expression of a pre-/postcondition property depends on the presence of the complete 'call' transitions. However, we should note that if such transitions are not included in the system, it is still possible to express the property, albeit with a considerably more complex formula.

The translation of our new AE template differs according to the type of clause. For the version where both the after and eventually clauses have boolean expressions, we have:

context `Classifier:`
  after: P
  eventually: Q

becomes

$$\nu Y.\,([-, \varnothing, P \wedge \neg \overleftarrow{P}]\mu Z.\,(Q \vee (\langle -, \varnothing, \mathbf{T}\rangle \mathbf{T} \wedge [-, \varnothing, \mathbf{T}]Z)) \wedge [-, \varnothing, \mathbf{T}]Y)$$

In the version with action and event, we have:

context `Classifier`:

after: e

eventually: a

becomes

$$\nu Y.\,([e, C, \mathbf{T}](\mu Z.\,[-a, \varnothing, \mathbf{T}]Z) \wedge [-, \varnothing, \mathbf{T}]Y)$$

where C includes a cell for any argument to the OCL event expression e. As it happens our examples did not include arguments: but, for example, if e is request(r) then C becomes $\{r\}$, so that our matching rules ensure that the actual parameter of the invocation of result is saved, in a cell called r, across any transition matching e. Thus a is able to mention r; for example, it might be self.collaborator.otherRequest(r+1).

We omit the obvious analogues for the mixed cases.

Finally, we show how to express in $\mathcal{O}\mu$(OCL) some of the properties not expressible in either OCL or our new template: the property 'after o receives message $m(p, r)$ it will eventually send message $n(r)$ to object p, unless p and r are the same object' is expressed as

$$\nu Y.\,([m(p, r), \{p, r\}, p \neq r]\mu Z.\,[-p.n(r), \varnothing, \mathbf{T}]Z) \wedge [-]Y$$

and the property 'each time o receives message $t()$ it will return a positive integer which is larger than the one it returned in response to the previous instance of message $t()$' is expressed as

$$\nu Z.\,[x = t(), \{x\}, x \leq \overleftarrow{x}]\mathbf{F} \wedge [x = t(), \{x\}, \mathbf{T}]Z \wedge [-(y = t()), \{y\}, \mathbf{T}]Z$$

where $x = 0$ in the initial cell valuation.

7 Conclusion and Future Work

In this paper we have presented logical foundations for adding temporal expressiveness to OCL, together with an example of how these might be used in practice to support richer contracts. Of course, the *after/eventually* template discussed here only represents one small application of the logic; it is not intended to be exhaustive but rather to illustrate that the technique can be used in a way which will be understandable to developers without special logical knowledge. There is a vast range of possible templates which could be developed and translated into $\mathcal{O}\mu$(OCL); experimentation will probably be required to determine which are useful in practice. For example, we might consider

- after/immediately: as mentioned in Section 4, we might wish to define a template which specifies self's immediate reaction to a certain stimulus, but will have to specify carefully what is meant by "immediately".

– provided/infinitely often: we might wish to define a template which specifies that provided some condition continues to hold (or perhaps, to hold infinitely often), some other condition will be true infinitely often. Such conditions are used in specifying *fairness*: it is often unreasonable to expect a part to continue to work no matter what, because it legitimately relies on correct functionality of some other part, but one wishes to specify that under fair conditions, i.e. when the relied-upon parts do work (often enough) the part will work correctly.

We have not discussed verification issues extensively in this paper but in fact the observational mu-calculus was designed with verification in mind. Considerable further work is needed, but in brief, we intend to extend the Dresden OCL kit to support the extra template described here, and to add a translator module to produce $\mathcal{O}\mu$ formulae from these templates. Alongside this we may extract a CCS process representing a labelled transition system from a UML model saved as XMI. Both the formula and the CCS process could then be fed into the Edinburgh Concurrency Workbench, which would use its existing abstraction techniques to check whether or not the process satisfies the formula. Because $\mathcal{O}\mu$ incorporates OCL as its lower-level logic, the translation can be kept quite simple which should enable us to give meaningful feedback to the user on the basis of this verification.

References

1. Boldsoft, Rational Software Corporation, and IONA. Response to the UML2.0 OCL RfP. http://www.klasse.nl/ocl/subm-draft-text.html, August 2001.
2. J.C. Bradfield and P. Stevens. Observational mu-calculus. In *Proceedings of FICS'98*, 1998. An extended version is available as BRICS-RS-99-5.
3. J.C. Bradfield and C.P. Stirling. Modal logics and mu-calculi: an introduction. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier, 2001.
4. D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 305–326. Kluwer, 2000.
5. D. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Object Technology Series. Addison-Wesley, 1999.
6. A. Hamie, R. Mitchell, and J. Howse. Time-based constraints in the Object Constraint Language. Technical Report CMS-00-01, University of Brighton, 2000.
7. A. Kleppe and J. Warmer. Extending OCL to include actions. In S. Kent and A. Evans, editors, *UML'2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.
8. OMG. *Unified Modeling Language Specification version 1.4 draft*, February 2001. OMG document 01-02-14 available from www.omg.org.
9. S. Ramakrishnan and J. McGregor. Extending OCL to support temporal operators. In A. Ulrich, editor, *ICSE'99 Workshop on Testing Distributed Component-based Systems*, Los Angeles, California, USA, May 1999.
10. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Object Technology Series. Addison-Wesley, 1999.