TIME-BASED CONSTRAINTS IN THE OBJECT CONSTRAINT LANGUAGE OCL

Ali Hamie, John Howse
School of Computing, Mathematical and Information Sciences,
University of Brighton, Brighton, UK.
{a.a.hamie@brighton.ac.uk, John.Howse@brighton.ac.uk}

Richard Mitchell
InferData Corporation, Austin TX 78759, US
{Richard.Mitchell@inferData.com}

ABSTRACT

The Object Constraint Language OCL is a textual specification language that supplements UML class diagrams for describing and expressing various constraints that can not be modelled by the diagrams. These constraints include invariants on classes and object types, preconditions and postconditions of operations. This paper describes an approach for extending OCL with time-based constraints in such a way so as not to compromise its simplicity. A time-based constraint describes how values can change between earlier and later states. The approach is essentially based on using @pre and oclIsNew in invariants as well as in postconditions of operations. In order to distinguish between invariants and time-based constraints we introduce the stereotype <<temporal constraint>>. We also introduce an operator eventually for expressing liveness constraints on attributes. We illustrate the approach by describing constraints such as constant attributes of an object, constant associations, and values increasing or decreasing over time.

KEY WORDS

OCL, UML, Invariants, Time-based Constraints

1. Introduction

The Unified Modelling Language UML [11] is widely accepted as the standard for object-oriented analysis and design. In particular UML class diagrams are popular for representing and modelling the static structure of object-oriented systems. However, not all structural details can be easily expressed in a class diagram. The Object Constraint Language OCL [10][12] which is part of UML, is a precise textual language for describing additional constraints on object-oriented models. OCL is similar to first order predicate logic and its constraints are boolean expressions that can be combined by logical connectives. Universal and existential quantifications are also supported. OCL constraints are used to describe invariants on classes and types, and preconditions and postconditions of operations.

There is a class of constraints which cannot be elegantly described in OCL. These constraints describe how values of attributes or associations change from earlier to later states, such as an operation's pre-state and its post-state. These constraints include expressing that an attribute or an association is constant, i.e their values don't change between earlier and later states. Other constraints are those which describe values increasing or decreasing from earlier to later states. We shall call such constraints time-based constraints.

There is already some work that extends OCL with temporal logic based on different approaches. In [1], OCL is extended with temporal constructs based on the observational mu-calculus. This results in logic that directly requires the developer to have a deep understanding of the mathematical background, so the authors suggest using "templates" with user-friendly syntax which then have to be translated to the logic. However, tool builders are still required to understand the logic. In [13], OCL is extended with some elements of linear temporal logic, where past and future operators are introduced. In [3], the object-based temporal logic is defined, which is based on the branching temporal logic CTL and a subset of OCL.

In this paper we describe an approach for incorporating time-based constraints within OCL which increases its expressive power without compromising its simplicity. Currently, OCL uses @pre in postconditions to refer to the value of a property immediately before the execution of an operation. The approach is based on using @pre in invariants as well as in postconditions. However, within an invariant, @pre indicates the value of a property in every

pre-state of an operation. This has the same effect as including the invariant in the postcondition of every operation. To distinguish invariants from time-based constraints, a stereotype <<temporal constraint>> is introduced to classify such constraints within object-oriented models. We illustrate the approach by describing constraints such as constant attributes and associations, and attributes with values increasing or decreasing over time. We also introduce an operator eventually to describe constraints which must hold at some time in the future. A similar approach is used in the behavioral interface specification language (JML) [6][7] for specifying classes and interfaces in Java, where time-based constraints are referred to as history constraints [8].

The paper is organised as follows. Section 2 is an informal introduction to using OCL, in object-oriented modelling, in particular with UML class diagrams. Section 3 introduces time-based constraints within OCL. Section 4 concludes with a summary and further work.

2. The OCL

The OCL is a specification language for describing constraints on object-oriented models. It is based on textual rather than symbolic syntax which makes it more accessible for specifying constraints on object-oriented models than other specification languages such as Z [9] and VDM [5]. The design of OCL is heavily influenced by the work of Cook and Daniels [2] which borrows heavily from Z. The constraints which are expressible using OCL are as follows:

- Invariants on Classes or Types that must hold at all times, i.e in every stable state.
- Preconditions which are constraints that must hold before the execution of an operation.
- Postconditions which are constraints that will hold after the execution of an operation under the appropriate precondition.
- Guards which are constraints on the transitions of an object from one state to another.

The constraints are described in the context of an object-oriented model represented as a UML class diagram, that is they cannot be stand alone constraints. The following example illustrates the use of OCL in describing some of these constraints, and the time-based constraints we are introducing.

2.1 Example Model

As an example, we use the class diagram in Figure 1 of a simple system for the scheduling of offerings of seminars to a collection of attendees by presenters who must be qualified for the seminar they present. The system has a

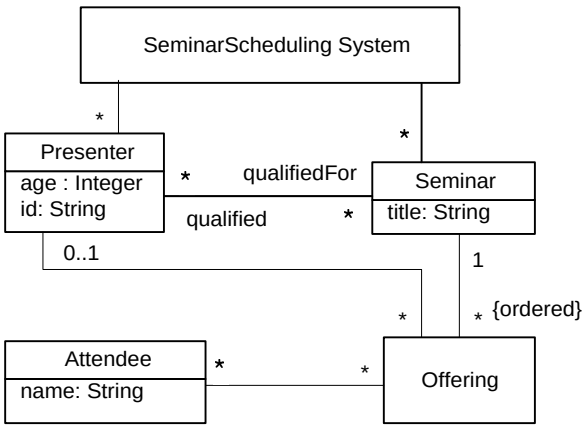

Figure 1.A class diagram for a seminar scheduling system

set of presenters who are qualified to present offerings of seminars, and each seminar offer might have a set of attendees associated with it. The model is expressed in UML [11] which consists of a set of notations for describing object-oriented models. A full description of UML can be found in [11] and a distilled description can be found in [4].

The language of classes, attributes and associations introduced by the class diagram automatically becomes part of OCL. That is the classes in the diagram (Presenter, Seminar, etc.) are automatically OCL types, and the attributes and association roles (age, id, qualified, etc.) are also part of the OCL language.

2.1.1 Invariants

The constraints which can be expressed using OCL are always connected to a UML object-oriented model. An *invariant* is a constraint which can be associated with a class, type or interface in a UML model. The invariant is expressed as a boolean expression which restricts or limits the value of an attribute or association role, or it can state a relationship between the values of attributes and association roles. The result of the expression must be true for all instances of the associated class in every stable state.

A simple invariant which may be appropriate on the class Presenter is that the age of each presenter must be greater than 18 years. This is expressed in OCL as follows:

```
context Presenter inv:
self.age > 18
```

The keyword context specifies the context of the invariant which in this case is the class Presenter. That is the object self belongs to the class Presenter (self can be omitted).

Another more complex constraint on the model is that a presenter who is assigned to present an offering must be qualified for the offering's seminar. In OCL, this can be expressed as follows:

```
context Offering inv:
self.presenter->notEmpty()  implies
  self.presenter.qualifiedFor->includes(self.seminar)
```

where not and implies are boolean operators for negation and implication respectively. includes is the set membership predicate. The expression self.presenter denotes the presenter associated with the offering self. In OCL, self.presenter can be used as a set or as a single element. In the first part of the invariant self.presenter is used as a set indicated by the "->". While in the second part is used as single element. That is the operator "_.presenter" is overloaded. "->" is also used to indicate operations on sets rather than individual objects. This is useful is the case of optional associations where it is necessary to check whether an object is associated with another before asserting anything.

2.1.2 Operation Specifications

The operation assign assigns a presenter to a seminar's offering. The precondition says that the presenter must be qualified for the seminar and that the offering is for that seminar. The postcondition says that the set of offering for the presenter is the old set augmented with the offering. The following is a specification of the operation assign in terms of a precondition and postcondition pair:

```
context Presenter:: assign(p : Presenter, o : Offering)
  pre: p.qualifiedFor->includes(o.seminar)
  post: p.offering = p.offering@pre->including(o)
```

In the postcondition @pre is used to refer to the value of an attribute or association at the precondition time, that is the value in the pre-state of an operation.

OCL is also used as a navigation language, where from a single or a collection of objects we can navigate associations to find all the associated objects. For example, self.presenter.qualifiedFor is a navigation expression denoting the set of seminars for which presenter self is qualified to present. Indeed, constraints are formed by using logical operators together with navigation expressions.

3. Time-Based Constraints

Invariants on classes or types in OCL express constraints that must hold at any point in time. That is in every observable state, each object of a class must satisfy the constraint. These constraints involve only one system state, where a state represents all the existing objects, their attribute values, and their associations at a point in time. Consequently, these kind of constraints can be statically checked. For example the above invariant says that in every state, the age of any presenter in the model must be greater than 18. However, the value of the attribute age may change from an earlier state to later state while keeping the constraint satisfiable.

Preconditions on operations are also predicates (boolean expressions) involving only one state. Postconditions are predicates which involve two system states, namely, the state before and after an operation is executed. A postcondition is only required to hold after the execution of an operation. There are other kinds of constraints which may involve more than one system state. Such constraints which we shall call "time-based" say how values can change from earlier to later states. Such constraints are useful when one needs to express constant properties or associations. This form of constraints often involves the values of properties in the previous and current states. For example, we might want to impose a constraint which says that the id of a presenter is unchanged over time, i.e it remains constant. In its present version, OCL could be used to express these kind of constraints at the expense of concise, short and readable specifications. This could be achieved by including such constraints in the postcondition of every operation on a class or type.

3.1 Constant Attributes

One way to express that an attribute is constant over time is to include @pre in invariants. For example, to express that the value of id does not change over time is by making the following assertion part of any postcondition of any operation in the model:

```
Presenter.allInstances->forAll(p:Presenter | p.id = p.id@pre)
```

where Presenter.allInstances denotes the set of all existing instances of type Presenter. It is easy to see that when there many of such constraints, this approach becomes impractical. Indeed one reason for having invariants is to avoid lengthy operation specifications by factoring the common constraints.

What is needed in OCL is an elegant way of incorporating time-based constraints which are used to say how values can change between earlier and later states, such as an operation's pre-state and its post-state, in such a way to preserve the simplicity of OCL. Our approach is based on factoring out such constraints by including identifiers such

as id@pre in invariants and introducing a stereotype <<temporal constraint>> to distinguish an invariant from a time-based constraint. Thus asserting that the id of a presenter does not change over time could be expressed as follows:

context Presenter temporal constraint:

self.id = self.id@pre

The meaning of this constraint is that the value of id cannot change, since in every pre-state and post-state (before and after the invocation of an operation), its value in the post-state, written id, must equal its value in the pre-state, written id@pre. In this case the pre-state and post-state are independent of any operation, and denote any two states in the system.

The above constraint applies only to objects that exists in the current and previous states. For example if self is created in the postcondition of an operation, then self.id@pre would be undefined. If undefined means unknown then there is no problem because we could always choose a value for self.id@pre equal to the value of self.id. If on the other hand undefined means non-denoting then we must deal with it properly since equating a value with undefined may lead to inconsistencies. However explicit checks can be incorporated to ensure that the object exists in the previous state as the following constraint shows:

context Presenter temporal constraint:

Presenter.allInstances->

forAll(p|Prsenter.allInstances@pre->includes(p)

implies p.id = p.id@pre)

that is if p is not newly created in the current state of the system then its id in the current state must be equal to its id in the previous state. This constraint could be expressed concisely by using the OCL operation oclIsNew as follows:

context Presenter temporal constraint:

not (self.oclIsNew()) implies self.id = self.id@pre

self.ocIsNew() is interpreted as saying that self is not newly created in the current state.

In general we could specify which operations are to establish the constraint by using the following syntax:

context Class temporal constraint:

OCLconstraint [op_1, ..., op_n]

where OCLconstraint is a boolean expression describing the time-based constraint, and op_1, ...op_n are operations. The semantics is that the constraint is established by the operations listed op_1, ...op_n. If the list of operations is empty, then by default the constraint is to be established by all operations.

Constant attributes can be specified in UML using the keyword *frozen*. However, our approach is more expressive since we can specify which operations establish a given constraint.

3.2 Constant Associations

The association between Seminar and Offering states that each offering is associated with exactly one seminar. A suitable constraint would be that an offering is associated with the same seminar until it is destroyed. That is the offering's seminar is constant. This can be expressed as follows:

context Offering temporal constraint:

not(self.oclIsNew()) implies

self.seminar = self.seminar@pre

which says that in every pre-state and post-state the values self.seminar and self.seminar@pre are the same.

Again, this can be specified in UML by placing the keyword *frozen* at the end of the association line. As in the case of attributes, our approach can specify which operations establish a given constraint for an association.

3.3 Values increasing or decreasing over time

Other useful time-based constraints are those that constrain values to increase or decrease over time. Let us say we want to have an attribute that allows objects to keep count (a concrete example would be to keep a retry count of how many times a computer has tried to dial up to an ISP). What type should the attribute be?. We could choose Integer, but this would allow negative values. Another choice would the Natural, but the value can go up by two or seventeen. An appropriate and more suitable choice is to have a type CountValue which has the constraint that operations can either increase the value by one or reset to zero. This can be expressed as follows:

context Type1 temporal constraint:

(self.countAtt.value = self.countAtt.value@pre + 1) or

(self.countAtt.value = 0)

where the attribute countAtt has the type CountValue.
In a similar way we express constraints where values are decreasing over time.

3.4 Liveness constraints

Liveness constraints can also be expressed by adding suitable operators. These constraints includes asserting that the value of a property will reach a certain value in the future or the value of a property at some point in the future will be the same in every pre-sate and post-state. These constraints can be expressed in OCL by the operator eventually. For example, the constraint that the age property of a presenter will be 50 in the future (we

don't allow our presenters to die young!) could be expressed as follows:

```
context Presenter temporal constraint:
eventually(self.age = 50)
```

The constraint that the number of presenters and the number of seminars will be the same sometimes in the future can be expressed as follows:

```
context SeminarSchedulingSystem  temporal constraint:
eventually(seminars->size() = self.presenters->size())
```

An alternative way to express liveness constraints on UML models is to introduce a new stereotype <<eventually>> which can be used to indicate an eventual constraint as:

```
context Presenter eventually:
self.age = 50
```

4. Conclusion

In this paper we presented an approach for describing time-based constraints on UML models using OCL. Time-based constraints are those constraints which say how values can change from earlier to later states such as an operation's pre-state and its post-state. This approach is based on using existing constructs within OCL namely by using property names postfixed with @pre in invariants as well as in postconditions. This was illustrated by expressing constant attributes and associations as well as attribute values increasing or decreasing over time. This provides a natural way of extending OCL without compromising its simplicity which is essential for its usability as a specification language by software engineers and modellers. We also extended the language by adding a new operator for describing liveness constraints which may hold at some point in the future.

In order to check the expressibility of this approach one needs to consider a more realistic case study which involves real-time constraints. The semantics of constraints will involve more than one state, that is state sequences are needed to evaluate time-based constraints. This will be left for future work.

References

[1] J. Bradfield, J. Filipe, and P. Stevens, Enriching OCL using observational mu-calculus. In Ralf-Detlef Kutsche and Herbert Weber, editors, Fundamental Approaches to Software Engineering, 5th International Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, 2002 proceedings, volume 230 of LNCS, pages 203-217, Springer, 2002.

[2] S. Cook, and J. Daniels, Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, 1994.

[3] D. Distefano, J. 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 Acadeim Publishers, 2000.

[4] M. Fowler, and K. Scott, UML Distilled. Addison-Wesley, 2003.

[5] B. C. Jones, Systematic Software Development using VDM. Prentice Hall 1990.

[6] G. Leavens, and A. Baker, and C. Ruby, JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, chapter 12, pages 175-188. Copyright Kluwer, 1999.

[7] G. Leavens, A. Baker, and C. Ruby, Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06t, 2003.

[8] B. Liskov, and J. Wing, A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16 (6):1811-1841, 1994.

[9] J. Spivey, The Z Notation: A Reference Manual. International Series in Computer Science. Prentice-Hall, New York, N. Y., second edition, 1992.

[10] Rational Software Corporation. The Object Constraint Language Specification Version 1.5. Available from http://www.rational.com, 2003.

[11] Rational Software Corporation. The Unified Modeling Language Version 1.5. Available from http://www.rational.com, 2003.

[12] J. Warmer, and A. Kleppe, The Object Constraint language: Precise Modelling with UML. Addison-Wesley, 1999.

[13] P. Zieman, M. Gogolla, An Extension of OCL with Temporal Logic. In Jan Jurjens, editor, Proceedings of UML'2002 Critical System Development (CSD 2002). technical Report, Technical University of Munich, 2002.