Time-Based Constraints in the Object Constraint Language

Ali Hamie, Richard Mitchell, John Howse
School of Computing and Mathematical Sciences,
University of Brighton, Lewes Rd., Brighton, UK.
http://www.biro.brighton.ac.uk/biro
e-mail: a.a.hamie@brighton.ac.uk

Abstract. The Object Constraint Language (OCL) is a specification language which could be used for describing constraints on object-oriented models and other modelling artifacts. OCL is a part of the Unified Modelling Language (UML) which is the *de facto* standard for object-oriented analysis and design. OCL is designed to be used by software engineers and modellers and requires only modest mathematical training. This is achieved by keeping it simple and by employing a textual syntax rather than mathematical symbols. The kind of constraints which can be described using OCL include invariants on classes, preconditions and postconditions of operations. OCL uses @pre to refer to the value of a property immediately before the execution of an operation. However, OCL does not support the description of time-based constraints which say how values can change between earlier and later states.

This paper describes an approach for incorporating time-based constraints within OCL in such a way so as not to compromise its simplicity. This approach is essentially based on using @pre in invariants as well as in postconditions of operations. To distinguish between invariants and time-based constraints we introduce the stereotype <<constraint>>. We also introduce an operator eventually for expressing liveness constraints, and an operator initially for describing initial constraints. We illustrate the approach by describing constraints such as constant attributes of an object, constant associations, and values increasing or decreasing over time.

Keywords: The OCL, UML, invariants, constraints, precondition, postcondition

1 Introduction

The Object Constraint Language (OCL) [8][13][15][16] is a precise textual language for describing constraints on object-oriented models. It is an optional part of the object-oriented analysis and design *de facto* standard UML [11] and complements diagrammatic notations in modelling object-oriented systems. OCL enables the description of constraints which can't be described using standard diagrammatic notation. These include the description of invariants, preconditions, and postconditions allowing the modeller to specify precise and detailed constraints on the behaviour of a model, without getting embroiled in implementation details. Kent [7] has developed a diagrammatic notation based on Venn diagrams for describing constraints which could be used in conjunction with UML/OCL.

OCL is the culmination of recent work in OO modelling [1][2] which has selected ideas from formal methods to integrate with diagrammatic, object-oriented modelling notation resulting in a more precise, robust and expressive notation. Syntropy [1] extended OMT [11] with a Z-like textual language for adding invariants to class diagrams and annotating transitions on state diagrams with preconditions and

postconditions. However, a preliminary study by Finney [3] indicates that graphic mathematical notations, such as those found in Z [12] and VDM [6] may make such specifications hard to read, even for programmers trained in the notation. In this respect OCL is, arguably, even more accessible to the general software engineer, adopting a simple non-symbolic syntax and restricting itself to a small set of core of concepts.

There are 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.

This paper describes 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 we introduce a stereotype <<constraint>> 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 two operators initially and eventually to describe constraints which must hold initially when creating objects, and constraints which must hold at some time in the future. A similar approach is used in the behavioral interface specification language (JML) [9] for specifying classes and interfaces in Java, where time-based constraints are referred to as history constraints [10].

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 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a specification language for describing constraints on object-oriented models. It is developed at IBM and it is part of the object-oriented *de facto* standard UML. 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 and VDM. The design of OCL is heavily influenced by the work of Cook and Daniels [1] which borrows heavily from Z [12]. The constraints which are expressible using OCL are as follows:

- Invariants on Classes or Types that must hold at all times.
- 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, that is they cannot be stand alone constraints. The following example illustrates the use of OCL in describing some of these constraints.

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 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 the UML which consists of a set of notations for describing object-oriented models. A full description of UML can be found in [14] and a distilled description can be found in [4].

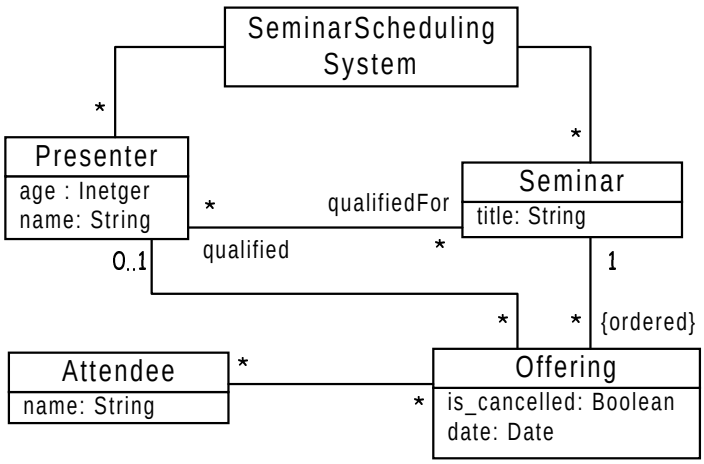

Figure 1: A class diagram for a seminar scheduling system

The language of classes, attributes and associations introduced by the class diagram automatically becomes part of OCL. That is the object types in the class diagram (Presenter, Seminar, etc.) are automatically OCL types, and the attributes and association roles (name, 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 at any point in time.

A simple invariant which may be appropriate on the class Presenter is that the age of each presenter must be greater than 18 years. Such an invariant can be shown in the class model as a text between curly brackets in a note box with a dotted line to the associated class as shown in Figure 2. The standard UML stereotype <<invariant>>

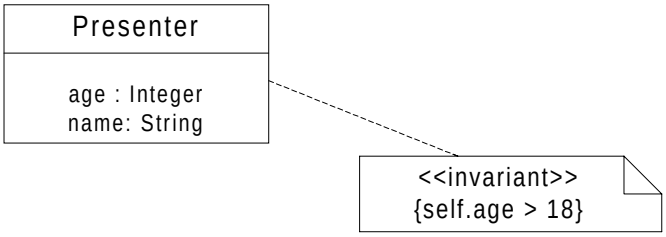

Figure 2: **Invariant shown as a note**

can be used to indicate an invariant constraint. However, in some cases invariants tend to take up too much space resulting in a cluttered diagram. This is the case when there are a large number of invariants. This can be overcome by writing invariants separately in a text document. For example, the invariant on the class Presenter can be expressed as follows:

context Presenter **invariant:**

self.age > 18

The keyword **context** specifies the context of the invariant which is the class Presenter in this case. That is the object self belongs to the class Presenter (self can be omitted). An alternative way to express this constraint is to use explicit declarations of variables.

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 **invariant:**

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 o.presenter denotes the presenter associated with the offering o. In OCL, o.presenter can be used as a set or as a single element. In the first part of the invariant o.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 indi-

vidual 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, s : Seminar, o : Offering)

pre: o.seminar = s and p.qualifiedFor->includes(s)

post: p.offering = p.offering@pre->union(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, o.presenter.qualifiedFor is a navigation expression denoting the set of seminars for which presenter o 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 invariant given in Figure 2 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 name of a presenter is unchanged over time, i.e it remains constant. In Syntropy [1] and Catalysis [2] constant attributes and associations are expressed by using the keyword const to annotating the end of the association line. However, constant constraints are only a subset of time-based constraints. 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 name 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.name = p.name@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 name@pre in invariants and introducing a stereotype <<**constraint**>> to distinguish an invariant from a time-based constraint. Thus asserting that the name of a presenter does not change over time could be expressed as shown in Figure 3.

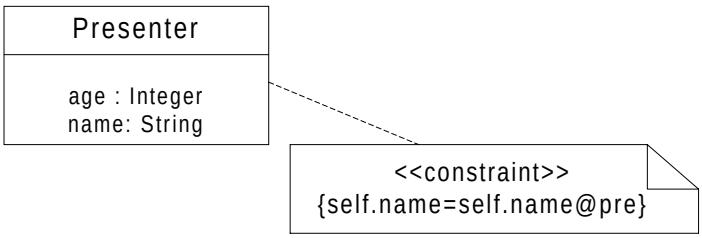

Figure 3: **Time-based constraint shown as a note**

This constraint can be written separately as follows

context Presenter **constraint:**

self.name = self.name@pre

The meaning of this constraint is that the value of name 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 name, must equal its value in the pre-state, written name@pre. In this case the pre-state and post-state are independent of any operation, and denote any two states in the system.

There is one problem with the above constraint. For instance if self is created in the postcondition of an operation, then self.name@pre would be undefined. If undefined means unknown then there is no problem because we could always choose a

value for self.name@pre equal to the value of self.name. 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. This can be easily solved by incorporating checks on whether self exists or not:

context Presenter **constraint:**

Presenter.allInstances->forAll(p | Presenter.allInstances@pre->includes(p)
implies p.name = p.name@pre)

that is if p is not newly created in the current state of the system then its name in the current state must be equal to its name in the previous state. This constraint could be expressed concisely by using a predicate new where p.new is true if p is newly created and false otherwise [5]. Such operator has been incorporated in the latest version of OCL [16] . Hence, we have:

context Presenter **constraint:**

not(self.new) implies self.name = self.name@pre

3.2 Constant association roles

The association between Seminar and Offering states that each offering is associated with exactly one seminar. If it is required to assert that the seminar an offering is associated with cannot change throughout its lifetime, one can use the following constraint:

context Offering **constraint:**

not(self.new) 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.

3.3 Values increasing or deceasing over time

Other useful time-based constraints such as values increasing or decreasing over time can also be expressed in a similar way. For instance, we could insist on the date property of an offering to increase over time. This can be described as the following time-based constraint

context Offering **constraint:**

Offering.allInstances->forAll(o | Offering.allInstances@pre->includes(o)
implies o.date >= o.date@pre)

Using the predicate new we have the following concise constraint:

context Offering **constraint:**

not(self.new) implies self.date >= self.date@pre

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 adding a suitable operator namely 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 **constraint:**

eventually(self.age = 50)

The constraint that the value of a property f in class A will be the same as the value of f in the pre-sate at some point in the future could be expressed as follows:

context A **constraint:**

eventually(self.f = self.f@pre)

An alternative way to express liveness constraints on UML models is to introduce a new stereotype <<eventual>> which can be used to indicate an eventual constraint. This is illustrated in Figure 4.

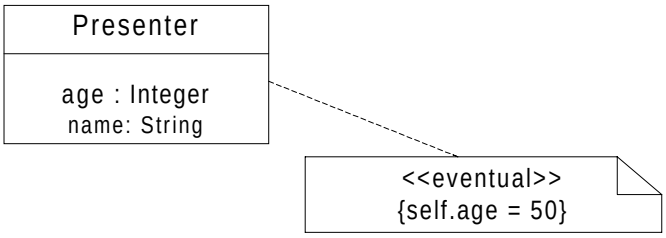

Figure 4: **Eventual constraint shown as a note**

3.5 Initial constraints

Initial constraints are those constraints which required to hold initially, for example, when an object is newly created. For this we could introduce an operator initially to describe initial values taken by properties of objects when they are created. For example, to say that the initial value of the attribute is_cancelled is false, we use the following:

context Offering **constraint:**

initially(self.is_cancelled = false)

This is useful if you have collection objects such as stack objects, where initially the stack is empty.

An alternative way to express initial constraints on UML models is to introduce a new stereotype <<initial>> which can be used to indicate an initial constraint as shown in Figure 5.

Another approach for incorporating time-based constraints in OCL is to represent time explicitly by having a type Time for representing time points, and by extend-

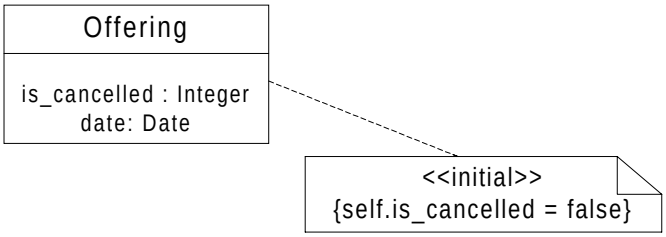

Figure 5: **Initial constraint shown as a note**

ing the @ operator. That is we use @t, where t is of type Time, to represent the value of a property at time t. This approach is similar to encoding temporal logic in first-order predicate logic. To describe the constraint that the name of the presenter is constant, we use:

context Presenter **constraint:**

self.name@t1 = self.name@t2

where t1 and t2 are any time points. This approach will be investigated further and will be compared with the approach presented here.

4 Summary and Further Work

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. This provides a natural way of extending OCL without compromising its simplicity which is essential for its usability as a constraint language by software engineers and modellers. We also added two new operators for describing initial constraints and for describing liveness constraints which may hold at some point in the future.

Other features which could be added to UML/OCL include describing frame conditions which state which attributes are allowed to be modified by an operation. In addition OCL could be extended by constructs for specifying exceptions

Acknowledgment

We are grateful to the BIRO research team at Brighton, in particular Stuart Kent and Franco Civello for many useful comments and feedback. This research was partly funded by the UK EPSRC under grant GR/K67304

References

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

2. D'Souza, D. and Wills, A. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1998. Draft and other related material available at http://www.trireme.com/catalysis.

3. Finney, K. Mathematical notation in formal specification: Too difficult for the masses? IEEE Transactions on Software Engineering, 22(2):158-159, February 1996.

4. Fowler, M. with Scott, K. UML Distilled. Addison-Wesley, 1997.

5. Hamie, A., Civello, F., Howse, J., Kent, S., Mitchell, R., Reflections on The Object Constraint Language. In the proceedings of the first international conference <<UML>>'98, LNCS, Springer, to appear 1999.

6. Jones B. C. Systematic Software Development using VDM. Prentice Hall 1990.

7. S. Kent, "Constraint Diagrams: Visualising Invariants in Object-Oriented Models",*Proc. of OOPSLA97*, ACM Press, 1997.

8. Kleppe, A., Warmer, J., and Cook, S., Informal Formality? The Object Constraint Language and its application in the metamodel. In the International Proceedings of <<UML>>'98 workshop, Mulhouse, France, June, 1998.

9. Leavens G., Baker A., and Ruby C., Preliminary Design of JML: A Behavioral Interface Specification Language for Java. TR98-06, June 1998.

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

11. Rumbaugh, J., Blaha, M., Premerali, W., Eddy, F. and Lorensen, W. Object-Oriented Modelling and Design. Prentice Hall, 1991.

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

13. Rational Software Corporation. The Object Constraint Language Specification Version 1.1. Available from http://www.rational.com, 1997.

14. Rational Software Corporation. The Unified Modeling Language Version 1.1. Available from http://www.rational.com, 1997.

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

16. Warmer, J., and Kleppe, A., OCL : The Constraint Language of the UML. JOOP, May, 1999.