Expressing Property Specification Patterns with OCL

Stephan Flake and Wolfgang Mueller
C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany
email: {flake,wolfgang}@c-lab.de

Abstract

The textual Object Constraint Language (OCL) is an official part of the Unified Modeling Language (UML). OCL is primarily used to formulate restrictions over UML models, in particular, invariants and operation pre- and post-conditions in the context of class diagrams. However, OCL is missing means to specify constraints over the dynamic behavior of a UML model. We have therefore developed a temporal extension of OCL that enables modelers to specify behavioral state-oriented constraints. That work provides an alternative to the rather cryptic temporal logic formulae that are commonly used to specify behavioral system properties.

This article now illustrates that our OCL extension allows for specifying all kinds of properties that are regarded as relevant in practice. We present according temporal OCL expressions for property specification patterns that have been identified in the area of formal specification.

Keywords: UML, Object Constraint Language, Patterns, Property Specification

1. Introduction

The Object Constraint Language (OCL) is part of the Unified Modeling Language (UML) since UML version 1.3 [10]. OCL is an expression language that enables modelers to formulate constraints in the context of a given UML model. It is used to specify invariants attached to classes, pre- and postconditions of operations, and guards for state transitions [14]. We can here only give an outline of the concepts of OCL and refer to the latest OCL 2.0 language definition proposal for more details [13].

OCL is a declarative language, not a programming language, i.e., evaluation of OCL expressions does not have side effects on the corresponding UML model. To integrate constraints into the visual modeling approach of UML, invariants, pre- and postconditions are modeled as comments and are attached to the respective graphical model elements in class diagrams. However, OCL constraints can become quite complex, such that they are often specified separately. The contextual class or operation is then explicitly provided in a preceding context clause.

Each OCL expression has a type. Besides user-defined model types (e.g., classes or interfaces) and some predefined basic types (e.g., Integer, Real, or Boolean), OCL also has a notion of object collection types (i.e., sets, sequences, and bags). Several useful operations are predefined to access and select objects from such object collections.

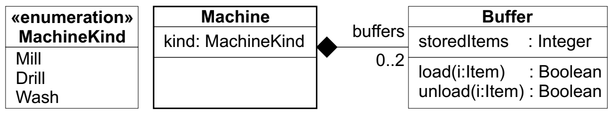

Figure 1. Parts of a UML Class Diagram

For example, consider the parts of a UML class diagram shown in Figure 1. Note that classes `Machine` and `Buffer` are related by an association with one association end called `buffers`. The following OCL invariant specifies that each object of class `Machine` that is of kind `Wash` must have exactly two associated buffers:

```
context Machine inv:
    self.kind = MachineKind::Wash
        implies self.buffers->size() = 2
```

We briefly explain how to read that OCL invariant. The class name that follows the keyword `'context'` specifies the class for which the following expression should hold. The keyword `'inv'` indicates that this is an invariant specification, i.e., for each object of the context class the following expression must evaluate to true at all times. But note that it is possible that an invariant is violated during execution of an operation. More precisely, an invariant therefore has to hold only for an object when none of its operations is currently executed. However, it is not formally defined in the official UML 1.5 specification when invariants are exactly to be checked, but it is commonly agreed that each time after the object's status has changed, its invariants have to hold (e.g., immediately after completion of an operation).

The (optional) keyword `self` refers to the object for which the expression is evaluated. Attributes, operations, and associations can be accessed by dot notation, e.g., `self.buffers` results in a (possibly empty) set of objects of class `Buffer` that are associated with the `Machine` object for which the constraint is currently evaluated. The arrow notation indicates that a collection of objects is manipulated by one of the pre-defined OCL collection operations. For example, operation `size()` applied to a collection returns the number of elements in that collection.

Though UML has received increasing attention to model software systems in recent years, it is missing sufficient means to specify constraints over the *dynamic behavior* of a UML model. However, it is essential to be able to formulate such temporal constraints already in early phases of development in order to specify correct system behavior, in particular in the domain of time-dependent systems. While other research approaches focus on UML Collaboration and Sequence Diagrams and consider temporal OCL constraints for event communication (e.g., [2, 9]), we here investigate consecutiveness of states and state transitions in UML Statecharts.

This article is based upon previous works concerning a temporal OCL extension that enables modelers to specify state-oriented real-time constraints [5, 7]. A formal semantics was defined by a mapping to time-annotated temporal logic formulae, but note that only a limited subset of the complete formal logic is obtained by that mapping [6]. In this article, we now show that the chosen approach is powerful enough to express system properties that frequently appear in practical systems development. We take the property specification patterns identified by Dwyer et al. [3, 4] and demonstrate that it is possible to formulate according temporal OCL expressions in each case. Note that beyond those general patterns, our OCL extension also covers explicit timing aspects, i.e., additional timing intervals can be attached to temporal OCL expressions to further delimit pattern scopes.

In the next section, we briefly present the pattern system of Dwyer et al. In Section 3, we give the mapping of property specification patterns to temporal OCL expressions. Due to space limitations, we can only provide an informal semantics description of the temporal OCL expressions.

2. Property Specification Patterns

Experiences in the domain of formal specification have shown that the full power of temporal logics, which allow for arbitrarily nested formulae, is not needed in practice. In this context, Dwyer et al. have developed a pattern system based upon more than 500 property specifications from different projects in the area of finite-state verification [3, 4]. That pattern system provides a structured set of commonly occurring property specifications and examples of how to translate these into different formal specification languages, such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL), or Graphical Interval Logic (GIL). These formulae can be directly applied in different verification tools, e.g., the model checking tools SPIN (accepts LTL) or SMV (accepts CTL).[1] As such verification tools require rather cryptic temporal logic formulae as one part of the input for the verification task, the pattern approach aims to support developers in a way that abstracts from the formal syntax of temporal logics.

2.1. Scopes

Dwyer et al. have identified different *scopes* applicable to a pattern. A scope is the part of the system execution path over which a pattern has to hold. Five basic kinds of scopes have been identified, as illustrated in Figure 2:

- Globally (i.e., the entire execution path),
- Before `R` (i.e., execution up to a state `R`),
- After `Q` (i.e., execution after a state `Q`),
- Between `Q` and `R` (i.e., all parts of the execution path from state `Q` to another state `R`), and
- After `Q` until `R` (i.e., all parts of the execution path from state `Q` to another state `R`, including those parts where `R` never occurs).

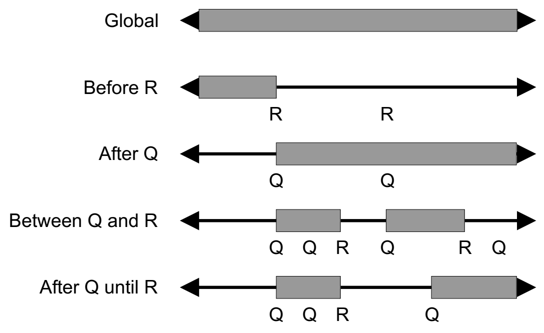

Figure 2. Specification Scopes of [3]

For state-delimited scopes with distinct delimiters `Q` and `R`, the interval in which the property is evaluated is closed at the left and open at the right end. Thus, the scope consists of all states beginning with the starting state and up to – but not including – the ending state. It is possible, however, to define scopes that are open-left and closed-right as well.

[1]For an introduction to temporal logics and an overview of verification tools, see, e.g., [1].

Note that most scopes may appear repetitively or with an unlimited future, as illustrated in Figure 2. These scopes are therefore embedded in *invariants*. Scope '`Before R`' is not an invariant, as we only investigate executions *up to the first occurrence* of `R` in this case. Patterns with that scope are only applied to paths starting at the initial state.

2.2. Patterns

The patterns themselves are hierarchically ordered as shown in Figure 3. In an online repository, for each pattern, each scope, and each formalism an according formal description is provided [3]. To illustrate the approach, we take the absence pattern as an example. Table 1 shows according CTL formulae for each scope in the context of the absence pattern.

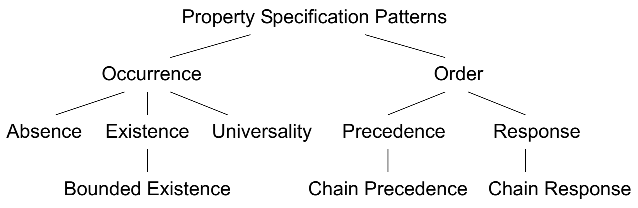

Figure 3. Property Specification Patterns

The absence pattern describes a part of an execution path that is free of a certain state `P`. We take a closer look at the pattern '`P is false before R`' in Table 1. A first intuitive attempt to specify an according CTL formula for that case would be `A(!P W R)`. This formula makes use of the *weak until* operator `W` and means that along all possible execution paths `P` is not entered from the initial state until the first state in which `R` is true, *if any*. In particular, if `R` is never entered along an execution path, then `P` must also not be entered along that path.

Table 1. CTL Formulae for Absence Pattern [3]

P is false . . .	
. . . globally	`AG(!P)`
. . . before `R`	`A((!P | AG(!R)) W R)`
. . . after `Q`	`AG(Q -> AG(!P))`
. . . between `Q` and `R`	`AG((Q & !R) -> A((!P |AG(!R)) W R))`
. . . after `Q` until `R`	`AG((Q & !R) -> A(!P W R))`

Dwyer et al. always consider the case that scope delimiters `Q` and `R` might not appear on execution paths. Now consider the case that on every execution path `P` becomes eventually true, but `R` will afterwards never be entered. In this case, the property '`P is false before R`' should also be true. However, our first formula `A(!P W R)` developed above does not cover this case and results in false in this case. One solution to resolve this issue is to add the sub-formula `(P & AG(!R))` as an alternative to sub-formula `!P`, resulting in

```
A( (!P | (P & AG(!R))) W R) .
```

As it holds $\neg a \vee (a \wedge b) \equiv \neg a \vee b$, we can simplify the latter formula to its final version as it appears in Table 1:

```
A((!P | AG(!R)) W R) .
```

As demonstrated, it always takes additional effort to include the case that scope delimiters `Q` and `R` might not appear at all on execution paths. Such assumptions unnecessarily complicate the resulting formulae. Instead, we propose a slightly different approach with inherent assumptions *requiring* that scope delimiters will eventually appear on all paths. Only if an assumption of such kind holds, a pattern can then be applied. Otherwise, a statement about validity cannot be given. By this approach, it is guaranteed that all possible executions really comply to the intended scope. Users of the pattern system need therefore pay less attention to whether delimiter states `Q` and `R` occur or not.

Moreover, mappings to respective temporal logic formulae, e.g., CTL, are significantly simplified and easier to adapt for further usage. As an example, Table 2 on the next page shows the absence pattern with additional assumptions and an according simplified mapping to CTL formulae. Assumptions can also be easily mapped to CTL and have to be checked separately. When an assumption is false over a given model, the actual property that is investigated cannot be validated.

While the patterns provided in the pattern system by Dwyer et al. already cover a broad range of requirements, it might still be necessary to adjust them for particular and more complex properties. There are a number of ways how this can be performed, e.g., by parameterization, logical combination, and changes in pattern scopes [3]. But note that users of the pattern system are usually not able to modify the temporal logic formulae without a concise understanding of the underlying semantics of the formal logics.

3. State-Oriented Temporal OCL Expressions

In this section, we demonstrate how to express patterns of the pattern system presented in Section 2 by means of our temporal state-oriented OCL extension. We here take the absence pattern as an example and provide according temporal OCL expressions in Table 3. To understand the OCL expressions in that table, we informally explain their semantics in the remainder.

Table 2. CTL Formulae for Absence Pattern with Additional Assumptions

Assumption	Pattern	CTL Formula
	`P` is globally false	`AG(!P)`
`R` becomes true on all paths [CTL: `AF(R)`]	`P` is false before `R`	`A(!P U R)`
`Q` becomes true on all paths [CTL: `AF(Q)`]	`P` is false after `Q`	`AG(Q -> AG(!P))`
`Q` and `R` always again become true on all paths [CTL: `AG AF(Q) & AG AF(R)`]	`P` is false between `Q` and `R`	`AG((Q & !R) -> A(!P U R))`
`Q` always again becomes true on all paths [CTL: `AG AF(Q)`]	`P` is false after `Q` until `R`	`AG((Q & !R) -> A(!P W R))`

Table 3. OCL Expressions for Absence Pattern (Assumptions implicitly as in Table 2)

`P` is false . . .	
. . . globally	`inv: not self.oclInConf(P)`
. . . before `R`	`init: self@post()->forAll(g | g->startsWith( Sequence{not P, R} ))`
. . . after `Q`	`inv: self.oclInConf(Q) implies self@post()->forAll(g | g->excludes(P))`
. . . between `Q` and `R`	`inv: self.oclInConf(Q) implies self@post()->forAll(g | g->startsWith( Sequence{not P, R} ))`
. . . after `Q` until `R`	`inv: self.oclInConf(Q) implies not self@post()->exists(g | g->startsWith( Sequence{not R, P} ))`

UML Statecharts are for modeling the reactive object behavior of objects. Basically, they are an object-oriented version of Harel Statecharts [8]. In a UML Statechart, the term 'current state' cannot be applied without causing confusion, as it can have composite (i.e., nested and orthogonal) states and thus may reside in more than one state at the same time. We better speak of a *state configuration*, i.e., the set of states that uniquely determines the currently active states in the Statechart. Consequently, in contrast to the original patterns, `P`, `Q`, and `R` denote configurations in the remainder. Nevertheless, note that in the simplest case a configuration consists of one state. As configurations uniquely determine the current state-related status of an object, conditions of form '`P and not Q`' are equal to the simple formula '`P`', as two distinct configurations of a Statechart can by definition never occur at the same time.

The following concepts and operations have been newly introduced to OCL to be able to express the specification patterns. Note that we keep compliant with the existing standard OCL syntax and reuse as often as possible existing collection operations like `forAll()`, `exists()`, `includes()`, and `excludes()`.

1. The only state-related operation of the current OCL standard as well as the new OCL 2.0 proposal is called `oclInState(s:OclState)`. It is defined over objects of user-defined classes that have an associated Statechart. Operation `oclInState(s:OclState)` returns true if state `s` is currently active.

   Additionally, we define and make use of operation `oclInConf(c:Set(OclState))` for Statechart configurations. This operation returns true if the object is in configuration `c` at time of evaluation.

2. In addition to OCL invariants declared by the keyword `inv`, we introduce a new clause called `init`. In contrast to an invariant over an object `obj` that has to hold each time after `obj`'s status has changed, the expression of an `init`-clause has to hold only at the starting point of execution. Nevertheless, note that the expression of the `init`-clause may be a temporal OCL expression.

3. Temporal OCL expressions are a new concept introduced to enable specification of dynamic, behavioral constraints. In our approach, temporal OCL expressions make use of a special *temporal operation* with signature `post(a:Integer,b:OclAny)`. To further emphasize that this is a temporal operation, we make use of a leading separator `@` instead of the common dot-notation. The operation can be applied to objects of user-defined classes that have an associated Statechart.

   When `@post(a,b)` is evaluated at a certain point of time `t`, we obtain the set of possible configuration sequences in the timing interval `[t+a,t+b]`. If parameters `a` and `b` are omitted, we set `a = 1` and `b = 'inf'` (short for infinity to cover infinite future executions).

4. We have defined an extended syntax for explicit specification of configuration sequences. This syntax is particularly tailored to the needs for formulating general execution paths that may be subject to some additional conditions. Basically, we allow that logical unary and binary operators such as 'not', 'and', 'or' are applied to sequence elements [6]. For the real-time domain, we also allow explicit timing intervals in this context, but note that these are not required for the general patterns we investigate in this article. E.g., the sequence specification

   ```
   Sequence{ not P [1,100], P [1,'inf'], Q }
   ```

   means that configuration P must not be true until within 100 time units configuration P is reached, and afterwards configuration Q must eventually become true. When the timing interval for one of the first $n - 1$ sequence elements is left out, it is implicitly set to [1,'inf'], but note that consecutive configuration specifications must still eventually become true (so-called *strong until* semantics).

5. We newly introduce the boolean operation startsWith(g:Sequence(T)), which can be applied to sequences of objects of some type T. That operation checks whether a given sequence starts with a sequence specified by parameter g.

   In particular, when T is equal to type Set(OclState) and the elements of T denote state configurations, we can make use of operation startsWith() to formulate restrictions over Statechart execution paths, using the syntax for configuration sequences as illustrated under item 4.

   Using operation startsWith() is similar to selecting a subsequence with the standard OCL operation subSequence(a:Integer,b:Integer) and then matching the extracted subsequence with g. But unfortunately, we cannot a priori provide an upper bound b from our particular viewpoint of possibly infinite execution runs, such that we cannot make use of existing OCL operations.

For the sake of completeness, Tables 4, 5, 6, and 7 at the end of this article provide according temporal OCL expressions for the other main property specification patterns.

Application Example. We consider the class diagram of Figure 1 and assume that class Machine has an associated Statechart modeling a process loop by means of consecutively running through configurations Idle, Loading, Working, and Unloading. In the respective Statechart in Figure 4, we have (informally) annotated the actions to load, transform, and unload an item by timing intervals for the estimated minimal and maximal times needed to execute these actions.

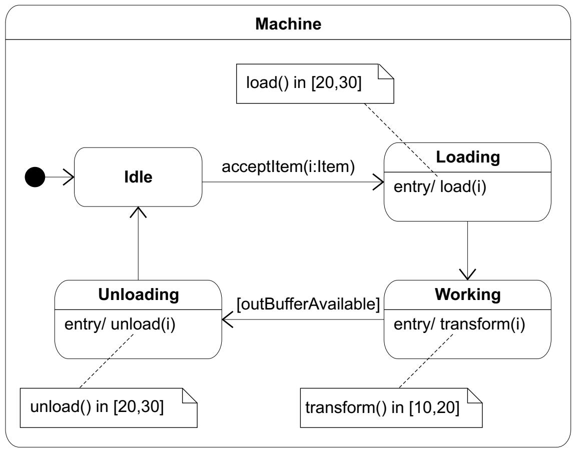

Figure 4. Machine Statechart

In this scenario, we require that 100 time units after loading of an item has started, the machine must have completely processed the item and changed to state Idle again. An according temporal OCL invariant is

```
context Machine
inv: self.oclInConf(Loading) implies
       self@post(1,100)->forAll( g |
         g->includes(Sequence{Working,Unloading,Idle}))
```

This kind of property is heavily related to what is called chained precedence and response in the original pattern system. But note that we additionally can make use of time limitations.

Expressing Assumptions with OCL. To capture the additional assumptions we make concerning the occurrence of scope delimiters, different approaches are imaginable. One idea uses only standard OCL language concepts. The expression

```
if <assumption> then <pattern>
                else OclUndefined
                endif
```

makes use of the three-valued logic of OCL that includes OclUndefined as the third logical value. For example, the complete OCL invariant for pattern 'P is false after Q' is defined as follows.

```
inv: if @post()->forAll(g | g->includes(Q)) then
       self.oclInConf(Q) implies
         self@post()->forAll(g | g->excludes(P))
     else
       OclUndefined
     endif
```

Note that OCL has a three-valued logic, i.e., OCL type Boolean actually comprises the values `true`, `false`, and `OclUndefined`. In the expression above, `OclUndefined` is returned when the if-condition does not hold. Unfortunately, such an expression cannot directly be mapped to a temporal logic like CTL or LTL due to a missing third logical value.

Another idea is to extend OCL and introduce a dedicated new clause, e.g., named `assume`, to express an assumption in the same manner as a precondition of an operation. For instance, the assumption '`R becomes true on all paths`' can then be expressed by

```
assume: @post()->forAll(g | g->includes(R)) .
```

Similarly, it has already been suggested by other authors to introduce means to formulate *exceptions* with OCL, such that undesired situations can be specified [12] and dealt with [11]. We can make use of such an approach to specify an according exception for each assumption simply by negating the assumption expression. When the exception evaluates to true, the assumption does not hold, and the respective pattern cannot be validated.

The advantage of this approach is that such assumption and exception expressions can directly be mapped to temporal logic formulae for further usage in verification tools.

4. Conclusion

We considered the property specification patterns by Dwyer et al. and found that some implicit assumptions about scope delimiters make it unnecessarily hard to map the patterns to a specific formal language like, e.g., CTL. We therefore separated implicit assumptions about scope delimiters from the actual patterns.

We then investigated whether the modified patterns can be expressed over UML Statechart configurations by means of OCL. It turned out that standard OCL concepts are not sufficient for that task. We therefore introduced some new operations and additional clauses to make OCL powerful enough to formulate all patterns accordingly.

We think that this work can help to increase the acceptance of formal specification in the domain of object-oriented modeling with UML. Developers familiar with the UML standard should easily understand our OCL extension, as – in contrast to other approaches – it keeps compliant with current OCL syntax and language concepts.

Acknowledgements

The work described in this article receives funding by the DFG project GRASP within the DFG Priority Programme 1064 'Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen'.

References

[1] B. Berard et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[2] M. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, volume 2391 of *LNCS*, pages 389–408. Springer, July 2002.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. A System of Specification Patterns, September 1998. URL: http://-www.cis.ksu.edu/santos/spec-patterns.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *21st International Conference on Software Engineering (ICSE99), Los Angeles, CA, USA*, May 1999.

[5] S. Flake and W. Mueller. A UML Profile for Real-Time Constraints with the OCL. In *UML 2002 – The Unified Modeling Language. Model Engineering, Concepts, and Tools. Dresden, Germany*, volume 2460 of *LNCS*, pages 179–195. Springer, 2002.

[6] S. Flake and W. Mueller. An OCL Extension for Real-Time Constraints. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*, pages 150–171. Springer, February 2002.

[7] S. Flake and W. Mueller. Specification of Real-Time Properties for UML Models. In *Proc. of the 35th Hawaii International Conference on System Sciences (HICSS-35)*, Hawaii, USA, January 2002. IEEE Computer Society.

[8] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[9] A. Kleppe and J. Warmer. Extending OCL to include Actions. In *UML 2000 – The Unified Modeling Language. Advancing the Standard. York, UK*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

[10] OMG. Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003. URL: http://www.omg.org/technology/documents/formal/uml.htm.

[11] S. Sendall and A. Strohmeier. Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML. In *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Toronto, Canada*, volume 2185 of *LNCS*, pages 391–405. Springer, October 2001.

[12] N. Soundarajan and S. Fridella. Modeling Exceptional Behavior. In *UML'99 – The Unified Modeling Language. Beyond the Standard. Fort Collins, CO, USA*, volume 1723 of *LNCS*, pages 691–705. Springer, 1999.

[13] J. Warmer et al. Response to the UML2.0 OCL RfP, Version 1.6 (Submitters: Boldsoft, Rational, IONA, Adaptive Ltd., et al.). OMG Document ad/03-01-07, January 2003.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

Table 4. OCL Expressions for Existence Pattern (Assumptions as in Table 2)

P becomes true . . .	
. . . globally	`init: self@post()->forAll(g | g->includes(P))`
. . . before R	`init: self@post()->forAll(g | g->startsWith( Sequence{not R, P} ))`
. . . after Q	`inv: self.oclInConf(Q) implies self@post()->forAll(g | g->includes(P))`

. . . between Q and R	`inv: self.oclInConf(Q) implies`
	`self@post()->forAll(g | g->startsWith( Sequence{not R, P} ))`

. . . after Q until R	`inv: oclInConf(Q) implies`
	`self@post()->forAll(g | g->startsWith( Sequence{not R, P} ))`

Table 5. OCL Expressions for Universality Pattern (Assumptions as in Table 2)

P is true . . .	
. . . globally	`inv: self.oclInConf(P)`
. . . before R	`init: self@post()->forAll(g | g->startsWith( Sequence{P, R} ))`

. . . after Q	`inv: self.oclInConf(Q) implies`
	`self@post()->forAll(g | g->forAll(conf | conf = P))`

. . . between Q and R	`inv: self.oclInConf(Q) implies`
	`self@post()->forAll(g | g->startsWith( Sequence{P, R} ))`

. . . after Q until R	`inv: self.oclInConf(Q) implies`
	`not self@post()->exists(g | g->startsWith(Sequence{not R, not P and not R}))`

Table 6. OCL Expressions for Precedence Pattern (Assumptions as in Table 2)

S precedes P . . .	
. . . globally	`init: not self@post()->exists(g | g->startsWith( Sequence{not S, P} ))`
. . . before R	`init: self@post()->forAll(g | g->startsWith( Sequence{not P, S or P} ))`

. . . after Q	`inv: self.oclInConf(Q) implies`
	`self@post()->forAll(g | g->startsWith( Sequence{Q, not P, S} ))`

. . . between Q and R	`inv: self.oclInConf(Q) implies`
	`self@post()->forAll(g | g->startsWith( Sequence{not P, S or R} ))`

. . . after Q until R	`inv: self.oclInConf(Q) implies`
	`not self@post()->exists(g | g->startsWith( Sequence{not S and not R, P} ))`

Table 7. OCL Expressions for Response Pattern (Assumptions as in Table 2)

S responds to P . . .	
. . . globally	`inv: self.oclInConf(P) implies self@post()->forAll(g | g->includes(S))`

. . . before R	`inv: self.oclInConf(P) implies`
	`self@post()->forAll(g | g->startsWith( Sequence{not R, S} ))`

. . . after Q	`inv: self.oclInConf(Q) implies`
	`self@post()->forAll(g | g->includes( Sequence{P, true[0,'inf'], S})`

. . . between Q and R	`inv: self.oclInConf(Q) implies self@post()->forAll(g |`
	`g->includes( Sequence{P, not R [0,'inf'], S, not R [0,'inf'], R} ))`

. . . after Q until R	`inv: self.oclInConf(Q) implies self@post()->forAll(g |`
	`g->startsWith( Sequence{not R, P, not R[0,'inf'], S} ))`