Temporal Constraint Support for OCL*

Bilal Kanso and Safouan Taha

SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie, F-91192 Gif-sur-Yvette cedex, France
{Bilal.Kanso,Safouan.Taha}@supelec.fr

Abstract. The Object Constraint Language is widely used to express precise and unambiguous constraints on models and object oriented programs. However, the notion of temporal constraints, controlling the system behavior over time, has not been natively supported. Such temporal constraints are necessary to model reactive and real-time systems. Although there are works addressing temporal extensions of OCL, they only bring syntactic extensions without any concrete implementation conforming to the OCL standard. On top of that, all of them are based on temporal logics that require particular skills to be used in practice.

In this paper, we propose to fill in both gaps. We first enrich OCL by a pattern-based temporal layer which is then integrated into the current Eclipse's OCL plug-in. Moreover, the temporal constraint support for OCL, that we define using formal scenario-based semantics, connects to automatic test generators and forms the first step towards creating a bridge linking model driven engineering and usual formal methods.

Keywords: OCL, Object-oriented Programming, Temporal constraints, Eclipse/MDT, Model-Driven Engineering, Formal Methods.

1 Introduction

The Object Constraint Language (OCL) is an expression-based language used to specify constraints in the context of object-oriented models [2]. It is equivalent to a first-order predicate logic over objects, but it offers a formal notation similar to programming languages. OCL may complete the specification of all object-oriented models, even if it is mostly used within UML diagrams.

The OCL constraints may be invariants that rule each single system state, or preconditions and postconditions that control a one-step transition from a pre-state to a post-state upon the call of some operation. Thus, it is not possible to express constraints of dynamic behavior that involve different states of the model at different instants. This is essentially due to the absence of the notion of time and events in OCL. This limitation seems to form the main obstacle which the use of OCL faces today in the verification and validation areas. The standard OCL published in [2] does not provide any means of featuring temporal quantification, nor of expressing temporal properties such as safety or liveness.

* This work was funded by the French ANR TASCCC project (ANR-09-SEGI-014) [1].

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 83–103, 2013.

Adding a temporal layer to the OCL language forms a primordial step towards supporting the automatic verification and validation of object-oriented systems.

In this paper, we propose a temporal extension of OCL that enables modelers/developers to specify temporal constraints on object-oriented models. We do so by relying on *Dwyers*'s patterns [3]. A temporal constraint consists in a pattern combined with a scope. A pattern specifies the behavior that one wants to exhibit/avoid, while a scope defines the piece of execution trace to which a given pattern applies. This allows us to write temporal OCL constraints without any technical knowledge of formalisms commonly used to describe temporal properties such as LTL or CTL logics. We integrated this extension into the Eclipse/MDT current OCL plug-in.

In this work, we are also interested in formal methods applied to oriented-object systems for guiding software testing. Indeed, we will use the temporal properties that were formally written in our OCL temporal extension, as test purposes. Test purposes are commonly used to focus on testing particular aspects of models, avoiding other irrelevant ones [4,5]. Thinking of functional and security properties when writing test purposes is a common practice, but it has not been automated. Despite the interest of test purposes in the process of test case derivation, the lack of formal methods for their description and tools for their automatic generation forms one of the serious obstacles which the use of testing techniques faces today in the industrial areas.

To support test purposes specification, we enrich our language with formal scenario-based semantics; the behavior to be tested is expressed as a temporal OCL expression, and then automatically translated into a regular expression. This latter is generic enough to be used by a large family of generation techniques of test cases from object-oriented models. After its integration into the Eclipse/MDT current OCL plug-in, our language provides a framework not only to constrain dynamic behavior of object-oriented systems, but other to generate functional tests for objects and verify their properties. The language is indeed used in the validation of smart card product security [1]. It provides a means to express security properties (provided by *Gemalto*) on UML specification of the *GlobalPlatform*, the latest generation smart card operating system. In this work [6], the test requirements are expressed as OCL temporal constraints described in our proposed language and then transformed into test scenarios. These are then animated using the *CertifyIt* tool, provided by the *Smartesting* company to generate test cases.[1]

This paper is organized as follows. Section 2 presents the OCL language while Section 3 discusses its limitations on the expression of temporal aspects. Section 4 recalls the related works. Section 5 describes our proposal for extending OCL to support time and events. Section 6 provides the formal scenario-based semantics of our language. Section 7 describes the implementation of the proposed extension in the Eclipse's OCL plug-in and Section 8 presents the use of our proposal as a test purpose framework within the TASCCC project. Finally, Section 9 concludes and presents the future work.

[1] `www.globalplatform.org`, `www.gemalto.com`, `www.smartesting.com`

2 Object Constraint Language (OCL)

OCL is a formal assertion language, easy to use, with precise and unambiguous semantics [2]. It allows the annotation of any object-oriented model, even if it is most used within UML diagrams. OCL is very rich, it includes fairly complete support for:

- *Navigation operators* to navigate within the object-oriented model,
- *Set/Sequence operations* to manipulate sets and sequences of objects,
- *Universal/Existential Quantifiers* to build first order (logic) statements.

We briefly recall these OCL capabilities by means of an example. The UML class diagram in Figure 1 represents the structure of a simple *software system*. This system has a *free_memory* attribute corresponding to the amount of free memory that is still available, and the following three operations:

- *load(app: Application)*: downloads the application *app* given as a parameter.
- *install()*: installs interdependent applications already loaded. Different applications can be loaded before a single call of *install()*, but only applications having all their dependencies already loaded are installed.
- *run(app: Application)*: runs the application *app* given as a parameter that should be both already loaded and installed.

A system keeps references to the previously installed applications using the association end-point *installed_apps*. An *Application* has a *size* attribute and keeps references to the set of applications it depends on using the association end-point *dependencies*. We will use this illustrative example along this work.

Fig. 1. A model example

Exp 1 describes three typical OCL expressions. The first expression *all_apps_dependencies_installed* verifies that every installed application has its dependencies installed as well. The *all_dependencies* expression is a recursive function that builds the transitive closure of the (noncyclic) *dependencies* association. The *may_install_on* expression is a boolean function which has a system as parameter and verifies that installing the application with its dependencies fits into the system's free memory.

```
context System
def: all_apps_dependencies_installed: Boolean =
     self.installed_apps->forAll(app:Application | self.installed_apps->
     includesAll(app.dependencies))

context Application
def: all_dependencies: Set(Application) =
     self.dependencies.all_dependencies->asSet()->including(self)

def: may_install_on(sys: System): Boolean =
 (self.all_dependencies - sys.installed_apps).size->sum() < sys.free_memory
```

Exp. 1. OCL Expressions

Exp 1 illustrates the OCL ability to navigate the model (*self.installed_apps*, *app.dependencies*), select collections of objects and manipulate them with functions (*including()*, *sum()*), predicates (*includesAll()*) and universal/existential quantifiers (*forAll()*) to build boolean expressions.

3 OCL Limitations

3.1 OCL is a First-Order Predicate Logic

OCL boolean expressions are first order predicate logic statements over a model state. They are written with a syntax which is similar to programming languages. Such OCL expressions are evaluated over a single system state, which is a kind of a snapshot given as an object diagram at some point in time. An object diagram is a particular set of objects (class instances), slots (attribute values), and links (association instances) between objects. For example, an equivalent first order statement of *all_apps_dependencies_installed* expression is:

$$\forall s \in Sys, \forall a, b \in App,\ (s, a) \in Ins \wedge (a, b) \in Dep \Rightarrow (s, b) \in Ins$$

where a *state* (object diagram) is a tuple $(Sys, App, Ins, Dep, free, size)$

- Sys is the set of *System* objects
- App is the set of *Application* objects
- $Ins \subseteq Sys \times App$ is the set *installed_apps* links, $(s, a) \in Ins$ iff the *Application* instance a is installed on the *System* instance s
- $Dep \subseteq App \times App$ is the set *dependencies* links, $(a, b) \in Dep$ iff the *Application* instance a depends on the *Application* instance b
- $free : Sys \to \mathbb{N}$ is the function that associates each *System* instance s to the amount of free memory available
- $size : App \to \mathbb{N}$ is the function that associates each *Application* instance a to its memory size.

The first order logic allows quantification over finite and infinite domains[2] contrary to the OCL language which has no free quantification over infinite domains such as $\mathbb{Z}$ or $\mathbb{N}$. Indeed, in OCL, one distinguishes three kinds of domains:

[2] Note that the first order logic over the set theory (with possibly many infinite sets) is undecidable.

- Set of objects.
- Set of some Primitive Type values.
- *Time* that is the set of all instants of the model's life. It corresponds to $\mathbb{N}$ if time is discrete, $\mathbb{Q}$ if time is dense or $\mathbb{R}$ if time is continuous.

The OCL expressions presented in Exp 1 are typical examples of OCL quantification (*forAll(), exists()*) over sets of objects (e.g. *self.dependencies*) and sets of primitive type values (e.g. *self.all_dependencies.size* of PrimitiveType::Integer). Since these sets are selections/subsets of an object diagram, they are finite by construction. Hence, there is no limitation to use OCL quantifiers over them. However, since *Time* is intrinsically infinite, quantification over it is restricted within OCL. This last point will be detailed in the next subsections.

3.2 Temporal Dimension

As previously mentioned, the OCL expressions are evaluated over a single system state at some point in time. But, the OCL language also provides some implicit quantification over time by means of OCL rules. An OCL *rule* is a temporal quantification applied to an OCL boolean expression, and may be an invariant of a class, a pre- or a post-condition of an operation.

The expression within an invariant rule has be to be satisfied throughout the whole life-time of all instances of the context class. The first expression in Exp 2 specifies the invariant which requires, in all system states, a nonempty free memory and the installation of dependencies of all installed applications. The precondition and postcondition are used to specify operation contracts. A precondition has to be true each time the corresponding operation is called, and a postcondition has to be true each time right after the corresponding operation execution has terminated. The second expression in Exp 2 describes the rule that provides the *load(app: Application)* contract. It requires that the application given as a parameter is not already installed and there is enough memory available to load it. Then, it ensures that the *free_memory* attribute is updated using the *@pre* OCL feature.

```
1 context System
2 inv : self.free_memory > 0 and all_apps_dependencies_installed = true

4 context System::load(app: Application):
5 pre : self.installed_apps->excludes(app) and self.free_memory > app.size
6 post: self.free_memory = self.free_memory@pre - app.size
```

Exp. 2. OCL rules

The operation parameters can be used within a pre or a post-condition rule, but the *@pre* OCL feature is only used within a post-condition rule. When *@pre* is used within the boolean expression of a post-condition rule, it is evaluated over two system states, one right before the operation call and one right after its execution. In other words, OCL expressions describe a single system state or a one-step transition from a previous state to a new state upon the call of some operation. Therefore, there is no way to make OCL expressions involving

different states of the model at different points in time. OCL has a very limited temporal dimension.

To illustrate the temporal limits of OCL, let us consider the following temporal properties for the example presented in Figure 1:

safety_1: each application can be loaded at most one time
safety_2: an application load must precede its run
safety_3: there is an install between an application loading and its run
liveness: each loaded application is installed afterwards

Such temporal properties are impossible to specify in OCL without at least enriching the model structure with state variables. In temporal logics [7], we formally distinguish the safety properties from the liveness ones. *Safety* properties for bad events/states that must not happen and *liveness* properties for good events/states that should happen. As safety properties consider finite behaviors, they can be handled by modifying the model and adding variables which save the system history. If we consider the first safety property, one solution is to save within a new attribute *loaded_apps* the set of applications already loaded, but not yet installed and then check in the *load(app: Application)* precondition that the loaded application is neither installed, nor loaded:

```
1 context System::load(app: Application):
2 pre : self.installed_apps->excludes(app) and
      self.loaded_apps->excludes(app) and self.free_memory > app.size
```

Even if specifying complementary temporal OCL constraints must not alter the model, such case-by-case techniques are of no use when specifying liveness properties that handle infinite behaviors.

In this work, we are mainly interested in temporal constraints from the temporal logics point of view, when they are ruling the dynamic behavior of systems. They specify absence, presence and ordering of the system life-time steps. A step may be a state that holds for a while or an event occuring at some point of time.

3.3 Events

An event is a predicate that holds at different instants of time. It can be seen as a function $P : Time \rightarrow \{true, false\}$ which indicates at each instant, if the event is triggered. The subset $\{t \in Time \mid P(t)\} \subseteq Time$ stands then for all time instants at which the event P occurs. When quantifying time, we need to select such subsets of $Time$ that correspond to events. We commonly distinguish five kinds of events in the object-oriented paradigm:

Operation call instants when a sender calls an operation of a receiver object
Operation start instants when a receiver object starts executing an operation
Operation end instants when the execution of an operation is finished
Time-triggered event that occurs when a specified instant is reached
State change that occurs each time the system state changes (e.g when the value of an attribute changes). Such an event may have an OCL expression as a parameter and occurs each time the OCL expression value changes.

OCL only provides an implicit universal quantification over *operation call* events within pre-conditions and a universal quantification over *operation end* events within post-conditions. However, it lacks the finest type of events which is *state change*. State change events are very simple, but powerful construct. It can replace other types of events. Suppose we add a chronometric clock that is now a part of our system. This common practice will create a new object *clock* within our system that has a *time* attribute. Each change of that attribute will generate a state-change event. A time-triggered event of some specified *instant* will be then one particular state-change in which the OCL boolean expression *clock.time = instant* becomes true.

To replace operation call, start and end events using the state-change event, we need to integrate the stack structure within the system model. We do not recommend this technique that is in contradiction to the model-driven engineering approach because it pollutes the system model with platform specific information and ruins all the abstraction effort.

3.4 Quantification

OCL has no existential quantification over time or events. For example, the second safety property we previously proposed needs existential quantification: **it exists** a *load()* operation call that precedes a *run()* operation call.

The other quantification limitation we identified is that OCL sets its few temporal quantification constructs within OCL rules, prior to the quantification over objects within the OCL expressions. Again, the second safety property needs quantifying over objects prior to quantifying over time: **for all** application instance *app*, **it exists** a *load(app)* operation call that precedes a *run(app)* operation call. We intend to relate the load event of the particular application *app* with its run. This quantification order is the way to define the relations we may need between events.

4 Related Work

Several extensions have been proposed to add temporal constraints to the OCL language. [8] presents an extension of OCL, called TOCL, with the basic operators of the common linear temporal logic (LTL). Both future and past temporal operators are considered. This paper only provides a formal description of the extension based on *Richters*'s OCL semantics [9]. It gives no explanation of how all presented formal notions can be implemented. In [10,11], authors propose an extension of OCL for modeling real-time and reactive systems. A general notion of time and event is introduced, providing a means to describe the temporal behavior of UML models. Then, OCL is enriched by (1) the temporal operator `@event` (inspired by the OCL operator `@pre`) to refer to the expression value at the instant when an event occurs, and (2) the time modal operators always and sometime. [12] proposes a version of CTL logic, called BOTL, and shows how to map a part of OCL expressions into this logic. Indeed, there is no extension of OCL by temporal operators, but a theoretical precise mapping of a

part of OCL into BOTL. [13] provides an OCL extension, called EOCL, with CTL temporal operators. This extension is strongly inspired by BOTL [12], and allows model checking EOCL properties on UML models expressed as abstract state machines. A tool (SOCLE), implementing this extension, is briefly presented with verification issues in mind; however, there is no tool available at the project site [14]. Similarly, Flake et al. [15] formalize UML Statechart within the Richters's OCL semantics and extend OCL with Clocked CTL in order to provide a sound basis for model-checking. [16] proposes templates (e.g. after/eventually template) to specify liveness properties. A template is defined by two clauses: a cause and a consequence. A cause is the keyword after followed by a boolean expression, while a consequence is an OCL expression prefixed by keywords like eventually, immediately, infinitely, etc. These templates are formally translated into observational μ-calculus logic. This paper gave no means to OCL developers to implement such templates. It only formally addresses some liveness properties; other liveness and safety properties are not considered. [17] adds to OCL unary and binary temporal operators such as until and always and [18] proposes past/future temporal operators to specify business components. Both [17] and [18]'proposals are far from being used in the context of concrete implementation conforming to the standard OCL [2]. For instance, in [18], an operator may be followed by user-defined operations (with possible side effects) that are not concretely in conformance with the standard OCL. Table 1 summarizes the state of the art and emphasizes the need for a complete approach.

Table 1. Related work

Approach	Temporal Layer	Event Constructs	Quantification Order	Tooling	Formal Semantics
Ziemann et al. [8]	LTL + past	no	no	no	trace semantics
Calegari et al. [10,11]	future/past modal operators	yes	no	not conforming to OCL standard	trace semantics
Distefano et al. [12]	CTL	no	no	no	BOTL
Mullins et al. [13]	CTL	no	no	not conforming to OCL standard	inspired by BOTL
Bradfield et al. [16]	template clauses (response pattern)	no	no	no	observational μ-calculus
Ramakrishnan et al. [17] Conrad et al. [18]	future/past modal operators	no	no	no	no
Flake et al [15]	Clocked CTL	state-oriented	no	no	trace semantics

Among temporal constraints we have the particular case of timings properties that are commonly used within the real-time systems development. Timings are static duration constraints between event occurrences, they are necessary to specify WCETs (Worst Case Execution Time), deadlines, periods... There are surprisingly many efforts to annotate statically this kind of constraints using UML profiles. MARTE [19] is an UML extension defining stereotypes (RTSpecification, RTFeature, RTAction) that annotate classes and operations to specify their timings.

5 OCL Temporal Extension

After identifying the OCL limitations that are absences of temporal operators, event constructs and free quantification (see Section 3), and after reviewing most existing OCL temporal extensions (see Section 4), we give in the following our contribution about OCL temporal extension:

- A *pattern-based language* contrary to most of OCL temporal extensions that are based on temporal logics such as LTL or CTL (see Section 4). The technicality and the complexity of these formalisms give rise naturally to difficulties even to the impossibility, in some cases, of using them in practice [3];
- Enrichment of OCL by the notion of *events* that is completely missing in the existing temporal extensions of OCL;
- A user-friendly syntax and formal *scenario-based semantics* of our OCL temporal extension (see Section 6);
- A *concrete implementation* conforming to the standard OCL [2]. In fact, all the works mentioned in Section 4 only address the way OCL has to be extended to deal with temporal constraints. The main purpose behind them was to use OCL in verification areas such as model checking. However, they did not reach this last step, at least not in practice, due to the absence of concrete implementations conforming to the standard OCL [2] of the proposed extensions.

5.1 Temporal Patterns

Formalisms such as linear temporal logic (LTL) and tree logic (CTL) have received a lot of attention in the formal methods community in order to describe temporal properties of systems. However, most engineers are unfamiliar with such formal languages. It requires a lot of effort to bridge the semantic gap between the formal definitions of temporal operators and practice. To shed light on this obstacle, let us consider the *safety_3* property, its equivalent LTL formula looks like:

$$\Box(load \wedge \neg run \Rightarrow ((\neg run \ \mathsf{U} \ (install \wedge \neg run)) \vee \neg \diamond run))$$

It means that each time ($\Box$) we have a load, this implies that there will be no run at least until (U) the install happens or there will be no run at all in the future ($\neg \diamond run$). To avoid such error-prone formulas, *Dwyer* et al. have proposed a pattern-based approach [3]. This approach uses specification patterns that, at a higher abstraction level, capture recurring temporal properties. The main idea is that a temporal property is a combination of one **pattern** and one **scope**. A scope is the part of the system execution path over which a pattern holds.

Patterns [3] proposes 8 patterns that are organized under a semantics classification (left side of Figure 2). One distinguishes occurrence (or non-occurrence) patterns from order patterns.

Occurrence patterns are: (*i*) Absence: an event never occurs, (*ii*) Existence: an event occurs at least once, (*iii*) BoundedExistence has 3 variants: an event

occurs k times, at least k times or at most k times, and (iv) Universality: a state is permanent.

Order patterns are: (i) Precedence: an event P is always preceded by an event Q, (ii) Response: an event P is always followed by an event Q, (iii) ChainPrecedence: a sequence of events $P_1, \ldots, P_n$ is always preceded by a sequence $Q_1, \ldots, Q_n$ (it is a generalization of the Precedence pattern), and (iv) ChainResponse: a sequence of events $P_1, \ldots, P_n$ is always followed by a sequence $Q_1, \ldots, Q_n$ (it is a generalization of the Response pattern as well).

Scopes [3] proposes 5 kinds of scopes (right side of Figure 2): (i) Globally covers the entire execution, (ii) Before Q covers the system execution up to the first occurrence of Q, (iii) After Q covers the system execution after the first occurrence of Q, (iv) Between Q and R covers time intervals of the system execution from an occurrence of Q to the next occurrence of R, and (v) After Q until R is same as the Between scope in which R may not occur.

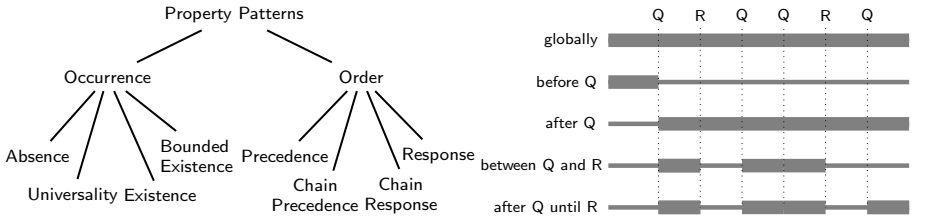

Fig. 2. Dwyer's patterns and scopes

Back to our temporal property *safety_3* : *there is an install between an application loading and its run.* It simply corresponds to the Existence pattern (exists *install*) combined with the Between scope (between load and run). It is clear that the patterns of *Dwyer* et al. dramatically simplify the specification of temporal properties, with a fairly complete coverage. Indeed, they collected hundreds of specifications and they observed that 92% of them fall into this small set of patterns/scopes [3]. Furthermore, a complete library is provided [20], mapping each pattern/scope combination to the corresponding formula in many formalisms (e.g. LTL, CTL, QREs, μ-calculus).

For these reasons, we adopt this pattern-based approach for the temporal part of our OCL extension and we bring enhancements to improve the expressiveness:

- *Dwyer* et al. have chosen to define scopes as right-open intervals that include the event marking the beginning of the scope, but do not include the event marking the end of the scope. We extend scopes with support to open the scope on the left or close it on the right. This adds one variant for both the Before and After scopes and three supplementary variants for the Between and After ... until scopes. We have chosen open intervals as default semantics.
- In *Dwyer* et al. work, Between and After ... until scopes are interpreted relative to the first occurrence of the designated event marking the beginning

of the scope (Figure 2). We kept this as default semantics and we provide an option to select the last occurrence semantics.
- To respect the classical semantical conventions of temporal logics, we renamed the After ... until scope as After ... **unless**. Then to improve the usability, we added the scope When that has an OCL boolean expression as a parameter and that covers the execution intervals in which this OCL expression is evaluated to true. The When scope is derived from the After ... unless scope as follows:

$$\textbf{When } P \equiv \textbf{After } \mathsf{becomesTrue}(P) \textbf{ unless } \mathsf{becomesTrue}(\textit{not } P)$$

  The becomesTrue event is introduced in Subsection 5.2.
- Order patterns describe sequencing relationships between events and/or chains of events. The *Dwyer* et al. semantics adopt non strict sequencing. For example, A, B (is) preceding B, C in both A, B, C and A, B, B, C executions. We add features to specify strict sequencing for an order pattern. For example, A, B (is) preceding **strictly** B, C only in the A, B, B, C execution. We provide same constructs to have strict sequencing within one chain of events, A, B to denote a non strict sequencing and $A; B$ for a strict one.
- In *Dwyer* et al. work, there is no construct equivalent to the temporal operator Next. For example, A (is) preceding C in both $A; C$ and $A; B; C$ executions. We add features to specify the Next temporal operator for an order pattern. For example, A (is) preceding **directly** C only in the $A; C$ execution. The directly feature is a particular case of strict sequencing.

These enhancements are inspired by our needs within the TASCCC project [1] and the *Dwyer*'s notes about the temporal properties that were not supported [20]. It is obvious that these enhancements improve the requirement coverage (i.e. 92%) shown by *Dwyer*, but we did not measure it precisely.

5.2 Events

Events are predicates to specify sets of instants within the time line. We discussed in Section 3 the different types of events in the object-oriented approach. There are operation (call/start/end) events, time-triggered events and state change events. We have seen that when integrating the clock into the system, time-triggered events are particular state change events. Hence, we only need to extend OCL with the necessary construct for both operation and state change events.

We aim to connect our OCL temporal extension to formal methods such as model-checking and test scenario generation. Formal methods are mainly based on the synchronous paradigm that has well-founded mathematical semantics and that allows formal verification of the programs and automatic code generation. The essence of the synchronous paradigm is the atomicity of reactions (operation calls) where all the occurring events during such a reaction are considered simultaneous. In our work, we will adopt the synchronous paradigm, and we then merge the operation (call/start/end) events into one call event, named isCalled, that leads the system from a pre-state to a post-state without considering neither observing intermediate change states.

isCalled: is a generic event construct that unifies both operation events and state change events. It has three optional parameters:

- op: is the called operation. The keyword *anyOp* is used if no operation is specified
- pre: is an OCL expression that is a guard over the system pre-state and/or the operation parameters. The operation invocation will lead to a call event only if this guard is satisfied by the pre-state of the call. If it is not satisfied, the event will not occur even if the operation is invoked.
- post: is an OCL expression that is a guard over the system post-state and/or the return value. The operation invocation will lead to a call event only if this guard is satisfied by the post-state of the call.

becomesTrue: is a state change event that is parameterized by an OCL boolean expression P, and designates a step in which P becomes true, i.e. P was evaluated to false in the previous state. In the object-oriented paradigm, a state change is necessarily a consequence of some operation call, therefore the **becomesTrue** construct is a syntactic sugar and stands for any operation call switching P to true (see Figure 3):

$$\mathsf{becomesTrue}(P) \equiv \mathsf{isCalled}(op: anyOp,\ pre: not\ P,\ post: P)$$

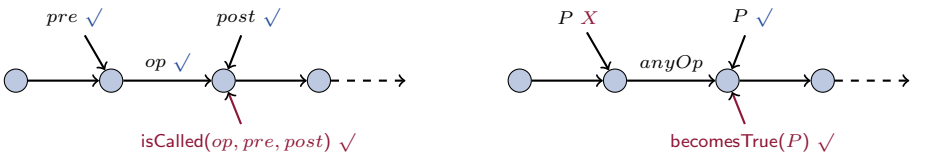

Fig. 3. Events

We also define two generic operators/constructors over events:

Disjunction: $ev1 \mid ev2$ occurs when $ev1$ occurs **or** $ev2$ occurs

Exclusion: $ev1 \setminus ev2$ occurs when $ev1$ occurs **and** $ev2$ does **not**

Other operators (Negation, Conjunction, ...) can be easily derived:

$$\begin{aligned}
&\textbf{not}(event) && \equiv \mathsf{isCalled}(anyOp, true, true) \setminus event\\
&\mathsf{becomesTrue}(P_1) \wedge \mathsf{becomesTrue}(P_2) && \equiv \mathsf{isCalled}(anyOp, \neg P_1 \wedge \neg P_2, P_1 \wedge P_2)\\
& && \neq \mathsf{becomesTrue}(P_1 \wedge P_2)
\end{aligned}$$

5.3 Quantification

Our OCL extension supports universal quantification over objects prior to quantification over time. The OCL feature let *Variables* in can be used within our OCL extension on the top of temporal expressions (see Section 7.3).

6 Semantics

Several formal semantics have been provided to describe the OCL language. These are not given in this paper, only the semantics of our OCL temporal extension are defined. Interested readers may refer to [2,9].

A test case is a scenario/sequence of operations calls. Since we are interested in test cases generation, we adopt a scenario-based semantics over the synchronous paradigm to formally describe our temporal extension. The essence of that paradigm is the atomicity of reactions (operation calls) where all the events occurring during such a reaction are considered as simultaneous. A reaction is one atomic call event, that leads the system directly from a pre-state to a post-state without going through intermediate states.

6.1 Events

We define the set of all atomic events of a given object model as follows:

Definition 1 (Alphabet of Atomic Events). *Let $\mathbb{O}$ be the set of all operations and $\mathbb{E}$ be the set of all OCL expressions of an object model $\mathcal{M}$. The* ***alphabet Σ of atomic events*** *is defined by the set $\mathbb{O} \times \mathbb{E} \times \mathbb{E}$.*

An atomic event $e \in \Sigma$ then takes the form: $e = (op, pre, post)$. It stands for a call of the operation op in a context where pre stands for the precondition satisfied in the pre-state and $post$ for the postcondition satisfied in the post-state.

We now give the formal meaning of the notion of events introduced in the grammar presented in Figure 5.

Definition 2 (Events). *Let Σ be the alphabet of atomic events, $\mathbb{O}$ be the set of all operations and $\mathbb{E}$ the set of all OCL expressions. An* ***event*** *is either an* $\mathsf{isCalled}(op, pre, post)$ *or* $\mathsf{becomesTrue}(\mathsf{P})$ *where:*

$$\begin{aligned} \mathsf{isCalled}(op, pre, post) &= \{(op, pre', post') \in \Sigma \mid pre' \Longrightarrow pre,\ post' \Longrightarrow post\} \text{ and} \\ \mathsf{becomesTrue}(P) &= \{(op, pre, post) \in \Sigma \mid op \in \mathbb{O},\ pre \Longrightarrow \neg P,\ post \Longrightarrow P\} \end{aligned}$$

Definition 2 calls for the following three comments:

- In our language, the operation op can be replaced by $anyOp$ the set of all operations as follows:
  $$\mathsf{isCalled}(anyOp, pre, post) = \bigcup_{op \in \mathbb{O}} \{(op, p, q) \in \Sigma \mid\ p \Rightarrow pre,\ q \Rightarrow post\}$$
- $\mathsf{becomesTrue}(P)$ is equivalent to $\mathsf{isCalled}(anyOp, \neg P, P)$. We keep this primitive to make our language easier to use.
- An event does not represent a single atomic event, but a specific subset of atomic events. It is intuitively the set of all atomic events in which the operation op is invoked, in a pre-state which implies the expression pre and leading to a post-state which implies the expression $post$. The set of all events is then defined[3] as the set 2^{Σ}.

[3] 2^X denotes the set of all subsets of X.

A disjunction (resp. exclusion) of events is an event. By considering events as subsets of Σ, the semantics of the disjunction (resp. exclusion) constructor $|$ (resp. $\backslash$) over events is given as a simple union (resp. minus) over sets.

Definition 3 (Operators over Events). *Let Σ be the alphabet of atomic events. The* ***disjunction operator*** $|$ *and the* ***exclusion operator*** $\backslash$ *over Σ are defined as follows:*

$$\begin{array}{ll} | \ : 2^{\Sigma} \times 2^{\Sigma} \to 2^{\Sigma} & \backslash \ : 2^{\Sigma} \times 2^{\Sigma} \to 2^{\Sigma} \\ \quad (E_1, E_2) \mapsto E_1 \cup E_2 & \quad (E_1, E_2) \mapsto E_1 - E_2 \end{array}$$

6.2 Scenarios

We introduce the notion of a scenario, which allows us to interpret our OCL temporal expressions. A *scenario* σ in a model $\mathcal{M}$ is a finite sequence $(e_0, \ldots, e_n) \in \Sigma^*$ of atomic events. Such a scenario embodies the temporal order between atomic event triggering, where the notion of time is implicitly specified. In a scenario $(e_0, \ldots, e_n)$, there is a logical time associated to the atomic event e_0 which precedes the logical time associated to the atomic event e_1, and so on.

In the following, for every scenario $\sigma \in \Sigma^*$ of length n, we write $\sigma = (\sigma(0), \ldots, \sigma(n-1))$. Thus, $\sigma(i)$ denotes the atomic event at index i and $\sigma(i:j)$ the part of σ containing the sequence of atomic events between i and j.

6.3 Temporal Expressions

We define here the semantics for our temporal expressions that are evaluated over event-based scenarios.

Definition 4 (Scopes). *Let $\mathbb{S}$ be the set of scopes defined in the grammar presented in Figure 5. The* ***semantics of a scope*** *$s \in \mathbb{S}$ is given by the function $[[s]]^s : \Sigma^* \longrightarrow 2^{\Sigma^*}$ defined for every $\sigma \in \Sigma^*$ of length n as follows:*

- $[[\mathsf{globally}]]^s(\sigma) = \{\sigma\}$
- $[[\mathsf{before}\ E]]^s(\sigma) = \{\sigma(0 : i-1) \ \mid \ \sigma(i) \in E \text{ and } \forall k, 0 \leq k < i, \sigma(k) \notin E\}$
- $[[\mathsf{After}\ E]]^s(\sigma) = \{\sigma(i+1 : n-1) \ \mid \ \sigma(i) \in E \text{ and } \forall k, 0 \leq k < i, \sigma(k) \notin E\}$
- $[[\mathsf{between}\ E_1\ \mathsf{and}\ E_2]]^s(\sigma) = \{\sigma(i_k + 1 : j_k - 1) \mid$
  $\forall k \geq 0, i_k < j_k < i_{k+1}, \sigma(i_k) \in E_1, \sigma(j_k) \in E_2,$
  $\forall m, i_k \leq m < j_k, \sigma(m) \notin E_2 \text{ and } \forall l, j_k < l < i_{k+1}, \sigma(l) \notin E_1\}$
- $[[\mathsf{after}\ E_1\ \mathsf{unless}\ E_2]]^s(\sigma) =$
  $\{\sigma(i_k + 1 : j_k - 1) \mid \forall k \geq 0, i_k < j_k < i_{k+1}, \sigma(i_k) \in E_1, \sigma(j_k) \in E_2,$
  $\forall m, i_k \leq m < j_k, \sigma(m) \notin E_2 \text{ and } \forall l, j_k < l < i_{k+1}, \sigma(l) \notin E_1\}$
  $\cup \{\sigma(i : n-1) \mid \sigma(i) \in E_1, \forall m \geq i, \sigma(m) \notin E_2\}$

Definition 5 (Patterns). *Let $\mathbb{P}$ be the set of patterns defined in the grammar presented in Figure 5. The* ***semantics of a pattern*** *$p \in \mathbb{P}$ is given by the function $[[p]]^p : \Sigma^* \longrightarrow \{true, false\}$ defined for every $\sigma \in \Sigma^*$ as follows:*

- $[[\mathsf{never}\ E]]^p(\sigma) \Leftrightarrow \forall i \geq 0, \sigma(i) \notin E$
- $[[\mathsf{always}\ P]]^p(\sigma) \Leftrightarrow [[\mathsf{never}(\mathsf{isCalled}(anyOp, \_, \neg P))]]^p(\sigma)$
- $[[E_1\ \mathsf{preceding}\ E_2]]^p(\sigma) \Leftrightarrow \forall i \geq 0, (\sigma(i) \in E_2 \Rightarrow \exists k \leq i, \sigma(k) \in E_1)$
- $[[E_1\ \mathsf{following}\ E_2]]^p(\sigma) \Leftrightarrow \forall i \geq 0, (\sigma(i) \in E_2 \Rightarrow \exists k \geq i, \sigma(k) \in E_1)$
- $[[\mathsf{eventually}\ E\ \alpha\ \mathsf{times}]]^p(\sigma) \Leftrightarrow \mathsf{card}\big(\{i \mid \sigma(i) \in E\}\big) \begin{cases} = k\ if\ \alpha = k \\ \geq k\ if\ \alpha = \mathsf{at\ least}\ k \\ \leq k\ if\ \alpha = \mathsf{at\ most}\ k \end{cases}$

Definition 6 (OCL temporal expressions). *The* ***semantics of an OCL temporal expression*** $(\mathsf{pattern}, \mathsf{scope}) \in \mathbb{P} \times \mathbb{S}$ *over a scenario* $\sigma \in \Sigma^*$*, denoted by* $\sigma \vDash (\mathsf{pattern}, \mathsf{scope})$*, is defined by:*

$$\sigma \vDash (\mathsf{pattern}, \mathsf{scope}) \iff \forall \sigma' \in [[\mathsf{scope}]]^s(\sigma),\ [[\mathsf{pattern}]]^p(\sigma')$$

Due to the lack of space in this paper, we do not provide the semantics of all variants of patterns and scopes that we defined in our temporal extension, interested readers may refer to [21].

7 Integration within the Eclipse/MDT Tool-Chain

7.1 Structure of Eclipse's OCL Plug-In

The Eclipse/MDT OCL Plug-in [22] provides an implementation of the OCL OMG standard for EMF-based models. It provides a complete support for OCL, but we will only focus on some capabilities that are represented and highlighted in red within Figure 4.

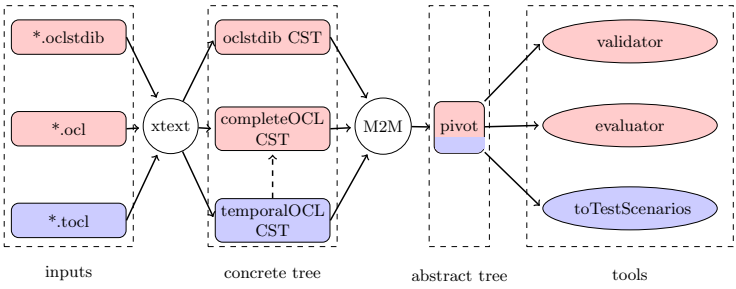

Fig. 4. Eclipse MDT/OCL 4.× with Temporal extension

On the left of Figure 4, there are two Xtext editors that support different aspects of OCL usage. The completeOCL editor for **.ocl* documents that contain OCL constraints, and the OCLstdlib editor for **.oclstdlib* documents that facilitates development of the OCL standard library. This latter is primarily intended for specifying new functions and predicates to use within OCL expressions.

In the middle of Figure 4, the architecture of the OCL plug-in is based around a pivot model. The pivot model isolates OCL from the details of any particular UML or Ecore (or EMOF or CMOF or etc.) meta-model representation. OCL expressions can therefore be defined, analyzed and evaluated for any EMF-based meta-model. Notice that most object-oriented meta-models (e.g. UML) are already specified within EMF.

From left to right, the Xtext framework [23] is used to transform the OCL constraints document to a corresponding Concrete Syntax Tree (CST). Then, using a Model to Model transformation (M2M), it generates the pivot model which corresponds to the Abstract Syntax Tree (AST). Notice that the CST and the AST are both defined within the OMG standard [2]. Finally on the right of Figure 4, the OCL plug-in provides interactive support to validate OCL expressions through their pivot model and evaluate them on model instances.

As highlighted in blue in Figure 4, we integrated our temporal extension within the Eclipse/MDT OCL tool-chain with respect to its architecture. We first extended the OCL concrete grammar to parse **.tocl* documents that contain temporal OCL properties. After that, we extended in Ecore both completeOCLCST and pivot meta-models with all the temporal constructs we defined. We kept both Xtext and M2M frameworks. Finally, in a join work with our partner *LIFC* within the TASCCC project, we developed a tool to transform temporal properties to test scenarios [1,6] (see Section 8).

Due to the lack of space in this paper, we do not give the implementation details on the temporalOCLCST structure and the pivot extension, but the temporal OCL plug-in is published with documentation under a free/open-source license [21].

7.2 Concrete Syntax

We extended the OCL concrete grammar defined within the OMG standard [2] and implemented it within the Eclipse/MDT plug-in. The syntax of our language for **.tocl* documents is summarized in Figure 5.

TempOCL ::= temp (name)? ':' *TempSpec*
TempSpec ::= *Quantif*? *Pattern Scope*
Quantif ::= let Variable (',' Variable)* in

Pattern ::= always OclExpression
| never *Event*
| eventually *Event* ((at least | at most)? integer times)?
| *EventChain* preceding(directly | strictly)? *EventChain*
| *EventChain* following (directly | strictly)? *EventChain*

Event ::= *CallEvent* ('|' *Event*)?
| *ChangeEvent* ('|' *Event*)?
EventChain ::= *Event* (',' *Event*)*
| *Event* (';' *Event*)*

Scope ::= globally
| before *Event* ('[' | ']')?
| after ('[' | ']')? *Event*
| between ('[' | ']')? last? *Event* and *Event* ('[' | ']')?
| after ('[' | ']')? last? *Event* unless *Event* ('[' | ']')?
| when OclExpression

CallEvent ::= isCalled '(' (anyOp | op : Operation)
(',' pre : OclExpression)?
(',' post : OclExpression)? ')'
ChangeEvent ::= becomesTrue '(' OclExpression ')'

Fig. 5. Grammar of the OCL temporal extension

In this figure, non-terminals are designated in *italics* and terminals in **bold**. (...)? designates an optional part and (...)* a repetitive part. Finally, the non-terminals imported from the standard OCL grammar (e.g. OclExpression) are underlined. This grammar represents the temporal layer we added to OCL expressions (temporal patterns, events constructs and support of quantification). Taking advantage of the integration within the Eclipse/MDT OCL, we developed, with the help of the Xtext framework, a temporal OCL editor which provides syntax coloring, code formatting, code completion, static validation (well formedness, type conformance...) and custom quick fixes, etc. Furthermore, there is an outline view that shows the concrete syntax tree of the temporal OCL property on-the-fly (while typing). Figure 6 illustrates a snapshot of the outline view.

7.3 Examples of Temporal Properties

In Exp 3, the temporal properties we identified in Section 3 are written using our OCL temporal extension. Due to our grammar, the temporal properties seem to be written in natural language. They are ruling call event occurrences with different patterns: following (strict), preceding (non-strict), existence and boundedexistence that are combined with globally and between scopes. Both *safety_2* and *safety_3* properties require quantification over objects prior to temporal operators to specify relations between events. For instance, in *safety_2* we need to specify that the load of an application *app* must precede the run of the same application *app*, and not any other. To do so, we introduced the variable *apptoInstall* which allows us to set the same parameter *apptoInstall* for both *load* and *run* operations.

```
1  context System
2  temp safety_1:
3            eventually isCalled(load(app:Application)) at most 1 times
4            globally

6  temp safety_2: let apptoInstall : Application in
7                      isCalled(load(app:Application), pre: app =
                           apptoInstall)
8            preceding isCalled(run(app:Application), pre: app =
                 apptoInstall)
9            globally

11 temp safety_3: let apptoInstall : Application in
12           eventually isCalled(install())
13           between    isCalled(load(app:Application), pre: app =
                 apptoInstall)
14           and        isCalled(run(app:Application), pre: app =
                 apptoInstall)

16 temp liveness:              isCalled(install())
17           following strictly isCalled(load(app:Application))
18           globally
```

Exp. 3. Temporal OCL constraints

The *safety_3* property is not relevant because having an install call between the load and the run does not ensure that the application will be really installed.

This will not happen if some dependencies are not loaded. To overcome this, we propose in Exp 4 two variants of the *safety_3* property. The *safety_3_v1* property ensures that there is a particular install call, leading to a post-state where the application is installed. The *safety_3_v2* property only specifies that the application becomes installed independently of any operation call (see the becomesTrue semantics in Subsection 5.2). It requires any operation call from a pre-state where the application was not installed to a post-state where it is installed.

```
temp safety_3_v1: let apptoInstall : Application in
        eventually isCalled(install(),
                            post: self.installed_apps->includes(apptoInstall))
        between     ...

temp safety_3_v2: let apptoInstall : Application in
        eventually becomesTrue(self.installed_apps->includes(apptoInstall))
        between     ...
```

Exp. 4. Variants of *Safety_3* property

8 Application: Test Purpose Framework

Testing nowadays programs leads naturally to an exponential state space. When reducing the state space, the testing process derivation may miss test cases of interest and yield irrelevant ones. Test purposes (test intentions) are viewed as one of the most promising directions to cope with this limit [4,5]. They are commonly used to guide the test generation techniques. A test purpose is a description of the part of the specification that we want to test and for which test cases are later generated.

Thinking of functional and security properties when writing test purposes is a common practice, but it has not been automated. We propose to automatically handle test purposes. We first specify the test purposes as OCL temporal properties using our extension. Then, we transform them automatically into regular expressions. This phase was achieved in a join work with our partner *LIFC* who generates automatically regular expressions from properties written in our OCL extension and measures the coverage of the properties [6]. Considering scenario-based semantics (see Section 6), the regular expressions generated are equivalent to the OCL temporal expressions from which they are derived. This automatic transformation is done based on the complete library given by *Dwyer* et *al.* [24,3], mapping each pattern/scope combination to the corresponding formula in many formalisms such as LTL, CTL, QRegExps and μ-calculus.

We choose regular expressions as an output language because they are generic enough to be used (with some adaptation) in large family of test generation techniques that are guided by test purposes. For instance, our framework connects naturally to the combinatorial test generation tool *Tobias* [25], that unfolds, in a combinatorial way, tests expressed as regular expressions. Furthermore, approaches such as [4,5,26] that describe their test purposes manually in the form of Labeled Transition systems (LTS) or Input-Labeled Transition systems

(IOLTS), could easily be targets of our framework. We only need to translate the regular expressions produced from the OCL temporal expressions into these two formalisms, which requires a little technical effort.

The first use of this test purpose framework is within the TASCCC project [1] which aims to automatize testing security properties on smart card products and experiment it on *GlobalPlatform*, a last generation smart card operating system. The process of test generation used in this project consists mainly of five phases:

1. Identifying security properties from the Common Criteria standard[4];
2. Writing these security properties using our OCL temporal extension and based on the *GlobalPlatform* UML model distributed by *Smartesting*;
3. Translating the OCL temporal properties into equivalent test scenarios that are regular expressions over an alphabet of API calls;
4. Transforming the test scenarios into test cases by means of *Tobias* [25];
5. Animating the generated test cases on the *GlobalPlatform*. This is performed by the *CertifyIt* tool of *Smartesting*.

Figure 6 is a snapshot of the Temporal OCL editor in which the *GlobalPlatform* security properties (extracted from Common Criteria) were entered and the corresponding regular expressions were generated. The visible property specifies that each logical channel must keep secured between the last successful call of ExternalAuthenticate and a command needing authentication.

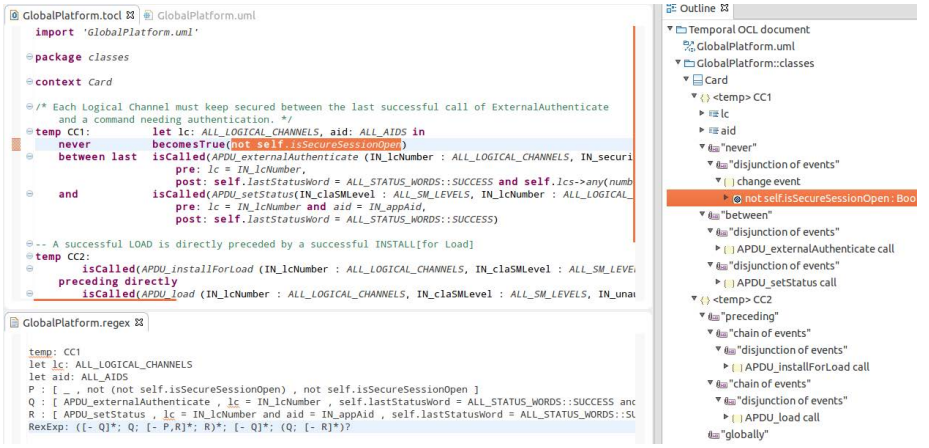

Fig. 6. The Temporal OCL editor (*GlobalPlatform* properties)

9 Conclusion

Although many temporal extensions of OCL exist, they have never been used convincingly in practice. To cope with this, we have presented a pattern-based extension of the OCL language to express temporal constraints on object-oriented

[4] `www.commoncriteriaportal.org`

systems. We defined our language with a formal scenario-based semantics to support the specification of test purposes and their automatic translation into regular expressions. We also developed this extension and integrated it into the Eclipse's OCL plug-in (version 4.×). As regards practical applications, our OCL extension is used in a test purpose framework dedicated to UML/OCL models in order to develop strategies to support the automatic testing of security properties on the smart card operating system *GlobalPlatform*.

Future work. As previously stated, adding temporal aspects to the OCL language could be a promising direction to explore model checking techniques. We intend to connect our language to usual model checking tools inspired by the work proposed by *Distefano* et al. in [12].

References

1. Projet TASCCC, Test Automatique basé sur des SCénarios et évaluation Critères Communs, `http://lifc.univ-fcomte.fr/TASCCC/`
2. Object Management Group. Object Constraint Language (February 2010), `http://www.omg.org/spec/OCL/2.2`
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Programming, pp. 411–420 (1999)
4. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. In: World Conference on Integrated Design and Process Technology, IDPT 2002, California, USA (2002)
5. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic Execution Techniques for Test Purpose Definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
6. Cabrera Castillos, K., Dadeau, F., Julliand, J., Taha, S.: Measuring Test Properties Coverage for Evaluating UML/OCL Model-Based Tests. In: Wolff, B., Zaïdi, F. (eds.) ICTSS 2011. LNCS, vol. 7019, pp. 32–47. Springer, Heidelberg (2011)
7. Baier, C., Katoen, J.P.: Principles of Model Checking. Representation and Mind Series. The MIT Press (2008)
8. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 351–357. Springer, Heidelberg (2004)
9. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 42–68. Springer, Heidelberg (2002)
10. Cengarle, M.V., Knapp, A.: Towards OCL/RT. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 390–409. Springer, Heidelberg (2002)
11. Calegari, D., Cengarle, M.V., Szasz, N.: UML 2.0 interactions with OCL/RT constraints. In: FDL, pp. 167–172 (2008)
12. Distefano, D., Katoen, J.P., Rensink, A.: On a temporal logic for object-based systems. In: Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems IV, Norwell, MA, USA, pp. 305–325 (2000)
13. Mullins, J., Oarga, R.: Model Checking of Extended OCL Constraints on UML Models in SOCLe. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 59–75. Springer, Heidelberg (2007)
14. SOCLe Project, `http://www.polymtl.ca/crac/socle/index.html`

15. Flake, S., Mueller, W.: Formal semantics of static and temporal state-oriented OCL constraints. Software and Systems Modeling (SoSyM) 2, 186 (2003)
16. Bradfield, J., Filipe, J.K., Stevens, P.: Enriching OCL Using Observational Mu-Calculus. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 203–217. Springer, Heidelberg (2002)
17. Ramakrishnan, S., Mcgregor, J.: Extending OCL to support temporal operators. In: 21st International Conference on Software Engineering (ICSE 1999) Workshop on Testing Distributed Component-Based Systems, LA, May 16-22 (1999)
18. Conrad, S., Turowski, K.: Temporal OCL: Meeting specifications demands for business components. In: Unified Modeling Language: Systems Analysis, Design, and Development Issues, pp. 151–166. Idea Publishing Group (2001)
19. Object Managment Group. UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) (November 2009)
20. Specification patterns, `http://patterns.projects.cis.ksu.edu`
21. OCL temporal extension (2012), `http://wwwdi.supelec.fr/taha/temporalocl/`
22. OCL (MDT), `http://www.eclipse.org/modeling/mdt/?project=ocl`
23. Xtext 2.1, `http://www.eclipse.org/Xtext/`
24. Spec Patterns, `http://patterns.projects.cis.ksu.edu/`
25. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 281–294. Springer, Heidelberg (2004)
26. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relations and test generation. Computer Networks and ISDN Systems 29(1), 49–79 (1996)