securityandtrust.lu

Software Verification and Validation Laboratory: *OCLR*: a More Expressive, Pattern-based Temporal Extension of OCL

Wei Dou, Domenico Bianculli and Lionel Briand

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

TR-SnT-2014-2

February 4, 2014

UNIVERSITÉ DU LUXEMBOURG

OCLR: a More Expressive, Pattern-based Temporal Extension of OCL

Wei Dou, Domenico Bianculli, and Lionel Briand

SnT Centre - University of Luxembourg, Luxembourg, Luxembourg
{wei.dou,domenico.bianculli,lionel.briand}@uni.lu

Abstract. Modern enterprise information systems often require to specify their functional and non-functional (e.g., Quality of Service) requirements using expressions that contain temporal constraints. Specification approaches based on temporal logics demand a certain knowledge of mathematical logic, which is difficult to find among practitioners; moreover, tool support for temporal logics is limited. On the other hand, a standard language such as the Object Constraint Language (OCL), which benefits from the availability of several industrial-strength tools, does not support temporal expressions.
In this paper we propose *OCLR*, an extension of OCL with support for temporal constraints based on well-known property specification patterns. With respect to previous extensions, we add support for referring to a specific occurrence of an event as well as for indicating a time distance between events and/or scope boundaries. The proposed extension defines a new syntax, very close to natural language, paving the way for a rapid adoption by practitioners. We show the application of the language in a case study in the domain of eGovernment, developed in collaboration with a public service partner.

1 Introduction

Complex software systems, such as modern enterprise information systems, call for the definition of requirements specifications that include both functional and non-functional aspects (such as QoS, Quality of Service). In both cases, the specifications might characterize (quantitative) aspects of the system that involve temporal constraints. Examples of these constraints are bounds on the sequence and/or number of occurrences of system events, possibly conjuncted with constraints on the temporal distance of events.

These types of specifications have been catalogued in various collections of property specification patterns, to help analysts and developers in expressing typical, recurrent properties of a system, using a generalized yet structured and precise form. The majority of property specification patterns have emerged in the context of concurrent, real-time critical systems [6, 12, 9], though there have been recent proposals of specification patterns for specific domains, like service-based applications [1]. In all cases, the patterns have been formalized in terms of some temporal logic, either the classic ones like LTL and CTL or a more specialized version like SOLOIST [2]. One problem in using a specification language

based on a temporal logic is that it requires a strong theoretical background, which is rarely found in practitioners. Moreover, tool support for the verification of properties expressed in temporal logic is prototypal and limited, at least if considered in the context of applying this kind of formal method at a scalable, industrial-grade level.

One of the specification languages that has found a significant consensus and adoption in industry is the Object Constraint Language (OCL) [10], used to specify constraints on models, and now a standard in the context of model-driven engineering practice. However, OCL does not support the specification of temporal requirements. There have been several research proposals to extend OCL with temporal constructs. Nevertheless, in the scope of a collaboration with a public service partner active in the domain of eGovernment, we found that the available temporal extensions of OCL do not meet the expressiveness requirements as determined in our field study, based on realistic specifications extracted from a collection of eGovernment business process descriptions.

In this paper we propose a new language, called *OCLR*, to fill the expressiveness gap that we found on the field. *OCLR* is an extension of OCL that supports temporal constraints based on some of the well-known property specification patterns. More specifically, we advance the state of the art by introducing support for referring to a specific occurrence of an event in scope boundaries as well as for indicating a time distance between events and/or scope boundaries. Our language extends OCL in a minimal fashion while maximizing the expressiveness of temporal properties; moreover, the syntax is very close to natural language, to encourage practitioners to use it. To show the feasibility of using *OCLR* in realistic scenarios, we include a case study in the context of an eGovernment application developed by our public service partner.

In the future, our intent is to adopt *OCLR* in the context of a larger project on model-driven run-time verification[1] of business processes. Since in this project we plan to leverage existing industrial-strength OCL tools, such as constraint verification engines, we decided to minimize, by design, the differences between the models underlying *OCLR* and OCL. We believe that making *OCLR* a *minimal* extension of OCL will make the translation[2] of *OCLR* expressions into regular OCL ones much easier than performing the same translation starting from expressions written in a language much more distant from OCL, such as a temporal logic.

The rest of this paper is structured as follows. In Sect. 2 We discuss the motivations for which and the context in which this work has been developed. Section 3 introduces *OCLR*, its syntax and the (informal) semantics[3]. In Sect. 4 we show the application of *OCLR* in a case study in the domain of eGovernment. We survey related work in Sect. 5. Section 6 concludes the paper, providing directions for future work.

[1] In fact *OCLR* stands for "OCL for Run-time verification".
[2] The translation from *OCLR* to OCL is out of the scope of this paper.
[3] The complete definition of the formal semantics of *OCLR* is available in the appendix.

2 Motivations

This work has been developed as part of an ongoing collaboration with CTIE (Centre des technologies de l'information de l'Etat), the Luxembourg national center for information technology. The main role of CTIE is to lead the development of electronic government (eGovernment) initiatives within Luxembourg, with the ultimate goal of delivering digital public services to citizens and enterprises, as well as improving the processes followed by the public administration.

The business processes designed for public administrations are usually highly complex and require the interaction of different stakeholders. In particular, they act as the "glue" to orchestrate different information systems, possibly by many different organizations, in an effort to foster cooperation of various administrations. Given the complexity and the many interactions foreseen for eGovernment business processes, designing effective and efficient processes to drive e-service delivery is one of the most challenging tasks for public administrations. For these reasons, their development is gradually moving towards model-driven techniques. This is the case for CTIE, which has developed in-house a model-driven methodology for designing eGovernment business processes.

Usually these processes are designed as compositions of services provided by different organizations, administrations, or third-party suppliers. A service integrator has to monitor the execution of the third-party services it uses to check whether they fulfill their obligations (both in terms of functional and non-functional properties), so that the business process itself can meet its requirements. Furthermore, it is also important to verify at run time whether the business process execution complies with the constraints specified during the modeling phase, to detect when a failure occurs and to possibly determine corrective actions. In this context, we are involved in a project on model-driven, run-time verification of (eGovernment) business processes.

One of the first steps of this project consisted in identifying the type of constraints to check at run time. We analyzed several applications developed by CTIE and scrutinized the requirements specifications associated with all use cases and business process descriptions. We were able to recast the majority of specifications written in natural language using the system of property specification patterns (and scopes) proposed by Dwyer et al. [6]. However, in some cases the original definitions proposed in [6] had to be extended to match the system specifications. For example, the definitions of property specification scopes, used to refer to the extent of a program execution over which a pattern must hold, had to be extended to support references to a specific occurrence of an event (not only the first one as in [6]), as in the requirement "event A shall occur before the *second* occurrence of event X". Another variant of this type of scope boundary that we found is the one with requirements on the distance between events, such as "event A shall occur *five time units before the second* occurrence of event X". In some cases, the requirements specifications had to be expressed in terms of some real-time specification patterns [12, 9], which quantitatively define distance among events and durations of events.

Based on the results of this phase, we pondered over the definition of a high-level specification language for expressing this type of constraints. The intrinsic temporal nature of the requirements specifications we found, including also real-time constraints, could have suggested to follow the direction of building on some temporal logic. However, specification languages based on temporal logic require a certain mathematical knowledge that is not easy and common to find among practitioners, such as business analysts or software engineers. Moreover, the array of tools available for the verification of temporal logic is limited, especially if one considers the additional requirement of applying them in realistic industrial contexts. Based on these limitations, given the model-driven engineering practice already in place at our public service partner, we decided to define our specification language as *an extension* of OCL. In this way, we can build on a language that is standardized, is known among practitioners, and has a wide set of well-established, industrial-strength tools, like constraints verification engines.

3 *OCLR*

The design of *OCLR* is based on Dwyer et al.'s property specification pattern system [6]. This system defines five scopes (*globally*, *before*, *after*, *between-and*, and *after-until*) and eight patterns (*universality*, *absence*, *existence*, *bounded existence*, *precedence*, *response*, *precedence chain*, and *response chain*).

In the definition of *OCLR* we decided to support all these scopes and patterns, with the following extensions:

- The possibility, in the definition of a scope boundary, to refer to a specific occurrence of an event, as in "before the second occurrence of event $X\ldots$". In the original definition of the pattern systems, boundaries of scopes refer implicitly to the first occurrence of an event.
- The possibility to indicate a time distance with respect to a scope boundary, as in "at least (at most) two time units before the n-th occurrence of event $X\ldots$".
- Support for expressing time distance between events occurrences, to express properties like a bounded response, such as "event B should occur in response to event A within 2 time units".

These design choices have been motivated by the type of properties that we have found while analyzing the requirement specifications of our public service partner, as well as by the lack of support for them in the current temporal extensions of OCL (see Sect. 5).

OCLR has been inspired by the design of Temporal OCL [11], another pattern-based temporal extension of OCL. As we will discuss in more detail in Sect. 5, Temporal OCL lacks the language features described above. Nevertheless, we borrow from it the notion of *event*, i.e., a predicate that specifies a set of instants within the time line; the specific types of events supported in the language are described in the following subsection.

3.1 Syntax

The syntax of *OCLR* (also inspired by the one of Temporal OCL [11]) is shown in Fig. 1: non-terminals are enclosed in angle brackets, terminals are enclosed in single quotes, and underlined italic words are non-terminals defined in the OCL grammar [10]. An ⟨*OCLR block*⟩ comprises a set of conjuncted ⟨*TemporalClauses*⟩ beginning with the keyword '`temporal`'. Each temporal clause contains a temporal expression that consists of a ⟨*scope*⟩ and a ⟨*pattern*⟩; the scope specifies the time slot(s) during which the property described by the pattern is checked.

⟨*OCLRBlock*⟩ ::= '`temporal`' ⟨*TemporalClause*⟩+
⟨*TemporalClause*⟩ ::= [⟨*simpleNameCS*⟩] '`:`' [⟨*Quantif*⟩] ⟨*TemporalExp*⟩
⟨*Quantif*⟩ ::= '`let`' ⟨*VariableDeclarationCS*⟩ '`in`'
⟨*TemporalExp*⟩ ::= ⟨*Scope*⟩ ⟨*Pattern*⟩
⟨*Scope*⟩ ::= '`globally`'
| '`before`' ⟨*Boundary1*⟩
| '`after`' ⟨*Boundary1*⟩
| '`between`' ⟨*Boundary2*⟩ '`and`' ⟨*Boundary2*⟩
| '`after`' ⟨*Boundary2*⟩ '`until`' ⟨*Boundary2*⟩
⟨*Pattern*⟩ ::= '`always`' ⟨*Event*⟩
| '`eventually`' ⟨*RepeatableEventExp*⟩
| '`never`' ['`exactly`' ⟨*IntegerLiteratureExpCS*⟩] ⟨*Event*⟩
| ⟨*EventChainExp*⟩ '`preceding`' [⟨*TimeDistanceExp*⟩] ⟨*EventChainExp*⟩
| ⟨*EventChainExp*⟩ '`responding`' [⟨*TimeDistanceExp*⟩] ⟨*EventChainExp*⟩
⟨*Boundary1*⟩ ::= [⟨*IntegerLiteratureExpCS*⟩] ⟨*SimpleEvent*⟩ [⟨*TimeDistanceExp*⟩]
⟨*Boundary2*⟩ ::= [⟨*IntegerLiteratureExpCS*⟩] ⟨*SimpleEvent*⟩ ['`at least`' *IntegerLiteratureExpCS* '`tu`']
⟨*EventChainExp*⟩ ::= ⟨*Event*⟩ ('`,`' ['`#`' ⟨*TimeDistanceExp*⟩] ⟨*Event*⟩)*
⟨*TimeDistanceExp*⟩ ::= ⟨*ComparingOp*⟩ ⟨*IntegerLiteratureExpCS*⟩ '`tu`'
⟨*RepeatableEventExp*⟩ ::= [⟨*ComparingOp*⟩ ⟨*IntegerLiteratureExpCS*⟩] ⟨*Event*⟩
⟨*ComparingOp*⟩ ::= '`at least`' | '`at most`' | '`exactly`'
⟨*Event*⟩ ::= (⟨*SimpleEvent*⟩ | ⟨*ComplexEvent*⟩) ['`|`' Event]
⟨*ComplexEvent*⟩ ::= '`isCalled`' '`(`' '`anyOp`' ['`,`' '`pre:`' ⟨*OCLExpressionCS*⟩] ['`,`' '`post:`' ⟨*OCLExpressionCS*⟩] '`)`' ['`\`' ⟨*Event*⟩]
⟨*SimpleEvent*⟩ ::= ⟨*SimpleCallEvent*⟩ | ⟨*SimpleChangeEvent*⟩
⟨*SimpleChangeEvent*⟩ ::= '`becomesTrue`' '`(`' ⟨*OCLExpressionCS*⟩ '`)`'
⟨*SimpleCallEvent*⟩ ::= '`isCalled`' '`(`' ⟨*OperationCallExpCS*⟩ ['`,`' '`pre:`' ⟨*OCLExpressionCS*⟩] ['`,`' '`post:`' ⟨*OCLExpressionCS*⟩] '`)`'

Fig. 1. Grammar of OCLR

The definitions of ⟨*Event*⟩s that can be used in a temporal expression are adapted from [11]. The keyword '`isCalled`' represents a *call event*, which corresponds to a call to an operation. Under the hypothesis of atomicity of operations, we merge into a single call event, the events corresponding to the call, the start, and the end of an operation. A call event has three parameters: the called operation; the precondition (optional, in the form of an OCL expression) that acts as guard over the system pre-state and the operation parameters for the actual call execution; the postcondition (optional, in the form of an OCL expression) that acts as guard over the system post-state and the return value of the call invocation. Notice that a call event is raised only if the operation is invoked *and* both the precondition and the postcondition are satisfied. The keyword '`anyOp`' is used if no operation is specified; in this case the call event becomes a state change event, from the state determined by the precondition to the state determined by the postcondition. The keyword '`becomesTrue`' denotes a state change event parameterized with the OCL expression provided as parameter: it corresponds to the state in which the input expression becomes true (which implies that in the previous state it evaluated to false). We also support the disjunction '`|`' and the exclusion '\' operations on events.

3.2 *OCLR* at Work

We now present some examples of properties that can be expressed with *OCLR*, in order to provide the reader with a high-level, intuitive understanding of the language. We consider the history trace shown in Fig. 2 and for each property indicate whether it is violated or not by the trace. First, we define the properties in English:

1. "Event C will happen 8 time units after the second occurrence of event X." (satisfied)
2. "Event A should happen within 30 time units after the first occurrence of event X." (satisfied)
3. "Event C will eventually happen after at least 3 time units since the first occurrence of event X; and it must happen before event Y if the latter happens." (violated)
4. "After the second occurrence of event X, event C will eventually happen exactly twice." (satisfied)
5. "Event C should happen at least once between every first occurrence of event X and the next event Y; the time interval between event X and the first occurrence of event C should be at least 5 time units." (violated)

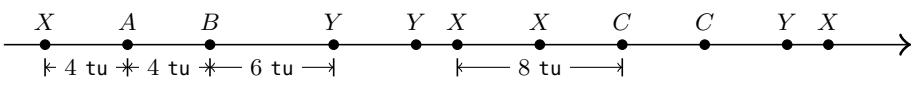

Fig. 2. Sample events traces

6. "Event B must happen at least 3 time units before the first occurrence of event Y." (satisfied)
7. "Before the first occurrence of event Y, once event X occurs, event A will happen followed by event B; the time interval between X and A is at least 3 time units." (satisfied)

The corresponding *OCLR* expressions are shown below:

1. `temporal: after 2` X `exactly 8 tu eventually` C
2. `temporal: after` X `at most 30 tu eventually` A
3. `temporal: after 1` X `at least 3 tu until` Y `eventually` C
4. `temporal: after 2` X `eventually exactly 2` C
5. `temporal: between` X `at least 5 tu and` Y `eventually at least 1` C
6. `temporal: before` Y `at least 3 tu eventually` B
7. `temporal: before` Y A`,` B `responding at least 3 tu` X

3.3 Informal Semantics

In this section we present the informal semantics of the scopes and the patterns supported in *OCLR* expressions; they correspond to non-terminals ⟨*Scope*⟩ and ⟨*Pattern*⟩, respectively. The full definition of the formal semantics is available in the appendix.

Scopes. For the description of scopes, we refer to the trace of events depicted in Fig. 3. We use symbols X and Y as shorthands for events that can be derived from the non-terminal ⟨*SimpleEvent*⟩.

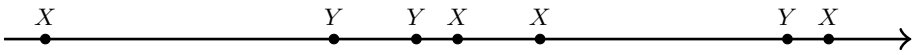

Fig. 3. A sample trace for the description of scopes

Before. This scope identifies a portion of a trace up to a certain boundary. The general template for this scope in *OCLR* is "`before` [`m`] X [⟨*ComparingOp*⟩ `n tu`]", where elements between brackets are optional, '`m`' and '`n`' are integers derived from the non-terminal ⟨*IntegerLiteratureExpCS*⟩, and '`tu`' stands for "time unit(s)". This template can be expanded in four forms: 1) "`before` X", 2) "`before` X ⟨*ComparingOp*⟩ `n tu`", 3) "`before m` X", 4) "`before m` X ⟨*ComparingOp*⟩ `n tu`". The first two forms are convenient shorthands for the third and fourth ones, respectively, with `m` = 1. The form "`before m` X" selects the portion of the trace up to the m-th occurrence of event X; see, for example, the top row in Fig. 4, where the interval from the origin of the trace up to the third occurrence of X is highlighted with a thick line. The form "`before m` X ⟨*ComparingOp*⟩ `n tu`" has three variants, depending on the possible expansions of non-terminal ⟨*ComparingOp*⟩:

- "`before m` X `at least n tu`" identifies the scope from the origin of the trace up to n time units before the m-th occurrence of X;
- "`before m` X `at most n tu`" identifies the scope starting at n time units before the m-th occurrence of X and bounded to the right by the m-th occurrence of X;
- "`before m` X `exactly n tu`" pinpoints the time instant at n time units before the m-th occurrence of X.

Examples of these three variants of scopes are shown with thick segments in Fig. 4, with `m` $= 3$ and with `n` $= 2$.

After. This scope identifies a portion of a trace starting from a certain boundary. It has a dual semantics with respect to the *before* scope. We provide an intuition of its semantics using Fig. 5, where the possible variants of this scope are represented as thick segments.

Between-And. This scope identifies portion(s) of a trace delimited by two boundaries. The general template for this scope in *OCLR* is "`between` [`m1`] X [`at least n1 tu`] `and` [`m2`] Y [`at least n2 tu`]", where elements between brackets are optional, '`m1`', '`m2`', '`n1`', '`n2`' are integers derived from the non-terminal ⟨*IntegerLiteratureExpCS*⟩, and '`tu`' stands for "time unit(s)". This template can be expanded in four forms:

- "`between` m_1 X [`at least` n_1 `tu`] `and` m_2 Y [`at least` n_2 `tu`]";
- "`between` X [`at least` n_1 `tu`] `and` m_2 Y [`at least` n_2 `tu`]";
- "`between` m_1 X [`at least` n_1 `tu`] `and` Y [`at least` n_2 `tu`]";
- "`between` X [`at least` n_1 `tu`] `and` Y [`at least` n_2 `tu`]".

The first form is the most general: it selects the single segment of the trace delimited by the m_1-th occurrence of event X and the m_2-th occurrence of event Y happening after the m_1-th occurrence of X. The second and third forms are shorthands for the first one, with `m1` $= 1$ and `m2` $= 1$, respectively. The fourth form is the closest to the original definition in [6], since it selects all the segments in the trace delimited by the boundaries. In this regard, notice the difference with respect to the expression "`between 1` X `and 1` Y", which selects the segment delimited by the first occurrence of X and the first occurrence of Y after X. In all forms it is possible to use the expression `at least n tu` when defining boundaries, with the same meaning described for the scope *before*. Four examples of the *Between-and* scope are shown in Fig. 6.

After-Until. This scope is similar to *Between-and*, with the difference that each identified segment extends to the right in case the event defined by the second boundary does not occur; this peculiarity can be noticed in the first two rows of Fig. 7, and compared with those in Fig. 6.

Globally. This scope corresponds to the entire trace shown in Fig. 3.

Note that all scopes but those using the '`exactly`' keyword do not include the events occurring at the boundaries of the scope itself.

Patterns. *OCLR* supports the eight patterns defined in [6].

Universality. It states that a certain event should *always* happen within the given scope.

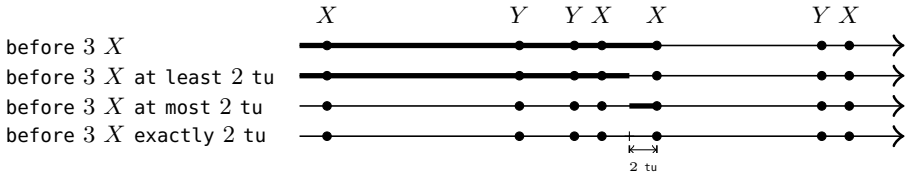

Fig. 4. Scope: before

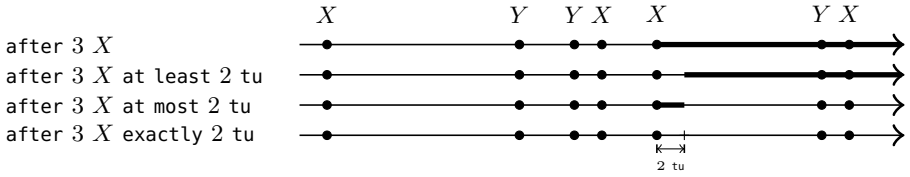

Fig. 5. Scope: after

Existence. It indicates that the given scope contains some occurrence(s) of a certain event. This pattern comes in four forms:

– "`eventually` A" means that the event A happens at least once;
– "`eventually at least m` A" means that A happens at least m times;
– "`eventually at most m` A" means that A happens at most m times;.
– "`eventually exactly m` A" means that A happens exactly m times.

The last three forms are variants of the *bounded existence* pattern, a subclass of the *existence* one.

Absence. It states that a certain event *never* occurs in the given scope. It is also possible to specify that a specific number of occurrences of the same event should not happen, as in "`never exactly 2` X", which says that X should never occur exactly twice.

Precedence. This pattern (also available in the variant called *precedence chain*) indicates the precondition relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the second event

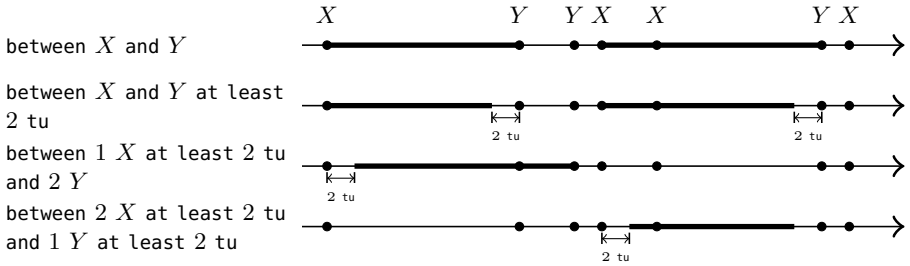

Fig. 6. Scope: between-and

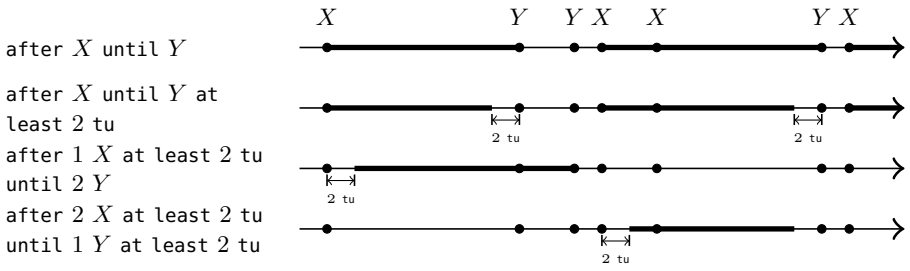

Fig. 7. Scope: after-until

(respectively, block) depends on the occurrence of the first event (respectively, block). Based on this original definition, we added support for timing information to enable expressing the time distance between two adjacent events. The semantics can be explained using the following example and the event trace in Fig. 8; the expression "A `preceding at most 10 tu` B, #`at least 5 tu` C" indicates that the event A is the precondition of the block "B followed by C", that the time distance between A and B is at most 10 time units, and the time distance (expressed using the `#` operator) between events B and C is at least 5 time units. Here, A (at the left of '`preceding`') represents the first block of the chain, while the expression "B, #`at least 5 tu` C" represents the second block (at the right of '`preceding`').

Response. This pattern (also available in the variant called *response chain*) specifies the cause-effect relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the first event (respectively, first block) leads to the occurrence of the second event (respectively, second block). The property "C, D `responding at most 10 tu` A, #`at least 5 tu` B" specifies that two successive events A and B stimulate the sequential occurrence of C and D, and the time interval between A and B should be at least 5 time units; the time interval between B (second element of the first block) and C (first element of the second block) should be at most 10 time units. This property is violated by the example in Fig. 8, because the time distance between A and B is only 4 time units.

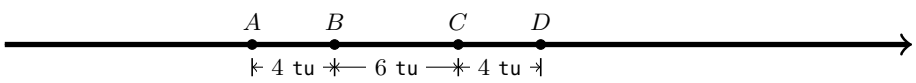

Fig. 8. Example trace for illustrating the precedence and response patterns

4 Applying *OCLR* in an eGovernment scenario

In this section we present a case study where we show the use of *OCLR* in the context of an eGovernment application developed by our public service partner. We illustrate some properties (selected from the 47 we analyzed) of a business process model related to three use cases. The goal is to investigate whether *OCLR* can precisely capture all temporal and timed properties of a real eGovernment system. The case study description has been sanitized for the purpose of not disclosing confidential information and also to obtain a model at the minimum level of detail required to illustrate and express the properties.

The scenario describes the *Identity Card Management (ICM)* business process, which is in charge of issuing and managing the ID cards of the diplomatic personnel of the country. A sanitized version of the conceptual model corresponding to this scenario is shown in Fig. 9. The *ICM* business process deals with the card requests, the production of the cards, and the returns of the cards once expired. The *ICM* process also keeps track of the state of a card (`CardState`), which can be, for example, `InCirculation` or `Expired`. A card `Request` can be in different states, such as `Approved`, `Denied`, and `InProgress`. Once a request for a card is submitted to the *ICM* system, it is evaluated and then either approved or denied. After the approval, the *ICM* system asks the production system to issue a physical card. The card will then be delivered to the applicant. The *ICM* also deals with events such as the damage, loss, or expiration of cards.

Sample properties. We now list the requirements specifications associated with three uses cases of the *ICM* system, and show how the corresponding properties can be expressed in *OCLR*.

Card Request. The following requirements are associated with the use case related to the card request:

R1 Once a card request is approved, the applicant is notified within three days; this notification has to occur before the production of the card is started.

R2 The applicant has to show up within five days from the notification to get her personal data collected.

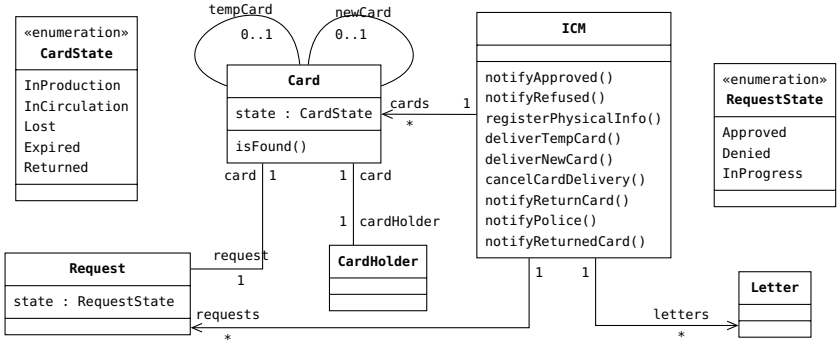

Fig. 9. Conceptual model of the ICM process

R3 If the applicant does not show up within five days after the second notification, the request will be denied and the applicant notified about the refusal.

```
1  context ICM
2  temporal R1: let r : Request in
3      before becomesTrue(r.card.state = CardState::InProduction)
4      isCalled(notifyApproved(r.applicant)) responding at most 3*24*3600 tu
5      becomesTrue(r.state = RequestState::Approved)
6  temporal R2: let r : Request in
7      after isCalled(notifyApproved(r.applicant)) at most 5*24*3600 tu
8      eventually isCalled(registerPhysicalInfo(r.applicant))
9  temporal R3: let r : Request in
10    after 2 isCalled(notifyApproved(r.applicant)) at least 5*24*3600 tu
11    eventually isCalled(notifyRefused(r.applicant))
```

Property R1 is expressed in lines 2–5. The *before* scope is delimited by the event that corresponds to a change in the state of the card (`c.state=CardState::InProduction`). The *response* pattern is bounded (time units are expressed in seconds) and requires the notification to the applicant (`notifyApproved`) to happen in response to a change in the state of the request (`r.state=RequestCard::Approved`). Property R2 (lines 6–8) combines an *after* scope with an *existence* pattern. A similar structure is used in R3 (lines 9–11), where the *after* scope uses the second occurrence of `notifyApproved` as the boundary.

Card Loss. The following requirements are associated with the use case related to the loss of a card:

L1 If a card is reported as lost to the *ICM* and has not been found yet, a temporary card will be sent to the card holder within 1 day.

L2 If the card has not been found yet, a new card will be delivered to the holder within five days after the report of the loss.

L3 After the card loss is reported, if the card is found, within at most three days the delivery of the new card will be canceled and a notification to return the temporary card will be sent.

```
1  context ICM
2  temporal L1: let c: Card in
3    after becomeTrue(c.state = CardState::Lost) at most 24*3600 tu
4    eventually isCalled(deliverTempCard(c.tempCard,
5                        pre: c.state=CardState::Lost))
6  temporal L2: let c : Card in
7    after becomeTrue(c.state = CardState::Lost) at most 5*24*3600 tu
8    eventually isCalled(deliverNewCard(c.newCard),
9                        pre: c.state=CardState::Lost)
10 temporal L3: let c: Card in
11   after becomeTrue(c.state = CardState::Lost)
12   isCalled(c.isFound(), pre: c.state=CardState::Lost)
13   preceding at most 3*24*3600 tu
14   isCalled(cancelCardDelivery(c.newCard),
15            pre: c.newCard.state <> CardState::InCirculation),
16   isCalled(notifyReturnCard(c.cardHolder))
```

Both properties L1 and L2 use an *after* scope combined with an *existence* pattern. Notice that in both cases the additional condition "card not found yet" is expressed as a precondition of the operation which is the argument of `isCalled` in the *existence* pattern (`deliverTempCard` and `deliverNewCard`). Property L3 combines an *after* scope with a *precedence chain* pattern, where the first block corresponds to finding the card (`isFound`) and the second block is the chain of `cancelCardDelivery` and `notifyReturnCard`.

Card Expiration. The following requirements are associated with the use case related to the expiration of a card:

E1 Once a card expires, the holder is notified to return the card at most twice.
E2 After five days from the second notification to the holder about the expiration of the card, if the card has not been returned yet, the police is notified.
E3 Once a card is returned, the holder will receive a confirmation within one day.

```
context ICM
temporal E1: let c:Card in
  after becomesTrue(c.state = CardState::Expired)
  until becomesTrue(c.state = CardState::Returned)
  eventually at most 2 isCalled(notifyReturnCard(c.cardHolder),
                                pre:c.state <> CardState::Returned)
temporal E2: let c:Card in
   after 2 isCalled(notifyReturnCard(c.cardHolder),
                    pre: c.state <> CardState::Returned)
   exactly 5*24*3600 tu
   eventually isCalled(notifyPolice(c.cardHolder),
                       pre: c.state <> CardState::Returned)
temporal E3: let c:Card in
  globally isCalled(notifyCardReturned(c.cardHolder),
                    pre: c.state = CardState::Returned)
  responding at most 24*3600 tu
  becomesTrue(c.state = CardState::Returned)
```

Property E1 uses an *after-until* scope, delimited by the events corresponding to the expiration of the card (`c.state=CardState::Expired`) and the return of the card (`c.state=CardState::Returned`). A *bounded existence* pattern is used to specify the maximum amount of notifications (`notifyReturnCard`) that can happen. In property E2 we use an *after* scope combined with the keyword '`exactly`' to pinpoint the exact time instant in which the police is notified (`notifyPolice`). Property E3 states an invariant of the system (using the *globally* scope) for the *response* pattern correlating the return of the card (`c.state=CardState::Returned`) to the notification to the holder (`notifyCardReturned`).

5 Related Work

There have been several proposals for extending OCL with support for temporal constraints. In the rest of this section we summarize them and discuss their differences and limitations with respect to *OCLR*.

One of the first proposals is OCL/RT [4], which extends OCL with the notion of timestamped events (based on the original UML abstract meta-class `Event`) and two temporal modalities, "always" and "sometimes". Events are associated with instances of classifiers and, by means of a special satisfaction operator, it is possible to evaluate an expression at the time instant when a certain event occurred. The OCL/RT extension allows for expressing real-time deadline and timeout constraints but requires to reason explicitly at the lowest-level of abstraction, in terms of time instants.

Cabot et al. [3] extend UML to use UML/OCL as a temporal conceptual modeling language, introducing the concepts of *durability* and *frequency* for the definition of temporal features of UML classifiers and associations. They define temporal operations in OCL through which it is possible to refer to any past state of the system. These operations are mapped into standard OCL by relying on the mapping of the temporally-extended conceptual schema into a conventional UML one, which explicitly instantiates the concepts of time interval and instant. However, the temporal operations are geared to express temporal integrity constraints on the model, rather than temporal properties correlating events of the system.

The majority of the proposals regarding temporal extensions of OCL are realized by extending the language with temporal operators/modalities borrowed from standard temporal logic, such as "always", "until", "eventually", "next". A preliminary work in this direction appeared in [5]. Lavazza et al. [13] define the Object Temporal Logic (OTL), which allows users to write temporal constraints on Real-time UML (UML-RT) models. In particular, it supports the concepts *Time*, *Duration* and *Interval* to specify the time distance between events. Nevertheless, the language is modeled after the TRIO temporal logic [14], and the properties are written using a low level of abstraction. Ziemann and Gogolla [17] proposes TOCL, an extension of OCL with elements of a linear temporal logic, to specify constraints on the temporal evolution of the system states. Being based on linear temporal logic, TOCL does not support real-time constraints. The work on Flake and Mueller [7] goes in a similar direction, proposing an extension of OCL that allows for the specification of past- and future-oriented time-bounded constraints. They do not support event-based specifications; moreover, the proposed mapping into Clocked LTL does not allow to rely on standard OCL tools. Kuester-Filipe and Anderson propose a liveness template for future-oriented time-bounded constraints, as those than can be captured with a *response* or *existence* pattern. This template is defined in terms of the real-time temporal logic of knowledge, interpreted over timed automata, to allow for formal reasoning. The expressiveness of this extension is very limited, since it supports only one template. Soden and Eichler [16] propose Linear Temporal OCL (LT-OCL) for languages defined over MOF meta-models in conjunction with operational semantics. LT-OCL contains the standard modalities of Linear Temporal Logic. The interpretation of LT-OCL formulae is defined in the context of a MOF meta-model and its dynamic behavior specified by action semantics using the M3Actions framework.

The approaches that are most similar to *OCLR* are those that extend OCL with support for Dwyer et al.'s property specification patterns [6]. Flake and Mueller [8] propose a state-oriented temporal extension of OCL for user-defined classes that have an associated Statechart. The pattern-based temporal expressions refer to configurations of Statecharts. With respect to *OCLR*, they do not support the specification in terms of events. Moreover, the expressions corresponding to the patterns are not first-class entities of the language, hence they are more verbose and less close to natural language. Robinson [15] presents a temporal extension of OCL called OCL_{TM}, developed in the context of a framework for monitoring of requirements expressed using a goal model. The temporal extension of OCL includes all the operators corresponding to standard LTL modalities, support for Dwyer et al.'s patterns and for timeouts in patterns. In this regard, it is very close to the expressiveness of *OCLR*, though it supports neither the reference to a specific occurrence of an event in scope boundaries nor the association of time shifts to boundaries (as *OCLR* does with the keywords '`at least`', '`at most`', '`exactly`'). Kanso and Taha [11] introduce Temporal OCL, a pattern-based temporal extension of OCL. As discussed in Sect. 3, *OCLR* borrows some language entities from Temporal OCL. Although the support for temporal patterns is very similar between the two languages, Temporal OCL does not allow references to specific event occurrences in scope boundaries and it lacks support for timing information, such as the distance between events and the distance from a scope boundary.

6 Conclusion and Future Work

A broad class of requirements for modern complex software systems involves temporal constraints, possibly enriched with timing information. Current approaches for specifying requirements either lack the expressiveness (as in the case of OCL) required for this new class of properties or require mathematical expertise (e.g., temporal logic). In this paper we presented *OCLR*, a novel temporal extension of OCL based on common property specification patterns, and extended with support for referring to a specific occurrence of an event in scope boundaries, and for specifying the distance between events and/or boundaries of the scope of a pattern. We presented the semantics of the language and its application to a case study in the domain of eGovernment.

This work has been developed as part of a broader collaboration with our public service partner CTIE, the Luxembourg state center for information technology, in the context of a project on model-based run-time verification of eGovernment business processes. We are currently working on defining the mapping between *OCLR* and OCL, in order to take advantage of the industrial-strength tools available to check OCL constraints. Our next steps will focus on defining a model-based run-time verification technique for properties written with *OCLR*, and integrating it in the business process run-time platform of our partner. We also plan to conduct an empirical study to assess the improvements provided by *OCLR* when adopted as specification language in the development life cy-

cle of our partner, and also to improve the language, integrating feedback from practitioners and adding support for other specification patterns [1].

Acknowledgments. This work has been supported by the National Research Fund, Luxembourg (FNR/P10/03). We would like to thank the members of the Prometa team at CTIE, in particular Lionel Antunes, Ludwig Balmer, Henri Meyer, Manuel Rouard, for their help with the analysis of the case study.

References

1. Bianculli, D., Ghezzi, C., Pautasso, C., Senti, P.: Specification patterns from research to industry: a case study in service-based applications. In: Proc. ICSE 2012. pp. 968–976. IEEE (2012)
2. Bianculli, D., Ghezzi, C., San Pietro, P.: The tale of SOLOIST: a specification language for service compositions interactions. In: Proc. FACS'12. LNCS, vol. 7684, pp. 55–72. Springer (2013)
3. Cabot, J., Olivé, A., Teniente, E.: Representing temporal information in UML. In: UML 2003, LNCS, vol. 2863, pp. 44–59. Springer (2003)
4. Cengarle, M., Knapp, A.: Towards OCL/RT. In: Proc. FME, LNCS, vol. 2391, pp. 390–409. Springer (2002)
5. Conrad, S., Turowski, K.: Temporal OCL: Meeting specification demands for business components. In: Unified Modeling Language: System Analysis, Design, and Development Issues, pp. 151–165. IGI Global (2001)
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. ICSE 1999. pp. 411–420. IEEE (1999)
7. Flake, S., Mueller, W.: Past- and future-oriented time-bounded temporal properties with OCL. In: Proc. SEFM 2004. pp. 154–163. IEEE (2004)
8. Flake, S., Müller, W.: Expressing property specification patterns with OCL. In: Software Engineering Research and Practice. pp. 595–603. CSREA Press (2003)
9. Gruhn, V., Laue, R.: Patterns for timed property specifications. Electron. Notes Theor. Comput. Sci. 153(2), 117–133 (2006)
10. Object Constraint Language (2012), http://www.omg.org/spec/OCL/ISO/19507/
11. Kanso, B., Taha, S.: Temporal constraint support for ocl. In: Proc. SLE 2012. LNCS, vol. 7745, pp. 83–103. Springer (2013)
12. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proc. ICSE '05. pp. 372–381. ACM (2005)
13. Lavazza, L., Morasca, S., Morzenti, A.: A dual language approach extension to UML for the development of time-critical component-based systems. Electron. Notes Theor. Comput. Sci. 82(6), 121–132 (2003)
14. Morzenti, A., Mandrioli, D., Ghezzi, C.: A model parametric real-time logic. ACM Trans. Program. Lang. Syst. 14, 521–573 (October 1992)
15. Robinson, W.N.: Extended OCL for goal monitoring. ECEASST 9 (2008)
16. Soden, M., Eichler, H.: Temporal extensions of OCL revisited. In: Proc. ECMDA-FA, LNCS, vol. 5562, pp. 190–205. Springer (2009)
17. Ziemann, P., Gogolla, M.: OCL extended with temporal logic. In: Perspectives of System Informatics, LNCS, vol. 2890, pp. 351–357. Springer (2003)

A Formal Semantics

This section presents the formal semantics of *OCLR*, using the concept of temporal linear traces.

A.1 Trace

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* is the pre-condition satisfied in the pre-state and *post* is the post-condition satisfied in the post-state.

Definition 2 (Poset of atomic events). *Let $\prec$ be the strict partial ordering on the alphabet of atomic events Σ. The poset of atomic events is denoted as $(\Sigma, \prec)$, in which for any $e_1 \neq e_2$, $e_1 \prec e_2$ indicates that e_1 occurs before e_2. Thus the corresponding non-strict partial ordering of the atomic events is denoted as $\preccurlyeq$.*

Definition 3 (Time distance). *Let $(\Sigma, \prec)$ be the poset of atomic events. The timestamp of an atomic event is the absolute occurrence time that is counted in* time unit *(tu); it is defined by the function $\tau : \Sigma \to \mathbb{N}^+$. The time distance between two different atomic events is the difference between their timestamp, i.e., $d(e_i, e_j) = \tau(e_j) - \tau(e_i)$, $e_i \prec e_j$.*

Definition 4 (Trace). *Let $(\Sigma, \prec)$ be the poset of atomic events. A trace λ is a finite timeline comprising a sequence of atomic events denoted as $(e_0, \ldots, e_{n-1})$ in which e_0 is its starting event and n is the length. The universal set of sub-traces is denoted as Λ.*

Given an n-length trace λ,

- the atomic event at index i is denoted as $\lambda(i)$;
- the time distance between $\lambda(i)$ and $\lambda(j)$ is denoted as $d(i, j)$ $(1 \leq i \leq j \leq n-1)$;
- the sub-trace from $\lambda(i)$ to $\lambda(j)$ is denoted as $\lambda(i:j)$ $(0 \leq i \leq j \leq n-1)$;
- a forward shift of an atomic event by t time units is denoted as $\lambda(i > t)$;
- a backward shift of an atomic event by t time units is denoted as $\lambda(i < t)$;
- a time constraint sub-trace $\lambda(i:j)$, $0 \leq i \leq j \leq n-1$ (i.e., left boundary shifts forwards by time distance t_1 and right boundary shifts backwards by time distance t_2) is denoted as $\lambda(i > t_1, j < t_2)$;

A.2 Event

AnyCallEvent Let Σ be the alphabet of atomic events, $\mathbb{O}$ be the set of all operations, $\mathbb{E}$ be the set of all OCL expressions. *AnyCallEvent* identifies any operation satisfied by a pair of pre- and post-conditions, which is defined by:

$$\begin{aligned}\text{AnyCallEvent} &= \text{isCalled}(\texttt{anyOp}, \texttt{pre}, \mathit{post}) \\ &= \{(o,p,q) \in \Sigma \mid o \in \mathbb{O}, p \in \mathbb{E}, q \in \mathbb{E} \text{ and } p \Rightarrow \mathit{pre}, q \Rightarrow \mathit{post}\}\end{aligned}$$

SimpleEvent Let Σ be the alphabet of atomic events, $\mathbb{O}$ be the set of all operations and $\mathbb{E}$ be the set of all OCL expressions. A *SimpleEvent* can be either a *SimpleCallEvent* or a *SimpleChangeEvent* defined below.

$$\begin{aligned}\text{SimpleCallEvent} &= \text{isCalled}(\mathit{op}, \mathit{pre}, \mathit{post}) \\ &= \{(o,p,q) \in \Sigma \mid \\ &\quad o \in \mathbb{O}, p \in \mathbb{E}, q \in \mathbb{E} \text{ and } o = \mathit{op}, p \Rightarrow \mathit{pre}, q \Rightarrow \mathit{post}\} \\ \text{SimpleChangeEvent} &= \text{becomesTrue}(P) \equiv \text{isCalled}(\texttt{anyOp}, \neg P, P)\end{aligned}$$

Hence a *SimpleChangeEvent* is defined by a special *AnyCallEvent* in which an OCL boolean expression P becomes true after some operation call that is not specified (`anyOp`).

Two binary operators are defined on events.

Disjunction: $E_1|E_2$ means E_1 occurs or E_2 occurs.

Exclusion: $E_1 \setminus E_2$ means E_1 occurs and E_2 does not occur.

ComplexEvent *ComplexEvent* is defined by:

$$\begin{aligned}\text{ComplexEvent} &= \text{AnyCallEvent} \\ &\quad \text{or AnyCallEvent} \setminus \text{Event}\end{aligned}$$

The operator *negation* can be expressed as a *ComplexEvent*.

Negation: `not` $E \equiv$ isCalled(anyOp, true, true) $\setminus E$

Event An event E can be specified by a *SimpleEvent* or a *ComplexEvent*, or either of them in disjunction with another event. It is defined by:

$$\begin{aligned}\text{Event} &= \text{SimpleEvent or ComplexEvent} \\ &\quad \text{or SimpleEvent} \mid \text{Event or ComplexEvent} \mid \text{Event}\end{aligned}$$

EventChain An *EventChain* is a chain of *Events* occurring in sequence with optional quantification of time distance between each pair of adjacent elements; it can degrade to a single *Event*. An m-length EventChain ($m > 1$) is denoted as $E_1, t_1, E_2, \ldots, t_{m-1}, E_m$. Therein $t_i (1 \leq i \leq m-1)$ represents the quantification of time distance between E_i, E_{i+1}, if it is available, having the form $t_i = \# \bowtie_i \delta_i$ `tu`, $\delta_i \in \mathbb{N}^+$, and the range of $\bowtie_i$ is {`at least, at most, exactly`}.

EventChain matching function Let λ be an n-length trace, E be a single element EventChain, $E_1, d_1, \ldots, E_{m-1}, d_{m-1}, E_m$ be an m-length EventChain ($m > 1$). The matching function **match** is defined for checking the occurrence of an EventChain over the trace.

$$\begin{cases} \text{match}(\lambda, E) \\ \qquad \Leftrightarrow \exists i, 0 \le i \le n-1, \lambda(i) \in E \\ \text{match}(\lambda, E_1, t_1, E_2, \ldots, t_{m-1}, E_m) \\ \qquad \Leftrightarrow \exists i_1, i_2, \ldots, i_m, 0 \le i_1 < i_2 < \cdots < i_m \le n-1, \\ \qquad\quad \lambda(i_1) \in E_1, \lambda(i_2) \in E_2, \ldots, \lambda(i_m) \in E_m \\ \qquad\quad \text{and } \forall j, 1 \le j \le m-1, \begin{cases} \mathrm{d}(i_j, i_{j+1}) \ge \delta_j \text{ if } \bowtie_j = \text{at least;} \\ \mathrm{d}(i_j, i_{j+1}) \le \delta_j \text{ if } \bowtie_j = \text{at most;} \\ \mathrm{d}(i_j, i_{j+1}) = \delta_j \text{ if } \bowtie_j = \text{exactly.} \end{cases} \end{cases}$$

where $t_j \in \{\text{“”}\} \cup \{\# \bowtie_j \delta_j \texttt{ tu} \mid 1 \le j \le m-1, \bowtie_j \in \{\texttt{ at least, at most, exactly}\}, \delta \in \mathbb{N}^+\}$.

Moreover, two functions **first**($EventChain$) and **last**($EventChain$) are defined to get the first and the last event, respectively.

A.3 Temporal Expressions

The semantics of temporal expressions considers the time distance, the events, and the trace. We assume the following types and ranges for the variables used in the semantics definition below.

- E, E_1, E_2 (for scopes) : SimpleEvent
- E (for patterns) : Event
- EC_1, EC_2 (for patterns) : EventChain
- a, c : $\{i \mid i \in \mathbb{N}^+ \text{ and } 0 \le i \le n-1\} \cup \{\text{“”}\}$
- b, d : $\mathbb{N}^+$
- $\alpha,\ \beta,\ \gamma,\ \theta,\ \delta,\ \eta,\ i,\ k,\ k_1,\ k_2,\ m,\ x,\ y : \{i \mid i \in \mathbb{N}^+ \text{ and } 0 \le i \le n-1\}$
- $|\lambda(i) \in E| = \begin{cases} 0, & \lambda(i) \notin E; \\ 1, & \lambda(i) \in E. \end{cases}$
- $\bowtie$: {at least, at most, exactly}
- $\triangle : \{\le, <, >, \ge, =, \ne\}$

Scopes Let $\mathbb{S}$ be the set of scopes defined in the grammar. A scope $s \in \mathbb{S}$ is a set of sub-traces of an n-length trace $\lambda \in \Lambda$ defined by the function $\phi_{[s]}(\lambda) : \Lambda \to 2^{\Lambda}$ as follows:

- $\phi_{[\texttt{globally}]}(\lambda) = \{\lambda\}$
- $$\begin{cases} \phi_{[\texttt{before } a\ E]}(\lambda) \\ \qquad = \left\{ \lambda(0:\gamma) \mid \lambda(\gamma) \in\ E \text{ and } \sum_{k=0}^{\gamma-1} |\lambda(k) \in E| = m \right\} \\ \phi_{[\texttt{before } a\ E \bowtie b\ \texttt{tu}]}(\lambda) \\ \qquad = \left\{ \lambda(\alpha:\beta) \mid \exists \theta, \lambda(\theta) \in E \text{ and } \sum_{k=0}^{\theta-1} |\lambda(k) \in E| = m \right\} \end{cases}$$

where

$$m = \begin{cases} 0, & \text{if a=“”;} \\ 0, & \text{if a=1;} \\ a-1, & \text{if } a > 1. \end{cases} \quad , \; \alpha = \begin{cases} 0, & \text{if } \bowtie = \text{at least;} \\ \theta < b, & \text{if } \bowtie = \text{at most; ,} \\ \theta < b, & \text{if } \bowtie = \text{exactly.} \end{cases}$$

$$\beta = \begin{cases} \theta, & \text{if } \bowtie = \text{at least;} \\ \theta, & \text{if } \bowtie = \text{at most;} \\ \theta < b, & \text{if } \bowtie = \text{exactly.} \end{cases}$$

$$- \begin{cases} \phi_{\texttt{[after } a \; E\texttt{]}}(\lambda) \\ \qquad = \left\{ \lambda(\gamma : n-1) \mid \lambda(\gamma) \in \; E \text{ and } \sum_{k=0}^{\gamma-1} |\lambda(k) \in E| = m \right\} \\ \phi_{\texttt{[after } a \; E \bowtie b \texttt{ tu]}}(\lambda) \\ \qquad = \left\{ \lambda(\alpha : \beta) \mid \exists \theta, \lambda(\theta) \in E \text{ and } \sum_{k=0}^{\theta-1} |\lambda(k) \in E| = m \right\} \end{cases}$$

where

$$m = \begin{cases} 0, & \text{if a=“”;} \\ 0, & \text{if a=1;} \\ a-1, & \text{if } a > 1. \end{cases} \quad , \; \alpha = \begin{cases} \theta > b, & \text{if } \bowtie = \text{at least;} \\ 0, & \text{if } \bowtie = \text{at most; ,} \\ \theta > b, & \text{if } \bowtie = \text{exactly.} \end{cases}$$

$$\beta = \begin{cases} n-1, & \text{if } \bowtie = \text{at least;} \\ \theta > b, & \text{if } \bowtie = \text{at most;} \\ \theta > b, & \text{if } \bowtie = \text{exactly.} \end{cases}$$

$$- \begin{cases} \phi_{\texttt{[between } E_1 \texttt{ and } E_2\texttt{]}}(\lambda \\ \qquad = \{\lambda(\alpha : \beta) \mid \lambda(\alpha) \in E_1, \lambda(\beta) \in E_2 \\ \qquad\quad \text{and } \forall k, 0 \leq \alpha < k < \beta \leq n-1, \lambda(k) \notin E_2 \\ \qquad\quad \text{and if } \exists i < \alpha, \lambda(i) \in E_1, \exists j, i < j < \alpha, \lambda(j) \in E_2\} \\ \phi_{\texttt{[between } E_1 \texttt{ and } E_2 \texttt{ at least } d \texttt{ tu]}}(\lambda) \\ \qquad = \{\lambda(\alpha : \beta) \mid \lambda(\alpha) \in E_1, \beta = \delta < d, \lambda(\delta) \in E_2 \\ \qquad\quad \text{and } \forall k, 0 \leq \alpha < k < \delta \leq n-1, \lambda(k) \notin E_2 \\ \qquad\quad \text{and if } \exists i < \alpha, \lambda(i) \in E_1, \exists j, i < j < \alpha, \lambda(j) \in E_2\} \\ \phi_{\texttt{[between } E_1 \texttt{ at least } b \texttt{ tu and } E_2\texttt{]}}(\lambda) \\ \qquad = \{\lambda(\alpha : \beta) \mid \alpha = \theta > b, \lambda(\theta) \in E_1, \lambda(\beta) \in E_2 \\ \qquad\quad \text{and } \forall k, 0 \leq \theta < k < \beta \leq n-1, \lambda(k) \notin E_2 \\ \qquad\quad \text{and if } \exists i < \theta, \lambda(i) \in E_1, \exists j, i < j < \theta, \lambda(j) \in E_2\} \\ \phi_{\texttt{[between } E_1 \texttt{ at least } b \texttt{ tu and } E_2 \texttt{ at least } d \texttt{ tu]}}(\lambda) \\ \qquad = \{\lambda(\alpha : \beta) \mid \alpha = \theta > b, \lambda(\theta) \in E_1, \beta = \delta < d, \lambda(\delta) \in E_2 \\ \qquad\quad \text{and } \forall k, 0 \leq \theta < k < \delta \leq n-1, \lambda(k) \notin E_2 \\ \qquad\quad \text{and if } \exists i < \theta, \lambda(i) \in E_1, \exists j, i < j < \theta, \lambda(j) \in E_2\} \end{cases}$$

$$
- \begin{cases}
\phi_{[\texttt{between } a\ E_1 \texttt{ and } c\ E_2]}(\lambda) \\
\quad = \Big\{ \lambda(\alpha:\beta) \mid \lambda(\alpha) \in E_1, \sum_{k_1=0}^{\alpha-1} |\lambda(k_1) \in E_1| = x \\
\quad\quad \text{and } \lambda(\beta) \in E_2, \sum_{k_2=\alpha+1}^{\beta-1} |\lambda(k_2) \in E_2| = y \Big\} \\
\phi_{[\texttt{between } a\ E_1 \texttt{ and } c\ E_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \\
\quad = \Big\{ \lambda(\alpha:\beta) \mid \lambda(\alpha) \in E_1, \sum_{k_1=0}^{\alpha-1} |\lambda(k_1) \in E_1| = x \\
\quad\quad \text{and } \beta = \delta < d, \lambda(\delta) \in E_2, \sum_{k_2=\alpha+1}^{\delta-1} |\lambda(k_2) \in E_2| = y \Big\} \\
\phi_{[\texttt{between } a\ E_1 \texttt{ at least } b \texttt{ tu and } c\ E_2]}(\lambda) \\
\quad = \Big\{ \lambda(\alpha:\beta) \mid \alpha = \theta > b, \lambda(\theta) \in E_1, \sum_{k_1=0}^{\theta-1} |\lambda(k_1) \in E_1| = x \\
\quad\quad \text{and } \lambda(\beta) \in E_2, \sum_{k_2=\theta+1}^{\beta-1} |\lambda(k_2) \in E_2| = y \Big\} \\
\phi_{[\texttt{between } a\ E_1 \texttt{ at least } b \texttt{ tu and } c\ E_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \\
\quad = \Big\{ \lambda(\alpha:\beta) \mid \alpha = \theta > b, \lambda(\theta) \in E_1, \sum_{k_1=0}^{\theta-1} |\lambda(k_1) \in E_1| = x \\
\quad\quad \text{and } \beta = \delta < d, \lambda(\delta) \in E_2, \sum_{k_2=\theta+1}^{\delta-1} |\lambda(k_2) \in E_2| = y \Big\}
\end{cases}
$$

where

$$
x = \begin{cases} 0, & \text{if a=“”and c}\neq\text{“”;} \\ 0, & \text{if a=1;} \\ a-1, & \text{if } a > 1. \end{cases}, \quad y = \begin{cases} 0, & \text{if c=“”and a}\neq\text{“”;} \\ 0, & \text{if c=1;} \\ c-1, & \text{if } c > 1. \end{cases}
$$

$$
- \begin{cases}
\phi_{[\texttt{after } E_1 \texttt{ until } E_2]}(\lambda) \\
\quad = \phi_{[\texttt{between } E_1 \texttt{ and } E_2]}(\lambda) \\
\quad\quad \bigcup \{ \lambda(\eta : n-1) \mid \forall k, 0 \leq \alpha < k \leq n-1, \lambda(k) \notin E_2 \} \\
\phi_{[\texttt{after } E_1 \texttt{ until } E_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \\
\quad = \phi_{[\texttt{between } E_1 \texttt{ and } E_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \\
\quad\quad \bigcup \{ \lambda(\eta : n-1) \mid \forall k, 0 \leq \alpha < k \leq n-1, \lambda(k) \notin E_2 \} \\
\phi_{[\texttt{after } E_1 \texttt{ at least } b \texttt{ tu until } E_2]}(\lambda) \\
\quad = \phi_{[\texttt{between } E_1 \texttt{ at least } b \texttt{ tu and } E_2]}(\lambda) \\
\quad\quad \bigcup \{ \lambda(\eta : n-1) \mid \forall k, 0 \leq \alpha < k \leq n-1, \lambda(k) \notin E_2 \} \\
\phi_{[\texttt{after } E_1 \texttt{ at least } b \texttt{ tu until } E_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \\
\quad = \phi_{[\texttt{between } E_1 \texttt{ at least } b \texttt{ tu and } E_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \\
\quad\quad \bigcup \{ \lambda(\eta : n-1) \mid \forall k, 0 \leq \alpha < k \leq n-1, \lambda(k) \notin E_2 \}
\end{cases}
$$

$$
- \begin{cases}
\phi_{\texttt{[after } a\ E_1 \texttt{ until } c\ E_2\texttt{]}}(\lambda) \\
\quad = \phi_{\texttt{[between } a\ E_1 \texttt{ and } c\ E_2\texttt{]}}(\lambda) \\
\quad\quad \bigcup \left\{ \lambda(\eta : n-1) \mid \lambda(\eta) \in E_1, \sum_{k_1=0}^{\eta-1} |\lambda(k_1) \in E_1| = x \right. \\
\quad\quad \left. \text{and } \sum_{k_2=\eta+1}^{n-1} |\lambda(k_2) \in E_2| \leq y \right\} \\
\phi_{\texttt{[after } a\ E_1 \texttt{ until } c\ E_2 \texttt{ at least } d \texttt{ tu]}}(\lambda) \\
\quad = \phi_{\texttt{[between } a\ E_1 \texttt{ and } c\ E_2 \texttt{ at least } d \texttt{ tu]}}(\lambda) \\
\quad\quad \bigcup \left\{ \lambda(\eta : n-1) \mid \lambda(\eta) \in E_1, \sum_{k_1=0}^{\eta-1} |\lambda(k_1) \in E_1| = x \right. \\
\quad\quad \left. \text{and } \sum_{k_2=\eta+1}^{n-1} |\lambda(k_2) \in E_2| \leq y \right\} \\
\phi_{\texttt{[after } a\ E_1 \texttt{ at least } b \texttt{ tu until } c\ E_2\texttt{]}}(\lambda) \\
\quad = \phi_{\texttt{[between } a\ E_1 \texttt{ at least } b \texttt{ tu and } c\ E_2\texttt{]}}(\lambda) \\
\quad\quad \bigcup \left\{ \lambda(\eta : n-1) \mid \eta = \theta > b, \lambda(\theta) \in E_1, \sum_{k_1=0}^{\theta-1} |\lambda(k_1) \in E_1| = x \right. \\
\quad\quad \left. \text{and } \sum_{k_2=\theta+1}^{n-1} |\lambda(k_2) \in E_2| \leq y \right\} \\
\phi_{\texttt{[after } a\ E_1 \texttt{ at least } b \texttt{ tu until } c\ E_2 \texttt{ at least } d \texttt{ tu]}}(\lambda) \\
\quad = \phi_{\texttt{[between } a\ E_1 \texttt{ at least } b \texttt{ tu and } c\ E_2 \texttt{ at least } d \texttt{ tu]}}(\lambda) \\
\quad\quad \bigcup \left\{ \lambda(\eta : n-1) \mid \eta = \theta > b, \lambda(\theta) \in E_1, \sum_{k_1=0}^{\theta-1} |\lambda(k_1) \in E_1| = x \right. \\
\quad\quad \left. \text{and } \sum_{k_2=\theta+1}^{n-1} |\lambda(k_2) \in E_2| \leq y \right\}
\end{cases}
$$

where

$$
x = \begin{cases} 0, & \text{if a=“”} \text{ and c}\neq\text{“”}; \\ 0, & \text{if a=1}; \\ a-1, & \text{if } a > 1. \end{cases}
, \; y = \begin{cases} 0, & \text{if c=“”} \text{ and a}\neq\text{“”}; \\ 0, & \text{if c=1}; \\ c-1, & \text{if } c > 1. \end{cases}
$$

Patterns Let $\mathbb{P}$ be the set of patterns defined in the grammar. The semantics of a pattern $p \in \mathbb{P}$ is given by the function $\varphi_{[p]}(\lambda) : \Lambda \to \{true, false\}$ defined for every $\lambda \in \Lambda$ as follows:

$$
- \begin{cases}
\varphi_{\texttt{[eventually } E\texttt{]}}(\lambda) & \Leftrightarrow \exists i \geq 0, \lambda(i) \in E \\
\varphi_{\texttt{[eventually} \bowtie \text{m } E\texttt{]}}(\lambda) & \Leftrightarrow \sum_{i=0}^{n-1} |\lambda(i) \in E_2| \ \triangle \ m
\end{cases}
$$

where

$$
\triangle \ = \begin{cases} \geq, & \text{if } \bowtie = \text{at least}; \\ \leq, & \text{if } \bowtie = \text{at most}; \\ =, & \text{if } \bowtie = \text{exactly}. \end{cases}
$$

- $\begin{cases} \varphi_{[\texttt{never } E]}(\lambda) & \Leftrightarrow \forall i \geq 0, \lambda(i) \notin E \\ \varphi_{[\texttt{never exactly m } E]}(\lambda) & \Leftrightarrow \sum_{i=0}^{n-1} |\lambda(i) \in E| \ \neq \ m \end{cases}$
- $\varphi_{[\texttt{always } E]}(\lambda) \Leftrightarrow \forall i \geq 0, \lambda(i) \in E$
- $\begin{cases} \varphi_{[EC_1 \texttt{ preceding } EC_2]}(\lambda) \\ \quad \Leftrightarrow \forall \mathbf{match}(\lambda, EC_2) \Rightarrow \\ \quad\quad \mathbf{match}(\lambda, EC_1) \text{ and } \mathbf{last}(EC_1) \prec \mathbf{first}(EC_2) \\ \varphi_{[EC_1 \texttt{ preceding} \bowtie b \texttt{ tu } EC_2]}(\lambda) \\ \quad \Leftrightarrow \forall \mathbf{match}(\lambda, EC_2) \Rightarrow \\ \quad\quad \mathbf{match}(\lambda, EC_1) \text{ and } \mathbf{last}(EC_1) \prec \mathbf{first}(EC_2) \\ \quad\quad \text{and } \mathrm{d}(\mathbf{last}(EC_1), \mathbf{first}(EC_2)) \triangle b \end{cases}$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \text{at least;} \\ \leq, & \text{if } \bowtie = \text{at most;} \\ =, & \text{if } \bowtie = \text{exactly.} \end{cases}$
- $\begin{cases} \varphi_{[EC_1 \texttt{ responding } EC_2]}(\lambda) \\ \quad \Leftrightarrow \forall \mathbf{match}(\lambda, EC_2) \Rightarrow \\ \quad\quad \mathbf{match}(\lambda, EC_1) \text{ and } \mathbf{last}(EC_2) \prec \mathbf{first}(EC_1) \\ \varphi_{[EC_1 \texttt{ responding} \bowtie b \texttt{ tu } EC_2]}(\lambda) \\ \quad \Leftrightarrow \forall \mathbf{match}(\lambda, EC_2) \Rightarrow \\ \quad\quad \mathbf{match}(\lambda, EC_1) \text{ and } \mathbf{last}(EC_2) \prec \mathbf{first}(EC_1) \\ \quad\quad \text{and } \mathrm{d}(\mathbf{last}(EC_2), \mathbf{first}(EC_1)) \triangle b \end{cases}$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \text{at least;} \\ \leq, & \text{if } \bowtie = \text{at most;} \\ =, & \text{if } \bowtie = \text{exactly.} \end{cases}$

Temporal Expression The semantics of a temporal expression (pattern, scope) $\in \mathbb{P} \times \mathbb{S}$ over a trace $\lambda \in \Lambda$ is defined by: $\lambda \vDash (pattern, scope) \Leftrightarrow \forall \lambda^{'} \in \phi_{[scope]}(\lambda),\ \varphi_{[pattern]}(\lambda^{'})$

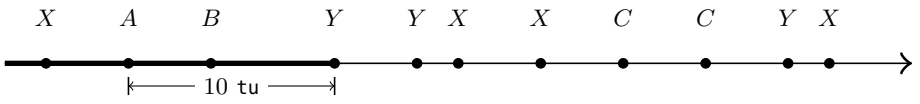

Fig. 10. An event trace

Consider an *OCLR* constraint like "eventually A at least 2 time units before Y". The black section in Fig. 10 is the scope restricted by "before Y". The temporal pattern "eventually A at least 2 time units" is evaluated using the semantics

described for *eventually*. Because the time interval between A and Y is 10 time units which is more than 2, this constraint is satisfied.