A Typed Language for Events

Sandra Alves[1]*, Sabine Broda[2]**, and Maribel Fernández[3]

[1] CRACS/INESCTEC, Faculty of Sciences, University of Porto
[2] CMUP, Faculty of Sciences, University of Porto
[3] Dept. of Informatics, King's College London

Abstract. We define a general typed language to deal with the notion of event in the context of access control systems. We distinguish between *generic* events, which represent the kind of actions that can occur in a system, and *specific* events, which represent actual occurrences of those kinds of actions. A relation is given associating specific to generic events, as well as a method for obtaining intervals from a history of events. We describe applications in access control systems with obligations.

Key Words: Event, Event Type, Access Control, Obligation, Rewriting

1 Introduction

The notion of event, as a particular action or happening taking place in a system, is a pervasive notion in today's computing (and real life) systems. Events can take up many forms, from messages exchanged over a network, to actions performed by users of the system, to occurrences of physical phenomena such as a disk error or a fire alarm.

In the context of access control policies, there are many situations when granting or denying access to certain resources depends on the occurrence of particular events. For example, in a hospital environment, an access control policy may specify that any doctor in the ward should have access to a patient p's medical records, if patient p suffers a cardiac arrest. Several access control models have been designed to deal with policies defined in terms of events (see, for example, [8, 3]). From the semantic point of view, the notions of action and event were extensively studied by Davidson [10]. Representation of events inspired by Davidson's work and adapted from Kowalski and Sergot's work on the event calculus [17] have been used in literature [5, 1]. In particular, in [1], events were used to define an abstract metamodel for access control and obligations. Obligations differ from permissions in the sense that, although permissions can be issued but not used, an obligation usually is associated with some mandatory action, which must be performed at a time defined by some temporal constraints

* Partially funded by FCT, Portuguese Foundation for Science and Technology within project UID/EEA/50014/2013.
** Partially supported by CMUP (UID/MAT/00144/2013), which is funded by FCT (Portugal) with national (MEC) and European structural funds through the programs FEDER, under the partnership agreement PT2020.

or by the occurrence of events. Therefore, effectively dealing with events is a key issue when reasoning about systems with obligations. The model described in [1] is an extension of Barker's category-based metamodel for access control [4] (CBAC): a notion of event adapted from [17] is used to describe a set of core axioms for defining obligations in an abstract way, without making any specific assumptions on the components of the system. In fact, two notions of events are defined in [1]: generic and specific. Generic events are used to represent the kind of events that can occur in a particular system, and specific events correspond to particular occurrences of events in a run of the system. The axiomatisation of the notion of obligation given in [1] relies on an *event typing* relation (associating specific events with generic events), and an *event interval* relation, which defines a link between an event that triggers a specific behaviour, and the event that terminates it. For example, the event associated to a fire alarm going off may start an emergency interval, which will be closed by the event associated to a call to the fire department.

In this paper we provide a general term-based language for events, and formally define the notions of *event typing* and *event interval*, to deal with event classification in a uniform way. Events are presented as typed-terms, built from a user-defined signature, that is, a particular set of typed function symbols that are specific to the system modelled. For each system we also define how to compute the events that close intervals initiated by previous events, based on a system specific function on generic events. This function allows us to extract intervals from a particular history (which is a sequence of events that have occurred in a system). Both event classification and interval computation have applications in access control and obligation management systems.

To summarise, the main contributions of the paper are the following:

– A general typed-language for events, and a typing relation associating specific and generic events;
– A general method for extracting event-intervals from a history;
– An implementation of this general method in Prolog together with methods for dealing with obligations, and an application of these methods in the context of obligation policies.

Overview: In Section 2, we recall some basic notions on term rewriting as well as the CBAC metamodel. In Section 3 we introduce a typed term language for representing events, and in Section 4 we recall the notion of event history and define an algorithm to extract event intervals from a history. Section 5 presents details of an implementation in Prolog of the relation between specific and generic events, the computation of intervals from history, as well as how this can be used in the obligation model. In Section 6 we discuss related work and finally, in Section 7, conclusions are drawn and further work is suggested.

2 Preliminaries

In this section we recall some basic notions and notations for term rewriting and access control policies involving obligations (see [2] and [1] for more details).

Term rewriting Term rewriting systems can be seen as programming or specification languages, or as formulae manipulating systems. We recall briefly the definition of first-order terms and term rewriting systems [2].

A signature $\mathcal{F}$ is a finite set of function symbols together with their (fixed) arity, where constants a are function symbols of arity zero. $\mathcal{X}$ denotes a denumerable set of variables $X_1, X_2 \ldots$, and $T(\mathcal{F}, \mathcal{X})$ denotes the set of terms built up from $\mathcal{F}$ and $\mathcal{X}$. Terms are identified with finite labeled trees. Positions are strings of positive integers. The subterm of t at position p is denoted by $t|_p$ and the result of replacing $t|_p$ with u at position p in t is denoted by $t[u]_p$. $\mathcal{V}(t)$ denotes the set of variables occurring in t. A term is ground (closed) if $\mathcal{V}(t) = \varnothing$. Substitutions are written $\theta = \{t_1/X_1, \ldots, t_n/X_n\}$ where t_i is assumed to be different from the variable X_i and $\mathsf{dom}(\theta) = \{X_1, \ldots, X_n\}$. We use Greek letters for substitutions and postfix notation for their application.

Given a signature $\mathcal{F}$, a *term rewrite system* on $\mathcal{F}$ is a set of rewrite rules $R = \{l_i \to r_i\}_{i \in I}$, where $l_i, r_i \in T(\mathcal{F}, \mathcal{X})$, $l_i \notin \mathcal{X}$, and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$. A term t *rewrites* to a term u at position p with the rule $l \to r$ and the substitution σ, written $t \to_p^{l \to r, \sigma} u$, or simply $t \to_R u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$. Such a term t is called *reducible*. Irreducible terms are said to be in *normal form*. We denote by $\to_R^+$ (resp. $\to_R^*$) the transitive (resp. transitive and reflexive) closure of $\to_R$. The subindex R will be omitted when it is clear from the context.

Access Control and Obligations The Category-Based Access Control (CBAC) metamodel [4] is an abstract framework for the definition of access control policies, which can be instantiated to derive well-known access control models, such as Role-Based Access Control [11], Bell-La Padula's model [6], and dynamic models [8, 3]. The latter permit the definition of access control policies where users' rights depend on their actions, or more generally, on events that happened in the system. In this paper, we present an event language and show how it can be applied in access control and obligation models. More precisely, we consider the extension of the CBAC metamodel that incorporates obligations, which in the following will be referred to as CBACO [1].

The CBAC metamodel is defined using a basic set of primitive, abstract notions: principals (which are the users of the system), resources (which are the objects that should be protected) and actions (which are the operations that users can perform on resources). These entities can be grouped into categories (in access control models we mostly consider categories of users). A category is a class of entities that share some property. Classic types of groupings used in access control, like a role, a security clearance, a discrete measure of trust, etc., are particular instances of the more general notion of category. Permissions, that is, pairs of action and resource, are assigned to categories of users rather than to individual users. Categories can be defined on the basis of e.g., user attributes, geographical constraints, resource attributes. In this way, permissions change in a dynamic and autonomous way (e.g., when a registered user has a birthday), unlike, e.g., role-based access control models, which require the intervention of a security administrator. Then, the axiomatic specification of the model allows us to derive, at any point, the rights of a principal by computing the principal's category and checking the permissions associated to it. In this way, an access request can be evaluated to decide whether it should be granted or denied. In CBACO, in addition to the basic notion of permissions available in CBAC, there are also two abstract notions of obligations, defined as follows.

Definition 1 (Obligation). *A* generic obligation *is a tuple (a, r, ge_1, ge_2), where a is an action, r a resource, and ge_1, ge_2 two event types (ge_1 triggers the obligation, and ge_2 ends it). If there is no starting event (resp., no ending event) we write $(a, r, \perp, ge)$ (resp., $(a, r, ge, \perp)$), meaning that the action on the resource must be performed at any point before an event of type ge (resp. at any point after an event of type ge).*

Example 1. Assume that in an organisation, the members of the security team must call the fire-department if a fire alarm is activated, and this must be done before they de-activate the alarm. This obligation could be represented by the tuple $(call, firedept, alarmON, alarmOFF)$.

Definition 2 (Duty). *A* duty *is a tuple (p, a, r, e_1, e_2, h), where p is a principal, a an action, r a resource, e_1, e_2 are two events and h is an event history that includes an interval opened by e_1 and closed by e_2. We replace e_1 (resp. e_2) with $\perp$ if there is no starting (resp. closing) event.*

Unlike access control models, which do not need to check whether the authorised actions are performed or not by the principals, obligation models need to include mechanisms to check whether duties were discharged or not. Specifically, obligation models distinguish four possible states for duties: *invalid* (when the duty is issued after the completion point); *fulfilled* (when carried out within the associated interval); *violated* (when not carried out within the associated interval, although issued with a valid interval) and *pending* (when has not yet been carried, but the interval is still valid). We refer the reader to [1] for the axiomatic and operational semantics of obligation policies in CBACO.

3 Events as Typed Terms

In this section we present a typed term language to represent events. We consider events as particular actions or happenings occurring at a particular time. Types are used to restrict the terms that correspond to events in our language.

In this section, and in the rest of the paper, we will present examples considering a hospital scenario, where several types of events can occur: patients can be triaged, admitted, receive consultation, be discharged, submitted to exams/procedures, etc. Sporadically there can also occur events such as fire alarms that can lead to the hospital evacuation, etc.

3.1 Types and Terms

Let b range over a finite set $\mathbb{B}$ of base types, and l over a finite set $\mathcal{L}$ of labels. The set $\mathbb{B}$ will always contain the types Tm and Ev, which are the types for time and event expressions respectively.

Definition 3 (Type). *The set $\mathbb{T}$ of types is built from $\mathbb{B}$:*

$$\tau \in \mathbb{T} ::= b \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \mid \tau_1 \to \tau_2$$

Record types, of the form $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, represent structures labelled with $l_1, \ldots, l_n$, with types $\tau_1, \ldots, \tau_n$ respectively.

We consider a (system specific) function $\mathsf{type} : \mathcal{F} \to \mathbb{T}$, which assigns a type to each function symbol in $\mathcal{F}$. If f is a function symbol with arity n, then $\mathsf{type}(f) = \tau_1 \to \cdots \to \tau_n \to \tau$, for some $\tau_1, \ldots, \tau_n, \tau \in \mathbb{T}$.

Because terms in our language can contain free variable occurrences, type declarations for variables must be taken into consideration when typing expressions. As usual, an *environment* env, is a set of declarations of the form $X : \tau$ where all the variables X are distinct.

We now present our language to model events. We consider event expressions as terms that can be built from other event expressions, atomic actions or sets of attributes (represented as labelled structures).

Definition 4 (Event Specification). *Consider $X \in \mathcal{X}$, $\tau \in \mathbb{T}$ and $f \in \mathcal{F}$, then* values *and* specifications *are defined in the following way:*

$$\begin{array}{ll} \nu & ::= X^\tau \mid f(\nu_1, \ldots, \nu_n), \qquad n \geq 0 \\ \mathsf{spec} & ::= \{\mathtt{l}_1 = \nu_1, \ldots, \mathtt{l}_n = \nu_n\}, \ n > 0 \end{array}$$

The value X^τ represents a term variable of type τ. An atomic value a is a particular case of a value of the form $f(\nu_1, \ldots, \nu_n)$ where $n = 0$. The event specification $\{\mathtt{l}_1 = \nu_1, \ldots, \mathtt{l}_n = \nu_n\}$ represents the structure with labels $\mathtt{l}_1, \ldots, \mathtt{l}_n$ and values $\nu_1, \ldots, \nu_n$ respectively.

Definition 5 (Generic and Specific Events). *A* generic event*, denoted by $ge(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) \in \mathcal{GE}$, is defined by an equation of the form:*

$$ge(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) = \{\mathsf{spec}_1, \ldots, \mathsf{spec}_m\}^C$$

where the variables $X_1^{\tau_1}, \ldots, X_n^{\tau_n}$ occur in the right-hand side of the equation. The expression $\{\mathsf{spec}_1, \ldots, \mathsf{spec}_m\}^C$ represents the compound generic event, formed from the generic event specifications $\mathsf{spec}_1, \ldots, \mathsf{spec}_m$. If $m = 1$, then just write $ge(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) = \mathsf{spec}$. Compound events represent sets of events that can occur separately in the history, but should be identified as a single event occurrence.

Specific events*, denoted by $e \in \mathcal{E}$, are defined in the following way, where $\mathsf{spec}^\varnothing$ denotes ground event specifications (see Definition 4):*

$$e ::= \{\mathsf{spec}_1^\varnothing, \ldots, \mathsf{spec}_n^\varnothing\}^C$$

As before, we write $\{\mathsf{spec}^\varnothing\}^C$ as $\mathsf{spec}^\varnothing$.

Example 2. We describe the action of a doctor P_1 reading the medical record of a patient P_2, by the generic event $gen\_read(P_1^{\mathcal{D}}, P_2^{\mathcal{P}})$, where $\mathcal{D}$ and $\mathcal{P}$ represent the types for *doctor* and *patient*, respectively:

$$gen\_read(P_1^{\mathcal{D}}, P_2^{\mathcal{P}}) = \{\mathsf{act} = read, \mathtt{doc} = P_1^{\mathcal{D}}, \mathtt{obj} = rec(P_2^{\mathcal{P}})\}$$

We state that an order to evacuate the neurology ward was issued by Chief Jones, with the specific event:

$$\{\mathsf{act} = \mathit{evacuate}, \mathtt{ward} = \mathit{neurology}, \mathtt{principal} = \mathit{chief\_jones}\}$$

The compound event *gen_pregnD* represents the events that must occur, before a doctor can make a pregnancy diagnosis.

$$\begin{aligned} \mathit{gen\_pregnD}(X^{\mathcal{P}}) = \{\ & \{\mathsf{act} = \mathit{lab\_test}, \mathtt{pat} = X^{\mathcal{P}}\}, \\ & \{\mathsf{act} = \mathit{ultrasound}, \mathtt{obj} = \mathit{abdomen}, \mathtt{pat} = X^{\mathcal{P}}\}\ \}^{C} \end{aligned}$$

3.2 Typing Rules

We now assign types to values and event specifications, to ensure that we only deal will well-typed entities (wrt. the type signature specific to each system). We use record types to type labelled structures, with an implicit notion of subtyping (inspired by Ohori's system with polymorphic record types [20]). For the moment we only consider (implicit) subtyping between record types, but this can later be extended to general event types.

Definition 6 (Typing Rules for Values, Specifications and Events). *A typing judgement is a declaration of the form* $\mathsf{env} \vdash \nu : \tau$, $\mathsf{env} \vdash \mathsf{spec} : \tau$, *or* $\mathsf{env} \vdash \mathit{ge}(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) : \mathsf{Ev}$. *We say that* ν *(resp.* spec*) has type* τ *given* env*, and write* $\mathsf{env} \vdash \nu : \tau$ *(resp.* $\mathsf{env} \vdash \mathsf{spec} : \tau$*), if the judgement can be derived using the following axioms and rules:*

$$\mathsf{env} \vdash X^{\tau} : \tau,\ \ \mathit{if}\ X : \tau \in \mathsf{env}$$

$$\frac{\mathsf{type}(f) = \tau_1 \to \cdots \to \tau_n \to \tau \quad \mathsf{env} \vdash \nu_1 : \tau_1 \quad \cdots \quad \mathsf{env} \vdash \nu_n : \tau_n}{\mathsf{env} \vdash f(\nu_1, \ldots, \nu_n) : \tau} \quad (n \geq 0)$$

$$\frac{\mathsf{env} \vdash \nu_1 : \tau_1 \quad \cdots \quad \mathsf{env} \vdash \nu_{n+k} : \tau_{n+k}}{\mathsf{env} \vdash \{\mathtt{l}_1 = \nu_1, \ldots, \mathtt{l}_n = \nu_n\} \cup \Gamma : \{\mathtt{l}_1 : \tau_1, \ldots, \mathtt{l}_n : \tau_n\}}$$

where, $\Gamma = \{\mathtt{l}_{n+1} = \nu_{n+1}, \ldots, \mathtt{l}_{n+k} = \nu_{n+k}\}$.

Given the rules above, a generic event expression $\mathit{ge}(X_1^{\tau_1}, \ldots, X_n^{\tau_n})$ *is well-typed given* env*, if* $\mathsf{env} \vdash \mathit{ge}(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) : \mathsf{Ev}$ *can be derived from:*

$$\frac{\mathit{ge}(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) = \{\mathsf{spec}_1, \ldots, \mathsf{spec}_m\}^{C} \quad \mathsf{env} \vdash \mathsf{spec}_i : \sigma_i\ (i = 1, \ldots, m)}{\mathsf{env} \vdash \mathit{ge}(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) : \mathsf{Ev}}$$

Given the condition on variables in Definition 5, if $\mathsf{env} \vdash \mathit{ge}(X_1^{\tau_1}, \ldots, X_n^{\tau_n}) : \mathsf{Ev}$ then env contains declarations $X_1 : \tau_1, \ldots, X_n : \tau_n$.

Example 3. Using the fact that $\mathsf{type}(\mathit{read}) = \mathcal{A}$, $\mathsf{type}(\mathit{dr.\ paul}) = \mathcal{D}$, $\mathsf{type}(\mathit{john}) = \mathcal{P}$, and $\mathsf{type}(\mathit{rec}) = \mathcal{P} \to \mathcal{R}$, we obtain

$$\mathsf{env} \vdash \{\mathsf{act} = \mathit{read}, \mathtt{doc} = P_1^{\mathcal{D}}, \mathtt{obj} = \mathit{rec}(P_2^{\mathcal{P}})\} : \{\mathsf{act} : \mathcal{A}, \mathtt{doc} : \mathcal{D}, \mathtt{obj} : \mathcal{R}\}$$

and, for $\mathsf{env} = \{P_1 : \mathcal{D}, P_2 : \mathcal{P}\}$, $\mathsf{env} \vdash \mathit{gen\_read}(P_1^{\mathcal{D}}, P_2^{\mathcal{P}}) : \mathsf{Ev}$ using the definition of *gen_read* given in Example 2.

Because we have an implicit subtyping rule for typing records, types are not unique. The event specification in the previous example, $\mathsf{spec} = \{\mathsf{act} = \mathit{read}, \mathtt{doc} = P_1^{\mathcal{D}}, \mathtt{obj} = \mathit{rec}(P_2^{\mathcal{P}})\}$, can also be typed with $\{\mathsf{act} : \mathcal{A}, \mathtt{obj} : \mathcal{R}\}$ and $\{\mathtt{obj} : \mathcal{R}\}$. In fact any non-empty subset of $\{\mathsf{act} : \mathcal{A}, \mathtt{doc} : \mathcal{D}, \mathtt{obj} : \mathcal{R}\}$ is a valid type for spec. In this paper we are doing type-checking (and not type-inference), therefore we do not focus on most general types for specifications. But this could be achieved by using the notion of kinds of records as it is done in [20].

We now define an instance relation, associating ground values (denoted $\nu^{\varnothing}$) to values and specific event specifications to generic event specifications, under a substitution.

Definition 7 (Instantiation). *We define the relation* $\vdash_\theta \nu^{\varnothing} :: \nu$ *(resp.* $\vdash_\theta \mathsf{spec}^{\varnothing} :: \mathsf{spec}$*), where* θ *is a substitution, in the following way:*

$$\frac{\vdash \nu^{\varnothing} : \tau}{\vdash_{\{\nu^{\varnothing}/X\}} \nu^{\varnothing} :: X^{\tau}} \qquad \frac{\vdash_{\theta_1} \nu_1^{\varnothing} :: \nu_1 \quad \cdots \quad \vdash_{\theta_n} \nu_n^{\varnothing} :: \nu_n}{\vdash_{\theta_1 \cup \cdots \cup \theta_n} f(\nu_1^{\varnothing}, \ldots, \nu_n^{\varnothing}) :: f(\nu_1, \ldots, \nu_n)} \quad (n \geq 0)$$

$$\frac{\vdash_{\theta_1} \nu_1^{\varnothing} :: \nu_1 \quad \cdots \quad \vdash_{\theta_n} \nu_n^{\varnothing} :: \nu_n}{\vdash_{\theta_1 \cup \cdots \cup \theta_n} \{\mathtt{l}_1 = \nu_1^{\varnothing}, \ldots, \mathtt{l}_n = \nu_n^{\varnothing}\} \cup \Gamma :: \{\mathtt{l}_1 = \nu_1, \ldots, \mathtt{l}_n = \nu_n\}}$$

where, $\Gamma = \{\mathtt{l}_{n+1} = \nu_{n+1}^{\varnothing}, \ldots, \mathtt{l}_{n+k} = \nu_{n+k}^{\varnothing}\}$*. Whenever we write* $\theta_1 \cup \cdots \cup \theta_n$*, we assume that* $\theta_1, \ldots, \theta_n$ *are compatible substitutions, in the sense that they do not assign different values to the same variable.*

Definition 8 (Event Instance). *The relation* $\vdash_\theta e :: \mathit{ge}(\overrightarrow{X})$*, extends the previous definition to event expressions in the following way:*

$$\frac{\mathit{ge}(\overrightarrow{X}) = \{\mathsf{spec}_1, \ldots, \mathsf{spec}_n\}^C \quad \vdash_{\theta_1} \mathsf{spec}_1^{\varnothing} :: \mathsf{spec}_1 \ \cdots \ \vdash_{\theta_n} \mathsf{spec}_n^{\varnothing} :: \mathsf{spec}_n}{\vdash_{\theta_1 \cup \cdots \cup \theta_n} \{\mathsf{spec}_1^{\varnothing}, \ldots, \mathsf{spec}_n^{\varnothing}\}^C :: \mathit{ge}(\overrightarrow{X})}$$

Example 4. Recall $\mathit{gen\_read}(P_1^{\mathcal{D}}, P_2^{\mathcal{P}})$, from Example 2. For the substitution $\theta = \{\mathit{dr.\ paul}/P_1, \mathit{john}/P_2\}$, we can derive

$$\vdash_\theta \{\mathsf{act} = \mathit{read}, \mathtt{doc} = \mathit{dr.\ paul}, \mathtt{obj} = \mathit{rec}(\mathit{john})\} :: \mathit{gen\_read}(P_1^{\mathcal{D}}, P_2^{\mathcal{P}}).$$

Proposition 1. *If* $\mathsf{env} \vdash \nu : \tau$ *(resp.* $\mathsf{env} \vdash \mathsf{spec} : \tau$*) and* $\vdash_\theta \nu^{\varnothing} :: \nu$ *(resp.* $\vdash_\theta \mathsf{spec}^{\varnothing} :: \mathsf{spec}$*), then:*

1. $\theta = \{\nu_1^{\varnothing}/X_1, \ldots, \nu_n^{\varnothing}/X_n\}$*, where* $\{X_1, \ldots, X_n\} = \mathcal{V}(\nu)$ *(resp.* $\mathcal{V}(\mathsf{spec})$*) and* $\vdash \nu_i^{\varnothing} : \tau_i$ *where* $X_i : \tau_i \in \mathsf{env}$.
2. $\vdash \nu^{\varnothing} : \tau$ *(resp.* $\vdash \mathsf{spec}^{\varnothing} : \tau$*).*

The instantiation relation defined in this section is syntactic (replacing variables by terms). Depending on the application and the kind of data used to define events, instantiation may require some computation; we call it a semantic instantiation in the latter case. Formally, semantic instantiation is defined in the context of an equational theory. Although we leave a complete study on different equational theories and its appropriateness for future work, in the next section we will deal with semantic instantiation for time expressions.

4 Event History and Intervals

In this section we will define the notions of event history and intervals in history, which are determined by events. We also show how the instance relation defined in the previous section can be used to extract intervals from a history of events that match two given generic events. A history of events corresponds to a specific sequence of events that occur in a particular time frame. To deal with time frames we need a language that appropriately deals with time.

4.1 Time Expressions and Time Constraints

In this subsection we define a language for expressions representing time and use this to encode the approach for dealing with events in [1], in this setting.

Definition 9 (Time Expressions and Constraints). *Let* c *range over a set* $\mathcal{S}$*, partially ordered by* $\leq$ *and closed under* $+$*. We define the set of* time expressions *and* time constraints *denoted* $\mathsf{t} \in \mathcal{T}$ *and* $\mathsf{tc} \in \mathcal{TC}$ *respectively, in the following way:*

$$\mathsf{t} ::= \mathsf{c} \mid X^{\mathsf{Tm}} \mid \mathsf{t} + \mathsf{c} \qquad\qquad \mathsf{tc} ::= \mathsf{t} \mid \mathsf{t}^{+} \mid \mathsf{t}^{+\mathsf{c}}$$

Time constraints can be seen as intervals $[\mathsf{t}_1,\ \mathsf{t}_2]$*, where* $\mathsf{t}_1 = \mathsf{t}_2$*, if the time constraint is a time expression;* $\mathsf{t}_2 = \infty$*, if the time constraint is of the form* t_1^{+}*; and* $\mathsf{t}_2 = \mathsf{t}_1 + \mathsf{c}$*, if the time constraint is of the form* $\mathsf{t}_1^{+\mathsf{c}}$*.*

Note that a constant time expression represents a specific instant in time, which can be particular to each modelled system. In a time expression of the form $\mathsf{t}+\mathsf{c}$, c can be seen as a duration. In the rest of the paper we will take $\mathcal{S}$ to be $\mathbb{N}$, but other constants can be considered (that is, we consider time constants as clock ticks from a fixed point in time).

Definition 10. *Let* $\sigma = \{\mathsf{c}_1/X_1, \ldots, \mathsf{c}_n/X_n\}$ *be a substitution. We define* $[\![\cdot]\!]\sigma$ *for time expressions and time constraints, as follows:*

$$\begin{array}{lll} [\![\mathsf{c}]\!]\sigma \ = \mathsf{c} & [\![\mathsf{t}^{+}]\!]\sigma \ = [[\![\mathsf{t}]\!]\sigma,\ \infty] & [\![\mathsf{t}+\mathsf{c}]\!]\sigma = [\![\mathsf{t}]\!]\sigma + [\![\mathsf{c}]\!]\sigma \\ [\![X]\!]\sigma = \sigma(X) & [\![\mathsf{t}^{+\mathsf{c}}]\!]\sigma = [[\![\mathsf{t}]\!]\sigma,\ [\![\mathsf{t}]\!]\sigma + [\![\mathsf{c}]\!]\sigma] & \end{array}$$

If $\mathcal{V}(\mathsf{t}_1) \cup \mathcal{V}(\mathsf{t}_2) \subseteq \mathsf{dom}(\sigma)$*, then* $\mathsf{t}_1 \preceq_\sigma \mathsf{t}_2$ *iff* $[\![\mathsf{t}_1]\!]\sigma \leq [\![\mathsf{t}_2]\!]\sigma$*. If* $\mathsf{t}_1, \mathsf{t}_2$ *are both ground we simply write* $\mathsf{t}_1 \preceq \mathsf{t}_2$ *instead of* $\mathsf{t}_1 \preceq_\varnothing \mathsf{t}_2$*.*

Another representation for events. Events in [1] are represented by finite sets of arity-2 facts, containing at least two necessary facts, $\mathtt{happens}(e, t)$ and $\mathtt{act}(e, a)$, where e is the identifier of the specific event, and t the time of its happening. Generic events are defined similarly, but can contain variables (as is the case in this paper), and in particular they always contain a variable (E) to be instantiated with the identifier of a specific event. A specific event e is an instance of a generic event ge, if there is a substitution σ such that $ge\sigma \subseteq e$. For example the events

$$e_1 = \{\texttt{happens}(e_1, 12.25), \texttt{act}(e_1, \mathit{activate}), \texttt{obj}(e_1, \mathit{alarm}), \texttt{subj}(e_1, \mathit{john})\}$$
$$e_2 = \{\texttt{happens}(e_2, 12.45), \texttt{act}(e_2, \mathit{deactivate}), \texttt{obj}(e_2, \mathit{alarm}), \texttt{subj}(e_2, \mathit{tom})\}$$

are instances, with respective substitutions $\sigma_1 = \{e_1/E, 12.25/T\}$ and $\sigma_2 = \{e_2/E, 12.25/T, \mathit{tom}/X\}$, of the generic events

$$\texttt{alarmON} \;= \{\texttt{happens}(E, T), \texttt{act}(E, \mathit{activate}), \texttt{obj}(E, \mathit{alarm})\}$$
$$\texttt{alarmOFF} = \{\texttt{happens}(E, T + 20), \texttt{act}(E, \mathit{deactivate}), \texttt{obj}(e, \mathit{alarm}), \texttt{subj}(E, X)\}$$

This is an example where the instantiation relation requires some computation (the instantiation of $T + 20$ with the substitution $12.25/T$ will produce 12.45 in σ_2). Note that the function in Definition 10 defines a semantic instantiation for time expressions and time constraints.

The encoding of this event representation is straightforward. Given a particular event, a record `spec` is created containing an entry $\texttt{fact} = \mathit{exp}$ for each fact $\texttt{fact}(e, \mathit{exp})$, except for $\texttt{happens}(e, t)$. The identifier e can, depending on necessity, either be omitted or be included as a particular entry $\texttt{id} = e$. Finally, the event will be represented by a pair $(\texttt{spec}, t)$, where t is a time expression (this notion will be formalised in the next section). The encoding of the events above is:

$$(\{\texttt{act} = \mathit{activate}, \texttt{obj} = \mathit{alarm}, \texttt{subj} = \mathit{john}\}, 12.25)$$
$$(\{\texttt{act} = \mathit{deactivate}, \texttt{obj} = \mathit{alarm}, \texttt{subj} = \mathit{tom}\}, 12.45)$$
$$(\texttt{alarmON} = \{\texttt{act} = \mathit{activate}, \texttt{obj} = \mathit{alarm}\}, T)$$
$$(\texttt{alarmOFF}(X) = \{\texttt{act} = \mathit{deactivate}, \texttt{obj} = \mathit{alarm}, \texttt{subj} = X)\}, T)$$

4.2 History of Events

We will now consider a history of events and define how one can relate events in a history to define appropriate intervals.

Definition 11 (History). *An* event history $h \in \mathcal{H}$ *is a sequence of distinct specific events in time of the form* $[E_1 = (e_1, \texttt{t}_1), \dots, E_n = (e_n, \texttt{t}_n)]$*, where* $\texttt{t}_1, \dots, \texttt{t}_n$ *are ground and such that for* $i < j$, $\texttt{t}_i \preceq \texttt{t}_j$*. A subsequence of* h *is called an* event interval *and it is represented as* $I = (E_i, E_j)$*, where* E_i, E_j *are respectively the first and the last event in the interval. We say that* E_i *opens the interval and* E_j *closes it. We use the constant* $\perp$ *to represent untimed events as a pair* $(e, \perp)$*.*

Example 5 (History).

$$\begin{aligned} h = [\ & (\{\texttt{act} = \mathit{enterHosp}, \texttt{patient} = \mathit{john}\}, 900), \\ & (\{\texttt{act} = \mathit{consult}, \texttt{patient} = \mathit{john}, \texttt{doc} = \mathit{dr.\ paul}\}, 1100), \\ & (\{\texttt{act} = \mathit{request}, \texttt{doc} = \mathit{dr.\ paul}, \texttt{patient} = \mathit{john}, \texttt{proc} = \mathit{x\text{-}ray}\}, 1110), \\ & (\{\texttt{act} = \mathit{perform}, \texttt{patient} = \mathit{john}, \texttt{doc} = \mathit{dr.\ mary}, \texttt{proc} = \mathit{x\text{-}ray}\}, 1125), \\ & (\{\texttt{act} = \mathit{read}, \texttt{doc} = \mathit{dr.\ paul}, \texttt{ward} = \mathit{neurology}, \texttt{obj} = \mathit{rec}(\mathit{john})\}, 1300) \end{aligned}$$

To compute intervals we will define a function that describes how events are linked to subsequent events in history. Because we want to consider different scenarios we will also consider different strategies to select intervals. A strategy is a function $\mathsf{strat} : 2^{\mathcal{I}} \to 2^{\mathcal{I}}$, which will be used to select elements from a set $\mathcal{I}$ of pairs of timed events (intervals). Examples of strategies can be *select-first*, *select-last*, *select-all*, etc. We will use $\mathscr{S}$ to represent the set of available strategies.

Definition 12. *A closing function* $\mathsf{cl} : \mathcal{GE} \times \mathcal{TC} \to 2^{\mathcal{GE} \times \mathcal{T} \times \mathscr{S}}$*, is a mapping associating to a pair* (ge, tc)*, a set of triples of the form* $(ge', \mathsf{t}', \mathsf{strat})$*, which are the generic events in time that are closed by the generic event ge provided that some constraints on* t' *and* tc *are satisfied, and selected by the function* strat*. In the rest of the paper we will assume a* select-first *strategy, and omit strategies from function* cl*.*

Example 6. In our hospital scenario, consider:

- $\mathsf{cl}(\mathit{exitHosp}(P, W), T^{+})) = \{(\mathit{triage}(P), T), (\mathit{inWard}(P, W), T)\}$
- $\mathsf{cl}(\mathit{releaseCR}(W), T^{+20})) = \{(\mathit{codeRED}(W), T)\}$

For instance $\mathsf{cl}(\mathit{releaseCR}(W), T^{+20})) = \{(\mathit{codeRED}(W), T)\}$ indicates that the specific event $\mathit{releaseCR}(\mathit{neurology})$ can close the event $\mathit{codeRED}(\mathit{neurology})$, provided that the former occurs at most 20 instants after the latter.

Closed and open intervals are key notions when dealing with obligations, as they allow us to determine the status of an obligation at a given point. In the next section we will use the notions of closed and open intervals (Definitions 13 and 14 below) in the context of the CBACO metamodel.

Definition 13. *Let* $h \in \mathcal{H}$*,* $(ge, \mathsf{tc}) \in \mathcal{GE} \times \mathcal{TC}$ *and* $(ge', \mathsf{t}) \in \mathcal{GE} \times \mathcal{T}$*, then* $\textsf{\textit{closed}}(ge', ge, h)$ *is the set of event intervals of the form* $((e_i, \mathsf{t}_i), (e_j, \mathsf{t}_j))$ *such that, for some compatible substitutions* θ_i, θ_j *and a substitution on time variables* σ*, one has:* $(ge', \mathsf{t}) \in \mathsf{cl}((ge, \mathsf{tc}))$*;* $\vdash_{\theta_i} e_i :: ge'$ *and* $\vdash_{\theta_j} e_j :: ge$*;* $\mathsf{t}_i = [\![\mathsf{t}]\!]\sigma$ *and* $\mathsf{t}_j \in [\![\mathsf{tc}]\!]\sigma$*. Then the function* $\mathsf{interval}(e_1, e_2, h)$ *is* ***true*** *if, for some compatible substitutions* θ_1 *and* θ_2*:* $\vdash_{\theta_1} e_1 :: ge_1$*,* $\vdash_{\theta_2} e_2 :: ge_2$ *and* $(e_1, e_2) \in \textsf{\textit{closed}}(ge_1, ge_2, h)$*, and* ***false*** *otherwise.*

Definition 14. *Let* $h \in \mathcal{H}$*,* $(ge, \mathsf{tc}) \in \mathcal{GE} \times \mathcal{TC}$ *and* $(ge', \mathsf{t}) \in \mathcal{GE} \times \mathcal{T}$*, then* $\textsf{\textit{open}}(ge', ge, h)$ *is the set of event intervals of the form* $((e_i, \mathsf{t}_i), \perp)$ *such that, for some substitution* θ_i *and a substitution on time variables* σ*, one has:* $(ge', \mathsf{t}) \in \mathsf{cl}((ge, \mathsf{tc}))$*;* $\vdash_{\theta_i} e_i :: ge'$*;* $\mathsf{t}_i = [\![\mathsf{t}]\!]\sigma$*, but there is not an event* (e_j, t_j)*, with* $\mathsf{t}_i \preceq \mathsf{t}_j$*, such that* $\vdash_{\theta_j} e_j :: ge$*, for a substitution* θ_j *compatible with* θ_i *and* $\mathsf{t}_j \in [\![\mathsf{tc}]\!]\sigma$*.*

In the defininions above, we do not distinguish between single and compound events, and assume that compound events appear in history. A more detailed and realistic treatment of coumpound events in history is left for future work.

5 A Prolog Implementation

In this section we describe a prototype implementation of the previous definitions in Prolog. Because Prolog programs are expressed in terms of relations, represented as facts and rules, Prolog is an ideal language to implement the notions defined in this paper. Backtracking, unification and logical variables are also useful features for our implementation (although in this prototype implementation we treat substitutions explicitly, a more efficient implementation can make use of Prolog's logic variables and unification to implicitly propagate substitutions and deal with compatibility of substitutions).

5.1 Defining Events, Event Typing and Intervals

For a particular system, the language of events is determined by the set of functors $\mathcal{F}$, its associated types given by function type, and the equations defining generic events, which can be represented as Prolog facts. For example, we can consider the following typed constants and functors for our hospital scenario:

```
type(neurology,ward).
type(dr_paul,doctor).
type(rec,arrow([patient],resource)).
ge(exitHosp,[var(P1,patient)], rec([lab(action,discharge),
                                    lab(patient,var(P1,patient)),
                                    lab(doc,var(P2,doctor))])).
cl((ge(exitHosp,[var(P,doctor),var(W,ward)]),plus(var(T,time))),
   [(ge(triage,[var(P,doctor)]),var(T,time)),
    (ge(inWard,[var(P,doctor),var(W,ward)]),var(T,time))]).
cl((ge(releaseCR,[var(W,ward)]),plus(var(T,time))),
   [(ge(codeRED,[var(W,ward)]),var(T,time))]).
```

Below we present the predicate ty(Theta, E, GE) implementing the relation $\vdash_\theta e :: ge$ from Definition 8.

```
ty([],A,A):- atomic(A).
ty([(X,Value)],Value,var(X,Type)):- typed([],Value,Type).
ty(Theta,fun(Name,CValues),fun(Name,Values)):- zip(CValues,Values,L),
                                               tyList(Theta,L).
ty(Theta,lab(Name,CValue),lab(Name,Value)):- ty(Theta,CValue,Value).
ty(Theta,rec(CL),rec(L)):- permut(CL,PCL),
                           zip(CL,L,LRec),
                           tyList(Theta,LRec).
ty(Theta,CSpec,ge(Name,Lvar)):- ge(Name,Lvar,Spec), ty(Theta,CSpec,Spec).
ty(Theta,comp(CSpec),ge(Name,Vars)):- ge(Name,Vars,comp(LSpec)),
                                      permut(CSpec,PCSpec),
                                      zip(PCSpec,LSpec,Specs),
                                      ty(Theta,Specs).
tyList([],[]).
tyList(Theta,[(CValue,Value)|L]):- ty(Theta1,CValue,Value),
                                   tyList(Theta2,L),
                                   compatible(Theta1,Theta2,Theta).
```

Where the predicate typed(Env, Value, Type) implements the typing relation $\mathsf{env} \vdash \nu : \tau$ from Definition 6. The predicate compatible(Theta1, Theta2, Theta) verifies if the two substitutions Theta1 and Theta2 are compatible, eliminating duplicated declarations. The predicate zip(L1, L2, L3), succeeds if in the list of pairs L3, each pair contains elements of lists L1 and L2 occurring at the same position (similar to the Haskell zip function). The predicate permut(L1,L2) succeeds if L2 is a permutation of L1.

The functions $\mathsf{closed}(ge', ge, h)$ and $\mathsf{open}(ge', ge, h)$ from Definition 13 can be computed using the predicates cinterval(GE1,GE2,H,(E1,E2,Sigma)) and ointerval(GE1,GE2,H,(E1,E2,Sigma)), respectively:

```
cinterval((ge(N1,V1),TC),(ge(N2,V2),T),H,((Ei,Ti),(Ej,Tj),Sigma)):-
    cl((ge(N1,V1),TC),LGEs), member((ge(N2,V2),T),LGEs),
    pick((Ei,Ti),H,RH), ty(Theta,Ei,ge(N2,Vs2)),
    pick((Ej,Tj),RH,_), ty(Theta,Ej,ge(N1,Vs2)),
    tsem(T,Sigma,Ti), tsem(TC,Sigma,Int), belongs(Tj,Int).
ointerval((ge(N1,V1),TC),(ge(N2,V2),T),H,((Ei,Ti),bot,Sigma)):-
    cl((ge(N1,V1),TC),LGEs), member((ge(N2,V2),T),LGEs),
    pick((Ei,Ti),H,RH), tsem(T,Sigma,Ti),ty(Theta,Ei,ge(N2,Vs2)),
    not cinterval((ge(N1,V1),TC),(ge(N2,V2),T),H,((Ei,Ti),(_,_),Sigma)).

closed(GE1,GE2,H,L):- findall(E,cinterval(GE1,GE2,H,E),L).
open(GE1,GE2,H,L):- findall(E,ointerval(GE1,GE2,H,E),L).
```

The predicate tsem(T, Sigma, Ti) implements the semantic for time expressions and time constraints. The predicate pick will pick an event from the history and return the rest of the history after that event.

5.2 Application: Obligation Models

In this section, we consider the rewrite-based semantics of CBACO [1], where the status of an obligation (a, r, ge_1, ge_2) (see Definition 1) for principal p in a given history h is computed using the following rewrite rule:

$$\begin{array}{l}\mathsf{eval\text{-}obligation}(p, a, r, ge_1, ge_2, h) \to if\ \mathsf{opar}(p, a, r, ge_1, ge_2)\ then \\ \quad \mathsf{append}(\mathsf{chk\text{-}cl}^*(\mathsf{closed}(ge_1, ge_2, h), p, a, r), \mathsf{chk\text{-}op}^*(\mathsf{open}(ge_1, ge_2, h), p, a, r)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad else\ [not\text{-}applicable]\end{array}$$

Here the function opar, specific to the system being modelled, is such that $\mathsf{opar}(p, a, r, ge_1, ge_2)$ holds if principal p has the generic obligation (a, r, ge_1, ge_2); append is a standard function that concatenates two lists; closed computes the sublists of h that start and finish with events e_1, e_2, which are respectively instances of ge_1, ge_2 (in this case e_2 closes the interval for this obligation). Similarly open returns the subhistories of h that start with an event e_1 (instance of ge_1) and for which there is no instance of ge_2 in h that closes the interval for this obligation. The function chk-cl with inputs h', p, a, r checks whether in the subhistory h' there is an event where the principal p has performed the action a on the resource r, returning a result fulfilled if that is the case, and violated otherwise. The function chk-op with inputs h', p, a, r checks whether in the subhistory

h' there is an event where the principal p has performed the action a on the resource r, returning a result fulfilled if that is the case, and pending otherwise. The functions chk-cl* and chk-op* do the same but for each element of a list of sub-histories, returning a list of results. Using the functions and relations defined in the previous sections, we can evaluate obligations according to the above specification. First, we give an alternative, equivalent specification for the evaluation of obligations, which is closer to the logic-programming implementation discussed in the previous subsection. Assuming that ty is the function that implements the instance relation $\vdash_\theta$ on events (that is, $\mathsf{ty}(e) = \{(ge, \theta) \mid \ \vdash_\theta e :: ge\}$), the status of an obligation can be computed using the following rule, where the extra variables in the right hand side are existentially quantified.

```
status(p, a, r, ge_1, ge_2, h) → if opar(p, a, r, ge_1, ge_2)
                                 then if ((e_1, t_1), (e_2, t_2)) ∈ closed(ge_1, ge_2, h)
                                   and (e, t) ∈ h and (ge, θ) ∈ ty(e) and
                                   geθ = {principal = p, action = a, resource = r} then
                                     if t_1 ≺ t ≺ t_2 then fulfilled else violated
                                 elseif ((e_1, t_1), ⊥) ∈ open(ge_1, ge_2, h)
                                   and (e, t) ∈ h and (ge, θ) ∈ ty(e) and
                                   geθ = {principal = p, action = a, resource = r} and
                                   t_1 ≺ t then fulfilled else pending
                                 else not-applicable
```

We are assuming that h contains all the events occurring in the system up to the moment where we wish to check the status of the obligation.

We now give the Prolog implementation for the rule that computes the status of an obligation. To deal with obligations we need Prolog facts/predicates to represent relations in CBACO. In particular, in order to implement assignment of obligations to principals, we have a predicate `opar(P,A,R,GE1,GE2)`. We assume the existence of a generic event `ge(par,[var(P,pl),var(A,act),var(R,res)])` with specification `rec([lab(principal,var(P,pl)),lab(action,var(A,act)), lab(resource,var(R,res))])`, where `pl,act,res` are the types for principal, actions and resources, respectively.

```
status(P,A,R,ge(N1,V1),ge(N2,V2),H,notapplicable):-
   not opar(P,A,R,GE1,GE2),!.
status(P,A,R,ge(N1,V1),ge(N2,V2),H,S):- opar(P,A,R,GE1,GE2),
   closed(GE1,GE2,H,CI), member((E1,E2),CI), event(P,A,R,H,(E,T)),
   chktime(T,T1,T2,S).
status(P,A,R,ge(N1,V1),ge(N2,V2),H,fulfilled):- opar(P,A,R,GE1,GE2),
   open(GE1,GE2,H,CI), member((E1,bot),CI), event(P,A,R,H,(E,T)),!,
   tsem(T1,Theta,Time1) tsem(T,Theta,Time), Time>=Time1.
status(P,A,R,ge(N1,V1),ge(N2,V2),H,pending):- opar(P,A,R,GE1,GE2),
   open(GE1,GE2,H,CI), member((E1,bot),CI).
event(P,A,R,H,(E,T)):- member((E,T),H),
   ty(Theta,E,ge(par,[var(P1,pl),var(A1,act),var(R1,res)])),
   member((P1,P),Theta), member((A1,A),Theta), member((R1,R),Theta).
checktime(Time,Time1,Time2,fulfilled):- tsem(Time1,Theta,T1),
                                         tsem(Time2,Theta,T2),
                                    tsem(Time,Theta,T), T>=T1,T2>=T.
checktime(Time,Time1,Time2,invalid):- tsem(Time1,Theta,T1),
                                      tsem(Time2,Theta,T2),
                                      tsem(Time,Theta,T), (T1>=T;T>=T2).
```

The rewrite-based specification of duties (see Definition 2) in CBACO [1] relies on auxiliary functions interval, and type, which are also specific to the system being modelled: $\mathsf{interval}(e_1, e_2, h)$ checks whether the event history includes an interval opened by e_1 and closed by e_2, and $\mathsf{type}(e, h)$ computes the generic event *ge*, of which e occurring in h is an instance (and that, in the rule below, is assumed to be unique).

$$\mathsf{duty}(p, a, r, e_1, e_2, h) \to \mathsf{opar}(p, a, r, \mathsf{type}(e_1, h), \mathsf{type}(e_2, h)) \ \mathit{and}\ \mathsf{interval}(e_1, e_2, h)$$

In [1] interval and type are assumed to be defined for each specific system, to respectively implement the relations *event interval* and *event typing.* In this paper we give general definitions/implementations of these relations. The implementation of a checker for duties in Prolog is straightforward, using the predicate defined above to compute intervals, which takes into account the type relation between events.

```
duty(P,A,R,E1,E2,H):- opar(P,A,R,GE1,GE2), cinterval(GE1,GE2,H,(E1,E2,_)).
```

6 Related Work

The notion of event has been treated in various settings in the literature, such as logic-based frameworks, algebraic approaches and query languages, amongst others. In the context of access control, Barker et al [5] have proposed a representation for events as sets of binary predicates, partially motivated by Davidson's view of events as action occurrences [10]. In this formalism, event descriptions are given as finite sets of ground 2-place facts (atoms) that describe an event, uniquely identified by $e_i, i \in \mathbb{N}$, and which includes three necessary facts: $happens(e_i, t_j)$, $act(e_i, a_l)$ and $agent(e_i, u_n)$, and n non-necessary facts. This was later used in [1] to model obligations in the CBAC metamodel, but considering only two necessary facts *happens* and *act*. This representation is claimed to be more flexible than a term-based representation with a fixed set of attributes. In our language, we do not fix necessary facts, although one can define them as part of the set of typed-functors. Furthermore, event specifications are given as records which may contain extra fields, so these sets of predicates can be easily encoded in our language. A less flexible representation was used in [8], in the context of distributed event-based access control. In this work, events are ground terms of the form $\mathsf{event}(e_i, u, a, t)$ where event is a data constructor of arity four, $e_i (i \in \mathbb{N})$ denotes unique event identifiers, u identifies a user, a is an action associated to the event, and t is the time when the event happened. When it is sufficient to know the chronological order of events, then the history can be ordered as to provide that information and the time parameter may be omitted.

The *Obligation Specification Language* (OSL) defined in [13], presents a language for events to monitor and reason about data usage requirements. The notions of obligational formulas/obligations defined in this paper are closely related to the notions of generic/concrete obligations in [1]. Therefore data usage as specified in [13] can be encoded within the CBACO metamodel. The notion of events presented in [13] is also similar to ours in some aspects, but where logical expressions are used to deal with intervals. The paper also presents a relation *refinesEv*, defining an instance relation between events. This relation is based on a subset relation on labels, as the instance relation in [1]. In our setting this instance relation between events is defined for parameterised generic events (i.e. containing variables), by the implicit subtyping on records but more generally using variable instantiation.

Still in the context of access control systems, Bertino et al.[7] proposed the Temporal Role-Based Access Control Model (TRBAC), using events to activate and deactivate roles. This was later used in [23, 22], to deal with security analysis in the presence of static temporal role hierarchies in RBAC. The *time models* used in these works also depend on the notion of time interval, but they use a simpler notion of interval that can easily be encoded in our language. The activation and de-activation of roles, as well as dealing with the so-called *safety problem* (i.e., administrative actions that can lead to a policy in which a user can acquire permissions that can compromise the security of the system), is not the purpose of events in our work. Nevertheless, this can achieved, through the assignment of users to categories in CBAC policies, based on some property depending on a temporal constraint.

An important notion in the above formalisms, and in the language described in this paper, is the notion of interval, which provides means to reason about assignment of status in [5] and status of obligations in [1]. Intervals as sequences that are initiated and terminated by events, and during which certain facts hold, are also a key aspect in the event calculus [17, 18]. The initial motivation of the event calculus was to deal with database updating, but it has been applied in a variety of settings [15, 9, 12, 16]. Like in the event calculus, we also consider intervals as being initiated and closed by events, however we do not reason (in general) about facts that hold at a certain point.

Time intervals and time constraints have also been used to appropriately deal with obligations in access control models [1, 14, 19, 21]. In most of these models time intervals are not defined by events, but as fixed points in time, which are easily represented in our language. Time constraints in [19] consider sequences of time intervals, to enforce systematic repetition of obligations. We do not consider this type of constraints, as repetition of obligations can be enforced through the definition of the categories for obligations, but our representation of time constraints could easily be adapted to consider sequences of intervals.

7 Conclusions

We have defined a language to represent events as typed-terms, built from a user-defined signature, to formally deal with the notions of event typing and event intervals in a uniform way, in the context of the CBACO metamodel. In a given system, intervals can be automatically extracted from a history of events by means of a relation that determines how events are closed in the system. This approach allows us to adequately define general functions to implement event typing and to compute event intervals, without having to know the exact type of events that we are dealing with. As future work we would like to extend this formalism to deal with notions such as conflicting events, and automatically generated events. Furthermore, we believe that a type-system for events could be useful in identifying patterns of events in history, which could lead to interesting applications in the context of event processing. We believe that the notions of intervals defined here could be useful in other contexts. In particular, it could be used to infer intervals where a particular status is valid, which can be applied in status-based access control models.

References

1. S. Alves, A. Degtyarev, and M. Fernández. Access Control and Obligations in the Category-Based Metamodel: A Rewrite-Based Semantics. In *Proceedings of LOPSTR'14*, volume 8981 of *LNCS*, pages 148–163. Springer, 2015.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, Great Britain, 1998.
3. S. Barker. Action-status access control. In *Proceedings of SACMAT'07*, pages 195–204. ACM, 2007.
4. S. Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of SACMAT'09*, pages 187–196. ACM, 2009.
5. S. Barker, M. J. Sergot, and D. Wijesekera. Status-Based Access Control. *ACM Transactions on Information and System Security*, 12(1):1:1–1:47, 2008.
6. D. E. Bell and L. J. Lapadula. Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, 1976.
7. E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-based Access Control Model. *ACM Transactions on Information and System Security*, 4(3):191–233, Aug. 2001.
8. C. Bertolissi, M. Fernández, and S. Barker. Dynamic Event-Based Access Control as Term Rewriting. In *Proceedings of DBSEC'07*, volume 4602 of *LNCS*, pages 195–210. Springer, 2007.
9. R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, and A. Bandara. Expressive Policy Analysis with Enhanced System Dynamicity. In *Proceedings of ASIACCS'09*, pages 239–250. ACM, 2009.
10. D. Davidson. *Essays on Actions and Events.* Oxford University Press, 2001.
11. D. Ferraiolo, R. Kuhn, and R. Chandramouli. *Role-Based Access Control.* Artech House, 2003.
12. M. Gelfond and J. Lobo. Authorization and Obligation Policies in Dynamic Systems. In *Proceedings of ICLP'08*, volume 5366 of *LNCS*, pages 22–36. Springer, 2008.
13. M. Hilty, A. Pretschner, D. A. Basin, C. Schaefer, and T. Walter. A Policy Language for Distributed Usage Control. In *Proceedings of ESORICS'07*, pages 531–546, 2007.
14. K. Irwin, T. Yu, and W. H. Winsborough. On the Modeling and Analysis of Obligations. In *Proceedings of CCS'06*, pages 134–143. ACM, 2006.

15. R. Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, 12(1-2):121–146, Jan. 1992.
16. R. Kowalski and F. Sadri. A Logic-based Framework for Reactive Systems. In *Proceedings of RuleML'12*, pages 1–15. Springer-Verlag, 2012.
17. R. Kowalski and M. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.
18. R. Miller and M. Shanahan. The Event Calculus in Classical Logic - Alternative Axiomatisations. *Electronic Transactions on Artificial Intelligence*, 3(A):77–105, 1999.
19. Q. Ni, E. Bertino, and J. Lobo. An Obligation Model Bridging Access Control Policies and Privacy Policies. In *Proceedings of SACMAT'08*, pages 133–142. ACM, 2008.
20. Ohori. A Polymorphic Record Calculus and its Compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov. 1995.
21. M. Pontual, O. Chowdhury, W. H. Winsborough, T. Yu, and K. Irwin. On the Management of User Obligations. In *Proceedings of SACMAT '11*, pages 175–184. ACM, 2011.
22. S. Ranise, A. T. Truong, and A. Armando. Scalable and precise automated analysis of administrative temporal role-based access control. In *Proceedings of SACMAT'14*, pages 103–114, 2014.
23. S. Ranise, A. T. Truong, and L. Viganò. Automated analysis of RBAC policies with temporal constraints and static role hierarchies. In *Proceedings of SAC'15*, pages 2177–2184, 2015.