Essaouini et al. EURASIP Journal on Information Security 2014, 2014:13
http://jis.eurasipjournals.com/content/2014/1/13

EURASIP Journal on Information Security
a SpringerOpen Journal

RESEARCH Open Access

Conflict detection in obligation with deadline policies

Nada Essaouini[1,2*], Frédéric Cuppens[1], Nora Cuppens-Boulahia[1] and Anas Abou El Kalam[2]

Abstract

Many papers have already provided models to formally specify security policies. In this paper, security policies are modeled using deontic concepts of permission and obligation. Permission rules are used to specify access control policies, while obligation rules are useful to specify other security requirements corresponding to usage control policies as the availability of information in its allotted time. However, when both permission and obligation concepts are used to express security policies, several different types of conflict can be raised and should be detected and managed. We are interested in this work in managing conflicts between obligations with deadlines and permissions. Thus, we first begin by formally defining the conflicting situations using the situation calculus. Afterwards, we provide an algorithm for searching a plan of actions, when it exists, which fulfills all the active obligations in a given situation in their deadlines with respect to the permission rules. The length of the plan is set in advance and can be calculated in the case where the sets of actions and fluents are finite to ensure the decidability of the solution search. Furthermore, in the plan search, the choice of the execution time of the elected actions obeys to equations and inequalities which need to be solved. For this purpose, we need a component allowing these equations and inequalities resolution. To illustrate our approach, we take an example inspired from existing laws in hospitals regulating deadlines for completion of patient medical records. The example is formally specified in our language and implemented in ECRC Common Logic Programming System ECLIPSE 3.5.2, which is equipped with Simplex algorithm for solving linear equations and inequalities over the reals. In the implementation, we show how the plan search can be optimized through the use of some heuristics and make some evaluation tests.

Keywords: Security policy; Conflict detection; Obligation with deadline; Situation calculus

Introduction

A security policy is often defined as permission, prohibition, obligation, and exemption rules. Permission and prohibition rules are used to specify access control policies. Obligation and exemption rules are useful to specify other security requirements corresponding to usage control policies [1,2]. In the usage control literature, two different types of obligation are generally considered called system obligation and user obligation [3]. When the security policy includes user obligation, these obligations should be associated with deadlines. When these obligations are activated, these deadlines provide the user with some time to enforce the obligation before violation occurs.

*Correspondence: nada.essaouini@telecom-bretagne.eu
[1]Télécom Bretagne, 2 rue Châtaigneraie, Cesson Sévigné Cedex 3, France
[2]Cadi Ayyad University, ENSA, Boulevard Abdelkrim Al Khattabi, Marrakesh, Marrakesh 40000, Morocco

The application of these rules to the same object may lead to conflicting situations. Preliminary work on the classification of conflicts are reported in [4], where several types of conflicts have been defined (see also [5,6]). Benferhat et al. [7] presents an approach based on possibilistic logic to deal with conflicts in prioritized security policies. However, there is another type of conflict which is not managed yet, namely, the conflict between obligations with deadlines. This kind of conflict could happen in the case of overlapping deadlines. For example: (i) The doctor is obliged to fill in the summary sheet within 1 h after the patient leaves. (ii) The surgeon must be vigilant in the operating room. If the doctor is a surgeon and he is in the operating room during a patient's leaving, and if the duration of the surgery ends 2 h after the patient's leaving, the surgeon cannot fill in the summary sheet of the patient because the surgery could end after the deadline associated with filling the summary sheet. Thus, there may be

Springer

© 2014 Essaouini et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

situations where it is impossible to meet certain obligation requirements of the security policy before their deadlines.

Conflicts between obligations with deadline are more complex to detect and manage. We need a model which manages how the information system evolves over the time. In this paper, we use a language based on deontic logic to specify security policies that include obligations with deadline. The advantage of deontic logic is that it provides means to consistently reason about deontic concepts as obligation and permission. Then, we suggest an approach based on the sequential temporal situation calculus [8] to give semantic to our language. The Situation Calculus allows us to analyze decidability and complexity of several useful problems:

- Temporal projection problem [9]: asks whether a formula holds after a sequence of actions is performed in the initial situation. This is useful to decide which rule can be applied to a given situation and detect violation.
- Planning [10]: given a goal formula, planning consists in finding a sequence of actions so that the goal is satisfied after executing this sequence of actions. We show how to detect, using planning task, if there is conflict between obligation with deadline rules. Then, we introduce the concept of legal plan to detect conflict between obligation and permission rules.

To illustrate our approach, we take the example of completion of medical records inspired from existing laws in hospitals [11]. This completion is regulated by obligation rules with deadline. Each rule specifies the associated deadline to complete each document in the patient record. Any latency on writing patient record could affect the information availability time for each patient which negatively impacts the quality of provided care. This has led some hospitals to specify sanctions when these deadlines are not respected, see for example the Ontario regulations [11]. The example shows a real need to have obligations with deadline in security policy and a real need to manage the conflicts between them.

The present work is an extended version of a previous conference paper [12]. The main contributions with respect to this extended version are the following:

- The initial proposed formalism is extended with two new modalities for expressing permissions and system obligations.
- Managing permissions induces a new type of conflict which occurs when it is impossible to find a sequence of permitted actions which leads to a situation where obligations are fulfilled in their deadlines. We formally define the situations which correspond to such conflicts by introducing the concept of a legal plan.
- The algorithm for detection of conflict between obligations with deadline initially proposed in the previous paper is extended to allow the detection of conflicts between permissions and obligations with deadlines.
- The previous implementation is extended to support the search for a legal plan. And finally, we made some evaluation tests and propose some optimization tools using heuristics.

This article is organized as follows. In Section 1, we give a motivation example. Section 2 presents the situation calculus. Section 3 explains how to define security policies that include obligations with deadline. This model is based on deontic logic, and a security policy is viewed as a set of deontic norms. Section 5 extends situation calculus to formally derive which actual norms apply in a given situation. In this section, we also formally define when an obligation with deadline is violated. Section 5 shows how to detect the presence of conflicting norms in the policy. In Section 6, we give the specification of the motivation example. In Section 7, we make the implementation of our model using the programming language GOLOG [13]. In this section, we make assessment on different situations that we built to simulate our model on the use case and discuss some performance evaluation. The related work is presented in Section 8. Finally, we present the conclusion and perspectives.

1 Motivation example

In the medical community, patient's record contains information about care provided to the patient during his stay in the hospital. The medical records are regulated by hospitals through legal texts [11]. These laws specify, in particular, the time given to doctors to complete patient records assigned to them. In hospitals where medical records are digitally stored, these rules may be expressed as obligations with deadlines. These rules aim to ensure the availability of medical information in expected time. In this section, we describe the impact of availability of medical information in expected time on the quality of patient care, and we give an example of obligations with deadline concerning completion of medical records.

1.1 Impact of deadlines to complete medical records on the availability of information

Studies have shown that patient care can be improved by timely sending a complete and accurate information on patient hospitalization to the practitioner [14-17]. In contrast, a breakdown of communication, due to delays in the transfer of information or incomplete information, can have serious consequences. For example, the physician who does not have access to the summary sheet of a patient hospitalization prepared by acute care services

is in an uncomfortable situation when the patient's life is in danger. Despite what has been raised by these studies on the importance of time when transferring patient information, other studies have noticed that in practice, there is a lag in the transfer of this information. Some of these studies noticed a significant delay between the time when the patient receives his leave and when the generalist physician received the advice [18,19]. Therefore, the hospitals are required to establish regulations so that the medical records are filled timely to ensure continuity of patient care. In what follows, we give some examples of rules concerning the deadline assigned to doctors to complete certain elements of patient's records.

1.2 Rules regarding the completion of patient's medical record

The law on public hospitals specifies that medical records must be filled for any person registered or admitted to a health facility [11]. Also, it specifies the elements that a medical record must contain. The law may specify the deadline given to doctors so that each element is present in the patient's record, and the appropriate measures when these deadlines are not respected, see for example the Ontario regulations [11]. Among the documents that must be found in the medical records are as follows: summary sheet, admission note, medical observation, operating protocol, and discharge note. The time to make these documents present in the folder of the user differs from one document to the other:

- The medical summary/summary sheet: When a doctor authorizes the patient assigned to him to leave the hospital, he must complete the medical summary of this patient before this latter leaves the hospital.
- Admission note: The doctor must complete the admission note of the patient assigned to him when he is admitted in the hospital within 30 min following his admission.
- Medical observation: The doctor must complete the medical observation of the patient assigned to him when he is admitted in the hospital within 40 min following his admission.
- Operating Protocol: The doctor who did a surgery for a patient assigned to him must complete the operating protocol of this patient within 100 min following the intervention.
- Discharge note: If a doctor authorizes the patient assigned to him to leave the hospital, he must complete his discharge note. The discharge note must be completed before the patient's leaving.

2 Situation calculus

The situation calculus [20] is a second-order logic language specially designed to represent the change in dynamic worlds. The ontology and axiomatization of the sequential situation calculus was extended to include time [8], concurrency, and natural actions [21]. However, in all cases, the basic elements of language are actions, situations, and fluents. The situation language used in this paper is described below.

2.1 The language

The language consists of the following ontology:

- All changes in the world are the results of actions. They are designated by terms of first-order logic. To represent the time in the situation calculus, we add a time argument in all instantaneous actions which is used to specify the exact time or time range in which the actions occur in world history. For example, *sign(Jean, dischargeNote(Mary), 100)* is the instantaneous action of signing the discharge note of *Mary* by *Jean* at the moment 100. The actions are instantaneous, but we can express actions with duration. For example, consider the following two instantaneous actions, *startConsultation(d, p, t)*, meaning d starts consultation of p at time t, and $endConsultation(d, p, t')$, meaning d ends consultation of p at time t'. The fluent *inConsultation(p, s)*, expressing the patient p is in consultation in the situation s, turns from false to true if there exists a time t and doctor d when the action *startConsultation(d, p, t)* is performed, and turns to false if there exists a time t' when the action $endConsultation(d, p, t')$ is performed. Thus, in situations where fluent *inConsultation(p, s)* is true, we can describe the properties of the world, such as the heartbeat of p per unit time, as a function of time that must be true during advancement of consultation.
- A possible history of the world, which is a sequence of actions is represented by the first-order terms denoted *situation*. The constant S_0 is the initial situation.
- There is a binary function symbol *do*; $\mathrm{do}(\alpha, s)$ denotes the situation resulting from the execution of the action α in the situation s. For example, *do(write(Jean, dischargeNote(Mary), 5), do(write(Jean, consultationReport(Mary), 8), do(write(Jean, admissionNote(Mary), 10),* S_0*)))* is the situation indicating the history of the world which consists of the execution of the sequence of actions *[write(Jean, admissionNote(Mary), 10), write(Jean, consultationReport(Mary), 8), write(Jean, dischargeNote(Mary), 5)]*.
- *Fluents* describing the facts of a state. There are two types of fluents: *relational fluents* and *functional fluents*. Relational fluents are symbols of predicates

which take a term of type *situation* as the last argument, which their truth values may vary from one situation to another. For example, *inConsultation(Mary, s)*, means that *Mary* is in consultation at situation *s*. Functional fluents are denoted by function symbols that take a situation as the last argument, which the truth of their function values changes from one situation to another. For example, *heartbeat(Mary, s)* denotes the number of heartbeats of *Mary* in situation *s*.
- There are also symbols of predicates and functions (including constants) denoting relations and functions independent of situations.
- A particular binary predicate symbol $<$, defines a strict order relation on situations; $s < s'$ means that we can reach s' by a sequence of actions starting from s. For instance, $\text{do}(a_2, \text{do}(a_1, S_0)) < \text{do}(a_4, \text{do}(a_3, \text{do}(a_2, \text{do}(a_1, S_0))))$.
- A second particular binary predicate symbol *Poss*, defines when an action is possible. $\text{Poss}(a, s)$ means that the action a can be executed in the situation s.
- A function symbol *time*: *time(a)* denotes the time when the action a occurs.
- A function symbol *start*: *start(s)* denotes the start time of the situation s.

2.2 Fundamental axioms

The basic axioms for the situation calculus, as defined in [22] and [23] are as follows:

- The second-order induction axiom:

$$(\forall P).\,[P(S_0) \wedge (\forall a, \sigma)(P(\sigma) \rightarrow P(do(a, \sigma)))] \rightarrow (\forall \sigma)P(\sigma)$$

The induction axiom says that to prove that property P is true in all situations, it is sufficient to prove that P is true in the initial situation S_0 (initialization step) and for all actions a and situations σ, if P is true in the situation σ, then P is still true in the situation $\text{do}(a, \sigma)$ (induction step). The axiom is necessary to prove properties true in all situations [24].

- The unique name axioms:

$$S_0 \neq \text{do}(a, s),$$
$$\text{do}(a, s) = \text{do}\left(a', s'\right) \rightarrow a = a' \wedge s = s'$$

- Axioms that define an order relation $<$ on situations:

$$\neg s < S_0,$$
$$s < \text{do}\left(a, s'\right) \leftrightarrow \left(\text{Poss}\left(a, s'\right) \wedge \text{start}\left(s'\right) \leq \text{time}(a) \wedge s \leq s'\right).$$

- The axiom: $\text{start}(\text{do}(a, s)) = \text{time}(a)$.

In addition to the axioms described above, we need to describe a class of axioms when we formalize an application domain:

- *Action precondition axioms*, one for each action:

$$\text{Poss}\left(A\left(\vec{x}, t\right), s\right) \leftrightarrow \phi\left(\vec{x}, t, s\right),$$

where $\phi\left(\vec{x}, t, s\right)$ characterizes the preconditions of the action A, it is any first-order formula with free variables among $\vec{x}$, t, and whose only term of sort of *situation* is s.
For example, a patient can leave the hospital if he is in the hospital.

$$\text{Poss}\left(\textit{leave}\left(p, t\right), s\right) \leftrightarrow \text{inpatient}(p, s)$$

Using predicate $\text{poss}(a)$, we can then recursively specify that a given situation s is executable.

$$\text{Executable}(s) \leftrightarrow$$
$$\left(\forall a, s'\right).\text{do}\left(a, s'\right) \leq s \rightarrow \left(\text{Poss}\left(a, s'\right) \wedge \text{start}\left(s'\right) \leq \text{time}(a)\right).$$

- *Successor state axioms*, one for each fluent. These axioms characterize the effects of actions on fluents and they embody a solution to the frame problem[a] for deterministic actions [23].
The syntactic form of successor state axiom for a relational fluent F is

$$\text{Poss}(a, s) \rightarrow$$
$$\left[F\left(\vec{x}, \text{do}(a, s)\right) \leftrightarrow \gamma_F^+\left(\vec{x}, a, s\right) \vee\right.$$
$$\left.\left(F\left(\vec{x}, s\right) \wedge \neg\gamma_F^-\left(\vec{x}, a, s\right)\right)\right],$$

where $\gamma_F^+\left(\vec{x}, a, s\right)$ and $\gamma_F^-(\vec{x}, a, s)$ indicate the conditions under which if the action a is executed in situation s, $F(\vec{x}, \text{do}(a, s))$ becomes true and false, respectively.
For example, the succession state axiom of fluent $\text{assigned}(p, d, s)$, meaning a patient p is assigned to a doctor d can be defined as follows:

$$\text{Poss}(a, s) \rightarrow$$
$$\text{assigned}\left(p, d, \text{do}\left(a, s\right)\right) \leftrightarrow \left[(\exists t)a = \text{assign}(p, d, t) \vee\right.$$
$$(\text{assigned}(p, d, s) \wedge \neg(\exists t)a = \text{revokeAssignment}(p, d, t) \wedge$$
$$\left.\neg(\exists t)a = \text{leave}(p, t))\right]$$

Here, $\gamma_F^+\left(\vec{x}, a, s\right)$ corresponds to the formula: $(\exists t)a = \text{assign}(p, d, t)$ and $\gamma_F^-\left(\vec{x}, a, s\right)$ is the formula: $(\exists t)a = \text{revokeAssignment}(p, d, t) \vee (\exists t)a = \text{leave}(p, t)$. The action *assign* makes the fluent *assigned* true, and the actions *revokeAssignment* and *leave* turn the fluent *assigned* to false.
It is assumed that no action can turn F to be both true and false in a situation, i.e., $\neg\exists s \exists a \gamma_F^+(\vec{x}, a, s) \wedge \gamma_F^-(\vec{x}, a, s)$.
For a functional fluent, the syntactic form of successor state axiom is

$$\text{Poss}(a, s) \rightarrow$$
$$\left[F\left(\vec{x}, \text{do}(a, s)\right) = y \leftrightarrow \gamma_F^+\left(\vec{x}, y, a, s\right) \vee\right.$$
$$\left.\left(y = F\left(\vec{x}, s\right) \wedge \neg\left(\exists y'\right)\gamma_F^-\left(\vec{x}, y', a, s\right)\right)\right],$$

where, $\gamma_F\left(\vec{x}, y, a, s\right)$ is any first-order formula with free variables among $\vec{x}$, y, a, t, and whose only term of sort of *situation* is s.

- Axioms describing the initial situation.
- In each application involving a particular action $A\left(\vec{x}, t\right)$, an axiom that gives the time of the action A: $time\left(A\left(\vec{x}, t\right)\right) = t$.

In the following, we denote $\text{Axioms} = \Sigma \cup A_{\text{ss}} \cup A_{\text{ap}} \cup A_{S_0}$, where

- Σ is the foundational axiomatic of the situation calculus.
- A_{ss} is a set of successor state axioms.
- A_{ap} is a set of action precondition axioms.
- A_{S_0} is a set of initial situation axioms. A_{S_0} is a set of sentences with the property that S_0 is the only term of sort situation mentioned by the fluents of a sentence of A_{S_0}. Thus, no fluent of a formula of A_{S_0} mentions a variable of sort situation or the function symbol *do*.

We denote $\text{Axioms} \vdash p$ the fact that the sentence p can be derived from the set of axioms *Axioms*. This kind of domain theories provides us with various reasoning capabilities, for instance planning [25]. Given a domain theory *Axioms* as above and a goal formula $G(s)$ with a single free-variable s, the planing task is to find a sequence of actions $\vec{a}$ such that

$$\text{Axioms} \vdash s_0 \leq \text{do}(\vec{a}, s_0) \wedge \text{Executable}(\text{do}(\vec{a}, s_0)) \wedge G(\text{do}(\vec{a}, s_0)),$$

where $\text{do}([a_1, \ldots, a_n], s)$ is an abbreviation for $\text{do}(a_n, \text{do}(a_{n-1}, \ldots, \text{do}(a_1, s)\ldots))$.

3 Security policy specification

The language we define to specify permissions and obligations in security policies is based on deontic logic of actions. We consider two modalities: permissions and obligations with deadline. They are called normative modalities in the following. Normative modalities are represented as dyadic conditional modalities. Permissions are specified using dyadic modality $P(\alpha|p)$, where α is an action of $\mathcal{A}$ and p is the condition of the permission. The condition is any formula built using fluents of $\mathcal{F}$ without situation. $P(\alpha|p)$ means that the action α is permitted when condition p holds. Obligations with deadline are specified using modality $O(\alpha < d|p)$ which intuitively means that when formula p starts to hold, there is an obligation to execute action α before the deadline condition d starts to hold. In the following, we assume that the deadline condition must be an atomic fluent predicate of $\mathcal{F}$. If the action α is executed before the deadline condition d starts to hold, then we shall say that the obligation is fulfilled. Else, we shall consider that the obligation is violated. We call *norm* a formula corresponding to a conditional permission or obligation with deadline. A security policy, $\mathcal{P}$ is a finite set of norms. We shall now use the situation calculus to formally define the semantics of these different modalities.

4 Actual norm derivation and violation detection

The situation calculus is extended with fluents $\text{Perm}(\alpha)$ (there is an actual permission to do α) and $\text{Ob}(\alpha < d)$ (the obligation to do α before deadline d starts to be effective), where α is an action of $\mathcal{A}$ and d is a fluent of $\mathcal{F}$. We first extend the set of axioms previously defined with a permission definition axiom for every fluent predicate $\text{Perm}(\alpha)$, $\alpha \in \mathcal{A}$. For this purpose, let P_α be the set of conditional permissions having the form $P(\alpha|p)$. We denote $\psi_{P_\alpha} = p_1 \vee \ldots \vee p_n$, where each p_i for $i \in [1, \ldots, n]$ corresponds to the condition of a permission in P_α. If $P_\alpha = \emptyset$, then we assume that $\psi_{P_\alpha} = \text{false}$. Using ψ_{P_α}, the successor state axiom for $\text{Perm}(\alpha, \sigma)$ is defined as follows:

$$\begin{aligned} &\text{Poss}\,(a, \sigma) \rightarrow \\ &\text{Perm}\,(\alpha, \text{do}\,(a, \sigma)) \leftrightarrow \left[\gamma^{+}_{\psi_{P_\alpha}}(a, \sigma) \vee \right. \\ &\left.\left(\text{Perm}\,(\alpha, \sigma) \wedge \neg\gamma^{-}_{\psi_{P_\alpha}}(a, \sigma)\right)\right] \end{aligned} \quad (1)$$

This axiom specifies that the permission to do an action becomes effective after the action that activates the context of the permission rule is executed. This permission remains effective until an action that turns the activation context to false is executed.

We can specify a predicate permitted which specifies that a given situation is secure with respect to access control requirements as follows:

$$\begin{aligned} &\text{Permitted}(S_0) \wedge \\ &(\forall a \forall \sigma)\,[\text{Permitted}\,(\text{do}\,(a, \sigma)) \leftrightarrow (\text{Perm}\,(a, \sigma) \wedge \text{Permitted}\,(\sigma))] \end{aligned}$$

We can then specify a condition to prove that the system specification represented by a given set of *Axioms* is secure with respect to access control requirements as follows:

$$\text{Axioms} \vdash (\forall \sigma)\,(\text{Executable}\,(\sigma) \rightarrow \text{Permitted}\,(\sigma))$$

which corresponds to proving that predicate permitted is an integrity constraint in every executable situation. We call this integrity constraint the 'close policy requirement'. It is easy to show that a sufficient condition to prove the close policy requirement consists in strengthening the action precondition axiom of every action α with the guarded condition that this action must be permitted:

$$\forall \sigma, \text{poss}\,(\alpha, \sigma) \leftrightarrow (\phi\,(\sigma) \wedge \text{Perm}\,(\alpha, \sigma))$$

Using permitted situations, we introduce a notion of *legal plan*. Given a goal formula G, a legal plan consists of finding a permitted situation that satisfies G.

Let us now turn to the obligation definition axiom for every fluent predicate $\text{Ob}(\alpha < d)$, where $\alpha \in \mathcal{A}$ and $d \in \mathcal{F}$. Notice that since the sets $\mathcal{A}$ and $\mathcal{F}$ are finite, we have

a finite set of successor state axioms to define for $\mathrm{Ob}(\alpha < d)$. We define $O_{\alpha,d}$ to be the set of conditional obligations with deadline in P having the form $\mathrm{Ob}(\alpha' < d'|p)$, such that $\alpha = \alpha'$ and d and d' are logically equivalent. We say that two fluent predicates d and d' are logically equivalent with respect to a set of *Axioms* if we can prove that $d \leftrightarrow d'$ is an integrity constraint of *Axioms*. We denote $\psi_{O_{\alpha,d}} = p_1 \vee \ldots \vee p_n$, where each p_i for $i \in [1, \ldots, n]$ corresponds to the condition of an obligation in $O_{\alpha,d}$. If $O_{\alpha,d} = \emptyset$, then we assume that $\psi_{O_{\alpha,d}} = \text{false}$. Using $\psi_{O_{\alpha,d}}$, the successor state axiom for $\mathrm{Ob}(\alpha < d)$ is defined as follows:

$$\begin{aligned}
&\mathrm{Poss}\,(a,\sigma) \rightarrow \\
&\mathrm{Ob}\,(\alpha < d, \mathrm{do}\,(a,\sigma)) \leftrightarrow \Big[\gamma^{+}_{\psi_{O_{\alpha,d}}}(a,\sigma) \vee \qquad (2)\\
&\Big(\mathrm{Ob}\,(\alpha < d,\sigma) \wedge \neg\,(a=\alpha) \wedge \neg\gamma^{+}_{d}(a,\sigma) \wedge \neg\gamma^{-}_{\psi_{O_{\alpha,d}}}(a,\sigma)\Big)\Big]
\end{aligned}$$

This axiom says that the obligation to do α before deadline d is activated when $\psi_{O_{\alpha,d}}$ starts to be true. This obligation is deactivated when it is fulfilled (i.e., action α is done) or it is violated (i.e., deadline d starts to be true) or condition $\psi_{O_{\alpha,d}}$ ends to be true (i.e., it is no longer relevant to do α).

We can characterize situations where the obligations are fulfilled by the following fluent:

$$\begin{aligned}
&\mathrm{Poss}(a,\sigma) \rightarrow \\
&\mathrm{Fulfil}\,(\alpha < d, \mathrm{do}(a,\sigma)) \leftrightarrow \Big[\big(\mathrm{Ob}(\alpha < d,\sigma) \wedge a = \alpha \wedge \neg\gamma^{+}_{d}(\alpha,\sigma)\big) \vee \\
&\mathrm{Fulfil}\,(\alpha < d,\sigma)\,\Big] \qquad (3)
\end{aligned}$$

Notice that if in a given situation σ, it simultaneously happens that the obligatory action is executed and the associated deadline is activated, then the decision is to consider that the obligation is violated and not fulfilled. This is called obligation with strict deadline. We can also define $O(\alpha \leq d|p)$ so that in the same situation, the obligation is fulfilled and not violated.

Finally, we define the succession state axiom of the fluent $\mathrm{Violated}(\alpha < d,\sigma)$:

$$\begin{aligned}
&\mathrm{Poss}(a,\sigma) \rightarrow \\
&\mathrm{Violated}\,(\alpha < d, \mathrm{do}(a,\sigma)) \leftrightarrow \Big[\big(\mathrm{Ob}\,(\alpha < d,\sigma) \wedge \gamma^{+}_{d}(a,\sigma)\big) \vee \\
&\mathrm{Violated}\,(\alpha < d,\sigma)\,\Big] \qquad (4)
\end{aligned}$$

This axiom specifies that an obligation to do α is violated, when the associated deadline comes true when it was still active, and it was never executed. The axiom also specifies that in a given situation σ, if it simultaneously happens that the obligatory action is executed and the associated deadline is activated, then the decision is to consider that the obligation is violated.

Concerning system obligations, we consider them as a special case of obligations with deadline, written as follows: $O(\alpha)$. As there is no deadline associated with these obligations, we assume that $\gamma^{+}_{d}(a,\sigma) = \gamma^{-}_{d}(a,\sigma) = \text{false}$. Thus, we can derive the succession state axiom characterizing the situations when system obligations are active using axiom 2.

$$\mathrm{Poss}(a,\sigma) \rightarrow \Big(\mathrm{Ob}(\alpha, \mathrm{do}(a,\sigma)) \leftrightarrow \gamma^{+}_{\psi_{O_\alpha}}(a,\sigma)\Big) \qquad (5)$$

This axiom says that the system obligation to do α is activated only in the situations when ψ_{O_α} starts to be true and they are deactivated immediately after. Thus, a system obligation should be fulfilled immediately after its activation. This can be derived using the axiom 3 as follows:

$$\begin{aligned}
&\mathrm{Poss}(a,\sigma) \rightarrow \\
&\mathrm{Fulfil}\,(\alpha, \mathrm{do}(a,\sigma)) \leftrightarrow [(\mathrm{Ob}(\alpha,\sigma) \wedge a = \alpha) \vee \mathrm{Fulfil}(\alpha,\sigma)]
\end{aligned}$$

When an obligation system is not executed immediately after its activation, a violation is detected using the following axiom:

$$\begin{aligned}
&\mathrm{Poss}(a,\sigma) \rightarrow \\
&\mathrm{Violated}\,(\alpha, \mathrm{do}(a,\sigma)) \leftrightarrow [(\mathrm{Ob}(\alpha,\sigma) \wedge \neg(a=\alpha)) \vee \mathrm{Violated}(\alpha,\sigma)]
\end{aligned}$$

5 Policy conflict detection

In this section, we define two kinds of conflict:

- conflict between obligation with deadline rules
- conflict between permission and obligation with deadline rules

The conflict between obligations is detected through the definition of the following situations:

- *situation locally enforceable* is defined in relation with a particular obligation and characterizes the fact that this obligation can be fulfilled by following the executable plan.
- *situation globally enforceable* characterizes the fact that all the active obligations can be enforced in an executable plan without violating the associated deadlines.

The conflict between permission and obligation with deadline rules is detected through the definition of the following situations:

- *situation legal locally enforceable* is defined in relation with a particular obligation and characterizes the fact that this obligation can be fulfilled by following an executable plan constituted of permitted actions (*legal plan*).
- *situation legal globally enforceable* characterizes the fact that all the active obligations can be enforced in an executable and legal plan without violating the associated deadlines.

5.1 Obligation conflicts

In a given situation, an active obligation is enforceable if it can be fulfilled following an executable plan.

$$\text{Enforceable}(\alpha < d, \sigma) \leftrightarrow [\text{Ob}(\alpha < d, \sigma) \wedge \left(\exists \sigma', \sigma < \sigma'\right) \left(\text{Fulfil}(\alpha < d, \sigma') \wedge \text{Executable}(\sigma')\right)]$$

It may happen that in one situation, every active obligation is enforceable, but it is still not possible to enforce all of them without violating the associated deadlines. If all the active obligations in a situation σ can be executed without violating at least one of them, we say that this situation is globally enforceable. To characterize this, we introduce the formula G-Enforceable(σ).

$$\text{G-Enforceable}(\sigma) \leftrightarrow (\exists \sigma', \sigma' > \sigma) (\forall \alpha, d) \left(\text{Ob}\left(\alpha < d, \sigma\right) \rightarrow \left(\text{Fulfil}\left(\alpha < d, \sigma'\right) \wedge \text{Executable}\left(\sigma'\right)\right)\right)$$

Proving that a given situation σ is globally enforceable, amounts to proving the existence of an executable situation where all the active obligations in σ are fulfilled. If such a situation does not exist, we say that the policy is *feasibility conflictual* in σ. If the set of actions and the set of fluents are finite, we can prove that the existence of such a situation is decidable and can be solved in NEXPTIME complexity. This complexity of planning in the situation calculus is high but is similar to other planners, like Strips for example [26].

5.2 Conflict between permission and obligation rules

In traditional deontic logic like Standard Deontic Logic (SDL), obligation implying permission is an an axiom of the logic. However, if the obligation is associated with a deadline, then $\text{Ob}(\alpha < d, \sigma) \wedge \neg\text{Perm}(\alpha, \sigma)$ may be satisfiable in some situation σ. As we consider the close security requirement, then every executed action must be explicitly permitted. Thus, we define the legal local requirement as follows:

$$L\text{-Enforceable}\left(\alpha < d, \sigma\right) \leftrightarrow \left[\text{Ob}\left(\alpha < d, \sigma\right) \wedge \left(\exists \sigma', \sigma < \sigma'\right) \left(\text{Fulfil}\left(\alpha < d, \sigma'\right) \wedge \text{Permitted}\left(\sigma'\right)\right)\right]$$

This means that there is a legal path that allows filling the active obligation. If in addition the path is executable, then the obligation is strongly enforceable.

$$S\text{-Enforceable}(\alpha < d, \sigma) \leftrightarrow \left[\text{Ob}(\alpha < d, \sigma) \wedge \left(\exists \sigma', \sigma < \sigma'\right) \left(\text{Fulfil}\left(\alpha < d, \sigma'\right) \wedge \text{Permitted}\left(\sigma'\right) \wedge \text{Executable}\left(\sigma'\right)\right)\right]$$

If all the active obligations in a situation σ can be executed in a permitted situation without violating at least one of them, we say that this situation is legal globally enforceable. The legally global requirement is defined as follows:

$$\text{LG-Enforceable}(\sigma) \leftrightarrow \left(\exists \sigma', \sigma < \sigma'\right) (\forall \alpha, \forall d) \left[\text{Ob}(\alpha < d, \sigma) \rightarrow \left(\text{Fulfil}\left(\alpha < d, \sigma'\right) \wedge \text{Permitted}\left(\sigma'\right)\right)\right]$$

The policy is considered legally conflictual in situation σ if it is not legally enforceable. If in addition the plan is executable, the situation is called strongly globally enforceable.

$$\text{SG-}\mathit{Enforceable}(s) \leftrightarrow \left(\exists s', s < s'\right) (\forall \alpha, \forall d) \left[\text{Ob}(\alpha < d, s) \rightarrow \left(\text{Fulfil}\left(\alpha < d, s'\right) \wedge \text{Legal}\left(s'\right) \wedge \text{Executable}\left(s'\right)\right)\right]$$

In what follows, we assume the existence of a temporal reasoning component that allows us to infer, for example, that $T_1 = T_2$ when $T_1 \leq T_1$ and $T_2 \leq T_2$, and we are able to solve linear equations and inequalities over the reals using the Simplex algorithm [27]. Algorithm 1 detects the different types of conflict we have defined using recursive search as defined in Algorithm 2. Note that in Algorithm 2, we allow the execution of parallel actions; otherwise, we can use constraints to specify the actions which cannot be done in parallel. These constraints can be compiled into precondition axioms of these actions [22]. In this work and to simplify, we omit the use of constraints. Note that we suppose that if a situation we check is globally enforceable (respectively legally globally enforceable), then this situation must be executable (respectively permitted). Proving that a situation is executable (respectively permitted) can be done using regression [23], where testing is reduced to proving first-order theorem in the initial situation.

Algorithm 1 ConflictDetection(s, N, conflictType)

Require: s: the situation to check
N: the maximal depth
conflictType: the type of the searched conflict,
"FC" for feasibility conflict,
"LC" for legally conflict and
"SC" for strong conflict

Ensure: No: if there is no conflict of type conflictType in the policy at situation *s*; otherwise, Yes.

$\mathcal{O} = \{\alpha \in \mathcal{A}$ **such that** $\text{Ob}(\alpha < d, s)\}$ {set of active obligations in *s*}
$s' \leftarrow$ recursiveSearch(s, N, $\mathcal{O}$, conflictType)
if $\neg(s' = \text{NULL})$ **then**
  return No {there is no conflict of type conflictType in the policy at *s* and s' is the plan which leads to fulfill all the active obligations in *s*}
else
  return Yes {there is a conflict of type conflictType in the policy at situation *s*}
end if

Algorithm 2 recursiveSearch(s, N, $\mathcal{O}$, conflictType)

Require: s: the current situation
N: the current depth (initially the given maximum depth)
$\mathcal{O}$: set of active obligations in s
conflictType: the type of the searched conflict,
"FC" for feasibility conflict,
"LC" for legally conflict and
"SC" for strong conflict

Ensure: Null: if the depth of the current path exceeds the given maximum depth or,
situation when all obligations in $\mathcal{O}$ are fulfilled if it exists otherwise,
the next situation to give to the next call for recursion

```
switch (conflictType)
case "FC":
  ℰ ← {a ∈ 𝒜, Poss(a,s) ∧ Start(s) ≤ Time(a)} {the set of actions that can lead from s to an eventual executable situation}
case "LC":
  ℰ ← {a ∈ 𝒜, Perm(a,s)} {the set of actions that can lead from s to an eventual legal situation}
case "SC":
  ℰ ← {a ∈ 𝒜, Poss(a,s) ∧ Start(s) ≤ Time(a) ∧ Perm(a,s)} {the set of actions that can lead from s to an eventual legal and executable situation}
end switch
while true do
  if N < 0 then
    return NULL
  end if
  for all a ∈ ℰ do
    s′ ← do(a,s)
    N ← N − 1
    if (∀α, d ∈ 𝒪)Fulfil(α < d, s′) then
      return s′
    end if
    s″ ← recursiveSearch(s′, N, 𝒪, conflictType)
    if ¬(s″ = NULL) ∧ (∀α, d ∈ 𝒪)Fulfil(α < d, s″))
    then
      return s″
    end if
    N ← N + 1
  end for
  return NULL
end while
```

6 Formal specification of the case study's security policy

To illustrate our approach, let us consider a security policy containing the following rules:

- *Rule 1*: The doctor *must* complete the admission note of the patient assigned to him within 30 units of time following his admission to the hospital.
- *Rule 2*: The doctor *must* complete the medical observation of the patient assigned to him within 30 units of time following his admission to the hospital.
- *Rule 3*: End deadline for completing the admission note of a patient must occur after 30 units of time of his admission to the hospital.
- *Rule 4*: End deadline for completing the medical observation of a patient must occur after 40 units of time of his admission to the hospital.
- *Rule 5*: The doctor is permitted to start writing observation or admission note of an inpatient assigned to him when he is not writing another document.
- *Rule 6*: The doctor is permitted to complete observation or admission note at least 5 units of time after he began to write it.

Normally, we should specify a permission rule for each action in $\mathcal{A}$. But for simplicity, we quote just one permission rule regarding the action of writing documents.

To give the specification of the example policy, we should determine the set of fluents $\mathcal{F}$, the set of the actions $\mathcal{A}$, the succession state axioms of all fluents, and the preconditions axioms of all actions.

- Set $\mathcal{F}$ of fluents:
  - assigned(p, d, s). The patient p is assigned to a doctor d in situation s.
  - inpatient(p, t, s). The patient p is admitted to the hospital at time t in the situation s.
  - leaving(p, s). The patient p leaves the hospital in s.
  - deadline$(type, p, t, s)$. The deadline to write document of type *type* concerning patient p created at time t is elapsed in s.
  - writingDoc$(d, \text{type}, p, t, t', s)$. d is writing the document of type *type* concerning patient p created at time t and began to be written at t' in s.
  - writtenDoc$(d, \text{type}, p, t, s)$. The document of type *type* concerning patient p and created at time t has been written in s by d.
  - doctor(d). d is a doctor.
- Set $\mathcal{A}$ of actions:
  - assign(p, d, t). The action to assign at time t the patient p to the doctor d.
  - revokeAssignment(p, d, t). The action to revoke at time t assignment of the patient p to the doctor d.

- patientAdmission(p, t). The action to admit at time t the patient p at the hospital.
- leave(p, t). The patient p leaves the hospital at time t.
- EndDeadline$(type, p, t, t')$: The action to warn at time t' that the accorded deadline for writing document of patient p expires.
- StartWrite(d, type, p, t), EndWrite(d, type, p, t): d starts (respectively ends) to write document of type corresponding to patient p at time t; type is one of the following elements: *Observation* or *AdmssionNote*.

- Fluent succession state axioms: Patient p will be assigned to doctor d when the action of assignment is executed, since then p remains assigned to d unless there is a revocation of assignment or the patient leaves the hospital.

$$\begin{aligned}&\text{Poss}(a, s) \rightarrow \\ &\text{assigned}(p, d, \text{do}(a, s)) \leftrightarrow \left[(\exists t) a = \text{assign}(p, d, t) \vee\right. \\ &(\text{assigned}(p, d, s) \wedge \neg(\exists t) a = \text{revokeAssignment}(p, d, t) \wedge \\ &\left.\neg(\exists t) a = \text{leave}(p, t))\right]\end{aligned} \tag{6}$$

Patient p is hospitalized if he was admitted to the hospital and did not leave.

$$\begin{aligned}&\text{Poss}(a, s) \rightarrow \\ &\text{inpatient}(p, t, \text{do}(a, s)) \leftrightarrow \left[a = \text{patientAdmission}(p, t) \vee\right. \\ &\left.\left(\text{inpatient}(p, t, s) \wedge \neg \left(\exists t'\right) a = \text{leave}(p, t')\right)\right]\end{aligned} \tag{7}$$

Action *EndDeadline* is executed to denote that the delay accorded to write documents is elapsed. When the deadline is considered expired, it remains expired forever.

$$\begin{aligned}&\text{Poss}(a, s) \rightarrow \\ &\text{deadline}(type, p, t, \text{do}(a, s)) \leftrightarrow \\ &\left[\left(\exists t'\right) a = \text{endDeadline}\left(\text{type}, p, t, t'\right) \vee \text{deadline}(\text{type}, p, t, s)\right]\end{aligned} \tag{8}$$

A document is in a writing process if its writing began before and is not completed yet.

$$\begin{aligned}&\text{Poss}(a, \sigma) \rightarrow \\ &\text{writingDoc}\left(d, \text{type}, p, t, t', \text{do}(a, s)\right) \leftrightarrow \\ &\left[a = \text{startWrite}(d, \text{type}, p, t') \vee \left(\text{writingDoc}(d, \text{type}, p, t, t', s) \wedge\right.\right. \\ &\left.\left.\neg \left(\exists t''\right) a = \text{endWrite}\left(d, \text{type}, p, t''\right)\right)\right]\end{aligned} \tag{9}$$

A document is considered written if the writing process is completed.

$$\begin{aligned}&\text{Poss}(a, \sigma) \rightarrow \\ &\text{writtenDoc}(d, \text{type}, p, t, \text{do}(a, s)) \leftrightarrow \\ &\left[\left(\exists t'\right) a = \text{endWrite}\left(d, \text{type}, p, t'\right) \vee \text{writtenDoc}\left(d, \text{type}, p, t, s\right)\right]\end{aligned} \tag{10}$$

- Action precondition axioms: In what follows, all action precondition axioms are written using equivalent conditions except *startWrite* action. This is because we consider that all these actions are always permitted (even if we did not mention the permission rules associated with them for simplicity). In contrast, precondition axiom of actions *startWrite* and *endWrite* will be rewritten later to take into account the permission rule associated with them. It is assumed that a patient can be assigned at any time to a doctor except if the patient is already assigned to him.

$$\text{Poss}\left(\text{assign}(p, d, t), \sigma\right) \leftrightarrow \left(\neg\text{assigned}(p, d, \sigma) \wedge \text{start}(\sigma) \leq t\right)$$

The assignment revocation of patients is necessarily applied to a patient who is already assigned to a doctor.

$$\text{Poss}\left(\text{revokeAssignment}(p, d, t), \sigma\right) \leftrightarrow \left(\text{assigned}(p, d, \sigma) \wedge \text{start}(\sigma) \leq t\right)$$

It is assumed that a patient assigned to a doctor can be admitted at any time to a hospital except if he is already hospitalized.

$$\begin{aligned}&\text{Poss}(\text{patientAdmission}(p, t), \sigma) \leftrightarrow (\text{assigned}(p, d, \sigma) \wedge \\ &\neg \left(\exists t'\right) \text{inpatient}\left(p, t', \sigma\right) \wedge \text{start}(\sigma) \leq t)\end{aligned}$$

Leaving the hospital concerns patients which are hospitalized.

$$\text{Poss}\left(\text{leave}(p, t), \sigma\right) \leftrightarrow \left(\left(\exists t'\right) \text{inpatient}\left(p, t', \sigma\right) \wedge \text{start}(\sigma) \leq t\right)$$

Action *EndDeadline*(type, p, t, t_{ed}) is executed when the accorded deadline to write document of type *type* is elapsed since the moment t of a patient hospitalization; deadline is 30 units of time when *type* is admission note and 40 units of time when it is medical observation. It is assumed in addition that it is not possible to execute the end of deadline for the same document more than once.

$$\begin{aligned}&\text{Poss}\left(\text{endDeadline}(\text{type}, p, t, t_{\text{ed}}), \sigma\right) \leftrightarrow \\ &\left[\text{inpatient}\left(p, t', \sigma\right) \wedge \left(\left(t_{\text{ed}} = t' + 40 \wedge \text{type} = \text{observation}\right) \vee\right.\right. \\ &\left.\left.\left(t_{\text{ed}} = t' + 30 \wedge \text{type} = \text{admissionNote}\right)\right) \wedge \neg\text{deadline}(\text{type}, p, t, \sigma)\right]\end{aligned}$$

It is assumed that the action to start writing a document is necessarily executed on a document that is not in a writing process.

$$\begin{aligned}&\text{Poss}\left(\text{startWrite}\left(d, \text{type}, p, t, t_{\text{sw}}\right), \sigma\right) \rightarrow \\ &\neg\text{writingDoc}\left(d, \text{type}, p, t, t', \sigma\right) \wedge \neg\text{writtenDoc}\left(d, \text{type}, p, t, \sigma\right) \wedge \\ &\text{start}(\sigma) \leq t_{\text{sw}}\end{aligned}$$

The writing end is applied to an ongoing writing document.

$$\begin{aligned}&\text{Poss}\left(\text{endWrite}\left(d, \text{type}, p, t, t_{\text{ew}}\right), \sigma\right) \leftrightarrow \\ &\left[\left(\exists t'\right) \text{writingDoc}\left(d', \text{type}, p, t, t', \sigma\right) \wedge \text{start}(\sigma) \leq t_{\text{ew}}\right]\end{aligned}$$

We can now give the specification of the aforementioned security rules.

$$\begin{aligned}
&\textit{Rule 1}: O\left(\text{write}\left(d,\text{type},p,t,t'\right) < \text{deadline}(\text{type},p,t) \middle| \text{doctor}(d)\wedge \text{assigned}(p,d)\wedge \text{inpatient}(p,t)\wedge \text{type}=\text{admissionNote}\right)\\
&\textit{Rule 2}: O\left(\text{write}\left(d,\text{type},p,t,t'\right) < \text{deadline}(\text{type},p,t) \middle| \text{doctor}(d)\wedge \text{assigned}(p,d)\wedge \text{inpatient}(p,t)\wedge \text{type}=\text{observation}\right)\\
&\textit{Rule 3}: O\left(\text{endDeadline}\left(\text{admissionNotes},p,t,t'\right) \middle| \text{inpatient}(p,t)\wedge t'=t+30\right)\\
&\textit{Rule 4}: O\left(\text{endDeadline}\left(\text{observation},p,t,t'\right) \middle| \text{inpatient}(p,t)\wedge t'=t+40\right)\\
&\textit{Rule 5}: P\left(\text{startWrite}\left(d,\text{type},p,t,t_{\text{sw}}\right) \mid \text{doctor}(d)\wedge\text{assigned}(p,d)\wedge \text{inpatient}(p,t,s)\wedge\text{type}=(\text{observation}\vee\text{admissionNote})\wedge \neg\left(\exists \text{type}',p',t',t''\right)\text{writingDoc}\left(d,\text{type}',p',t',t''\right)\right)\\
&\textit{Rule 6}: P\left(\text{endtWrite}\left(d,\text{type},p,t,t_{\text{ew}}\right) \mid \text{writingDoc}(d,\text{type},p,t,t')\wedge t_{\text{ew}}\geq t'+5\right)
\end{aligned}$$

According to the specification of obligation rules, we can see that we have one set of conditional obligations with deadlines: $O_{\text{write}(d,\text{type},p,t,t_{\text{w}}),\text{deadline}(\text{type},p,t,s)}$. The corresponding formula $\psi_{O_{\text{write}(d,\text{type},p,t,t_{\text{w}}),\text{deadline}(\text{type},p,t,\sigma)}}$, after simplification, is as follows:

$$\psi_{O_{\text{write}(d,\text{type},p,t,t_{\text{w}}),\text{deadline}(\text{type},p,t,\sigma)}} \leftrightarrow \text{doctor}(d)\wedge\text{assigned}(p,d,\sigma)\wedge\text{inpatient}(p,t,\sigma)\wedge \text{type}=(\text{observation}\vee\text{admissionNote}))$$

According to axiom (2), to derive concrete obligations, we should calculate the following formulas:

$$\begin{aligned}
&\gamma^{+}_{\psi_{O_{\text{write}(d,\text{type},p,t,t_{\text{w}}),\text{deadline}(\text{type},p,t,\sigma)}}}(a,\sigma)\leftrightarrow \text{assigned}(p,d,\sigma)\wedge a=\text{patientAdmission}(p,t)\\
&\gamma^{-}_{\psi_{O_{\text{write}(d,\text{type},p,t,t_{\text{w}}),\text{deadline}(\text{type},p,t,\sigma)}}}(a,\sigma)\leftrightarrow a=\text{revokeAssignment}(p,d,t')\vee a=\text{leave}(p,t')
\end{aligned}$$

The above formula is calculated using the succession state axiom of fluent *assigned* (6). Finally, the formula $\gamma^{+}_{\text{deadline}(\text{type},p,t)}(a,\sigma)$ is calculated using succession state axiom of fluent *deadline* (8).

$$\gamma^{+}_{\text{deadline}(\text{type},p,t)}(a,\sigma)\leftrightarrow a=\text{endDeadline}(\text{type},p,t,t')$$

Thus, the concrete obligations concerning rules 1 and 2 are as follows:

$$\begin{aligned}
&\text{Poss}(a,\sigma)\rightarrow\\
&\text{Ob}(\text{write}(d,\text{type},p,t,t_{\text{w}}),\text{do}(a,\sigma))\leftrightarrow\\
&\big[(\text{assigned}(p,d,\sigma)\wedge a=\text{patientAdmission}(p,t))\vee\\
&\big(\text{Ob}(\text{write}(d,\text{type},p,t),\sigma)\wedge\\
&\neg(\exists t')a=\text{endWrite}(d,\text{type},p,t,t')\wedge\\
&\neg(\exists t')a=\text{endDeadline}(\text{type},p,t,t')\wedge\\
&\neg(\exists t')a=\text{revokeAssignment}(p,d,t')\wedge\neg(\exists t')a=\text{leave}(p,t')\big)\big]
\end{aligned}$$

Similarly, we can derive the situations where the system obligations are active.

$$\begin{aligned}
&\text{Poss}(a,\sigma)\rightarrow\\
&\text{Ob}\left(\text{endDeadline}(\text{type},p,t,t',\text{do}(a,\sigma)\right)\leftrightarrow\\
&\big[(a=\text{patientAdmission}(p,t)\wedge\\
&\left(\left(\text{type}=\text{admissionNote}\wedge t'=t+30\right)\vee\text{type}=\text{observation}\wedge\right.\\
&\left.t'=t+40\right))\vee\\
&\left(\text{Ob}(\text{endDeadline}(\text{type},p,t,t'),\sigma\right)\wedge\\
&\neg\left(\exists t'\right)a=\text{endDeadline}\left(\text{type},p,t,t'\right)\wedge\neg\left(\exists t'\right)a=\text{leave}\left(p,t'\right))\big]
\end{aligned}$$

Using rule 5 and succession state axiom of fluent *assigned* (6), the actions and the conditions under which *startWrite* will be permitted are given by the following formula:

$$\begin{aligned}
&\gamma^{+}_{\psi_{P_{\text{startWrite}(d,\text{type},p,t,t_{\text{sw}})}}}(a,\sigma)\leftrightarrow\\
&\big[\text{assigned}(p,d,\sigma)\wedge\text{type}=(\text{observation}\vee\text{admissionNote})\wedge\\
&\neg\left(\exists\text{type}',p',t',t'_{\text{sw}}\right)\text{writingDoc}\left(d,\text{type}',p',t',t'_{\text{sw}},\sigma\right)\wedge\\
&a=\text{patientAdmission}(p,t)\big]\vee\\
&\big[\text{inpatient}(p,t,\sigma)\wedge(\exists\text{type}',p',t',t'_{\text{sw}})\text{writingDoc}\left(d,\text{type}',p',t',t'_{\text{sw}},\sigma\right)\wedge\\
&a=\text{endWrite}\left(d,\text{type}',p',t',t_{\text{ew}}\right)\big]
\end{aligned}$$

On the other side, the actions and the conditions under which *startWrite* will be no longer permitted are given by the following formula:

$$\begin{aligned}
&\gamma^{-}_{\psi_{P_{\text{startWrite}(d,type,p,t,t_{\text{sw}})}}}(a,\sigma)\leftrightarrow\\
&\left(\exists t'\right)a=\text{revokeAssignment}\left(p,d,t'\right)\vee\left(\exists t'\right)a=\text{leave}\left(p,t'\right)\vee\\
&\left(\exists\text{type}',p',t',t''\right)a=\text{startWrite}\left(d,\text{type}',p',t',t''\right)
\end{aligned}$$

Then, the active permission for startWrite is calculated using axiom (1).

$$\begin{aligned}
&\text{Poss}(a,\sigma)\rightarrow\\
&\text{Perm}\left(\text{startWrite}\left(d,\text{type},p,t,t_{\text{sw}}\right),\text{do}(a,\sigma)\right)\leftrightarrow\\
&\big[\text{assigned}(p,d,\sigma)\wedge\text{type}=(\text{observation}\vee \textit{admissionNote})\wedge &&(11)\\
&\big[(\neg(\exists\text{type}',p',t',t'_{\text{sw}})\text{writingDoc}(d,\text{type}',p',t',t'_{\text{sw}},\sigma)\wedge &&(12)\\
&a=\text{patientAdmission}(p,t))\vee &&(13)\\
&(\text{inpatient}(p,t,\sigma)\wedge &&(14)\\
&\left(\exists\text{type}',p',t',t'_{\text{sw}}\right)\text{writingDoc}\left(d,\text{type}',p',t',t'_{\text{sw}},\sigma\right)\wedge &&(15)\\
&a=\text{endWrite}(d,\text{type}',p',t',t_{\text{ew}}))\big]\vee &&(16)\\
&\left(\text{Perm}\left(\text{startWrite}\left(d,\text{type},p,t,t_{\text{sw}}\right),\sigma\right)\wedge\right.\\
&\neg\left(\exists t'\right)a=\text{revokeAssignment}\left(p,d,t'\right)\wedge\\
&\neg\left(\exists t'\right)a=\text{leave}\left(p,t'\right)\wedge\\
&\left.\neg\left(\exists\text{type}',p',t',t'_{\text{sw}}\right)a=\text{startWrite}\left(d,\text{type}',p',t',t'_{\text{sw}}\right)\right)
\end{aligned}$$

The lines 11, 12, and 13 of the axiom above express the fact that a doctor which is not writing a document is permitted to write the observation and the admission note of

a patient assigned to him as soon as this patient is admitted in the hospital. The lines 11, 14, 15, and 16 express the fact that when a patient is hospitalized and his doctor is writing a document, the doctor will be permitted to write the observation and the admission note of this patient after he completes the writing document. Finally, the precondition axiom for starting writing documents is as follows:

$$\mathrm{Poss}(\mathrm{startWrite}(d,\mathrm{type},p,t,t_{\mathrm{sw}}),\sigma) \leftrightarrow$$
$$\mathrm{Perm}(\mathrm{startWrite}(d,\mathrm{type},p,t,t_{\mathrm{sw}}),\sigma) \wedge$$
$$\neg\mathrm{writingDoc}(d,\mathrm{type},t,\sigma) \wedge \neg\mathrm{writtenDoc}(d,\mathrm{type},p,t,\sigma)$$

Using rule 6 and the succession state axiom (10) of *writingDoc*, we can calculate the situation when the end for writing document is permitted.

$$\mathrm{Poss}(a,\sigma) \rightarrow$$
$$\mathrm{Perm}(\mathrm{endWrite}(d,\mathrm{type},p,t,t_e),\mathrm{do}(a,\sigma)) \leftrightarrow$$
$$a = \mathrm{startWrite}(d,\mathrm{type},p,t,t_s) \wedge t_e \geq t_s + 5 \vee$$
$$\mathrm{Perm}(\mathrm{endWrite}(d,\mathrm{type},p,t,t_e),\sigma) \wedge$$
$$\neg a = \mathrm{endWrite}(d,\mathrm{type},p,t,t_e')$$

7 Implementation

We implement our model using the logic programming language Golog [8,13], based on the situation calculus. Regarding our need to solve linear equations and inequalities, we use the Common Logic Programming System ECLIPSE 3.5.2, which provides a built-in Simplex algorithm for solving linear equations and inequalities over the reals.

The point of departure for the implementation is to get the list of all active obligations in a given situation S. This is given using the predicate *activeObligations(ActiveObligationsList,S)*:

```
activeObligations
(ActiveObligationsList,S):-findall(Rule,ob(Rule,S),
                             ActiveObligationsList).
```

Given the list of active obligations, seeking the situation where all these obligations are fulfilled is made using the procedure *plan*.

```
proc(plan(N,L),
       ?(all(r,member(r,L) =>
       fulfil(r))):?(reportStats)#
       ?(N > 0):
       pi(a,?(primitiveAction(a)):a):
       ?(-badSituation):
       pi(n,?(n is N-1):plan(n,L))).
```

Here, *primitiveAction* is a predicate characterizing all the actions of the domain. If $N = 0$, the execution of the procedure ends. If $N > 0$, a primitive action a is selected. The Golog interpreter checks if the selected action a is possible and verifies that $\mathrm{start}(s) \leq \mathrm{time}(a)$, where s is the current situation. If so, a is executed and $\mathrm{do}(a,s)$ becomes the new current executable situation.

We can change the following instruction of Golog:

```
do(E,S,do(E, S)) :- primitiveAction(E), poss(E,S),
                    start(S,T1),time(E, T2),T1 <= T2.
```

and replace it with the following statement for searching a legal plan:

```
do(E,S,do(E, S)) :- primitiveAction(E),perm(E, S),
                   start(S,T1),time(E,T2),T1 <= T2.
```

In our implementation, we make no change in the Golog interpreter, but every precondition axiom of an action includes the fact that this action is permitted using the fluent *perm*. Thus, we test whether a situation is strongly enforceable or not using the predicate *sEnforceable(N,S1)*.

```
sEnforceable(N,S1):- initializeCPU,
                     activeObligations
                     (ActiveObligationsList,S1),
                     do(plan(N,
                     ActiveObligationsList),S1,S),
                     prettyPrintSituation(S).
```

Let us start by seeing how we can write some axioms of our example using Golog. The complete description of axioms is described in Additional file 1.

7.1 Examples of succession state axioms

```
ob(write(D,Type,P,T),
   do(A,S)):- (assigned(P,D,S),
              A=patientAdmission(P,T));
              (ob(write(D,observation,P,T),S),
              not A=endWrite(D,observation,P,T,T1),
              not A=endDeadline(observation,P,T,T2),
              not A=leave(P,T3),
              not A=revokeAssignation(P,D,T4)).

fulfil(write(D,Type,P,T),
      do(A,S)):- (ob(write(D,Type,P,T),S),
                 (A=endWrite(D,Type,P,T,T2));
                 fulfil(write(D,Type,P,T),S)).
```

7.2 Examples of action precondition axioms

```
poss(startWrite(D,observation,
    P,T,T1),S):- inpatient(P,T,S),
                 assigned(P,D,S),
                 not writingDoc(D,Type1,P1,T3,T4,S),
                 not writtenDoc(D,observation,P,T,S).

poss(endWrite(D,Type,P,T,T1),S):- writingDoc
                                  (D,Type,P,T,T2,S),
                                  T1 $>= T2+5.
```

In addition to the succession state axioms and precondition axioms of actions, we suppose having the following axioms in the initial situation s_0.

```
start(s0, 0).
doctor(jean).
```

7.3 Description of bad situations

The badSituation test is used to remove partial plans which are known in advance to be unsuccessful. For example, a branch resulting from the execution of an action that disables an active obligation can be eliminated. A branch resulting from the execution of an action that activates the deadline corresponding to an active obligation may also be removed. We will see in the following how this can be done in the implementation of our example.

If a violation of an active obligation occurs after the execution of an action, it is no longer necessary to continue searching a solution from the resulting situation.

```
badSituation(do(A,S)):- A=endDeadline(Type,P,T,T1),
                        not fulfil(write
                        (D,Type,P,T),S),!.
badSituation(do(A,S)):- A=endDeadline(Type,P,T,T1),
                        poss(endDeadline(Type1,
                        P1,T2,T3),S),T3 $< T1,!.
badSituation(S):- ob(endDeadline(Type,P,T,T1),S),
                  start(S,T2), not (T1$>=T2),!.
```

If an active obligation is deactivated after the execution of an action, it is no longer necessary to continue searching a solution from the resulting situation.

```
badSituation(do(A,S)):- A=leave(P,T),!.
badSituation(do(A,S)):- A=revokeAssignation(P,D,T),!.
```

The construction of these bad situations can be done using the succession state axioms of fluent *Ob* and *Fulfil*. Thus, these optimizations can be generalized to any policy without losing the completeness of the planning search. In our example, the execution of actions *patientAdmission* and *assign* activates other obligations and have no impact on the fulfillment of obligations which are already active. Therefore, the path resulting from their execution can be eliminated in the solution search.

```
badSituation(do(A,S)):- A=assign(P,D,T),!.
badSituation(do(A,S)):- A=patientAdmission(P,T),!.
```

This optimization is closely related to our example because there is nothing that prevents to have actions in the policy that are necessary to fulfill obligations, but their execution leads to activate other obligations simultaneously.

We have performed tests that check the strongly enforceability of situations constructed as follows: the first situation checked by the first test, denoted $s1$ is the result of the assignment of a single patient $p1$ to the doctor jean at time 4 followed by his admission in the hospital at time 5.

```
test1:- sEnforceable(6,do(patientAdmission(p1,5),
        do(assign(p1,jean,4),s0))).
```

The next situation $s2$ is the assignment of another patient $p2$ to jean at time 6 from the situation $s1$ followed by the admission of $p2$ in the hospital at time 7.

```
test2:- sEnforceable(12,do(patientAdmission(p2,7),
        do(assign(p2,jean,6),
        do(patientAdmission(p1,5),
        do(assign(p1,jean,4),s0))))).
```

We build 20 tests. Their complete description is in Additional file 2. The planning depth research is calculated as follows. In our application domain, there are seven actions, four of them are removed from the planning through the specification of *badSituation*. The remainder actions are *startWrite*, *endWrite*, and *endDeadline*. In the database, there is one doctor Jean and two types of documents, so for each patient, these actions are possible twice, one for each type of document. When one of these actions is executed, it is not possible to execute it again according to their precondition axioms. Thus, when all these actions are performed one after the other, it is no longer possible to perform other actions except those which are discarded from the planning search. Thereby, whenever a patient is added, the minimum depth ensuring the decidability of solution research is increased by six.

We conducted two series of tests depending on the deadlines associated with the obligations to write documents. The experiment was run on a machine equipped with an Intel 32 bit, 2.60 GHz, x4 processor, and 3.8 GB RAM, running ECLIPSE 3.5.2 on ubuntu Linux (v.13.04).

7.4 The first series of tests

The deadline for writing admission note is 30 units of time, and the observation is 40 units of time (see Additional file 3). In this series of tests, the maximum number of patients, who can be admitted in the hospital and assigned to *jean*, without causing conflict between the obligations is 4. Indeed, the policy is not conflictual in the first four situations. As example, the following legal plan generates a situation when all the active obligations in the situation $s1$ are fulfilled which means that $s1$ is strongly globally enforceable.

```
[eclipse 2]: test1.

  CPU time (sec): 0.00

[assign(p1,jean,4),patientAdmission(p1,5),
startWrite(jean, admissionNote,p1,5,_{3}74),
endWrite(jean,admissionNote,p1,5,_{1}095),
startWrite(jean,observation,p1,5,_{2}264),
endWrite(jean,observation,p1,5,_{3}112),
endDeadline(admissionNote,p1,5,35),
endDeadline(observation,p1,5,45)]
more? n.

Linear Store:

_{3}112 $>= 15+1*_{2}321+1*_{1}187+1*_{4}31
_{2}264 $>= 10+1*_{1}187+1*_{4}31
_{1}095 $>= 10+1*_{4}31
_{3}74 $>= 5
```

The above plan contains uninstantiated temporal variables. The value of these variables is just constrained by the inequalities in ECLIPSE's linear constraint store, although there may be cases when plans are fully specified like the following third test which proves that the policy remains consistent after the admission of three patients.

```
[eclipse 4]: test3.

 CPU time (sec): 0.03

[assign(p1,jean,4),patientAdmission(p1,5),
assign(p2,jean,6),patientAdmission(p2,7),
assign(p3,jean,8),patientAdmission(p3,9),
startWrite(jean,admissionNote,p3,9,9),
endWrite(jean,admissionNote,p3,9,14),
startWrite(jean,admissionNote,p2,7,14),
endWrite(jean,admissionNote,p2,7,19),
startWrite(jean,admissionNote,p1,5,19),
endWrite(jean,admissionNote,p1,5,24),
startWrite(jean,observation,p3,9,24),
endWrite(jean,observation,p3,9,29),
startWrite(jean,observation,p2,7,29),
endWrite(jean,observation,p2,7,34),
startWrite(jean,observation,p1,5,34),
endDeadline(admissionNote,p1,5,35),
endDeadline(admissionNote,p2,7,37),
endWrite(jean,observation,p1,5,39),
endDeadline(admissionNote,p3,9,39),
endDeadline(observation,p1,5,45),
endDeadline(observation,p2,7,47),
endDeadline(observation,p3,9,49)]
more? n.
```

Finally, the fifth test shows how the admission of a fifth patient produces a conflict.

```
[eclipse 6]: test5.

No (1460.33s cpu)
```

7.5 The second series of tests

The deadline for writing admission note is 1,000 units of time, and the observation is 1,100 units of time (see Additional file 4).

In this series of tests, we check 20 situations. Table 1 summarizes the results obtained and the execution time for each tested situation. Figure 1 shows how the execution time for finding a plan is increasing with the number of active obligations in these situations. On average, there are $2 \times nb$ actions, which can be executed from a given situation, where nb is a number of admitted patients. Moreover, the plan to achieve the desired goal is made up of $nb \times 6$ actions. Then, the number of worlds to explore is the order of $(2 \times nb)^{nb \times 6}$. This explains the increment in the time duration, each time a patient is admitted.

8 Related work

Most traditional security models are static and respond to access requests just by yes (accept) or no (deny). Recently, there are more and more works on security models that model obligations [3,28-31]. Formalization of obligations differs from one model to another. In XACML [32], obligations are all operations that must be met in conjunction with the application of the authorization decision. In [3,28,33], distinction is made between provisions and obligations. Provisions are actions or conditions that must be met before authorizing access. Obligations are actions that must be executed by users or system after the access is given. The ABC model (authorization, obligation, and condition) [34] was specifically designed to express security policies including usage control constraints. The expression of a constraint to be satisfied before the use of an object can be expressed as contextual authorization. However, constraints to meet during or after the use of an object relate to obligations that the user must follow. The NOMAD model [35] is based on a formalization in temporal and deontic logic to express contextual obligations which should be met before, during, or after the execution of an action. It is also possible to specify a deadline after which some obligation will be considered violated if the action was not performed. Authors in [36], define a core language to specify the access and usage control requirements and then give a formalism based on the logic of temporary actions (TLA) [37] to specify the behavior of the policy controller in charge of evaluating such policy. In this approach, a permission is associated with two conditions, the first must be true at the time of query evaluation,

Table 1 The summary of second series of tests

Tests	Patient number	Active obligation number	The minimum research depth	CPU time (s)	Strongly globally enforceable?
*s*1	1	4	6	0	Yes
*s*2	2	8	12	0.01	Yes
*s*3	3	12	18	0.03	Yes
*s*4	4	16	24	0.06	Yes
*s*5	5	20	30	0.1	Yes
*s*6	6	24	36	0.2	Yes
*s*7	7	28	42	0.36	Yes
*s*8	8	32	48	0.66	Yes
*s*9	9	36	54	1.1	Yes
*s*10	10	40	60	1.78	Yes
*s*11	11	44	66	2.78	Yes
*s*12	12	48	72	4.16	Yes
*s*13	13	52	78	6.03	Yes
*s*14	14	56	84	6.04	Yes
*s*15	15	60	90	8.60	Yes
*s*16	16	64	96	11.95	Yes
*s*17	17	68	102	16.43	Yes
*s*18	18	72	108	21.75	Yes
*s*19	19	76	114	28.89	Yes
*s*20	20	80	120	37.97	Yes

This table describes the most important parameters influencing the time of executions: the number of active obligations and the depth of the solution search.

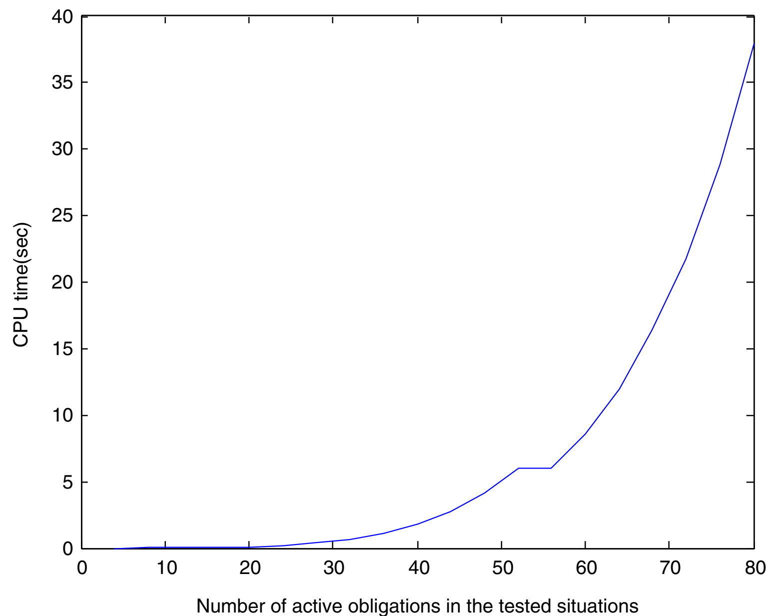

Figure 1 CPU utilization vs. Number of active obligations in the tested situations. This figure shows the cpu time elapsed before giving a first solution where all active obligations in the situations tested are fulfilled.

and the second must always be true as long as access is in progress. The authors also introduce a concept to reset a current access. Regarding obligations, they are associated with two conditions. Once the first condition is satisfied, the obligation is triggered, then the controller sends a notification to the user to perform the appropriate obligation, the second condition determines when the obligation should be considered violated. If the user does not satisfy the obligation before the second condition becomes true, a penalty is applied to him. The authors in [38] talk about what they called deontic conflicts. The types of conflict that the authors have classified in this category are those that occur between permission and prohibition and those which occurs between obligation and obligation waiver. As in the used formalism, the authors do not use prohibition and obligation waiver modalities, they do not deal with these conflicts in their work. But in this category, there is another kind of conflict which is the conflict between the obligations with deadlines and permissions. In our work, this conflict is detected when there is no plan consisting of permitted actions that lead to fulfilling an obligation requirement in its deadline. In other words, it is possible that in a given situation, a mandatory action is permitted and it can be fulfilled in its deadline, but it is not possible to execute because it is necessary to first execute other actions which are not permitted. Certainly, the authors define another type of conflict called temporal conflicts which occur when two deontic assignments at the same time initiate and terminate obligation. This is a particular case of what we detect in what we call the global conflict between the obligations with deadlines. Indeed, in a given situation, it may be possible to fulfill an active obligation in its deadline but given that there are other active obligations, at the same time it is not possible to fulfill them together without violating one of them. The conflict in the temporal constraints is actually a special case of a 'logical' conflict which we detect with the concept of executable plan.

9 Conclusions

In this paper, we use deontic modalities to specify close security policies including obligations with deadline. Then, we use the temporal sequential situations calculus to derive concrete permissions and obligations. Furthermore, we show how the situation calculus allows us to detect if there is a policy conflict in a given situation using the planning task. Moreover, we have illustrated our approach by using a case study from the health care community. Specifically, we are interested in obligations with deadlines concerning completion of the patients' medical records. We show how we can use our language to express the obligations of this example. In addition, we present the implementation that we did, using the logic programming language based on Golog.

On the other hand, obligations are generally associated with penalties when their violation occurs. The regulation of a hospital may specify the penalties triggered when medical records are not completed on due time. For example, if medical folders are not completed on time, the medical records department can establish for the president of the Executive Committee of the Medical Council of doctors the list of doctors and the number of folders that remain incomplete for each of them. While receiving this list, the director of professional services can inform by warning all doctors of the list that their privileges are automatically suspended until they complete their late folders. Our ongoing work along these lines consists, when we have a conflict, to specify whether the subject is accountable for this conflict. Furthermore, when we have several subjects, it is important to specify whether the violation of an obligation is not due to a violation of another subject obligation that has indirectly delayed fulfillment of the first obligation.

Our future work also includes conflict resolution. A conflicting situation means that there are subjects who cannot accomplish their active obligations one after another without violation. In such a situation, it will be interesting to know if there are active obligations of another subject in the same situation that could solve this conflict and then derive if this subject has enough free time to perform some active obligations of the first subject. This is one of the possible solutions to solve the conflict. We can also detect the rules responsible for the conflict and then update the policy with new deadlines for these rules.

Endnote

[a] The difficulty in logic of expressing the dynamics of a situation without explicitly specifying everything that is not affected by the actions.

Additional files

Additional file 1: conflictualSituation.pl. This is a prolog file which contains the description of the implementation of the studied example.

Additional file 2: test.pl. This is a prolog file which contains the description of the 20 performed tests. Each test represent a situation. Each situation is the result of an admission of another patient relative to the previous situation.

Additional file 3: deadlines_30_40.pl. This is a prolog file which initializes the deadline values for the first series of tests.

Additional file 4: deadlines_1000_1100.pl. This is a prolog file which initializes the deadline values for the second series of tests.

Competing interests
The authors declare that they have no competing interests.

Acknowledgements
This work is supported by the ITEA2 Predykot project (Grant agreement no.10035).

Received: 21 March 2014 Accepted: 8 August 2014
Published online: 10 September 2014

References

1. F Cuppens, N Cuppens-Boulahia, T Sans, Nomad: a security model with non atomic actions and deadlines, in *CSFW*, (2005), pp. 186–196
2. Y Elrakaiby, F Cuppens, N Cuppens-Boulahia, Formal enforcement and management of obligation policies. Data Knowl. Eng. **71**(1), 127–147 (2012)
3. M Hilty, A Pretschner, D Basin, C Schaefer, T Walter, A policy language for distributed usage control, in *Proceedings of the 12th European Conference on Research in Computer Security (ESORICS'07)* (Springer-Verlag Berlin, 2007), pp. 531–546
4. JD Moffett, MS Sloman, Policy conflict analysis in distributed system management. J. Organ. Comput. **4**(1), 1–22 (1994)
5. E Bertino, S Jajodia, P Samarati, Supporting multiple access control policies in database systems, in *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (IEEE Computer Society Washington, D.C. 1996), p. 94
6. G Dinolt, L Benzinger, M Yatabe, Combining components and policies, in *Proceedings of the Computer Security Foundations Workshop VII* (Franconia, USA, 1994)
7. S Benferhat, R El Baida, F Cuppens, A stratification-based approach for handling conflicts in access control, in *SACMAT 2003, Proceedings of the 8th ACM Symposium on Access Control Models and Technologies, Como, Italy* (ACM New York, 2003), pp. 189–195. ISBN:1-58113-681-1
8. R Reiter, Sequential, temporal Golog, in *Principles of Knowledge Representation and Reasoning: Proceedings of the 6th International Conference (KR' 98)* (Morgan Kaufmann, San Francisco, 1998), pp. 547–556
9. S Hanks, D McDermott, ed. by ML Ginsberg, Default reasoning, nonmonotonic logics, and the frame problem, in *Readings in Nonmonotonic Reasoning* (Morgan Kaufmann, San Francisco, 1987), pp. 390–395
10. C Green, Application of theorem proving to problem solving, in *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI'69)* (Morgan Kaufmann, San Francisco, 1969), pp. 219–239
11. RRO, Règlement 965. Gestion Hospitalière. Dossiers de renseigments personnels sur la santé (1990)
12. N Essaouini, F Cuppens, N Cuppens-Boulahia, AAE Kalam, *Proceedings of the eighth International Conference on Availability, Reliability and Security*. (IEEE Computer Society, Los Alamitos, 2013)
13. HJ Levesque, R Reiter, Y Lesperance, F Lin, RB Scherl, GOLOG: a logic programming language for dynamic do- mains. J. Logic Program. **31**(1–3), 59–83 (1997)
14. JI Balla, WE Jamieson, Improving the continuity of care between general practitioners and public hospitals. Med. J. Aust. **161**(11–12), 656–659 (1994)
15. P Bolton, A quality assurance activity to improve discharge communication with general practice. J. Qual. Clin. Pract. **21**, 69–70 (2001)
16. DC Adams, JB Bristol, KR Poskitt, Surgical discharge summaries: improving the record. Ann. R. Coll. Surg. Engl. **75**(2), 96–99 (1993)
17. PJ Embi, *Perceived Impact of Computerized Physician Documentation on Education and Clinical Practice in a Teaching Hospital*. (Oregon Health & Science University, Portland, 2002)
18. RJ Mageean, Study of "discharge communications" from hospital. Br. Med. J. **293**(6557), 1283–1284 (1986)
19. AN Raval, GE Marchioriz, JM Arnold, Improving the continuity of care following discharge of patients hospitalized with heart failure: is the discharge summary adequate? Can. J. Cardiol. **19**(4), 365–370 (2003)
20. J McCarthy, *Situations, Actions and Causal Laws*, Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing. (M Minskys, ed.) (MIT Press, Cambridge, 1968), pp. 410–417
21. R Reiter, ed. by L Aiello, J Doyle, and S Shapiro, Natural actions, concurrency and continuous time in the situation calculus, in *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference* (Morgan Kaufmann, San Francisco, 1996), pp. 2–13
22. F Lin, R Reiter, State constraints revisited. J. Logic Comput. **4**, 655–678 (1994). Special issue on actions and processes
23. R Reiter, ed. by V Lifschitz, The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression, in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy* (Academic Press San Diego, 1991), pp. 359–380
24. R Reiter, Proving properties of states in the situation calculus. Artif. Intell. **64**(2), 337–351 (1993)
25. CC Green, ed. by B Meltzer, D Michie, Theorem proving by resolution as a basis for question-answering systems, in *Machine Intelligence, volume 4* (American Elsevier New York, 1969), pp. 183–205
26. R Fikes, NJ Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving. Artif. Intell. **2**(3/4), 189–208 (1971)
27. TW Frühwirth, A Herold, V Küchenhoff, T Le Provost, P Lim, E Monfroy, M Wallace, ed. by G Comyn, NE Fuchs, and M Ratcliffe, Contraint logic programming - an informal introduction, in *Proceedings of the Second International Logic Programming Summer School on Logic Programming in Action (LPSS '92)* (Springer-Verlag London, pp. 3–35
28. C Bettini, S Jajodia, XS Wang, D Wijesekera, Obligation monitoring in policy management, in *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), 5–7 June 2002* (IEEE Computer Society Monterey, 2002)
29. N Damianou, N Dulay, E Lupu, M Sloman, The ponder policy specification language. Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY '01), 18–38 (2001)
30. Q Ni, E Bertino, J Lobo, An obligation model bridging access control policies and privacy policies, in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT '08)* (ACM New York, 2008), pp. 133–142
31. R Craven, J Lobo, J Ma, A Russo, E Lupu, A Bandara, Expressive policy analysis with enhanced system dynamicity, in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS '09)* (ACM New York, 2009), pp. 239–250
32. Oasis, Extensible access control markup language tc v2.0, normative xacml 2.0 documents. OASIS Standard. http://www.oasis-open.org/specs/index.php. Accessed 1 Feb 2005
33. M Hilty, D Basin, A Pretschner, ed. by S di Vimercati, P Syverson, and D Gollmann, On obligations, in *Computer Security-ESORICS 2005, Lecture Notes in Computer Science*, vol. 3679 (Springer Berlin, 2005), pp. 98–117
34. J Park, R Sandhu, The UCONABC usage control model. ACM Trans. Inf. Syst. Secur. **7**(1), 128–174 (2004)
35. F Cuppens, N Cuppens-Boulahia, T Sans, Nomad: a security model with non atomic actions and deadlines, in *18th IEEE Computer Security Foundations Workshop (CSFW)* (Aix en Provence, France, 2005)
36. T Sans, F Cuppens, N Cuppens-Boulahia, A framework to enforce access control, usage control and obligations. Ann. Telecomm. **62**(11–12), 1329–1352 (2007)
37. L Lamport, The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
38. S Goedertier, J Vanthienen, ed. by J Eder, S Dustdar, Designing compliant business processes with obligations and permissions, in *Business Process Management Workshops, Lecture Notes in Computer Science*, vol. 4103 (Springer Berlin, 2006), pp. 5–14

doi:10.1186/s13635-014-0013-5
Cite this article as: Essaouini *et al.*: **Conflict detection in obligation with deadline policies.** *EURASIP Journal on Information Security* 2014 **2014**:13.