In the Nick of Time: Proactive Prevention of Obligation Violations

David Basin
Information Security
ETH Zurich, Switzerland
Email: basin@inf.ethz.ch

Søren Debois
Process and System Models Group
IT University of Copenhagen, Denmark
Email: debois@itu.dk

Thomas T. Hildebrandt
Process and System Models Group
IT University of Copenhagen, Denmark
Email: hilde@itu.dk

***Abstract*—We present a system model, an enforcement mechanism, and a policy language for the proactive enforcement of timed provisions and obligations. Our approach improves upon existing formalisms in two ways: (1) we exploit the target system's existing functionality to avert policy violations proactively, rather than compensate for them reactively; and, (2) instead of requiring the manual specification of remedial actions in the policy, we automatically deduce required actions directly from the policy. As a policy language, we employ timed dynamic condition response (DCR) processes. DCR primitives declaratively express timed provisions and obligations as causal relationships between events, and DCR states explicitly represent pending obligations. As key technical results, we show that enforceability of DCR policies is decidable, we give a sufficient polynomial time verifiable condition for a policy to be enforceable, and we give an algorithm for determining from a DCR state a sequence of actions that discharge impending obligations.**

I. Introduction

Many security requirements can be decomposed into provisions and obligations [4], [17], [26]. Provisions specify conditions or properties dependent on the present and the past. They cover most traditional access control requirements. For example, access to customer records is granted to users in the role of customer-relations manager, provided customer consent was previously granted. Obligations, in contrast, impose conditions on the future that an agent or process should fulfil. For example, a hospital may need to delete patient records within 14 days of a patient's release.

Provisions and their enforcement by access control mechanisms are well understood. Obligations are less well understood, and subject to active research [1], [3], [10]–[13], [18], [20], [22]–[25], [31], [35]. Enforcement of obligations is difficult as, to be enforceable, obligations must be associated with deadlines. A simple but limited enforcement mechanism is to associate obligations with access control rules, whereby the enforcement mechanism immediately takes the obliged action when the rule grants access, e.g., logging the taken action. Alternatively, obligations may be associated with deadlines, whose expiration triggers remedial actions to be taken by the access control mechanism.

The state of the art generally handles obligations in limited ways, like those suggested above. The theory of how to handle obligations is underdeveloped, especially when deadlines are involved. With few exceptions, policy violations are not prevented, they are remediated. Namely, the enforcement mechanism witnesses a deadline expiring, but is powerless to prevent the concomitant policy violation and is reduced to taking remedial actions after the fact, such as logging, lowering a reputation, etc. Moreover, the enforcement mechanism's interaction with the target system is often too limited for effective obligation enforcement or the exact extent of the mechanism's control over the target system is unclear. While an enforcement mechanism can intercept actions and prevent them from happening, it cannot, a priori, force the target system to take action when required. Existing mechanisms tend to take only actions *independent* of the target system's functionality, such as logging or sending notifications. We expand on these points and discuss exceptions in Section VI.

Approach and Results. We tackle the problem of proactive policy enforcement and present an enforcement mechanism that directs the target system to prevent policy violations. Not every policy can be enforced, and enforceability depends on the enforcement mechanism's exact powers over the target system. We distinguish between whether the enforcement mechanism can (1) *control* an action by denying that it happens at a given point in time, (2) proactively *cause* an action to happen in the target system, or (3) merely *observe* that an action happens in the target system.

The above distinctions are critical. For some actions, e.g. a patient at a hospital dies, it is neither meaningful for an enforcement function to deny nor cause the action to happen. In other cases it may make sense for a mechanism to control whether the action is allowed, but not for the mechanism to cause the action to happen by itself. For instance, a hospital IT system may be able to deny the immediate re-admission of a released patient; however, it cannot outright cause a patient to be readmitted, as that would require the patient's consent. Finally, some actions may be both controllable and causable, e.g., the enforcement mechanism can both deny and proactively cause the transfer of records from local data storage to a remote archive, depending on the exact circumstances. Such distinctions between controllable (in the sense of denying) and uncontrollable (but observable) actions are well-known in other areas, such as supervisory-control theory [32]. Here the supervisor plays the role of the enforcement mechanism; however, in the classical supervisory-control theory, the supervisor does not cause actions in the target system.

These observations motivate the following technical questions, answered in this paper. Given the ability to *cause* certain actions to happen in the target system, how do we decide whether a given policy is in fact enforceable? Can we efficiently compute a sequence of actions sufficient to resolve a given impending obligation violation? And in what sense can this be done transparently in that the enforcement mechanism alters the target system's behaviour only when absolutely required by the policy?

Answering these questions necessarily involves a system model, a policy language, and an enforcement mechanism. The *system model* must clarify how the target system and the enforcement mechanism interact and what the mechanism can and cannot cause or control in the target system. The *policy language* must be expressive enough to formulate realistic policies, yet constrained enough that we can efficiently compute (1) whether a policy is enforceable and (2) which actions are needed to avert impending obligation violations. Finally, the *enforcement mechanism* connects the system model and the policy, using the latter to compute action sequences for execution by the former.

In our policy language, it must be possible to express declaratively timed properties of both the past and the future, close to their natural-language formalisation. Moreover, we need a run-time representation that can be updated as events occur, and we must also be able to plan the actions to be taken to avoid violating obligations. Standard approaches, based on metric temporal logics for specifications and automata for the run-time representation, depend on a translation from formulas into automata, which suffers from state-space explosion. The formalism that we will use, *Timed Dynamic Condition Response Processes* (DCR processes, for short), combines the declarative specification and run-time representation, avoiding this translation. As we will show, this allows us to compute plans and enforce policies at run-time based directly on the declarative specification, avoiding translation to an exponentially larger operational model.

Untimed DCR processes were introduced in [8] as a process language for describing DCR graphs [15], a declarative graphical process notation. DCR processes are expressive enough for many security applications: For untimed properties they are equivalent in their expressivity to Büchi automata [8], but their language primitives are rather different. Instead of specifying processes in terms of states and transitions, a DCR process describes the causal relationships between events declaratively in terms of conditions and responses, and describes dynamic conflict relationships between events in terms of exclusions and inclusions. To allow for the specification of timed polices, we conservatively extend the DCR process language introduced in [8] to allow for describing timed DCR graphs [16].

Contributions. Conceptually, we present a system model, a policy language, and an enforcement mechanism for timed provisions and obligations. Our enforcement mechanism *deduces* those actions necessary to avert policy violations and *proactively causes* the target system to take these actions *in the nick of time*, that is, whenever a deadline is about to be missed.

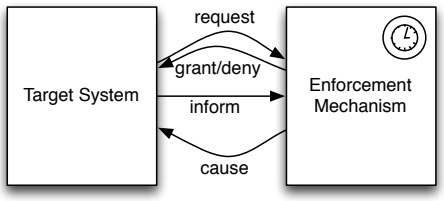

Fig. 1. System Model

This approach improves upon existing formalisms in two ways: (1) we exploit the target system's existing functionality to avert proactively policy violations, rather than to compensate reactively for them, and, (2) rather than requiring the manual specification of remedial actions in the policy, we automatically deduce relevant actions directly from the policy.

Technically, we show that the enforceability of policies expressed as timed DCR processes is decidable but NP-hard. We then give a sufficient polynomial time verifiable condition for a DCR policy to be enforceable. Moreover, we give an algorithm that, given a DCR state of an enforceable DCR policy, computes a sequence of actions that, when executed on the target system, will discharge impending obligations.

As proof-of-concept, we have built a prototype implementation of the algorithms in this paper. The implementation is available on-line at http://dcr.itu.dk/obligations along with simulations of the examples presented in this paper.

Scope. We focus exclusively on policies governing the sequencing of actions. This approach plays to the strengths of the DCR formalisms and helps focus the presentation on the central issue of proactive policy enforcement. We leave open extensions to the policy language, like the addition of events dependent on data, and the question of whether and how comparable enforcement mechanisms can be realised in other formalisms.

Overview. In Section II we present our system model and define enforcement. We present timed DCR processes in Section III, and give examples of policies in Section IV. We show in Section V when and how a DCR policy is enforceable and report briefly on a prototype implementation of a DCR policy enforcement point. Finally, in Sections VI and VII we discuss related work and draw conclusions. Proofs and most lemmas have been relegated to Appendix A.

II. System Model and Enforcement

A. System Model

Our system model is depicted in Figure 1. The target system and the enforcement mechanism (also called a Policy Enforcement Point, or simply PEP) are independently running *processes*, which interact in three distinct ways:

1) Whenever the target system wishes to undertake some *controllable* action, it requests permission from the enforcement mechanism (upper arrow, left to right), which

will return either "grant" or "deny" (upper arrow, right to left). The target system actually undertakes the requested action iff the enforcement mechanism responds "grant".
2) Whenever the target system performs an *uncontrollable* action, it informs the enforcement mechanism that it does so (middle arrow).
3) Finally, a subset of actions of the target system, its *causable actions*, are available to be triggered by the enforcement mechanism, as indicated by the bottom arrow labelled "cause".

(1) and (2) ensure that the target system and the enforcement mechanism are synchronised. (1) makes it possible to suppress controllable actions, thereby enforcing provisions. Through (3), the architecture supports the proactive policy enforcement. Note that the target system may take internal actions not observable by the enforcement mechanism; the policy enforced must be independent of such internal actions.

In this paper we will restrict our attention to discrete time systems (as in, e.g., [5], [27]) and assume that enforcement mechanisms can only proactively cause actions within a time-step, just before a tick of time. For example, if the target system needs to undertake action a within deadline d, the enforcement mechanism can, before some tick within the deadline d, force the system to undertake action a, using the lower arrow. Note that the enforcement mechanism relies only on its own clock (indicated by the clock symbol in the diagram). The model does not stipulate synchronisation between the clocks of the enforcement mechanism and the target system.

In practice, there are various ways that the abstract communication in the above model can be realised. For example, suppose the target system is an HTTP/REST component within a larger system, and that "actions" are the external invocation of its own APIs, or its own invocation of other components' APIs. The upper arrow can simply and crudely be implemented by the enforcement mechanism intercepting and selectively dropping incoming and outgoing requests; TCP's failure semantics will handle the rest. The middle arrow is just listening to messages representing uncontrollable actions. The lower arrow is the enforcement mechanism issuing HTTP requests against the target system's API. A more relaxed implementation might see the enforcement mechanism realising the upper arrow as its own HTTP/REST API.

B. Target System and Policies

Prior to formalising systems and policies, we first introduce relevant notation. For a finite alphabet Σ of actions, we write Σ^* for the set of finite words over Σ, Σ^ω for the infinite words, and let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. As usual, we refer to a set $\mathcal{L}$ of words as a *language*. We write concatenation of words w and v as $w \cdot v$ and write $v \sqsubseteq w$ iff v is a prefix of w, i.e. $w = v \cdot v'$ for some word v'. We write $w \setminus x$ for the word w with every occurrence of the symbol $x \in \Sigma$ removed. Finally, we let $\mathsf{prefixes}(\mathcal{L})$ be the prefix closure of the language $\mathcal{L}$ and write ϵ for the empty word.

We account for time by requiring a special symbol tick in our alphabets; the passage of time is indicated by the occurrence of this tick action. We say that a word is *non-Zeno* if it is finite or contains infinitely many tick actions; a language $\mathcal{L}$ is non-Zeno if every word in it is non-Zeno.

Now, following the system model, a target system is a language over an alphabet of uncontrollable actions, controllable actions, causable actions, and time.

Definition 1: A *target system* $(\mathcal{S}, \Sigma, \Gamma, \Delta)$ comprises a prefix-closed language $\mathcal{S} \subseteq \Sigma^*$ with $\mathsf{tick} \in \Sigma$; *controllable actions* $\Gamma \subseteq \Sigma \setminus \{\mathsf{tick}\}$; and *causable actions* $\Delta \subseteq \Sigma \setminus \{\mathsf{tick}\}$.

In this definition, prefix-closure ensures that the target system S produces actions consecutively. Time is considered neither controllable nor causable: it can be neither suppressed nor caused. We consider only the finite behaviour of the target system due to our focus on the enforcement of safety properties in the form of provisions and deadlines. Still, timed security policies may speak of safety properties that are never fulfilled for finite sequences, such as the property, that "an examination must be performed every other month". Consequently, we define policies as non-Zeno languages over both finite and infinite sequences.

Definition 2: A *security policy* $\mathcal{P}$ over an alphabet Σ with $\mathsf{tick} \in \Sigma$ is a non-Zeno language $\mathcal{P} \subseteq \Sigma^\infty$.

Example 3: As a running example, consider a *target system* $t = (\mathcal{S}, \Sigma, \Gamma, \Delta)$ with alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$, causable actions $\Delta = \{\mathsf{a}\}$, controllable actions $\Gamma = \{\mathsf{b}\}$, and language $\mathcal{S}$ defined by the regular expression $(\mathsf{a} + \mathsf{b} + \mathsf{tick})^*$, i.e., any finite sequence over a, b, and tick. We want to enforce the *policy* $\mathcal{P}$ defined by the ω-regular expression $(\mathsf{a}^+\ \mathsf{tick})^\omega$, i.e., b is never allowed, and every tick is preceded by at least one a.

C. Enforcement Mechanisms

Now, what should an enforcement mechanism accomplish? We define here *abstractly* the requirements for such a mechanism. Later, in Section V, we give a concrete instance.

Following [24], [31], the result of enforcement should be a stream of actions performed jointly by the target system and the enforcement mechanism. Clearly, the mechanism should ensure that this stream complies with the policy being enforced. However, rather than retroactively *translating* the output of the target system as done in most previous work, our focus is on *proactive* enforcement: the enforcement mechanism *directs* the target system by suppressing and causing actions. It cannot cause actions not enabled in the target system to happen, i.e., it does not add to the possible behaviours.

To capture what it means to direct the target system, we define the *directed language* of a monotone, idempotent function $m : \mathcal{S} \to \mathcal{S}$ as the least fixed point of "directed steps" extending the execution of an already directed trace w by an action a into $m(w \cdot a)$.

Definition 4: Let $(\mathcal{S}, \Sigma, \Gamma, \Delta)$ be a target system, and let $m : \mathcal{S} \to \mathcal{S}$ be a monotone and idempotent function. Define the *directed language* $\mathcal{D}_m$ of m inductively by

1) $m(\epsilon) \in \mathcal{D}_m$
2) $m(w \cdot a) \in \mathcal{D}_m$ whenever $w \in \mathcal{D}_m$, $a \in \Sigma$, and $w \cdot a \in \mathcal{S}$.

Note that monotonicity (i.e. if $v \sqsubseteq w$ then $m(v) \sqsubseteq m(w)$) and idempotency (i.e. $m(m(v)) = m(v)$) means respectively, that the function m does not retroactively change its past decisions and it agrees with its own directives.

An enforcement mechanism for a target system can now be defined as a function $m : \mathcal{S} \to \mathcal{S}$ for which the directed steps respect the constraints of controllable and causable actions, and only cause actions immediately before a tick action.

Definition 5: An *enforcement mechanism* for a target system $(\mathcal{S}, \Sigma, \Gamma, \Delta)$ is a recursive, monotone, and idempotent function $m : \mathcal{S} \to \mathcal{S}$ satisfying:

1) $m(\epsilon) = \epsilon$
2) for all $v \in \mathcal{D}_m$ and $a \in \Sigma$ with $v \cdot a \in \mathcal{S}$, we have $m(v \cdot a) = m(v) \cdot w$ where w satisfies:
   a) if $a = \mathsf{tick}$ then $w \in \Delta^* \cdot \mathsf{tick}$;
   b) if $a \in \Gamma$ then $w \in \{\epsilon, a\}$; and
   c) otherwise $w = a$.

When m is an enforcement mechanism, we will call $\mathcal{D}_m$ the *enforcement language*.

The conditions on the function m formalise that: (1) m will wait for time to pass or the target system to take an action before acting; (2a) m cannot stop the passage of time, but may cause actions in Δ to be taken just before a tick; (2b) m may either suppress or preserve controllable actions; and (2c) m must preserve uncontrollable actions.

Example 6: We define a function $m : \mathcal{S} \to \mathcal{S}$ for the target system t of Example 3. The function m removes all bs and inserts a before tick when necessary.

$$m(w) = n(w \setminus \mathsf{b})$$

$$n(w) = \begin{cases} \epsilon & \text{when } w = \epsilon \\ \mathsf{a} \cdot \mathsf{tick} & \text{when } w = \mathsf{tick} \\ n(w \cdot \mathsf{tick}) \cdot \mathsf{a} \cdot \mathsf{tick} & \text{when } w = w' \cdot \mathsf{tick} \cdot \mathsf{tick} \\ n(w') \cdot x & \text{otherwise, assuming wlog } w = w' \cdot x \end{cases}$$

It is straightforward to verify that both m and n are recursive, monotone, and idempotent functions, and to prove by induction on w that the directed language $\mathcal{D}_m$ of m is exactly the set $\mathcal{D}_m = \mathsf{prefixes}\big((\mathsf{a}^+ \ \mathsf{tick})^\omega\big)$. Moreover, it is easy to see that m satisfies the remaining conditions of Definition 5.

We proceed to consider correctness and transparency. Our notion of correctness necessarily deviates from standard notions. In the presence of pending obligations, the current (corrected) output $m(v)$ might not be a word of the policy. However, it must be extensible to one that is. Intuitively, the extension w discharges pending obligations, taking the output string $m(v) \cdot w$ back into the policy language.

Definition 7: Let $\mathcal{P}$ be a policy over Σ. An enforcement mechanism m is *correct* for $\mathcal{P}$ iff for all $v \in \mathcal{D}_m$ there exists some $w \in \Sigma^\infty$ such that $v \cdot w \in \mathcal{P}$.

A violation of the policy $\mathcal{P}$ is a word u that has no possible extension to a word in the policy: no matter what the target system subsequently does, it will *never* get u back within the bounds of the policy. This situation would arise if, for example, an impermissible action was executed, or a deadline was missed. Our notion of correctness is such that a correct enforcement mechanism will tolerate no such *finite violations*

Lemma 8: Define the *finite violations language* $\bar{\mathcal{P}}$ of a policy $\mathcal{P}$ over Σ by $\bar{\mathcal{P}} = \Sigma^* \setminus \mathsf{prefixes}(\mathcal{P})$. If m is correct for policy $\mathcal{P}$ then for all $v \in \mathcal{D}_m$ we have $m(v) \notin \bar{\mathcal{P}}$.

Note that this language is closely related to the notion of *bad prefixes* for languages over infinite words, defined in [21].

We now formulate transparency in terms of the finite violations language:

Definition 9: Let m be an enforcement mechanism for a target system $(\mathcal{S}, \Sigma, \Gamma, \Delta)$. We say that m is *transparent* iff for all $v \in \mathcal{D}_m$ and $a \in \Sigma$ such that $v \cdot a \in \mathcal{S}$, whenever $m(v \cdot a) \neq m(v) \cdot a$ then $m(v) \cdot a \in \bar{\mathcal{P}}$.

That is, a transparent enforcement mechanism modifies an action a iff taking the action would violate the policy.

Example 10: Continuing Example 6, we saw that the enforcement language of $\mathcal{D}_m$ is exactly the prefixes of the policy $\mathcal{P} = (\mathsf{a}^+ \ \mathsf{tick})^\omega$ and hence m is a correct enforcement mechanism. It is straightforward to verify by cases on the last symbol of w that it is also transparent. Note that while the policy language $\mathcal{P}$ contains no finite words, the enforcement language $\mathcal{D}_m$ does: $\mathsf{prefixes}(\mathcal{P}) = \mathcal{D}_m$.

The question remains of how to construct useful enforcement mechanisms and build practical, running PEPs based on them. In the coming sections, we will show how timed DCR processes can be used for this purpose.

III. Policy Specification Language

We present here timed DCR processes, the semantics of which defines a security policy over an alphabet Σ in the sense of Definition 2. The language is closely based on the core DCR process language [8], conservatively extended to make it possible to express timed DCR graphs as introduced in [16].

DCR processes are about *events* $\mathcal{E}$ and *constraints* between events. Constraints define under what circumstances (1) events may or may not happen, and (2) under what circumstances they may be required to happen in the future or be dynamically excluded from (or re-included to) the process.

In general, each event $e \in \mathcal{E}$ has an associated label $\ell(e) \in \Sigma \setminus \{\mathsf{tick}\}$. The set Σ (which includes tick) will be used as the (finite) alphabet for defining the language recognised by a timed DCR process. For simplicity, we restrict our attention to DCR processes where $\mathcal{E} = \Sigma \setminus \{\mathsf{tick}\}$ and $\ell(e) = e$.

A. Syntax

A DCR process $P = [M]\ T$ comprises a *marking* M and a *term* T; the full syntax is given in Figure 2. Here $\mathbf{N}$ is the set of natural numbers, excluding 0. The marking M specifies the state of events; the term T specifies both constraints between events, and the effects on that state of executing events. We explain terms and markings separately.

Terms. Terms describe constraints and effects between events as follows.

- A *condition* $e \bullet\!\!\xleftarrow{k} f$ imposes the *constraint* that for the event e to happen, the event f must either previously

$$
\begin{array}{rll}
T, U ::= & e \bullet\!\!\xleftarrow{k} f & \text{condition, } k \in \mathbf{N} \cup \{0\} \\
| & e \bullet\!\!\xrightarrow{d} f & \text{response, } d \in \mathbf{N} \cup \{\omega\} \\
| & e \rightarrow\!+ f & \text{inclusion} \\
| & e \rightarrow\!\% f & \text{exclusion} \\
| & e \diamond\!\leftarrow f & \text{milestone} \\
| & T \mid U & \text{parallel} \\
| & 0 & \text{unit} \\
\\
\Phi ::= & (h, i, r) & \text{event state} \\
M, N ::= & M, e : \Phi & \text{marking} \\
| & \epsilon & \\
P, Q ::= & [M]\, T & \text{process}
\end{array}
$$

Fig. 2. DCR Process Syntax.

have happened at least k time units ago, or currently be excluded. Note that k is a natural number or zero.

- A *response* $e \bullet\!\!\xrightarrow{d} f$ imposes the *effect* that when e happens, f becomes pending (obliged) and must happen within d time units or be excluded. Note that the deadline d is a natural number or infinity ("eventually"), but cannot be zero—one cannot require things to happen "now".
- An *exclusion* $e \rightarrow\!\% f$ imposes the *effect* that when e happens, it *excludes* f. An excluded event cannot happen; it is ignored as a condition. Moreover, it need not happen if pending, unless it is subsequently re-included.
- An *inclusion* $e \rightarrow\!+ f$ imposes the *effect* that when the event e happens, it re-includes the event f.
- A *milestone* $e \diamond\!\leftarrow f$ imposes the *constraint* that for the event e to happen, the event f must be either not pending or excluded.

If several condition (response) constraints are defined between the same two events in T, the process will have the maximal delay (minimal deadline). An untimed process [8] corresponds to a timed process with all delays 0 and all deadlines ω, so we write $e \bullet\!\!\rightarrow f$ for $e \bullet\!\!\xrightarrow{\omega} f$ and $f \bullet\!\!\leftarrow e$ for $f \bullet\!\!\xleftarrow{0} e$.

Example 11: Recall the obligation of the hospital given in the introduction, where the event delete ("the patient's record is deleted") must happen within 14 days after the event release ("a patient is released from the hospital"). We specify this obligation with a timed response relation:

$$\mathsf{release} \bullet\!\!\xrightarrow{14d} \mathsf{delete}\,.$$

If we instead wish to model the provision that the event archive ("archiving data"') cannot be followed by the event unarchive ("delete archived data") for at least 8 years, we use a timed condition:

$$\mathsf{unarchive} \bullet\!\!\xleftarrow{8y} \mathsf{archive}\,.$$

Markings. The marking M is a finite map from events to triples (h, i, r), called the *event state*. The first component, $h \in \mathbf{N} \cup \{0, \bot\}$, indicates whether the event happened, and if so how many ticks ago, i.e., the event's age. Namely, $h \in \mathbf{N} \cup \{0\}$ represents that the event happened h ticks ago and $\bot$ represents that it did not happen. Note that an event may happen several times, and the state only records the age of the last occurrence. The second component, $i \in \{\bot, \top\}$, is a boolean indicating whether the event is currently *(i)ncluded*. Finally, the third component, $r \in \mathbf{N} \cup \{0, \omega, \bot\}$, indicates whether the event is a *pending (r)esponse*, that is, obliged to happen in the future, and if so a possible deadline. Here $r = \bot$ represents that it is not pending, a natural number $r \in \mathbf{N} \cup \{0\}$ represents an unfulfilled obligation that the event should happen within r time steps, and ω represents that the event is obliged to happen eventually, i.e. without any fixed deadline. We write $\mathsf{dom}(M)$ for the domain of the marking M and take $\mathsf{dom}([M]\, T) = \mathsf{dom}(M)$. As is commonly done for environments, we write markings as finite lists of pairs of events and states, e.g., $M = e_1 : \Phi_1, \ldots, e_k : \Phi_k$. We understand the extension $M, e : \Phi$ of such an environment M to be undefined when $e \in \mathsf{dom}(M)$.

B. Enabledness and effects

A process $[M]\, T$ has some subset of its events *enabled*. Enabled events can be *executed* and will, when executed, apply an *effect* to the marking M, yielding a new marking M'. We shall use these notions in Section III-C to define a timed labelled transition system (LTS) for a given DCR process $[M]\, T$, which has markings as states and event executions and time steps as transitions. Given this LTS, we can then define the language accepted by $[M]\, T$.

The notions of *enabled* events and *effects* are given by the judgement $[M]\, T \vdash e : E, I, R$, defined in Figure 3. The judgement should be read: "in the marking M under the constraints T, e is enabled and will when executed have the effect of excluding events E, including events I, and recording the pending responses R, possibly with deadlines."

Rules 1–2 formalise constraints between events. The first rule says that if f is a *condition* for e, then e can happen only when (1) it is itself included, and (2) if f is included, then f previously happened at least k steps ago. The second says that if f is a *milestone* for e, then e can happen only if (1) it is itself included, and (2) if f is included, then it is not pending.

Rules 3–5 formalise the effects an event might have on other events (and itself). The third rule says that if f is a *response* to e with deadline d and e is included, then e can happen with the effect of making f pending with deadline d. The fourth (respectively fifth) rule says that if f is *included* (respectively *excluded*) by e and e is included, then e can happen with the effect of including (respectively excluding) f.

Rules 6–8 formalise the composition of rules. The sixth rule says that for the completely unconstrained process 0, an event e can happen if it is currently included. The seventh rule says that a relation allows any included event e to happen without effects when e is not constrained by that relation; that is, when e is not the relation's left-hand–side event. Finally, the last rule accounts for compositionality: it says that if both T_1 and T_2

$$\frac{i \Rightarrow h \geq k}{[M, f : (h, i, \_), e : (\_, \top, \_)]\ e \bullet\!\!\xleftarrow{k} f \vdash e : \emptyset, \emptyset, \emptyset} \quad (1)$$

$$\frac{i \Rightarrow r = \bot}{[M, f : (\_, i, r), e : (\_, \top, \_)]\ e \diamond\!\!\leftarrow f \vdash e : \emptyset, \emptyset, \emptyset} \quad (2)$$

$$\frac{}{[M, e : (\_, \top, \_)]\ e \bullet\!\!\xrightarrow{d} f \vdash e : \emptyset, \emptyset, \{f : d\}} \quad (3)$$

$$\frac{}{[M, e : (\_, \top, \_)]\ e \rightarrow\!+\ f \vdash e : \emptyset, \{f\}, \emptyset} \quad (4)$$

$$\frac{}{[M, e : (\_, \top, \_)]\ e \rightarrow\!\% \ f \vdash e : \{f\}, \emptyset, \emptyset} \quad (5)$$

$$\frac{}{[M, e : (\_, \top, \_)]\ 0 \vdash e : \emptyset, \emptyset, \emptyset} \quad (6)$$

$$\frac{e \neq f}{[M, e : (\_, \top, \_)]\ f\ \mathcal{R}\ f' \vdash e : \emptyset, \emptyset, \emptyset} \quad (7)$$

$$\frac{[M]\ T_1 \vdash e : E_1, I_1, R_1 \qquad [M]\ T_2 \vdash e : E_2, I_2, R_2}{[M]\ T_1 \mid T_2 \vdash e : E_1 \cup E_2, I_1 \cup I_2, R_1 \cup R_2} \quad (8)$$

Fig. 3. Enabling and effects. We write "_" for "don't care" and write $\mathcal{R}$ for any of the relations $\bullet\!\!\xleftarrow{k}$,$\diamond\!\!\leftarrow$, $\bullet\!\!\xrightarrow{d}$, $\rightarrow\!+$, or $\rightarrow\!\%$.

allow an event to execute, then so does $T_1 \mid T_2$, with the effect of the event being the union of the component effects.

We proceed to define how the effects derived in Figure 3 affect a marking. Suppose e is enabled in the process $[M]\ T$ with the effect $\delta = (E, I, R)$. We first register in the state of e that it happened now (setting the age to 0) and that it is no longer pending (setting the response deadline to $\bot$). Formally, we define $e\langle M\rangle$ inductively by $e\langle\epsilon\rangle = \epsilon$ and $e\langle M, f : (h, i, r)\rangle = e\langle M\rangle, f : (h', i', r')$, where $(h', i', r') = (0, i, \bot)$ if $e = f$ and otherwise $(h', i', r') = (h, i, r)$. We then apply the effect $\delta = (E, I, R)$ of the event, that is, excluding the events in E, including the events in I and registering response deadlines given in R. Formally, we inductively define $\delta\langle M\rangle$ by $\delta\langle\epsilon\rangle = \epsilon$ and

$$\delta\langle M, f : (h, i, r)\rangle = \delta\langle M\rangle, f : \big(h, \underbrace{(i \wedge f \notin E) \vee f \in I}_{\text{included?}}, r'\big), \quad (9)$$

where $r' = \mathsf{min}\{d \mid f : d \in R\}$ if $f : d \in R$ and $r' = r$ otherwise. That is, if $f : d \in R$ then f is marked pending with the minimal deadline d for which $f : d \in R$; otherwise, it keeps the deadline recorded in the current state. Note that if an event is both excluded and included by the effect, inclusion takes precedence.

Example 12: Consider the process

$$[\mathsf{release} : (\bot, \top, \bot), \mathsf{delete} : (\bot, \top, \bot)]\ \mathsf{release} \bullet\!\!\xrightarrow{14d} \mathsf{delete}\,. \quad (10)$$

Both events have the same state: they have not been previously executed, they are included, and they are not pending, i.e., have no obligation to eventually execute. Following Figure 3, we find that both events are enabled with the following effects:

$$\mathsf{release} : \emptyset, \emptyset, \{\mathsf{delete} : 14d\} \qquad \text{and} \qquad \mathsf{delete} : \emptyset, \emptyset, \emptyset\,.$$

Applying first the event release to the marking of (10), we obtain the following (where we highlight changes in grey):

$$\mathsf{release}\langle\mathsf{release} : (\bot, \top, \bot), \mathsf{delete} : (\bot, \top, \bot)\rangle = \mathsf{release} : (\,0\,, \top, \bot), \mathsf{delete} : (\bot, \top, \bot)\,.$$

Now applying the effect $(\emptyset, \emptyset, \{\mathsf{delete} : 14d\})$ we get

$$(\emptyset, \emptyset, \{\mathsf{delete} : 14d\})\langle\mathsf{release} : (0, \top, \bot), \mathsf{delete} : (\bot, \top, \bot)\rangle = \mathsf{release} : (0, \top, \bot), \mathsf{delete} : (\bot, \top, \,\mathsf{14d}\,)\,.$$

That is, release is registered as having happened now (age 0) and delete is registered with deadline $14d$. A second example:

$$[\mathsf{archive} : (\bot, \top, \omega), \mathsf{unarchive} : (\bot, \top, \bot)]\ \mathsf{unarchive} \bullet\!\!\xleftarrow{8y} \mathsf{archive}\,.$$

This process has only the single enabled and pending event $\mathsf{archive} : \emptyset, \emptyset, \emptyset$. The unarchive event is not enabled, as it has an unfulfilled condition. Executing archive yields the following marking (this time skipping the intermediate steps):

$$\mathsf{archive} : (\,0\,, \top, \,\bot\,), \mathsf{unarchive} : (\bot, \top, \bot)\,.$$

In the new marking, archive is registered as having just happened (age 0) and no longer pending (response deadline $\bot$). The event unarchive is still not enabled, as the age of archive must be at least $8y$ for unarchive to be enabled.

Example 13: Returning to the running example, we represent that the event a should happen before every tick by making it initially pending and having itself as a response with deadline 1, and we suppress b by making it initially excluded:

$$[\mathsf{a} : (\bot, \top, 0), \mathsf{b} : (\bot, \bot, \bot)]\ \mathsf{a} \bullet\!\!\xrightarrow{1} \mathsf{a}\,. \quad (11)$$

When applying the event a to the marking, a gets the state $(\,0\,, \top, \,\bot\,)$, i.e. age 0 and no longer pending, but when we subsequently apply the effect $(\emptyset, \emptyset, \{\mathsf{a} : 1\})$, the deadline is set to 1 thus yielding the marking $[\mathsf{a} : (\,0\,, \top, \,1\,), \mathsf{b} : (\bot, \bot, \bot)]$.

C. Transition semantics

Prior to defining the LTS of a DCR process $[M]\ T$, we must account for time. First, we define the function deadline inductively on markings:

$$\mathsf{deadline}(\epsilon) = \omega$$
$$\mathsf{deadline}(M, e : (h, i, r)) = \mathsf{min}\{r', \mathsf{deadline}(M)\}$$

where $r' = r$ if $i = \top$, else $r' = \omega$. That is, deadlines of excluded events are ignored. We use the deadline function to ensure that time cannot progress beyond any deadline of an included event. When time advances, we update the marking by incrementing histories ("it is now $k + 1$ steps since the

$$\frac{[M]\ T \vdash e : \delta}{T \vdash M \xrightarrow{e} \delta\langle e\langle M\rangle\rangle}\ [\textsc{event}] \qquad \frac{\mathsf{deadline}(M) > 0}{T \vdash M \xrightarrow{\mathsf{tick}} \mathsf{tick}\langle M\rangle}\ [\textsc{time}]$$

Fig. 4. Transition semantics.

event e happened") and decrementing deadlines ("there are now $k-1$ steps left before we must do event e"). Formally, we inductively define the effect of tick on a marking:

$$\mathsf{tick}\langle\epsilon\rangle = \epsilon,$$
$$\mathsf{tick}\langle M, e : (h, i, r)\rangle = \mathsf{tick}\langle M\rangle, e : (h+1, i, \max\{0, r-1\}),$$

where $\bot + 1 = \bot$, $\bot - 1 = \bot$ and $\omega - 1 = \omega$. Using $\mathsf{tick}\langle - \rangle$ and deadline, we give timed transition semantics to processes in Figure 4. The *event transition* $M \xrightarrow{e} M'$ applies the effect of an enabled event e to the marking M. Note that in general more than one event e might be enabled. The *time transition* $M \xrightarrow{\mathsf{tick}} M'$ advances the time by one in the marking M. In examples so far, we have given relative time with units like $14d$ or $8y$. These units are just a convenience; one may normalise them to seconds, in which case they would be 1209600 or 252460800 respectively.

The transitions give rise to a timed event labelled transition system (LTS).

Definition 14: A timed DCR process $[M]\ T$ defines a timed event LTS, $\mathsf{lts}([M]\ T) = (\mathcal{M}, \mathcal{E}', \to, M, \ell', \Sigma)$, where the components are: *events* $\mathcal{E}' = \mathcal{E} \uplus \{\mathsf{tick}\}$; *transitions* $\to \subseteq \mathcal{M} \times \mathcal{E}' \times \mathcal{M}$, where $M \xrightarrow{\alpha} M'$ iff $T \vdash M \xrightarrow{\alpha} M'$; *states* $\mathcal{M} = \{M' \mid M \to^* M'\}$; *initial* state M; and *labelling* defined by $\ell'(e) = \ell(e)$ for $e \in \mathcal{E}$ and $\ell'(\mathsf{tick}) = \mathsf{tick}$.

Definition 15: A run of $\mathsf{lts}([M]\ T)$ is a finite or infinite sequence of transitions starting from the initial state $M = M_0 \xrightarrow{\alpha_0} \cdots$, for $\alpha_i \in \mathcal{E}'$.

The LTS has a notion of *acceptance*: a run is accepting iff it is non-Zeno and every response is eventually discharged. Recall that we write "_" for "don't care".

Definition 16: A run is *live* iff for every state M_i, if whenever an event e is pending in M_i, i.e., $M_i(e) = (\_, \top, d)$ for $d \neq \bot$, then there exists some $j \geq i$ such that either $M_j \xrightarrow{e} M_{j+1}$ or e is excluded in M_j, i.e., $M_j(e) = (\_, \bot, \_)$. A run is *accepting* iff it is live and non-Zeno. A *trace* of a process $[M]\ T$ is a possibly infinite word $s = (s_i)_{i \in I}$ such that $[M]\ T$ has an accepting run $M_i \xrightarrow{\alpha_i} M_{i+1}$ with $s_i = \ell(\alpha_i)$. The *timed language* $\mathsf{lang}(P)$ of a process P is the set of traces of P, which defines a security policy according to Definition 2.

Note that the set of traces is not necessarily prefix-closed, e.g. the timed language of the running example process (11) is indeed the policy $(\mathsf{a}^+\ \mathsf{tick})^\omega$.

Example 17: The following is a run of the process of (10):

$$\begin{aligned}
&\mathsf{release} : (\bot, \top, \bot), \mathsf{delete} : (\bot, \top, \bot)\\
\xrightarrow{\mathsf{release}}\ &\mathsf{release} : (\ 0\ , \top, \bot), \mathsf{delete} : (\bot, \top, \ 14\mathrm{d}\ )\\
\xrightarrow{\mathsf{tick}}\ &\mathsf{release} : (\ 1\ , \top, \bot), \mathsf{delete} : (\bot, \top, \ 13\mathrm{d}\ )\\
\xrightarrow{\mathsf{delete}}\ &\mathsf{release} : (1, \top, \bot), \mathsf{delete} : (\ 0\ , \top, \ \bot\ ).
\end{aligned}$$

This run is non-Zeno and accepting. Other runs exist:

- $\mathsf{release} \cdot \mathsf{tick}$ is a non-accepting finite run.
- $\mathsf{release} \cdot \mathsf{release} \cdot \ldots$ is an infinite Zeno run.
- $\mathsf{tick} \cdot \mathsf{tick} \cdot \ldots$ is an infinite, non-Zeno, accepting run.

IV. DCR Policy Examples

We now illustrate the mechanics of specifying provisions and obligations in a DCR process. We use these examples to clarify the subsequent discussion and results about the proactive enforceability of obligations. We consider a typical healthcare data retention policy like [19] from [2, Section 3.3].

A. Data protection in hospitals

Hospitals balance the dual requirements of protecting patients' privacy while documenting treatments given and procedures followed. This tension is resolved by retaining patient records in a central hospital database during the treatment and moving the records to a restricted-access archive shortly after the patient's release. In practice, a policy might look like this:

1) Records must be deleted within 14 days of release.
2) Records must not be deleted if archival is pending.
3) Records must be archived for at least 8 years.
4) Records must *not* be deleted should the patient be re-admitted within the 14 days.

We formalise this policy as a DCR process. For clarity, we consider only a single fixed patient and set of records. The events are as follows:

release	The patient is released.
delete	The patient's records are deleted from the central database.
archive	The patient's records are copied from the central database to the restricted long-term archive.
unarchive	The archived records are deleted.
readmit	The patient is re-admitted.

We formalise next the constraints T_0 and effects. We model the obligation 1) that records must be deleted within 14 days of the patient's release using a response.

$$\mathsf{release} \bullet\!\xrightarrow{14d} \mathsf{delete}$$

The provision in (2) is modelled using a milestone relation.

$$\mathsf{delete} \diamond\!\leftarrow \mathsf{archive}$$

Recall that a milestones means (in this case) that as long as we have an unfulfilled obligation to execute archive, we cannot execute delete.

Next, we model the retention requirement in (3) that records must be archived by an (unbounded) response

$$\mathsf{release} \bullet\!\rightarrow \mathsf{archive}$$

and we model the timed provision that they should remain for 8 years after archival by a condition relation with a time constraint.

$$\mathsf{unarchive} \bullet\!\xleftarrow{8y} \mathsf{archive}$$

Note, this rule says that 8 years must pass before archived records can be deleted, not that they must in fact be deleted.

If the patient is re-admitted, we must remove the obligation to delete records (4). We model this using exclusion.

$$\mathsf{readmit} \rightarrow\!\!\!\!\% \ \mathsf{delete}$$

Once the patient is (re-)released, we must reinstate the obligation to delete records. We model this using inclusion.

$$\mathsf{release} \rightarrow\!\!+ \ \mathsf{delete}$$

Putting these rules together and re-arranging them for clarity, we obtain the following set of rules:

$$\begin{aligned} T_0 = \ & \mathsf{release} \bullet\!\!\xrightarrow{14d} \mathsf{delete} \mid \mathsf{release} \bullet\!\!\rightarrow \mathsf{archive} \\ & \mid \mathsf{release} \rightarrow\!\!+ \ \mathsf{delete} \mid \mathsf{delete} \diamond\!\!\leftarrow \mathsf{archive} \\ & \mid \mathsf{readmit} \rightarrow\!\!\!\!\% \ \mathsf{delete} \\ & \mid \mathsf{unarchive} \bullet\!\!\xleftarrow{8y} \mathsf{archive}\,. \end{aligned} \tag{12}$$

As for markings, we assume that initially, the patient is already admitted. Although the textual description of the model above does not say so explicitly, it stands to reason that it should not be possible to delete patient records prior to release. Hence, for our initial marking, we leave that event excluded:

$$\begin{aligned} M_0 = \ & \mathsf{release} : (\bot, \top, \bot), \\ & \mathsf{delete} : (\bot, \bot, \bot), \mathsf{archive} : (\bot, \top, \bot), \\ & \mathsf{unarchive} : (\bot, \top, \bot), \mathsf{readmit} : (\bot, \top, \bot)\,. \end{aligned}$$

Altogether, our model is $P_0 = [M_0]\ T_0$.

B. Example runs

We consider a few runs of this model, presenting them as tables. Each row starts with the event leading to the current marking, which is represented by the rest of the row.

The common case: provisions and obligations. Refer to Figure 5. The first row describes the initial marking M_0. No event is previously executed as all h (happened) columns are $\bot$. All events but delete are included as can be seen from the i (included) columns. No events are pending since all r columns are $\bot$. We walk through the sequence of events.

1) release is executed. The age of release is set to "just executed" ($h = 0$). Looking at our constraints T_0 in (12), we see that executing release imposes obligations to delete within 14 days and to eventually archive; these obligations are reflected in the marking by the change to $r = 14d$ for the former and $r = \omega$ for the latter. Moreover, T_0 also stipulates that delete is now included, which is reflected in the marking by the change to $i = \top$.
2) $4d$ passes. The age of release and the deadline for delete are respectively incremented and decremented by 4 days. Note that the "eventual" deadline ω for archive remains as $\omega - 4d = \omega$.
3) archive is executed. This meets the deadline of archive, which is then cleared.
4) $1d$ passes. The age and deadlines are updated.
5) delete is executed. The deadline for delete is cleared. Note that unarchive still cannot execute as it is conditional on archive having happened at least $8y$ in the past.
6) $10y$ passes. The condition unarchive $\bullet\!\!\xleftarrow{8y}$ archive is now fulfilled, hence unarchive is now executable.
7) unarchive is executed.

An attempted violation. Will records be deleted? Consider the run in Figure 6. The event delete has deadline 0 at the end of the run and is still included. That means that the minimum response deadline, $\mathsf{deadline}(M)$ is now 0, and so the tick-transition cannot fire, that is, time cannot advance. (Obviously, an enforcement mechanism cannot rely on stopping time; rather, it must take care to avoid finding itself in a situation where advancing time would violate the policy. We will return to this in Section V-A.) Hence, the model prevents this potential policy violation by refusing to let time pass in this state. To allow time to pass again, we must either re-admit the patient (thereby excluding the pending delete event), releasing the patient again (thereby extending the deadline of delete by another 14 days), or delete the records. Neither re-admitting nor releasing the patient again make sense as a general way to enforce the policy. And we cannot delete the records straight away since, because of the milestone delete $\diamond\!\!\leftarrow$ archive and the fact that archive is pending, we must first archive the records—see Figure 7.

As we will see in Section V, this policy is enforceable only if either the event readmit, the event release, or both the events archive and delete can be executed by the PEP. Also note that if, for instance, only archive but not delete can be controlled by the PEP, then one can consider policy redesign. For instance, one can introduce an event notify (notifying the IT Department's Data Retention Officer of undeleted data) that is controllable by the PEP, is enabled when the deadline of delete is met, and cancels the delete event by excluding it.

$$\mathsf{release} \bullet\!\!\xrightarrow{14d} \mathsf{notify} \mid \mathsf{notify} \bullet\!\!\xleftarrow{14d} \mathsf{release} \mid \mathsf{notify} \rightarrow\!\!\!\!\% \ \mathsf{delete}\,.$$

This is an instance of a general *escalation* pattern for dealing with uncontrollable events with deadlines. It is suggestive of the way obligations are often traditionally treated: rather than preventing the original policy from being violated, the enforcement mechanism triggers remedial actions such as notifications or logging information for subsequent audits.

Re-admission—dynamic exclusion of obligations. Figure 8 presents the case where a patient is readmitted. The initial markings in the first two events follow the first example: the patient is release'ed, and four days pass (row 4d). The patient is readmit'ted. Following the rules T_0, readmit excludes the delete event ($i = \bot$). Ten days pass (row 10d). Notice that the deadline for delete reaches zero. However, this zero-deadline *does not* prevent time from progressing further: because delete is excluded, this zero does not contribute to the computation of the minimum deadline $\mathsf{deadline}(M)$. So another four days pass (row 4d). By the definition of $\mathsf{tick}\langle - \rangle$, the deadline for

Event	release			delete			archive			unarchive			readmit		
	h	i	r	h	i	r	h	i	r	h	i	r	h	i	r
–	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
release	0	$\top$	$\bot$	$\bot$	$\top$	14d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
4d	4d	$\top$	$\bot$	$\bot$	$\top$	10d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
archive	4d	$\top$	$\bot$	$\bot$	$\top$	10d	0	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
1d	5d	$\top$	$\bot$	$\bot$	$\top$	9d	1d	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
delete	5d	$\top$	$\bot$	0	$\top$	$\bot$	1d	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
10y	10y5d	$\top$	$\bot$	10y	$\top$	$\bot$	10y1d	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
unarchive	10y5d	$\top$	$\bot$	10y	$\top$	$\bot$	10y1d	$\top$	$\bot$	0	$\top$	$\bot$	$\bot$	$\top$	$\bot$

Fig. 5. "Common case" run.

Event	release			delete			archive			unarchive			readmit		
	h	i	r	h	i	r	h	i	r	h	i	r	h	i	r
–	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
release	0	$\top$	$\bot$	$\bot$	$\top$	14d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
14d	14d	$\top$	$\bot$	$\bot$	$\top$	0	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$

Fig. 6. "Attempted violation" run.

archive	14d	$\top$	$\bot$	$\bot$	$\top$	0	0	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
delete	14d	$\top$	$\bot$	0	$\top$	$\bot$	0	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$

Fig. 7. Continuation of the "attempted violation" run.

Event	release			delete			archive			unarchive			readmit		
	h	i	r	h	i	r	h	i	r	h	i	r	h	i	r
–	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
release	0	$\top$	$\bot$	$\bot$	$\top$	14d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
4d	4d	$\top$	$\bot$	$\bot$	$\top$	10d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
readmit	4d	$\top$	$\bot$	$\bot$	$\bot$	10d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	0	$\top$	$\bot$
10d	14d	$\top$	$\bot$	$\bot$	$\bot$	0	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	10d	$\top$	$\bot$
4d	18d	$\top$	$\bot$	$\bot$	$\bot$	0	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	14d	$\top$	$\bot$
release	18d	$\top$	$\bot$	$\bot$	$\top$	14d	$\bot$	$\top$	ω	$\bot$	$\top$	$\bot$	14d	$\top$	$\bot$

Fig. 8. "Dynamic exclusion of obligations" run.

the excluded delete event cannot become negative and thus remains zero. When the patient is finally again release'ed, release has its usual effects: it includes delete, and *resets its deadline to* $14d$. Note the semantics: an event can have only one response deadline; setting a new response deadline cancels the previous one.

V. Proactive Enforcement of DCR Policies

We now consider what is necessary for a DCR policy to be proactively enforceable. That is, under what circumstances does an enforcement mechanism exist for the policy?

A. Enforceability and Time-locks

It might happen that the deadline for meeting an obligation has arrived, but the action required to meet the obligation would violate the (overall) security policy.

Example 18: Suppose we were to add the constraints

$$\mathsf{early} \bullet\!\!\xrightarrow{1y} \mathsf{unarchive} \mid \mathsf{archive} \rightarrow\!\!\!\!\!\!\% \; \mathsf{early}$$

to the data retention policy T_0 given previously in (12). These new constraints model that the patient can request that his records are unarchived early, no later than a year after the request, but this request can only be made before archival. These new constraints would contradict the old constraint that unarchival cannot happen before 8 years after archival:

	release			unarchive			archive		
	h	i	r	h	i	r	h	i	r
–	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
release	0	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	ω
early	0	$\top$	$\bot$	$\bot$	$\top$	1y	$\bot$	$\top$	ω
archive	0	$\top$	$\bot$	$\bot$	$\top$	1y	0	$\top$	$\bot$
1y	1y	$\top$	$\bot$	$\bot$	$\top$	0	1y	$\top$	$\bot$

In the last state, the event unarchive must happen now, but it cannot due to the required delay of 8 years since archive. The deadline would be extended by re-executing early, but early is excluded. This process is not deadlocked: release can still happen. It is, however, *time-locked*: it has reached a marking

from which time cannot possibly advance.

Definition 19 ([16]): A process $[M]\ T$ is *time-locked* iff no marking M' reachable from M has $T \vdash M' \xrightarrow{\mathsf{tick}} M''$.

We conclude that a DCR policy is *not* necessarily enforceable, even if all events are controllable and causable: If there is a time-lock in a policy, then there are target systems for which no correct transparent enforcement mechanism exists, since one cannot prevent time from advancing.

Theorem 20: Let $\mathcal{P}$ be the language of a DCR process P that has a time-lock. Assume that $\mathcal{P}$ non-empty and take as target system $(\mathsf{prefixes}(\mathcal{P}), \mathsf{dom}(P) \cup \{\mathsf{tick}\}, \mathsf{dom}(P), \mathsf{dom}(P))$. Then no transparent enforcement mechanism m is correct for $\mathcal{P}$.

We can decide whether a DCR policy has a time-lock, using that its LTS is extended bisimilar [14] to a finite one (see the Appendix).

Proposition 21: It is decidable whether a DCR process P is time-lock free.

But decidable is not the same as feasible:

Definition 22: Let $P = [M_1]\ T$ be a DCR process, and let e be an event of P. We say that e *is eventually executable in* P iff there is a transition sequence $M_1 \xrightarrow{e_1} M_2 \xrightarrow{e_2} \cdots \xrightarrow{e_n} M_{n+1} \xrightarrow{e} N$.

Lemma 23: There exists an NLOGSPACE-reduction from boolean satisfiability to eventual executability for time-lock–free DCR processes.

Theorem 24: (a) Deciding eventual executability of any e in any DCR process P is NP-hard; and (b) deciding time-lock–freedom for DCR processes is NP-hard.

B. Avoiding Time-locks

We have just seen that the existence of an enforcement mechanism for a DCR policy depends at least on the policy's time-lock–freedom. We now give a polynomial-time computable sufficient condition for a DCR policy to be time-lock free, which also supports the effective computation of a sequence of events that averts violating a given deadline. We call such a DCR policy "resolvable". In Section V-C, we use this notion to define a correct, transparent enforcement mechanism for resolvable DCR policies.

Definition 25 (Resolvability): Let $P = [M]\ T$ be a DCR process and $S \subseteq \mathsf{dom}(M)$ a subset of events. We say that P is *S-resolvable* iff for any marking M' reachable from M with $\mathsf{deadline}(M') = 0$, there exists a $\{e_1, \ldots, e_n\} \subseteq S$ such that (1) the following is a transition sequence:

$$M' \xrightarrow{e_1} M'_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} M'_n$$

and (2) $\mathsf{deadline}(M'_n) > 0$. Assume a particular choice of such a sequence and define $\mathsf{resolve}([M']\ T)$ to be the (partial) function that exhibits this choice.

That is, in any reachable state with one or more deadlines about to be missed, one may execute some sequence of the events in the set S to avoid missing those deadlines.

Lemma 26: A DCR process P is time-lock–free iff it is S-resolvable for some S.

Example 27: It is easily verified that the policy T_0 given in Section IV is S-resolvable for $S = \{\mathsf{release}\}$, or $S = \{\mathsf{readmit}\}$, or $S = \{\mathsf{delete}, \mathsf{archive}\}$.

Note that a DCR process P can be time-lock–free and yet have no way to accept, e.g., $[e : (\bot, \top, \omega)]\ e \bullet\!\!\leftarrow^{0} e$. This lemma has the subtle consequence that resolvability does *not* entail that any trace of P can be extended to an accepting one.

From Proposition 21 and Theorem 24, we can bound the complexity of determining resolvability, for some S.

Proposition 28: Let P be a DCR process and let $S \subseteq \mathsf{dom}(M)$. It is decidable, but NP-hard, whether a P is S-resolvable.

Given NP-hardness, we need tractable approximations to resolvability. We begin by considering which events may reach deadlines. We call these *busy* events.

Definition 29: An event e is *busy* in $[M]\ T$ iff there exists a reachable marking M' with $e : (i, \top, k) \in M'$ for some $k \neq \bot$ and some i.

Note that as defined, busy events may have the indefinite deadline ω. We can identify busy events in time polynomial in the size of the DCR process:

Lemma 30: An event e is busy in $[M]\ T$ only if for some $k \neq \bot$ either $e : (i, \top, k) \in M$ for some i or $f \bullet\!\!\rightarrow^{k} e \in T$ for some f.

The only way to violate an obligation is to neither execute nor exclude a busy event when its deadline comes up. Clearly, either (1) the busy event must be under the enforcement mechanism's control or (2) an alternative event that excludes the busy event must be under the enforcement mechanism's control. In the following, we will focus just on the first option. To this end, we define *dependable* events.

Definition 31: An event e is *dependable* in P iff in any P' reachable from P, the event e is either excluded or enabled.

The following proposition follows immediately.

Proposition 32: Let P be a DCR process, let B be the set of busy events of P, and suppose every event in B is dependable. Then P is B-resolvable.

This proposition gives us an approximation for resolvability that can be computed in time polynomial in P, since we can approximate the busy events using Lemma 30 and dependable events with those that have no conditions or milestones. In practice, however, it is unlikely that every busy event is dependable. For example, in the example of Section IV, delete is busy, but not dependable: it has a milestone from archive, and so may be included but not enabled. We will need to consider the dependencies of busy events.

Definition 33: Let $P = [M]\ T$, and let e be an event of P. The *inhibitors* of e is the set

$$I(e) = \{f \mid \exists k.e \bullet\!\!\leftarrow^{k} f \vee e \diamond\!\!\leftarrow f\}\,.$$

Let the *inhibition graph* $I(P)$ of P be the directed graph that has nodes $\mathsf{dom}(P)$ (P's events) and an edge from f to e iff $f \in I(e)$. For a subset of events X, define the *inhibition subgraph for* X as the subgraph of $I(P)$ comprising every event f with a path to some $e \in X$ in $I(P)$ and every edge

on those paths. We denote this graph $\hat{I}_P(X)$ and its set of nodes $I_P(X)$, dropping the subscript P when it is clear from context. We call the set $I(X)$ the *inhibition closure* of X.

The inhibitors of an event e is (the transitive closure of) those events that may cause e to be not executable, because they are milestones or conditions for e.

Definition 34: Let X be a set of events of a DCR process $[M]\ T$. We say that X is *dependable* iff the following three conditions hold.

1) The inhibition graph $\hat{I}(X)$ is acyclic.
2) For any two nodes e, f in $\hat{I}(X)$, if there is a relation $e \bullet\!\!\xrightarrow{k} f$ for some k or a relation $e \rightarrow\!+\ f$, then there is a path from e to f.
3) For any two nodes e, f in $\hat{I}(X)$, if there is a relation $f \bullet\!\!\xleftarrow{k} e$ for some k, then $k = 0$.

Intuitively, a set of events is dependable if, when we build the graph of those events' dependencies (milestones and conditions), the dependencies form a directed acyclic graph (1), and we can eventually execute any event in the set by simply executing the events in that acyclic graph in a topological order, i.e., "bottom-up".

Care must be taken because of the interplay between responses and milestones. We must avoid the situation where resolving a dependency by executing it re-blocks a previously resolved dependency by either reinstating a milestone or a condition (2). Similarly, we must ensure that no event in the set depends any other event in the set with a delay (3).

Theorem 35 (Approximation of Resolvability): Let P have busy events B. If B is dependable, then P is $I(B)$-resolvable. Moreover, there exists a polynomial time algorithm computing $\mathsf{resolve}$ of Definition 25 for every P' reachable from P.

The actual algorithm is simply this: Given a subset $X \subseteq B$, $\mathsf{resolve}(B)$ is the topological sort of $\hat{I}(X)$.

Example 36: We return to our data retention example P_0 of Equation (12) on page 8. Inspecting the relations, we see that the inhibition graph has only the two edges:

$$\mathsf{archive} \mapsto \mathsf{unarchive} \qquad \mathsf{archive} \mapsto \mathsf{delete}\,.$$

The busy events of P_0 are over-approximated by Lemma 30 to $B = \{\mathsf{delete}, \mathsf{archive}\}$. The inhibition subgraph and inhibition closure of B are then simply $\hat{I}_{P_0}(B) = \mathsf{archive} \mapsto \mathsf{delete}$ and $I_{P_0}(B) = \{\mathsf{archive}, \mathsf{delete}\}$. It is straightforward to verify that $I_{P_0}(B)$ is dependable. By Theorem 35, it is thus decidable in polynomial time that P_0 is indeed $\{\mathsf{archive}, \mathsf{delete}\}$-resolvable. Summing up, if we have the ability to execute archive and delete we can *guarantee* compliance for obligations.

C. Proactive DCR Policy Enforcement

With the mechanics of statically avoiding time-locks in DCR policies in place, we now define an enforcement mechanism, from which we derive a prototype implementation of a PEP. First note that a DCR process directly defines a policy language in the sense of Definition 2. We define an enforcement mechanism for a process P.

Definition 37: Define a function $m(\cdot,\cdot)$ that, for a given DCR policy P with $\mathsf{dom}(P) \subseteq \Sigma \backslash \{\mathsf{tick}\}$ and action $a \in \Sigma$, says what actions to take.

$$m(P,a) = \begin{cases} \mathsf{resolve}(P) \cdot \mathsf{tick} & (i) \\ \qquad \text{when } a = \mathsf{tick} \wedge \mathsf{deadline}(P) = 0 & \\ \epsilon \quad \text{when } a \in \mathsf{dom}(P) \text{ and} & (ii) \\ \qquad a \text{ is } \textit{not} \text{ executable in } P & \\ a \quad \text{otherwise} & (iii) \end{cases}$$

We lift m to finite sequences of actions. We must advance the "current" DCR policy whenever the target system takes an action. Let $a\langle P\rangle = P'$ iff $P \xrightarrow{a} P'$. This is well-defined because the transition semantics of DCR processes in Figure 4 assigns at most one transition for any a, P and each event has a unique label. Moreover, define $\mathsf{advance}(P, \epsilon) = P$ and $\mathsf{advance}(P, a \cdot w) = \mathsf{advance}(a\langle P\rangle, w)$, for $w \in \Sigma^*$. Then the *DCR enforcement mechanism* m_P is given inductively by

$$\begin{aligned} m_P(\epsilon) &= \epsilon \\ m_P(w \cdot a) &= m_P(w) \cdot m(\mathsf{advance}(P, m_P(w)), a)\ . \end{aligned}$$

Note that m_P is not defined for all P, since $\mathsf{resolve}$ (Definition 25) is only defined on resolvable DCR processes.

Theorem 38: Let $t = (\mathcal{S}, \Sigma, \Gamma, \Delta)$ be a target system, and let P be a DCR process with $\mathsf{dom}(P) = \Sigma$ and dependable busy events $B \subseteq \Delta$, and assume every event $\Sigma \setminus \Gamma$ is enabled in any P' reachable from P. Then m_P is a correct, transparent enforcement mechanism.

From Lemma 8, we know then that m_P steers clear of the finite violations language:

Corollary 39: Let P and m_P be defined as in the Theorem 38. Then for all $w \in \mathcal{D}_m$ we have $w \notin \bar{\mathcal{P}}$.

To apply DCR policy enforcement in practice, we require:

1) A target system $(\mathcal{S}, \Sigma, \Gamma, \Delta)$,
2) A DCR policy P over Σ such that P is Δ-resolvable, and every P' reachable from P has every $e \in \Sigma \setminus \Gamma$ enabled, and
3) A PEP implementing m_P.

For (1), note that the causable actions Δ must be enabled in the target system whenever they are to be caused according to the policy P: otherwise the PEP cannot rely on executing them to avert deadlines. In practice, this either requires (some) white-box knowledge of the target system, for example, the knowledge that these actions are always available when needed or that the actions are always enabled, e.g. the actions are provided as a web service.

For (2), apply Theorem 35 to get resolvability. We do not address here the means of ensuring that uncontrollable events, i.e., events in $\Sigma \setminus \Gamma$, are always enabled. Several options exists along the lines of Theorem 35. A very pragmatic approach is simply to require the events $\Sigma \setminus \Gamma$ to be completely unconstrained in P. Alternatively, it suffices for the set $\Sigma \setminus \Gamma$ to be empty. This will be the case when the target system must ask the PEP for permission on all (policy-relevant) actions and is analogous to the principle of complete mediation for access control policies.

```
// A PEP takes as input an Observation which is
// either an (attempted) transition, or a deadline
// approaching.
type Observation =
  | Transition of DCR.event
  | Deadline   of DCR.event
  | Inform     of DCR.event

// A PEP produces as output a Reaction. (Code for
// acting on the Reaction is not included here.)
type Reaction =
  | Grant
  | Deny
  | Cause of DCR.event list
  | Ignore

// DCR-PEP. Takes a current DCR policy-state P and
// an Observation, and produces a Reaction and a
// new DCR policy-state.
let PEP P observation =
    match observation with
    | Deadline e ->
        Cause (resolve P e), P            // (i)
    | Transition e ->
        if (DCR.is_executable P e) then
          Grant, DCR.execute e P          // (iii)
        else
          Deny, P                         // (ii)
    | Inform e ->
         Ignore, DCR.execute e P          // (iii)
```

Fig. 9. F# implementation of m_P.

For (3), we have constructed a prototype PEP implementing m_p, built on top of the DCR process engine of [7]. This prototype is available on-line at http://dcr.itu.dk/obligations.

Our prototype PEP repeatedly waits to either be consulted about an action by the target system, or for a deadline to come close, in either case acting as m of Definition 37: For actions, it grants or denies depending on whether the corresponding event is executable in the current DCR policy state. When actions are granted, the state is advanced by executing the event in it. For deadlines, the PEP instructs the target system to issue actions as given by resolve. Since we have assumed that non-controllable events $\Sigma \setminus \Gamma$ are always enabled, they need no special treatment.

We give the actual F# code implementing m in Figure 9. Note the comments connecting various branches to cases of Definition 37. Note too the use of the resolve algorithm provided of Definition 25, guaranteed to exist by Theorem 35. Our prototype uses the policy $\mathcal{P}$ also as the target system; this is enough for experimentation and simulation in the DCR Workbench. To lift the prototype to an actual enforcement mechanism, it suffices to update the driver function producing the `Observation` inputs and executing the `Reaction` outputs of Figure 9 to one that interacts, say, via REST, with an actual system, as dictated by the system model of Section II.

VI. Related Work

In this section, we compare our approach with related policy specification languages for handling obligations and with alternative enforcement mechanisms.

A. Specification

The idea of adding obligations to policies was first proposed by Minsky and Lockman [26]. They augmented permissions with tasks to be executed within a stated deadline after an access request is granted. This deadline corresponds, for example, to the passage of time or is triggered by other events. Park and Sandhu examined obligations in the context of usage control [28] and Bettini et al. [4] systematically considered the combination of provisions and obligations within a datalog formalism. In both cases, the emphasis is on policy specification (and, in the case of [4], also analysis), rather than applications to enforcement and monitoring.

Obligations have also been used in languages such as PONDER [6] and XACML [34]. Obligations there are triggered by events: an event's occurrence directly results in actions to discharge the obligation. This is in contrast to our work where obligations are associated with events that become pending but need not be immediately discharged.

Temporal logics are, of course, well suited for specifying properties based on the past (provisions) and future (obligations). For example, linear-time temporal logics (LTL) have been used in [3], [12], [18], [35] to formalise regulations with obligations and usage-control policies. LTL formulas are closely related to DCR processes: Core (untimed) DCR processes are equivalent in their expressivity to ω-regular languages [8], whereas (propositional) LTL is equivalent to a subset of ω-regular languages, namely the star-free languages [33]. DCR processes also have the advantage that their operational semantics does not depend on the possibly exponential translation to (e.g. Büchi) automata, which is most often employed for LTL; see, for example, the use of LTL and Büchi automata to formalise (untimed) obligations in [9].

B. Enforcement

Obligations are fundamentally more difficult to enforce than provisions. For provisions, policy monitoring and enforcement are equivalent. Any enforceable policy can be monitored as the enforcement mechanism's denial of an action signifies a policy violation when that action takes place. Conversely, a monitor can serve as a policy decision point for an enforcement mechanism. When policies contain obligations, there are monitorable policies that cannot be enforced, in particular when one distinguishes, as we have done (see also [20]), between actions that can be controlled by the enforcement mechanism versus those that can only be observed [1].

Our solution is for the (standard) setting where one interacts with the target system as a black box, which one can neither examine nor modify. Even with these restrictions, enforceability is a rich concept that depends on the underlying mechanism used [11], [13], for example, whether the mechanism can only suppress actions, like security automata [31], or can initiate or change actions as with edit automata [23], [24] or mandatory results automata (MRA) [25].

As with our approach, edit automata and MRA share the ability to proactively change behaviour to enforce policies.

They differ though in that neither formalism handles real-time constraints, which is an essential aspect of obligations. Their system models are quite different too. Our system model essentially extends that of security automata with additional target-system interfaces of controllable and causable actions. The MRA system model interposes the enforcement mechanism between the target system and an execution platform. For edit automata, no distinction is made between observable and controllable events and hence it remains open what an edit automaton can actually control on the target system.

A number of enforcement mechanisms have been proposed for obligations, like those for PONDER and XACML, which do not involve any sophisticated calculations. They enforce obligations simply by executing the appropriate actions (such as logging information) when the obligation to do so arises. Li et al. [22] provide operational extensions to XACML enforcement such as support for pre-obligations and on-going obligations, as introduced in the UCON model [28]—see also Elrakaiby et al. [10] who study enforcement of UCON-style obligations in the context of active databases.

Finally, there is related work in other communities. For example, in supervisory-control theory for discrete event systems [29], a supervisor process (in our work, enforcement mechanism) attempts to control the behaviour of a generator process (in our work, the target system) over its controllable events, which corresponds to our mechanism without the ability to force actions. Subsequent extensions of supervisory-control theory to timed discrete event systems [5], [32] include delays and deadlines of each event and a designated subset of so-called *forcible* events. The supervisor process can now disable the tick (time) actions in case a forcible event is enabled. But the target system may choose to perform a non-forcible event; indeed the supervisor cannot enforce that a forcible event happens as in our approach. Also closely related is the research of Renard et al. [30] who present an automata-based framework for enforcing regular timed properties with uncontrollable events. Their enforcement mechanisms work by buffering and delaying input events so that the output satisfies a specified policy. Again, in contrast, our mechanisms do not buffer and delay, but actively schedule controllable events to prevent obligations' violations.

VII. Conclusions and Future Work

Obligations are a central notion found in a wide range of security policies such as those for usage control, data protection and privacy, and digital rights management. We have investigated their specification and enforcement in a black-box setting where the enforcement mechanism proactively computes and schedules events needed to prevent policy violations. As not all policies in this setting are enforceable, we have also established complexity results for enforceability and given a sufficient polynomial-time verifiable condition for a policy to be enforceable.

The present work investigates avoiding the violation of obligations by proactively taking action. We have assumed that the sequence of actions caused proactively can always be executed within a single time-step. If time is measured in hours or days this is usually reasonable, but for finer granularities of time or if resolving an obligation requires delays, one needs to support tick actions *during* enforcement. This support would require generalising Definition 34 of dependable sets.

The approach of this paper is not limited to obligations: one can reasonably consider proactively avoiding the violations of provisions too. For example, suppose we have the provision "action a can happen only if b previously happened", and suppose the string w contains no occurrences of b. The present approach would suppress a, i.e., $m(w \cdot a) = m(w)$. However, we might instead *cause* the action b to happen before allowing a: $m(w \cdot a) = m(w) \cdot b \cdot a$.

Finally, with respect to the usefulness of proactive enforcement in practice, the proof of the pudding is in the eating. We are presently engaged in a large-scale case study of policy enforcement with a major European railway service. While it seems that our policy language is well-suited for capturing temporal constraints between actions, the formalism presented here does not handle actions depending on data. An extension to handle data is currently underway.

References

[1] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security*, 16(1), 2013.

[2] D. Basin, F. Klaedtke, and S. Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, pages 23–33, New York, NY, USA, 2010. ACM Press.

[3] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, May 2015.

[4] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *J. Network and System Mgmt.*, 11(3):351–372, 2003.

[5] B. Brandin and W. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39:329–342, Feb 1994.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.

[7] S. Debois, T. T. Hildebrandt, and T. Slaats. Concurrency and asynchrony in declarative workflows. In *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lect. Notes Comput. Sci.*, pages 72–89. Springer, 2015.

[8] S. Debois, T. T. Hildebrandt, and T. Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 143–160, 2015.

[9] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Obligations and their interaction with programs. In J. Biskup and J. Lopez, editors, *12th European Symposium On Research In Computer Security (ESORICS)*, volume 4734 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 2007.

[10] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia. Formal enforcement and management of obligation policies. *Data & Knowledge Engineering*, 71(1):127 – 147, 2012.

[11] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382, 2012.

[12] C. Giblin, A. Y. Liu, S. Müller, B. Pfitzmann, and X. Zhou. Regulations expressed as logical models (REALM). In *Proceedings of the 18th Annual Conference on Legal Knowledge and Information Systems*, volume 134 of *Frontiers Artificial Intelligence Appl.*, pages 37–48, Amsterdam, The Netherlands, 2005. IOS Press.

[13] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Progr. Lang. Syst.*, 28(1):175–205, 2006.
[14] M. Hennessy and C. Stirling. The power of the future perfect in program logics. *Information and Control*, 67(1):23 – 52, 1985.
[15] T. T. Hildebrandt and R. R. Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In *3rd Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES)*, volume 69 of *EPTCS*, pages 59–73, 2010.
[16] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, and F. Zanitti. Contracts for cross-organizational workflows as timed dynamic condition response graphs. *J. Log. Algebr. Program.*, 82(5-7):164–185, 2013.
[17] M. Hilty, D. Basin, and A. Pretschner. On obligations. In *Proc. ESORICS*, Springer LNCS 3679, pages 98–117, 2005.
[18] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A Policy Language for Distributed Usage Control. In *Proc. ESORICS*, pages 531–546, 2007.
[19] The health insurance portability and accountability act of 1996 (hipaa). Public Law, 1996.
[20] B. Katt, X. Zhang, R. Breu, M. Hafner, and J.-P. Seifert. A general obligation model and continuity: Enhanced policy enforcement engine for usage control. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 123–132, New York, NY, USA, 2008. ACM.
[21] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, 2001.
[22] N. Li, H. Chen, and E. Bertino. On practical specification and enforcement of obligations. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 71–82, New York, NY, USA, 2012. ACM.
[23] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Secur.*, 4(1–2):2–16, 2005.
[24] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inform. Syst. Secur.*, 12(3), 2009.
[25] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proceedings of the 15th European Symposium on Research in Computer Security*, volume 6345 of *Lect. Notes Comput. Sci.*, pages 87–100, Heidelberg, Germany, 2010. Springer.
[26] N. H. Minsky and A. D. Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, pages 92–102, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.
[27] J. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel Distributed Systems*, 1(2):170–183, April 1990.
[28] J. Park and R. Sandhu. The UCON ABC Usage Control Model. *ACM Transactions on Information and Systems Security*, 7:128–174, 2004.
[29] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
[30] M. Renard, Y. Falcone, A. Rollet, S. Pinisetty, T. Jéron, and H. Marchand. Enforcement of (timed) properties with uncontrollable events. In M. Leucker, C. Rueda, and F. D. Valencia, editors, *Theoretical Aspects of Computing, 12th International Colloquium*, volume 9399, pages 542–560. Springer, 2015.
[31] F. B. Schneider. Enforceable security policies. *ACM Trans. Inform. Syst. Secur.*, 3(1):30–50, 2000.
[32] J. Thistle. Supervisory control of discrete event systems. *Mathematical and Computer Modelling*, 23(1112):25 – 53, 1996.
[33] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. MIT Press, 1990.
[34] *eXtensible Access Control Markup Language (XACML) V. 3.0 OASIS Standard*.
[35] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Trans. Inform. Syst. Secur.*, 8(4):351–387, 2005.

APPENDIX

A. Proofs

Here we provide the remaining proofs for all lemmas and theorems in the paper.

Proof of Lemma 8: Let $v \in \mathcal{D}_m$. Observe that from Definition 5, $m(v)$ is finite. It follows by the definition of correctness that $m(v) \in \mathsf{prefixes}(\mathcal{P})$; but then $m(v) \notin \bar{\mathcal{P}}$. □

Proof of Theorem 20: Suppose m is correct and transparent. By the definition of a time-lock, there exists a $w \in \mathsf{prefixes}(\mathcal{P})$ such that w is a shortest sequence exhibiting a time-lock in $\mathsf{lts}(P)$. Because m is transparent, $w \in \mathcal{D}_m$ and $m(w) = w$. Because w exhibits a time-lock, w is not accepting and so $w \notin \mathcal{P}$. Let $v \in \mathsf{dom}(P)^*$. By the definition of a time-lock, $w \cdot v$ is also not accepting. But then m is not correct, which is a contradiction. □

Proof of Proposition 21: Immediate from Proposition 43. □

Proof of Lemma 23: Let B be a boolean satisfiability problem over atoms, conjunction and negation, that is, B is a string generated by the language

$$S ::= x \mid S \wedge S \mid \neg S\,.$$

Construct a DCR process $[\![B]\!]$ which has, for each non-leaf node n of the abstract syntax tree of B, events $n^{\mathsf{t}}, n^{\mathsf{f}}$ and $n^{\mathsf{b1}}, n^{\mathsf{b2}}, n^{\mathsf{b3}}$; and for each atom a nodes $a^{\mathsf{t}}, a^{\mathsf{f}}$ and $a^{\mathsf{b1}}, a^{\mathsf{b2}}$. For each non-leaf node n, add relations as follows.

1) For each atom a, add relations $a^{\mathsf{b1}} \bullet\!\!\leftarrow a^{\mathsf{b1}}$ and $a^{\mathsf{b2}} \bullet\!\!\leftarrow a^{\mathsf{b2}}$; and $a^{\mathsf{t}} \bullet\!\!\leftarrow a^{\mathsf{b1}}$ and $a^{\mathsf{f}} \bullet\!\!\leftarrow a^{\mathsf{b2}}$ and $a^{\mathsf{t}} \rightarrow\!+\, a^{\mathsf{b2}}$ and $a^{\mathsf{f}} \rightarrow\!+\, a^{\mathsf{b1}}$.
2) For each non-leaf node n, add relations $n^{\mathsf{b1}} \bullet\!\!\leftarrow n^{\mathsf{b1}}, n^{\mathsf{b2}} \bullet\!\!\leftarrow n^{\mathsf{b2}}, n^{\mathsf{b3}} \bullet\!\!\leftarrow n^{\mathsf{b3}}$.
3) For each non-leaf node $n = u \wedge v$ add relations $n^{\mathsf{t}} \bullet\!\!\leftarrow n^{\mathsf{b1}}$, $n^{\mathsf{t}} \bullet\!\!\leftarrow n^{\mathsf{b2}}$ and $n^{\mathsf{f}} \bullet\!\!\leftarrow n^{\mathsf{b3}}$; and $u^{\mathsf{t}} \rightarrow\!\% \, n^{\mathsf{b1}}$, $v^{\mathsf{t}} \rightarrow\!\% \, n^{\mathsf{b2}}$; and $u^{\mathsf{f}} \rightarrow\!\% \, n^{\mathsf{b3}}$, $v^{\mathsf{f}} \rightarrow\!\% \, n^{\mathsf{b3}}$.
4) For each non-leaf node $n = \neg u$ add relations $n^{\mathsf{t}} \bullet\!\!\leftarrow u^{\mathsf{f}}$ and $n^{\mathsf{f}} \bullet\!\!\leftarrow u^{\mathsf{t}}$.

Define a marking M_0 where every event is $(\bot, \top, \bot)$, except a^{b1} and a^{b2} which must be $(\bot, \bot, \bot)$. Consider a run of $[\![B]\!]$ executing a maximal number of distinct events of $[\![B]\!]$. Note that for each atom a; either a^{t} or a^{f} has been executed, but not both; define from this an assignment $a \mapsto \top$ or $a \mapsto \bot$. By induction, each non-leaf node n has n^{t} executed iff this assignment evaluates n true and n^{f} executed iff it evaluates n false. But then for the root node r of B, r^{t} is executed in some run iff B is satisfiable and r^{f} is executed in some run iff it is not; i.e., r^{t} is eventually executable in $[\![B]\!]$ iff B is satisfiable. The DCR process $[\![B]\!]$ has $O(|B|)$ nodes and relations, and so this reduction is in NLOGSPACE. By inspection of the reduction, we see that $[\![B]\!]$ contains no responses or initially pending events; it is then trivially time-lock–free. □

Proof of Theorem 24: (a) is immediate from Lemma 23. For (b), let B be a boolean satisfiability problem, suppose $([\![B]\!], e)$ is the corresponding eventual execution problem, and suppose $[\![B]\!] = [M]\ T$. Choose $f \notin \mathsf{dom}(M)$. Then $[f : (\bot, \top, 0), M]\ T \mid f \bullet\!\!\stackrel{0}{\leftarrow} e$ is time-lock free iff f is eventually executable iff B has a satisfying assignment. □

Proof of Lemma 26: Suppose P is not S-resolvable for any S. By definition, there exists then a state from which time cannot advance after any finite sequence of events, and so P

has a time-lock. Suppose instead P is S-resolvable for some S, and suppose for a contradiction that P' is reachable from P and is time-locked. But then $\mathsf{resolve}(P')$ is defined and leads to a state in which time has advanced. Contradiction. □

Proof of Lemma 30: Immediate from the Definition of busy events. □

Proof of Theorem 38: We show (a) that m_P is a function, (b) that it is an enforcement mechanism in the sense of Definition 5, and (c) that it is a correct such mechanism.

For (a), by Theorem 35, the function $\mathsf{resolve}$ is defined on every reachable marking of P, and so m_P is a function.

For (b), it is straightforward to show by cases on $m(\cdot,\cdot)$ that m_P is monotone and idempotent. For the properties, (1) is immediate by the definition of m_P. (2a-c) are immediate by the definition of m.

For (c), m_P is correct. Consider $m_P(v)$ for some $v \in \mathcal{D}_m$. Note inductively using the definition of $m_P(v)$ and that $\Sigma \setminus \Gamma$ are always enabled that $m_P(v)$ is a run of P. We must prove that this run is accepting or can be extended to an accepting one. If it is already accepting, then we are done, so suppose it is not. Again by Theorem 35 B is resolvable, so any pending response can be discharged; it follows that $m_P(v)$ can be extended to an accepting run. Finally, m_P is transparent, which follows straightforwardly from the definition of m. □

Lemma 40: If X is dependable for $[M]\ T$, then so is any $Y \subseteq X$.

Proof: Immediate from the definition. □

Proof of Theorem 35: Suppose M' is reachable from M, and suppose X is the maximal set of events satisfying $x : (i_x, 1, 0) \in M'$ for $x \in X$ and some i_xs. We must prove that there is some finite set of events $\{e_1, \ldots, e_m\} \subseteq I(X)$ such that

$$M' \xrightarrow{e_1} M'_1 \xrightarrow{e_2} \cdots \xrightarrow{e_m} M'_m \tag{13}$$

is an event transition sequence and $\mathsf{deadline}(M'_m) > 0$. Clearly each $x \in X$ is busy, $X \subseteq B$, and X is dependable by Lemma 40. Because X is dependable, $\hat{I}(X)$ is acyclic. Let $e_1, \ldots, e_n$ be a topological sorting of $\hat{I}(X)$. We prove by induction on n that:

1) a subsequence of $e_1, \ldots, e_n$ is a transition sequence from M',
2) at each M'_i, e_i is either enabled or excluded, and
3) at each M'_j with $j > i$, e_i is excluded if it was at M'_i, pending only if it is excluded, and executed otherwise.

For $k = 1$, by the definition of $\hat{I}(X)$, e_1 has no conditions or milestones and so is either enabled or excluded (2). If it is enabled, we keep and execute it, otherwise we forget about it (1). (3) is vacuously true. For $k > 1$ we know that all of the conditions and milestones of e_k are among $e_1, \ldots, e_{k-1}$, and by the induction hypothesis (2) and (3) each of those are either executed or excluded. By Definition 34 of dependability, item 3, if e_k has a condition to an earlier e_j, it is with time constraint 0. Thus e_k is enabled iff it is included. If it is excluded we have (1,2). If it is included and so enabled, we execute it to establish (1,2). By Definition 34 of dependability, item 2, executing e_k does not make any of $e_1, \ldots, e_{k-1}$ pending or included, establishing (3).

Clearly, each $x \in X$ are among $e_1, \ldots, e_n$, and by construction, each x is either excluded or executed in M'_n. By the definition of dependability, if follows that $\mathsf{deadline}(M'_n) > 0$.

This proof also provides an algorithm for computing resolve: Simply compute $\hat{I}(X)$ from its definition—clearly in polynomial time—and topologically sort the resulting graph. □

B. Finite LTS for Timed DCR Processes

We define a finite LTS. To simplify the definition, we use the same limit for all events, namely the maximal delay occurring in the term T identified by the function $\mathsf{maxdelay}$ defined as follows:

$$\begin{aligned}
\mathsf{maxdelay}(e \bullet\!\!\xleftarrow{k} f) &= k \\
\mathsf{maxdelay}(e\ \mathcal{R}\ f) &= 0 \qquad (\text{when } \mathcal{R} \neq e \bullet\!\!\xleftarrow{k} f) \\
\mathsf{maxdelay}(T \mid U) &= \max\{\mathsf{maxdelay}(T), \mathsf{maxdelay}(U)\}\,.
\end{aligned}$$

We then inductively define a function that truncates a marking to a given maximum delay, taking $\epsilon_{|k} = \epsilon$, and

$$M, e : (h, i, r)_{|k} = M_{|k}, e : (\mathsf{min}\{k, h\}, i, r)\,.$$

We say that markings M and $M'$are k-equivalent if $M_{|k} = M'_{|k}$, and define the bounded LTS as the quotient.

Definition 41: Let $[M]\ T$ be a DCR process with $k = \mathsf{maxdelay}(T)$. We define the *bounded* LTS $\mathsf{lts}_b([M]\ T) = (\mathcal{M}_{|k}, \mathcal{E}', \to, M_{|k}, \ell', \Sigma)$, with components *events* $\mathcal{E}' = \mathcal{E} \uplus \{\mathsf{tick}\}$; transitions $\to \subseteq \mathcal{M}_{|k} \times \mathcal{E}' \times \mathcal{M}_{|k}$, with $M_{|k} \xrightarrow{\alpha} M'_{|k}$ iff $T \vdash M_{|k} \xrightarrow{\alpha} M'$; *states* $\mathcal{M}_{|k} = \{M'_{|k} \mid M \to^* M'\}$, *initial state* $M_{|k}$, and *labelling* $\ell'(e) = \ell(e)$, for $e \in \mathcal{E}$ and $\ell'(\mathsf{tick}) = \mathsf{tick}$.

Finiteness is immediate by noting that the first state component, the age, is bounded by the maximum delay, while the third state component, the response deadline, is bounded by the maximum response given by the initial marking and all response relations $e \bullet\!\!\xrightarrow{k} f$.

We define accepting runs for the bounded LTS as for the timed event LTS.

Lemma 42: Let $[M]\ T$ be a DCR process with $k = \mathsf{maxdelay}(T)$. Then $T \vdash M \xrightarrow{\alpha} M'$ iff $T \vdash M_{|k} \xrightarrow{\alpha} M''$ with $M''_{|k} = M'_{|k}$.

Proof: By structural induction on T, we have $[M]\ T \vdash e : \delta$ iff $[M_{|k}]\ T \vdash e : \delta$. Now note that $((e : \delta)\langle M\rangle)_{|k} = ((e : \delta)\langle M_{|k}\rangle)_{|k}$ and $\mathsf{tick}\langle M\rangle_{|k} = \mathsf{tick}\langle M_{|k}\rangle_{|k}$. The lemma follows using the definition of transitions in Figure 4.

□

Proposition 43: Let $P = [M]\ T$ be a DCR process, and let $k = \mathsf{maxdelay}(T)$. Then $P = [M]\ T$ in $\mathsf{lts}(P)$ and $P_{|k} = [M_{|k}]\ T$ in $\mathsf{lts}_b(P)$ are bisimilar and the languages of the two LTSes coincide, that is, $\mathsf{lang}(\mathsf{lts}(P)) = \mathsf{lang}(\mathsf{lts}_b(P))$.

Proof: Assume $k = \mathsf{maxdelay}(T)$. By Lemma 42 it follows that $\{(M', M'_{|k}) \mid M \to^* M'\}$ is a bisimulation. Moreover, for any M, the second and third components of $M_{|k}$ are identical, so this bisimulation is an *extended* bisimulation in the sense of [14] that respects the acceptance of runs. □