

# Analysis and Applications of Timed Service Protocols

Julien Ponge, Benatallah Boualem, Fabio Casati, Farouk Toumani

▶ **To cite this version:**

Julien Ponge, Benatallah Boualem, Fabio Casati, Farouk Toumani. Analysis and Applications of Timed Service Protocols. ACM Transactions on Software Engineering and Methodology, 2010, 19 (4), 10.1145/1734229.1734230 . inria-00483952

**HAL Id: inria-00483952**

**https://inria.hal.science/inria-00483952**

Submitted on 17 May 2010

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Analysis and Applications of Timed Service Protocols

JULIEN PONGE
Université Blaise Pascal, Clermont-Ferrand, France
University of New South Wales, Sydney, Australia

BOUALEM BENATALLAH
University of New South Wales, Sydney, Australia

FABIO CASATI
University of Trento, Italy

FAROUK TOUMANI
Université Blaise Pascal, Clermont-Ferrand, France

Preliminary version, December 2008

*This is a preliminary version of the article to appear in ACM Transactions on Software Engineering and Methodology.*

**Abstract**

Web services are increasingly gaining acceptance as a framework for facilitating application-to-application interactions within and across enterprises. It is commonly accepted that a service description should include not only the interface, but also the *business protocol* supported by the service. The present work focuses on the formalization of an important category of protocols that include time-related constraints (called *timed protocols*), and the impact of time on compatibility and replaceability analysis. We formalized the following timing constraints: C-Invoke constraints define time windows within which a service operation can be invoked while M-Invoke constraints define expirations deadlines. We extended techniques for compatibility and replaceability analysis between timed protocols by using a semantic-preserving mapping between timed protocols and timed automata, leading to the identification of a novel class of timed automata, called *protocol timed automata* (PTA). Specifically, PTA exhibit a particular kind of silent transitions that strictly increase the expressiveness of the model, yet they are closed under complementation, making every type of compatibility or replaceability analysis decidable. Finally, we implemented our approach in the context of a larger project called ServiceMosaic, a model-driven framework for web service life-cycle management.

## 1 Introduction

Service descriptions are specifications of the syntactic or semantic properties of a service that are made available to potential clients for the purpose of (i) assisting developers in creating clients that can correctly use and interact with a service, and (ii) enabling the selection, either at design time or at runtime, of services that match the clients' needs. Today, service descriptions typically include the interface definition, the transport-level properties

(both can be specified in WSDL), and may also include *business protocol* definitions, that is, the specification of the possible message exchange sequences (conversations) that are supported by the service [11]. Protocols can be specified using BPEL [43]) or any of the many other languages developed for this purpose (e.g., [11, 17]).

Providing service descriptions is not in itself sufficient to facilitate development and binding. In addition to descriptions, we need methods and software tools for analyzing service descriptions to (i) identify if interaction between a client and a service is possible; (ii) if it is, identify which conversations can be carried out between two services, to help developers to check if these include all and only the desired ones; (iii) if it is not, understand mismatches between protocols and, if possible, create adapters to allow interactions to occur. We generally refer to this kind of analysis as *compatibility/replaceability analysis* [38, 11].

The need for formal methods and software tools for such type of analysis is widely recognized, and many approaches have been developed to this end, including some by the authors. In [11, 10, 12], we presented an approach and a model for business protocols as well as a framework for compatibility analysis. This paper focuses on the important category of protocols that include time-related constraints (called *timed protocols* in the following). Time is a crucial abstraction that has been studied in several works in research fields such as workflow systems [49, 29, 20] and even web services [18, 36, 30]. There are countless examples of behaviors that involve timing issues in any kind of protocol [11], from business protocol for web services (e.g., see the *RosettaNet PIPs* [48]), to interactions between traditional web-based services and users (see E-Commerce web sites such as *Travelocity* or *Amazon*), to lower level protocols such as TCP. Time-related behaviors range from session timeouts to "logical" deadlines with different kinds of behaviors (e.g., seats reserved on a flight needs to be paid within $n$ hours otherwise they are released). Existing approaches in the field of service-oriented computing mainly consider time for performing traditional model checking (e.g., detecting deadlocks).

Given the importance of considering time-related properties, we present concepts and techniques, supported by a tool, for performing compatibility and replaceability analysis between timed protocols. Furthermore, we show how the analysis and the tool can be leveraged to perform the same analysis between BPEL processes or between a BPEL process and a protocol. The availability of such concepts and tools is quite useful in that it allows to assess compatibility and replaceability in both top-down and bottom-up development approaches. In top-down scenarios a client is designed starting from its external interface and protocol, and we are faced with the problem of binding it with a (compatible) service. In bottom-up approaches the development starts from the composition of services (e.g., the BPEL process), and quite often no protocol description is made available as part of the documentation (and if it is made available, we have no guarantee that it correctly describes the message exchanges actually supported by the process). In this case, we are then faced with the problem of verifying if a selected partner service is compatible with our composition, either by looking at the partner's defined protocol, if available, or by looking at the partner's process (again, if available). If none of the above is available then compatibility analysis can only be based on the WSDL interface description and it is necessary limited). Compatibility analysis between processes (composite services) is very useful in integration scenarios within an enterprise where process (e.g., BPEL) specifications are often available. Solving the above problem and supporting these protocol-based analysis scenarios require tackling a number of challenges that we address in this paper. More precisely, the main contributions of this paper are:

- The first step consists in **defining a protocol model**. We did so in our previous

work, and we briefly summarize the results in this paper to make it self-contained. In particular, an informal model for timed protocols has been presented in [9], with a first approach being formalized in [8] as a subset of the model that we present in this paper. Here, **we revisit and extend** concepts and techniques defined earlier for basic protocols to make them applicable to timed protocols. As it is often the case in such tasks, when designing our extended model we were facing a trade-off between expressiveness and complexity / readability of the model. While in general new abstractions may provide benefits, they may also make the model too complex, thereby rendering it unusable for practitioners. To find a reasonable balance between expressiveness and complexity, we conducted an analysis of real world e-commerce portals to identify the service abstractions that are useful and commonly needed in many practical situations [13].

- The next step addressed in this paper lies in the extension of the protocol operators (e.g., intersection, difference), needed to handle the **compatibility and replaceability analysis**, to the context of timed protocols and the investigation of their computational properties. Indeed, the introduction of time aspects **adds significant complexity to the problem**. Many formal models enabling explicit representation of time exist (e.g., *timed automata*, *timed petri-nets*), all showing extreme difficulties to handle algorithmic analysis of timed models [4, 34]. For example, *timed automata*, which are today one the most used modeling formalism to deal with timing constraints, suffer from undecidability of many problems such as language inclusion and complementation that are fundamental to system analysis and verification tasks. Such problems have been shown to be sensitive to several criteria (e.g., density of the time axis, type of constraints, presence of silent transitions, etc). In this paper we show that the set of protocol manipulation and comparison operators that are essential for realizing protocol analysis are decidable. We show this by establishing a reversible, **semantic-preserving mapping to *timed automata*** [2] that, incidentally, identifies a new class that we called *protocol timed automata*. This class exhibits *silent* or *$\varepsilon$-transitions* with clock resets, making it strictly more expressive than general timed automata for which there is no closure under the complementation operator, and hence the language inclusion problem is not decidable. These properties turn out to be the key for realizing the protocol operators. Protocol timed automata are a **novel class of timed automata** which is closed under complementation and for which the language inclusion problem is decidable, despite the presence of $\varepsilon$-transitions with clock resets.

- We tackle the problem of examining **compatibility between processes** (and specifically BPEL processes with time-related constructs such as alerts and durations). This is very important as protocol definitions are not always available (even for the services we develop), while BPEL specifications are. Furthermore, BPEL is widely used also as a protocol language and therefore addressing BPEL significantly increases the practical applicability and impact of the research presented here. To this end, we build on top of the protocol analysis operators by devising a mechanism to extract timed protocol specifications from BPEL code and by then using the operators to analyze (in)compatibility between BPEL processes and the services they are supposed to interact with.

- All the features described here have been **implemented in a tool** as part of a larger project called *ServiceMosaic* [14] which aims at developing a model-driven framework

for web service life-cycle management. In addition to process and protocol analysis as discussed here, *ServiceMosaic* also includes facilities for designing protocols and for discovering protocol models from service execution logs [41]. A demonstration featuring the ServiceMosaic tools has been presented in [39].

Preliminary versions of this work that consider restricted forms of timed protocols have been already published in conference papers [9, 8, 45]. More precisely, [9, 8] deals with a timed protocol model without C-Invoke constraints and which includes M-Invoke constraints that only refer to the last transition while [45] considers timed protocol model in which explicit transitions cannot be fired after the expiration of the implicit transitions. This paper extends our previous work in the following directions: (i) it considers a more general timed protocol model without restrictions on the usage of the C-Invoke and M-Invoke constraints. As a consequence, a more in-depth technical analysis is required in order to set up an adequate mapping to timed automata and to prove the closure properties of the operators, (ii) it extends the protocol analysis approach in order to be able to compare BPEL processes or a BPEL process and a protocol and illustrates on a typical usage scenario how the proposed approach works.

The remainder of this paper is organized as follows. Section 2 motivates the need for time-related constraints and introduces both informally and formally the *timed protocol* model used in this paper. Section 3 extends to the context of timed protocols the approach of protocol compatibility and replaceability analysis as well as the protocol manipulation and comparison operators introduced in [11]. Sections 4 and 5 contain the main technical contribution of this work. Section 4 describes a semantic-preserving mapping between timed protocols and timed automata and leverages it to prove the main computational properties of the considered timed protocol operators for which we give algorithms in section 5. Section 6 presents our prototype platform and illustrates it on a typical usage scenario related to service development. In particular, this section describes how the presented protocol analysis approach can be applied on BPEL processes. Section 7 discusses some limitations of the proposed approach as well as related work. We conclude in Section 8.

## 2 Timed protocol modeling

This section starts by motivating the need of time-related constraints using three examples related to web services and business processes where timing constraints play a critical role: an application, an application integration standard and a web service composition language. Then, it describes the *timed protocols* model both formally and informally.

- *E-Commerce portals.* The sales condition notice of many E-Commerce portals provide temporal constraints. Let us consider a classic example of a plane ticket seller portal. A potential purchaser is usually allowed to put a seat on hold for a day or two before a confirmation and payment. In case the buyer does not confirm the purchase after the delay, or if it does not cancel the reservation, the seller will implicitly release the seat holding and cancel the purchase process. There are other examples in the field of E-Commerce. For example goods selling portals are usually entitled by law regulations to allow a buyer to return a purchase within a short delay such as one week after the delivery. Also, they are often constrained to respect delays when dealing with customers for operations such as the deliveries or the refunding of returned purchases.
- *RosettaNet PIPs.* RosettaNet [48] is an industrial consortium which aims at facilitating the transactions among the supply chains of trading partners. It consists

of many specifications called the *Partner Interface Processes* (PIPs) that represent the processes involved in those transactions. The PIPs have been applied to the production systems of companies such as Intel. The PIPs can be potentially implemented using service-oriented architectures. The PIPs exhibit time-dependent behaviors. For example the PIP 3A4 specifies how a seller and a buyer can process a purchase order. More specifically, it specifies the following timing constraints: (i) the *PurchaseOrderRequestAction* and the *PurchaseOrderConfirmationAction* must be acknowledged within 2 hours, and (ii) the reply to the *PurchaseOrderRequestAction* must be sent within 24 hours.

- *BPEL processes.* The *Business Process Execution Language* [43] is the major specification in the field of long-running web services orchestrations. BPEL provides several time-related constructions such as the *wait* activity which causes a given process to "sleep" for a given amount of time or until a date has been reached. The other activity is the use of *pick* containers for which timers can be defined, for example to trigger a timeout handler if a message has not been received after a given delay, or when a given date has been reached.

## 2.1 Overview of the model

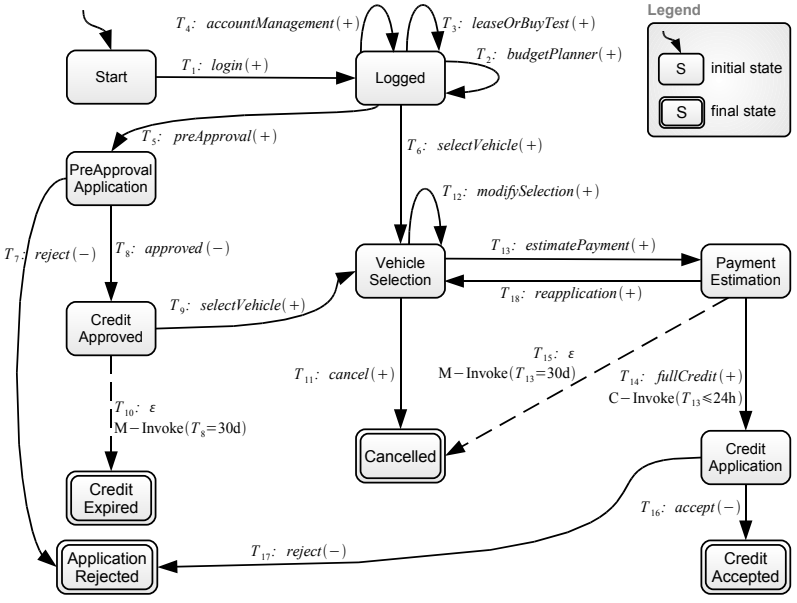



Figure 1: A timed protocol of an online financing service.

We propose an extension of the *business protocol* model [11, 12] which is built upon the traditional state-machine formalism. Indeed, state-based models have been commonly used to model the behavior of systems, due to the fact that they are simple and intuitive. In the model, states represent the different phases that a service may go through during its interaction with a requester. A transition label is a message supported by the service. It has a polarity which is positive (+) if the message is incoming, or negative (−) if it is outgoing. Transitions are triggered when their associated messages are sent or received. Hence, a state identifies a set of outgoing transitions, and therefore a set of possible messages that can be either sent or received when the conversation with a requester is in this state. For example, the protocol depicted on Figure 1 (inspired by the *Ford Credit*

*web portal*) is initially in the *Start* state, and requesters begin a conversation by sending a *login* message, moving to the *Logged* state.

In the figure, the initial state is indicated by an unlabeled entering arrow while final states are double-circled. A conversation is accepted when ending in such a final state. Hence, a sequence of messages $login(+) \cdot preApproval(+) \cdot reject(-)$ is a conversation supported by the protocol, while $selectVehicle(+) \cdot login(+)$ is not: no transition for a $selectVehicle$ message is available from the *Start* state, the two messages cannot be ordered this way, and the conversation does not end in a final state. Business protocols must be deterministic, as the requesters always needs to be able to determine in which state the service is, otherwise much of the purpose of the protocol specification would be lost. We consider the following two extensions to the base protocol model:

1. C-Invoke constraints specify time windows within which a transition *can* be fired. Outside of those time windows, the transition is disabled, and exchanging the associated message results in an error.
2. M-Invoke constraints specify *when* a transition is automatically fired.

The obtained model is called *timed (business) protocol model*. C-Invoke constraints can be attached to *explicit* transitions for which a message is exchanged between the service provider and its requesters. The absence of C-Invoke constraint on an explicit transition means that it can be fired from its source state at any time. By contrast, M-Invoke constraints are associated to *implicit* transitions. They model state changes in conversations once a delay has elapsed (a typical example being a timeout). Implicit transitions are analogous to the silent transitions in the automata theory [35] and we associate the empty word $\varepsilon$ as the label of those transitions. However, and unlike usual silent transitions, implicit transitions are mandatorily fired whenever their associated M-Invoke constraints are evaluated to true.

Continuing with the example protocol depicted on Figure 1, it is indicated that a full credit application is accepted only if it is received at most 24 hours after a payment estimation has been made. This behavior is specified by tagging the transition $T_{14} : fullCredit(+)$ with a time constraint C-Invoke($T_{13} \leq 24h$), i.e., $T_{14}$ can only be fired within a time window $[0h, 24h]$ after $T_{13}$ has been fired. $T_{10}$ has a constraint M-Invoke($T_8 = 30d$), meaning that once a pre-approval application has been approved ($T_8$), a requester is given 30 days to select a vehicle ($T_9$). If it does not continue the conversation by sending a $selectVehicle$ message within the next 30 days, then the service provider will automatically fire $T_{10}$ and move to the $CreditExpired$ state, ending the conversation. Finally, it should be noted that the presence of an implicit transition from a given state affects the time constraints of the explicit transitions outgoing from the same state. Indeed, $T_{10}$ implies that $T_9$ can only be fired within a time window matching the 30 days. Hence, a constraint C-Invoke($T_8 < 30d$) is implicitly associated with $T_9$ because of the M-Invoke constraint of $T_{10}$.

## 2.2 Formal model

### 2.2.1 Syntax

Before giving the definition of timed protocols, we need to formalize the C-Invoke and M-Invoke constraints. Let $\mathcal{X}$ be a set of variables referring to transition identifiers: if $r$ is a transition then $T_r \in \mathcal{X}$ is the variable referring to this transition. We consider the two kinds of time constraints defined over a set of variables $\mathcal{X}$ using the following grammars:

- C-Invoke($c$) with $c$ defined as follows:

$$c ::= x \; op \; k \mid x - x' \; op \; k \mid c \wedge c \mid c \vee c$$

with $op \in \{=, \neq, <, >, \leq, \geq\}$, $x \in \mathcal{X}$, $x' \in \mathcal{X}$ and $k \in \mathbb{Q} \cup \{\perp\}$, where $\mathbb{Q}$ denotes the set of positive rational numbers,

- M-Invoke($c$) with $c$ defined as follows:

$$c ::= (x = k) \wedge c' \mid c \wedge c \mid c \vee c$$

with $x \in \mathcal{X}$, $k \in \mathbb{Q} \cup \{\perp\}$ and $c'$ being defined like in the grammar of C-Invoke constraints.

The following is the definition for timed business protocols, extending the business protocols model [11, 12].

**Definition 2.1.** *A* timed business protocol *is a tuple* $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{X}, \mathcal{C}, \mathcal{R})$ *where:*

- $\mathcal{S}$ *is a finite set of states, with* $s_0 \in \mathcal{S}$ *being the initial state.*
- $\mathcal{F} \subseteq \mathcal{S}$ *is the set of final states. If* $\mathcal{F} = \emptyset$*, then* $\mathcal{P}$ *is said to be an empty protocol.*
- $\mathtt{M} = \mathtt{M_e} \cup \{\varepsilon\}$ *is a finite set of messages* $\mathtt{M_e}$ *augmented with the empty message* $\varepsilon$*. For each message* $m \in \mathtt{M_e}$*, we define a function* Polarity$(\mathcal{P}, m)$ *which will be positive* $(+)$ *if* $m$ *is an input message in* $\mathcal{P}$*, and negative* $(-)$ *if* $m$ *is an output message in* $\mathcal{P}$*.*
- *We assume that each transition* $r \in \mathcal{R}$ *is identified by a unique identifier* $id(r)$*.* $\mathcal{X} = \{T_i \mid \exists r \in \mathcal{R},\ T_i = id(r)\}$ *is a set of clock variables defined over the set of transitions* $\mathcal{R}$*.*
- $\mathcal{C}$ *is a set of time constraints defined over a set of variables* $\mathcal{X}$*. The absence of a constraint is interpreted as a constraint which always evaluates to* `true`*.*
- $\mathcal{R} \subseteq \mathcal{S}^2 \times \mathtt{M} \times \mathcal{C}$ *is a finite set of transitions. Each transition* $(s, s', m, c)$ *identifies a source state* $s$*, a target state* $s'$*, a message* $m$ *and a constraint* $c$*. We say that the message* $m$ *is enabled from a state* $s$*. When* $m = \varepsilon$*,* $c$ *must be a M-Invoke constraint, otherwise* $c$ *must be either a C-Invoke constraint or* `true`*.*

In the sequel, we use the notation $\mathcal{R}(s, s', m, c)$ to denote the fact that $(s, s', m, c) \in \mathcal{R}$. To enforce determinism, we require that a protocol has only one initial state, and that for every state $s$ and every two transitions $(s, s_1, m_1, c_1)$ and $(s, s_2, m_2, c_2)$ enabled from $s$, we have either $m_1 \neq m_2$ or $c_1 \wedge c_2 \equiv \mathtt{false}$.

### 2.2.2 Semantics

Before defining the timed protocol semantics, we introduce first the notion of variable valuation.

**Variable interpretation** We consider as a time domain the set of positive reals $\mathbb{R}_{\geq 0}$ augmented with a special element $\perp$ to denote the fact that a transition has never been taken yet. Let $\mathcal{X}$ be a set of variables valued in $\mathbb{R}_{\geq 0}$. A variable valuation $\mathcal{V} : \mathcal{X} \rightarrow \mathbb{R}_{\geq 0} \cup \{\perp\}$ is a mapping that assigns to each variable $x \in \mathcal{X}$ a time value $\mathcal{V}(x)$.

We note by $\mathcal{V}_t(x)$ the valuation of $x$ at an instant $t$. In the beginning (i.e., at instant $t_0 = 0$) we assume that all of the variables are set to $\perp$, i.e., $\mathcal{V}_{t_0}(T_i) = \perp$, $\forall T_i \in \mathcal{X}$.

Then, a variable valuation at a time $t_j$ is completely determined by a protocol execution. Consider for example an execution $\sigma = s_0 \cdot (m_0, t_0) \cdot s_1 \ldots s_{n-1} . (m_{n-1}, t_{n-1}) \cdot s_n$ of a

protocol $\mathcal{P}$. The valuation of a clock variable $T_i$ at time $t_j$, with $0 < j \leq n$, is defined as follows:

$$\mathcal{V}_{t_j}(T_i) = \begin{cases} \mathcal{V}_{t_{j-1}}(T_i) + (t_j - t_{j-1}) \\ 0 \text{ if } T_i = id(\mathcal{R}(s_j, s_{j+1}, m, c)) \text{ is fired from } s_j \text{ to } s_{j+1} \end{cases}$$

It should be noted that for any $r \in \mathbb{R}_{\geq 0}$, $k \in \mathbb{Q}$ and any comparison operator $op \in \{<, \leq, =, \neq, >, \geq\}$:

$$\begin{cases} \bot + r = \bot \\ \bot - r = \bot \\ \bot \ op \ k = \texttt{false} \\ \bot \ op \ \bot = \texttt{true} \text{ if } op \in \{=, \leq, \geq\} \\ \bot \ op \ \bot = \texttt{false} \text{ otherwise} \end{cases}$$

Given a variable valuation $\mathcal{V}$ and a constraint C-Invoke($c$) (respectively, M-Invoke($c$)), we denote by $c(\mathcal{V})$ the constraint obtained by substituting each variable $x$ in $c$ by its value $\mathcal{V}(x)$. A variable valuation $\mathcal{V}$ satisfies a constraint C-Invoke($c$) (respectively, M-Invoke($c$)) if and only if $c(\mathcal{V}) \equiv \texttt{true}$. In this case, we write $\mathcal{V} \models$ C-Invoke($c$) (respectively, $\mathcal{V} \models$ M-Invoke($c$))

**Timed conversations** Timed conversations are inspired from the notion of *timed words* in timed automata as defined in [2].

Let $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{X}, \mathcal{C}, \mathcal{R})$ be a timed protocol. A *correct execution* (or simply, an execution) of $\mathcal{P}$ is a sequence $\sigma = s_0 \cdot (m_0, t_0) \cdot s_1 \ldots s_{n-1} \cdot (m_{n-1}, t_{n-1}) \cdot s_n$ such that:

1. $t_0 \leq t_1 \leq \ldots \leq t_n$ (i.e., the occurrence of times increase monotonically). As usual, we also assume non-Zenoness [2] of the sequences of time (i.e., we cannot have infinite sequences in finite time),
2. $s_0$ is the initial state and $s_n$ is a final state of $\mathcal{P}$, and
3. $\forall j \in [1, n]$, we have: $\mathcal{R}(s_{j-1}, s_j, m_{j-1}, c_{j-1})$ and $\mathcal{V}_{j-1} \models c_{j-1}$.

As an example, the sequence $\sigma' = Start \cdot (login(+), 0) \cdot Logged \cdot (preApproval(+), 1) \cdot PreApprovalApplication \cdot (approved(-), 3) \cdot CreditApproved \cdot (\varepsilon, 33) \cdot CreditExpired$ is a correct execution of the financing service protocol depicted on Figure 1.

If $\sigma = s_0 \cdot (m_0, t_0) \cdot s_1 \ldots s_{n-1} \cdot (m_{n-1}, t_{n-1}) \cdot s_n$ is a correct execution of $\mathcal{P}$, then the sequence $tr(\sigma) = (m_0, t_0) \ldots (m_{n-1}, t_{n-1})$ forms a *timed trace* which is compliant with $\mathcal{P}$. Continuing with the example, the execution $\sigma'$ of the above service protocol leads to the timed trace $tr(\sigma') = (login(+), 0) \cdot (preApproval(+), 1) \cdot (approved(-), 3) \cdot (\varepsilon, 33)$.

During an execution $\sigma$ of a protocol $\mathcal{P}$, the externally observable behavior of $\mathcal{P}$, hereafter called *timed conversation* of $\mathcal{P}$ and noted $conv(\sigma)$, is obtained by removing from the corresponding timed trace $tr(\sigma)$ all of the non observable events (i.e., all of the pairs $(m_i, t_i)$ with $m_i = \varepsilon$). For example, during the previous execution $\sigma'$, the observable behavior of the financing service is described by the timed conversation $obs(\sigma') = (login(+), 0) \cdot (preApproval(+), 1) \cdot (approved(-), 3)$.

In the following, given a protocol $\mathcal{P}$, we denote by $Traces(\mathcal{P})$ the set of the *timed traces* which are compliant with $\mathcal{P}$ and with $Tr(\mathcal{P})$ the set of timed conversations of $\mathcal{P}$.

**Timed interactions** Timed conversations describe the externally observable behavior of timed protocols and, as we will show below, are essential to analyze the ability of two services to interact correctly. Consider for example the protocol $\mathcal{P}$ depicted on Figure 1 and its reversed protocol $\mathcal{P}'$ obtained by reversing the polarity of the messages (i.e., input messages become outputs and vice-versa).

We can observe that when $\mathcal{P}'$ interacts with $\mathcal{P}$ by following a given timed conversation, $\mathcal{P}$ follows exactly a conversation with the reversed polarities of the messages. For example, if during such an interaction the timed conversation of $\mathcal{P}$ is $(login(+), 0) \cdot (selectVehicle(+), 1) \cdot (estimatePayment(+), 10) \cdot (fullCredit(+), 30) \cdot (accept(-), 100)$, then the timed conversation of $\mathcal{P}'$ is $(login(-), 0) \cdot (selectVehicle(-), 1) \cdot (estimatePayment(-), 10) \cdot (fullCredit(-), 30) \cdot (accept(+), 100)$.

In this case, we call the path $(login, 0) \cdot (selectVehicle, 1) \cdot (estimatePayment, 10) \cdot (fullCredit, 30) \cdot (accept, 100)$ a *timed interaction trace* of $\mathcal{P}$ and $\mathcal{P}'$. The polarities of the messages that appear in interaction traces are not defined. Indeed, each input message $m$ of one protocol matches an output message $m$ of the other protocol.

More precisely, let $\mathcal{P}$ and $\mathcal{P}'$ be two timed protocols and let $\tau = (a_0, t_0) \cdot (\ldots) \cdot (a_n, t_n)$ be a sequence of events in which the polarities of the messages are undefined. Then $\tau$ is a timed interaction trace of $\mathcal{P}$ and $\mathcal{P}'$ if and only if there exist two timed conversations $\sigma_1$ and $\sigma_2$ such that:

1. $\sigma_1 \in Tr(\mathcal{P})$ and $\sigma_2 \in Tr(\mathcal{P}')$, and
2. $\sigma_1$ is the reverse conversation of $\sigma_2$ (i.e., the conversation obtained from $\sigma_2$ by inverting the polarities of the messages), and
3. $\tau = Unp(\sigma_1) = Unp(\sigma_2)$ where $Unp(\sigma)$ denotes the trace obtained from $\sigma$ by removing the polarities of the messages.

# 3 Protocol analysis concepts

Our approach for protocol compatibility and replaceability analysis is based on a set of protocol manipulation and comparison operators that were introduced in [11]. The proposed operators are build on existing relations and operators defined in the context of timed automata (e.g., language inclusion, intersection, complementation, etc). We briefly recall hereafter both the type of analysis that we target and the operators that we use to do it. We describe the implementation of our operators using already-known constructs on timed automata in Section 5.

## 3.1 Protocol analysis

*Compatibility* analysis is concerned with verifying whether two services can converse. It is necessary for both static and dynamic binding, and it also aids in managing evolution as it helps verify that a modified client can still interact with a certain service. More precisely, we identified two compatibility classes. *Partial compatibility* between two protocols $P_1$ and $P_2$ implies that at least one conversation can be carried out between two services implementing these protocols. A protocol $P_1$ is *fully compatible* with $P_2$ if $P_2$ can support all message exchanges that $P_1$ can generate (the inverse is not required to be true). Ideally, if we have developed a service $S$ characterized by protocol $P$, at binding time we will want to look for services that have a protocol with which $P$ is fully compatible, so that every message exchange that our service can generate is understood by our partner.

*Replaceability* analysis identifies whether a service can act as a substitute of another one, either in general or when interacting with certain requesters. Such an analysis involves finding the set of conversations that both services can support even if they are not equivalent. This is useful to determine whether a new version of a service (protocol) can support the same conversations as the previous one or whether a newly defined service can support the conversations that a given standard specification mandates. As in the case of compatibility, we identified several replaceability classes. *Protocol equivalence* occurs

when two protocols support exactly the same conversations. *Protocol subsumption* occurs when a protocol supports at least all of the conversations of another one. Finally,

*Protocol replaceability w.r.t. a client protocol* occurs when a protocol $P_1$ can replace a protocol $P_2$ when interacting with a client protocol $P_c$ if every valid conversation between $P_2$ and $P_c$ is also a valid conversation between $P_1$ and $P_c$. This latter definition is helpful in managing evolution, as when we update our service we may want to check that it can still communicate with the same clients it was interacting before.

We now recall the operators that fully characterize the compatibility and replaceability classes mentioned above and give examples of their usage.

### 3.2 Protocol operators

Operands of the algebra are protocols and operators are special operations defined on protocols that enable, for example, to determine intersection and difference among protocols or to identify which conversations can and cannot be supported when a service is used in place of another one.

| **Operator name** | **Symbol** | **Semantics** |
|---|---|---|
| Compatible Composition | $\|\|^{\mathtt{TC}}$ | $\mathcal{P} = \mathcal{P}_1 \ \|\|^{\mathtt{TC}} \ \mathcal{P}_2$ is a protocol $\mathcal{P}$ such that $T \in Tr(\mathcal{P})$ iff $T$ is an interaction trace of $\mathcal{P}_1$ and $\mathcal{P}_2$ |
| Intersection | $\|\|^{\mathtt{TI}}$ | $\mathcal{P} = \mathcal{P}_1 \ \|\|^{\mathtt{TI}} \ \mathcal{P}_2$ is a protocol $\mathcal{P}$ such that $Tr(\mathcal{P}) = Tr(\mathcal{P}_1) \cap Tr(\mathcal{P}_2)$ |
| Difference | $\|\|^{\mathtt{TD}}$ | $\mathcal{P} = \mathcal{P}_1 \ \|\|^{\mathtt{TD}} \ \mathcal{P}_2$ is a protocol $\mathcal{P}$ that satisfies the following condition: $Tr(\mathcal{P}) = Tr(\mathcal{P}_1) \setminus Tr(\mathcal{P}_2)$ |
| Projection | $\left[\|\|^{\mathtt{TC}}\right]$ | Let $\mathcal{P} = \mathcal{P}_1 \ \|\|^{\mathtt{TC}} \ \mathcal{P}_2$. $\left[\mathcal{P}_1 \ \|\|^{\mathtt{TC}} \ \mathcal{P}_2\right]_{\mathcal{P}_i}$, with $i \in \{1, 2\}$, is the protocol obtained from $\mathcal{P}_1 \ \|\|^{\mathtt{TC}} \ \mathcal{P}_2$ by defining the polarity function of the messages as follows: $Polarity(\left[\mathcal{P}_1 \ \|\|^{\mathtt{TC}} \ \mathcal{P}_2\right]_{\mathcal{P}_i}, m) = Polarity(\mathcal{P}_i, m), \forall m \in \mathtt{M}$ |

Table 1: Protocol manipulation operators semantics.

We informally describe the protocol manipulation operators below, while their formal semantics are presented in Table 1.

- *Parallel composition*, denoted as $\|^{\mathtt{TC}}$, takes two input timed protocols and returns a *timed interaction protocol* that captures the possible interactions between them. A timed interaction protocol has simply no messages polarities. More precisely, the resulting timed interaction protocol describes the set of timed interaction traces of the input protocols.
- *Projection*, used to project the polarity of one protocol on the parallel composition of two protocols, is denoted as $\left[\mathcal{P}_1 \ \|^{\mathtt{TC}} \ \mathcal{P}_2\right]_{\mathcal{P}_1}$.
- *Intersection*, denoted as $\|^{\mathtt{TI}}$, takes two input timed protocols and returns one timed protocol that captures the timed conversations that they have in common.
- *Difference*, denoted as $\|^{\mathtt{TD}}$, takes two input timed protocols and returns one that captures the timed conversations that are supported by the first input protocol but not by the second one.

We give examples of operators-based compatibility and replaceability analysis on Figure 2. $P_1$ and $P_2$ are only partially compatible, as $\left[P_1 \ \|^{\mathtt{TC}} \ P_2\right]_{P_2} \not\equiv P_2$. By using the difference operator to compute $P_2 \ \|^{\mathtt{TD}} \ \left[P_1 \ \|^{\mathtt{TC}} \ P_2\right]_{P_2}$, we get the set of conversations that are supported by $P_2$ but not by $P_1$ which yield to a partial compatibility ($P_1$ does not support receiving a $c$ message after 10 units of time). $P_4$ can be replaced by $P_3$ as it supports

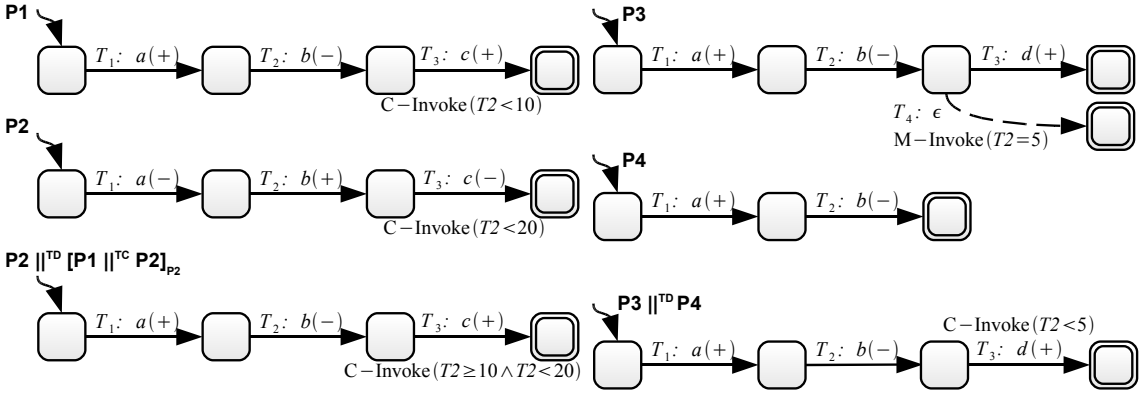



Figure 2: Compatibility and replaceability analysis.

all of the conversations that $P_4$ supports, denoted as $P_3 \sqsubseteq P_4$ (every valid conversation of $P_3$ is also a valid conversation of $P_4$). The converse is however not true as illustrated by $P_3 \parallel^{\mathsf{TD}} P_4$: $P_3$ cannot handle $d$ messages.

| **Class** | **Characterization** |
|---|---|
| Partial compatibility of $\mathcal{P}_1$ and $\mathcal{P}_2$ | $\mathcal{P}_1 \parallel^{\mathsf{TC}} \mathcal{P}_2$ is not *empty* |
| Full compatibility of $\mathcal{P}_1$ and $\mathcal{P}_2$ | $\left[\mathcal{P}_1 \parallel^{\mathsf{TC}} \mathcal{P}_2\right]_{\mathcal{P}_1} \equiv \mathcal{P}_1$ |
| Replaceability of $\mathcal{P}_1$ by $\mathcal{P}_2$ | $\mathcal{P}_2 \sqsubseteq \mathcal{P}_1$ |
| Equivalence of $\mathcal{P}_1$ and $\mathcal{P}_2$ w.r.t. replaceability | $\mathcal{P}_1 \equiv \mathcal{P}_2$ |
| Replaceability of $\mathcal{P}_2$ by $\mathcal{P}_1$ w.r.t. a client protocol $\mathcal{P}_C$ | $\left[\mathcal{P}_C \parallel^{\mathsf{TC}} \mathcal{P}_2\right]_{\mathcal{P}_2} \sqsubseteq \mathcal{P}_1$ or equivalently $\mathcal{P}_C \parallel^{\mathsf{TC}} (\mathcal{P}_2 \parallel^{\mathsf{TD}} \mathcal{P}_1)$ is *empty* |
| Replaceability of $\mathcal{P}_2$ by $\mathcal{P}_1$ w.r.t. a role $\mathcal{P}_R$ | $(\mathcal{P}_R \parallel^{\mathsf{TI}} \mathcal{P}_2) \sqsubseteq \mathcal{P}_1$ |

Table 2: Characterization of the compatibility and replaceability classes.

The formalization of the compatibility and replaceability classes that was introduced in [8] as well as the characterization of these classes using the aforementioned operators remain unchanged in the context of our extended protocol model (see Table 2). However, due to the expressiveness of the timed protocol model used in this paper, the decision problems underlying protocol analysis (e.g., closure properties of the operators, decidability of protocol subsumption) must be investigated in this new context. As described below, we conducted our investigation using a formal framework based on timed automata theory.

# 4 From timed protocols to protocol timed automata

The previous sections have introduced the model of timed business protocols which is suitable for describing and analyzing the external behavior of web services in presence of timing constraints. In turn, the model of timed automata [2] has been extensively studied as an extension of classical automata [35] with real-valued clocks and conditions on the transitions. Given the extensive research that has been made on timed automata, we chose to: (i) use timed protocols as a conceptual model, and (ii) use timed automata for implementing the behavior of timed protocols, and (iii) adapt and/or extend theoretical properties on timed protocols from existing work on timed automata. To achieve this task, we give a mapping from timed protocols to timed automata. However, defining such a

mapping is not a trivial task. Indeed, as we will see later, M-Invoke constraints are not straightforward to implement using timed automata.

Note that we do not directly use timed automata for modeling service protocols at the conceptual level as we believe that this will be a difficult and error prone task. The difficulties in defining the semantic-preserving mapping that we will see hereafter as well as the complexity of the obtained timed automata are strong arguments in favor of timed protocols.

This section is structured as follows. We first introduce timed automata and then provide a formal definition of the model used in this paper. Then we illustrate the challenges of converting a timed protocol into a timed automaton that correctly implements its behavior. We describe the technique for performing such a mapping and finally, we give a characterization of the obtained class of timed automata.

### 4.1 Quick overview on timed automata

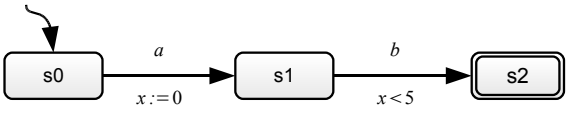



Figure 3: A sample timed automaton $A$.

Timed automata were introduced in [2] as an extension of classical automata [35] to model real-time systems. We take as an example the timed automaton $A$ depicted on Figure 3. At first sight, $A$ is much like a "normal" automaton: it has locations (e.g., $s_0, s_1$ and $s_2$) as well as switches with labels over the alphabet $\Sigma = \{a, b\}$. There is one initial location $s_0$ while $s_2$ is an accepting location. Timing constraints are added in $A$ by making use of a clock $x$ which is a continuous variable over the set of real-valued numbers $\mathbb{R}_{\geq 0}$. The automaton $A$ recognizes the set of timed words $a \cdot b$ such that $b$ is recognized at most 5 units of time after $a$. To do that, the clock $x$ is reset to 0 when the automaton switches from the location $s_0$ to $s_1$ on the symbol $a$. Again, a switch can reset an arbitrary number of clocks. Initially, every clock is set to 0 and then, they grow synchronously as time evolves. Note that, time elapses in the locations, while the switches are instantaneous. Clocks can be used in constraints attached to the switches, called *guards*, and that can enable or disable a switch depending on how guards are evaluated. Here, the clock $x$ is used in the guard of the $b$-labeled switch so that $b$ cannot be recognized when $(x \geq 5)$ is true. More precisely, a *timed word* over an alphabet $\Sigma$ is a finite sequence $(a_0, t_0) \cdot (a_1, t_1) \cdots (a_n, t_n)$ of symbols $a_i \in \Sigma$ that are paired with nonnegative real numbers $t_i \in \mathbb{R}_{\geq 0}$ such that $t_0 \leq t_1 \leq \cdots \leq t_n$. For example $w = (a, 0) \cdot (b, 1)$ is a timed word where $b$ has been recognized 1 unit of time after $a$. A timed language over an alphabet $\Sigma$ is a set of timed word over $\Sigma$. Timed automata recognize timed languages. For example, the automaton $A$ of Figure 3 recognizes the time language $\{(a, t) \cdot (b, t') \mid t - t' < 5\}$. In the sequel, we denote by $\mathcal{L}(\mathcal{A})$ the timed language recognized by a given automaton $A$.

Over the years, several classes of timed automata with different level of expressiveness have been studied leading to many results regarding the usual verification problems (e.g., reachability, language inclusion, etc) [4]. We present below the timed automata model used in our context.

## 4.2 Target timed automata model

We consider a timed automata model that enables *silent transitions*, noted hereafter $\varepsilon$ and called $\varepsilon$-labeled switches. As usual, a timed automaton is defined using a set of clocks $X$ while the set of clocks constraints over $X$, noted $\mathcal{C}(X)$, is built using boolean combinations of atomic constraints of the form $x \# c$ with $x \in X$, $\# \in \{=, \neq, <, \leq, >, \geq\}$ and $c \in \mathbb{Q}$. We also allow diagonal constraints of the form $(x_1 - x_2 \# k)$ with $x_1, x_2 \in X$ and $k \in \mathbb{Q} \cup \perp$. Diagonal constraints are known not to increase the expressiveness of the model but allow more representation conciseness [24].

**Definition 4.1.** *A timed automaton $A$ is a tuple $A = (\Sigma, L, L^0, L^f, X, E)$ where:*

- *$\Sigma$ is a finite* alphabet*, and $\epsilon$ denotes the empty word in $\Sigma^*$,*
- *$L$ is a finite set of* locations *(or states), with $L^0 \in L$ being the* initial location*,*
- *$L^f \subseteq L$ is the set of* final locations *(or accepting states),*
- *$X$ is a finite set of clocks,*
- *$E \subseteq L \times \mathcal{C}(X) \times \Sigma \cup \{\varepsilon\} \times 2^X \times L$ is a finite set of switches (or transitions) $e = (l, g, a, r, l') \in E$ from $l$ to $l'$, where $g$ is the guard, $r$ is the set of clocks to be reset and $a$ is the label.*

A clock valuation $v$ for a set $X$ of clocks is a mapping from $X$ to $\mathbb{R}_{\geq 0} \cup \{\perp\}$ that assigns to each clock $x \in X$ a value $v(x)$ in $\mathbb{R}_{\geq 0} \cup \{\perp\}$. A clocks valuation $v$ satisfies an atomic constraint $(x \# c)$ if and only if $(v(x) \# c)$ is true. This allows to check whether a guard $g$ can be satisfied by a clocks valuation $v$, denoted as $v \models g$. Given $d \in \mathbb{R}_{\geq 0}$, $v' = v + d$ is the clocks valuation such that $v'(x) = v(x) + d$ for each $x \in X$. Also, for $r \subseteq X$, $v' = [r \leftarrow 0]v$ denotes a clock valuation such that $v'(x) = 0$ if $x \in r$ and $v'(x) = v(x)$ if $x \in X \setminus \{r\}$.

As usual, each clock of a timed automaton is a real valued variable that records the amount of time that has elapsed since the last time the clock was reset. However, unlike "standard" timed automata, and in the same spirit as [3], we assume that the values of all clocks are initially equal to $\perp$ (i.e., *undefined*). Then, the first time any clock $x$ is reset to 0, its valuation starts to grow synchronously w.r.t. other clocks as time evolves. Detailed semantics of timed automata and discussion of "classic" verification problems (e.g., location reachability, closure under complementation) can be found in [2].

## 4.3 Informal overview of the challenges

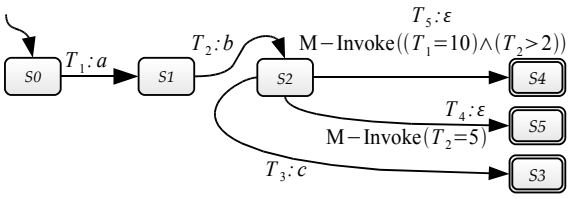



Figure 4: A sample timed protocol $\mathcal{P}$ used as a mapping running example.

We use the timed protocol from Figure 4 as a running example throughout this section. It allows us to illustrate the challenges in creating a timed automaton that behaves like a timed protocol. At first sight, the translation may look straightforward. However as we will see, preserving the behavior of M-Invoke is not an easy task. The translation is performed in two steps.

The first step consists in a straightforward procedure that converts a timed protocol $\mathcal{P}$ into a timed automaton $A$ as follows. States of $\mathcal{P}$ are translated into locations in $A$

(e.g., the state $s_0$ of $\mathcal{P}$ is mapped to a location $l_0$). The initial state is mapped into an initial location while final states are mapped to final locations. Each explicit (respectively, implicit) transition in $\mathcal{P}$ is translated into a switch in $A$ with the message name as its label (respectively, a $\varepsilon$-labeled switches). For example, in $\mathcal{P}$, $T_1 : \mathcal{R}(s_0, s_1, a, \texttt{true})$ is translated into a switch $e_1 = (l_0, \texttt{true}, a, r, l_1)$ in $A$.

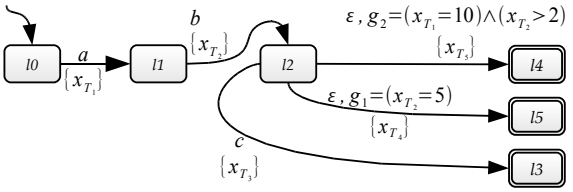



Figure 5: A sample timed automaton that does not enforce M-Invoke semantics.

The transition identifiers from $\mathcal{P}$ are used to generate clocks in $A$. Indeed, each identifier generates a unique clock which is solely reset on its corresponding switch. For example the transition $T_2$ generates a clock $x_{T_2}$ in $A$ which is only reset on the switch that was mapped from $T_2$. This way, we can know when a switch was fired, just like in the timed protocol $\mathcal{P}$. The conversion of a C-Invoke constraint from $\mathcal{P}$ is also direct. For example, a constraint C-Invoke($T_1 < 3$) is mapped to a guard ($x_{T_1} < 3$). At this stage, the mapping of $\mathcal{P}$ to $A$ yields the timed automaton of Figure 5.

Let us now have a closer look on the mapping of implicit transitions. As mentioned before, implicit transitions are automatically fired whenever their associated M-Invoke constraints are evaluated to true. This semantics is not enforced in the target timed automata by simply mapping implicit transitions into $\varepsilon$-labeled switches with a direct translation of M-Invoke constraints into guards. Continuing with our example, the guard $g_1$ and $g_2$ of the timed automata of Figure 5, may potentially become satisfied at the same time, and hence the $c$-labeled transition can be fired while the guard $g_2$ of the $\varepsilon$-labeled switches is evaluated to true. To capture correctly the semantics of M-Invoke constraints in the timed automata, $\varepsilon$-labeled must be preempted with respect to other transitions that can be fired from the same location. To achieve this goal, additional constraints must be added during the mapping phase in order to ensure that once a guard of a given $\varepsilon$-labeled switch becomes true, all the other switches starting from the same location must be deactivated.

To see how such constraints can be enforced, let us have a closer look at the various cases that may occur when entering a location that offers one $\varepsilon$-labeled switch. We illustrate that again on the location $l_2$ of of Figure 5 but by considering only the switch $c$ and one $\varepsilon$-labeled switch, namely the one associated with the guard $g_1$. The generalization to more than one $\varepsilon$-labeled is straightforward, and will be detailed later. Three cases are possible. The first one is that $l_2$ is entered before $g_1$ has been satisfied (i.e., while the constraint ($x_{T_1} < 10$) is true). In this case $c$ can be fired as long as ($x_{T_1} < 10$) is true. The second case is that $l_2$ is entered when the valuation of $x_{T_1}$ is such that ($x_{T_1} > 10$): $c$ can be fired since $g_1$ will never be satisfied in the future. Finally, third one is that $l_2$ is entered while ($x_{T_1} < 10$) is satisfied and the automaton remains at location $l_2$ while time elapses until an instant $t$ such that the clock valuation becomes such that ($x_{T_1} \geq 10$) is true. In this case $c$ can be fired only if, in the past (i.e., at an instant $t' < t$), when ($x_{T_1} = 10$) was true, ($x_{T_2} > 2$) was false. Note that, in this later case to decide if a transition $c$ can be fired safely at a given instant $t$ it is necessary to conduct a reasoning about the *'past'* (i.e., the values of the clocks at an instant $t' < t$).

To sum up, a direct translation of the M-Invoke constraints to guards is not enough to properly capture the implicit transitions semantics in timed automata. Because of that, more elaborated constraints need to be added into the guards. More specifically:

1. the mapping needs to rewrite some guards to enforce the expected behavior of M-Invoke constraints, and
2. there is a need for knowing the exact clock valuations when a location is entered:
   (a) to know the status of each equality clause in each $\varepsilon$-labeled switches (e.g., when $l_2$ is entered, do we have $(x_{T_2} < 5)$, $(x_{T_2} > 5)$ or $(x_{T_2} = 5)$?)
   (b) to know if the $\varepsilon$-labeled switches guards will be satisfiable or not (e.g., $g_2$) when their equality clause is satisfied.

In particular, knowing the valuation of each clock when a location was entered is important for enabling/disabling some switches. For example when entering $l_2$, if $x_{T_2}$ was already greater than 5, then we know that the $\varepsilon$-labeled switch with guard $g_1$ will have no influence on the execution of the $c$-labeled switch nor on the other $\varepsilon$-labeled switch.

In timed automata, as clocks evolve synchronously, the difference between 2 clocks $x_1$ and $x_2$ is a constant until one of them is reset to zero. In the following, we use *diagonal constraints* of the form $(x_1 - x_2 \ \# \ k)$, with $k \in \mathbb{Q} \cup \bot$, to capture the clock valuations when locations are entered. More precisely, for each location offering at least one $\varepsilon$-labeled switch, we add a clock that is attached to this location. Such a clock is reset on every incoming switch of the considered location. For example on Figure 5, we add a clock $y_{l_2}$ which is reset on the $b$-labeled switch. Then, the difference between any clock $x_e$ and such a "location clock" $y_l$ is the exact value of $x_e$ when the location $l$ was entered. Indeed, when the location is entered, the valuation of $y_l$ is 0 as the clock has just been reset. Given that the difference between the two clocks remains a constant while $l$ remains the current location, $(x_e - y_l)$ is the clock valuation of $x_e$ when $l$ was entered.

Using this technique to capture clock valuations at a time a location is entered, the second step of the mapping can be done by rewriting the guards of every switch whose source location offers $\varepsilon$-labeled switches. The rewriting must take care of allowing normal (i.e., non $\varepsilon$-labeled switches) to recognize input symbols when there is no conflict with $\varepsilon$-labeled switches and deactivate them otherwise. To this purpose, we will introduce new clock constraints to capture *when* a given switch is allowed with respect to the guard of a $\varepsilon$-labeled switch.

Going back to the example of Figure 5 with a new clock $y_{l_2}$ having been added, observe that $g_1$ must enable the other switches in the following two cases: $(x_{T_2} < 5)$, and $(x_{T_2} > 5) \wedge (x_{T_2} - y_{l_2} > 5)$. While the first case is rather easy (i.e., $l_2$ is entered before the equality clause has been satisfied), the second case uses the valuation of $x_{T_2}$ when $l_2$ was entered. $(x_{T_2} - y_{l_2} > 5)$ is only true if $l_2$ was entered when $(x_{T_2} > 5)$ was true.

In a similar manner, $g_2$ enables the other switches in the following cases: $(x_{T_1} < 10)$, $(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} > 10)$, $(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} \leq 10) \wedge (x_{T_2} - x_{T_1} \leq -8)$, and $(x_{T_1} = 10) \wedge (x_{T_2} - x_{T_1} \leq -8)$. The first two cases are similar to the case of $g_1$. The third case enables the other switches after the equality clause of $g_2$ has been verified if the clause $(x_{T_2} > 2)$ is false when $(x_{T_1} = 10)$ is true. Indeed, $(x_{T_2} - x_{T_1} \leq -8) = (x_{T_2} - x_{T_1} \leq 2 - 10)$ and when $x_{T_1} = 10$, this reduces to $(x_{T_2} \leq 2)$ which is the negation of $(x_{T_2} > 2)$. Hence the clause $(x_{T_2} - x_{T_1} \leq -8)$ is able to check when $g_2$ cannot be satisfied. Finally the fourth case is similar as it enables the other switches when the equality clause is satisfied if the rest of $g_2$ cannot be completely satisfied.

The correct mapping of $\mathcal{P}$ to $A$ is given on Figure 6, where the "permits" constraints are just the cases that we mentioned above. For example, $\text{permits}(g_1) = (x_{T_2} < 5) \bigvee x_{T_2} > 5) \wedge (x_{T_2} - y_{l_2} > 5)$.

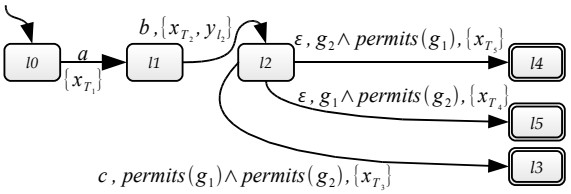



Figure 6: A sample protocol timed automaton that does enforce M-Invoke semantics.

### 4.4 Enforcing M-Invoke constraints in protocol timed automata

To enforce M-Invoke semantics, the following behavior needs to be captured (examples are taken from Figure 5).

1. When a $\varepsilon$-labeled switch guard becomes satisfied, all other switches must be immediately disabled so as to make it the only switch that can be taken. An example is when $g_1$ becomes satisfied in $l_2$: the two other switches must be disabled.
2. When a location offering a $\varepsilon$-labeled switch is entered after the equality clause of its guard has been satisfied, the other switches must not be disabled. For example, if $l_2$ is entered while $x_{T_2} > 5$, the other switches must not be disabled.
3. When the guard of a $\varepsilon$-labeled switch cannot be satisfied when its equality clause is satisfied, the other switches must not be disabled. Let us consider $g_2$ while the current location is $l_2$. In case $(x_{T_1} = 10)$ is satisfied but $(x_{T_2} > 2)$ is not, the two other switches must not be disabled.

The enforcement is done by rewriting the constraints using two new ones. First, we define a constraint "inhib" over a $\varepsilon$-labeled transition guard of the form $g = (x = k) \wedge g'$. The role of this constraint is to capture the cases where $g'$ is false, thus making the switch that has $g$ as its guard inactive. As we will see, this constraint plays a critical role in enforcing the M-Invoke constraints in protocol timed automata. Then, we will provide another constraint, called "permits", also defined over a $\varepsilon$-labeled transition guard. It defines when it allows other switches from the same source location to become actionable. This constraint relies on the introduction of clocks that are attached to locations so as to capture the clocks valuation when locations are entered. It uses "inhib" that we introduce in the following definition.

**Definition 4.2** (inhib constraint)**.** *Let a guard $g := (x = k) \wedge g'$ of a $\varepsilon$-labeled switch defined over a $\varepsilon$-labeled switch $l \rightarrow l'$ such as $x$ is a clock over $\mathbb{T} = \mathbb{R}_{\geq 0} \cup \{\perp\}$, $k$ is a constant in $\mathbb{Q} \cup \{\perp\}$ and $g'$ is any clocks constraint: $g' = (x_1 \#_1 k_1) \wedge \cdots \wedge (x_j \#_j k_j) \wedge (x_{j+1} - x'_{j+1} \#_{j+1} k_{j+1}) \wedge \cdots \wedge (x_m - x'_m \#_m k_m)$ (for any $i \in \{1, \cdots, m\}$: $x_i, x'_i \in X \cup Y$, $k_i \in \mathbb{Q} \cup \{\perp\}$, $\#_i$ is any comparison operator) and $Y$ is the set of clocks that we add to record when a location is entered by resetting them on the switches that are incoming to a given location.*

*We define the constraint "*inhib*" such that:* $\text{inhib}(g) = (x_1 - x \ \texttt{not}(\#_1) \ k_1 - k) \vee \cdots \vee (x_j - x \ \texttt{not}(\#_j) \ k_j - k) \vee (x_{j+1} - x'_{j+1} \ \texttt{not}(\#_{j+1}) \ k_{j+1}) \vee \cdots \vee (x_m - x'_m \ \texttt{not}(\#_m) \ k_m)$

*In the case where $g'$ is not defined (i.e., $g = (x = k)$), then* $\text{inhib}(g) = \texttt{false}$.

Going back to the timed automaton of Figure 5: $\text{inhib}(g_1) = \texttt{false}$ and $\text{inhib}(g_2) = (x_{T_2} - x_{T_1} \leq -8)$. Without loss of generality, we chose to reduce the $\varepsilon$-labeled switch guard $g$ to a unique conjunction in the previous definition to simplify the notations. The case where $g$ is a disjunction is easy: we consider it as multiple $\varepsilon$-labeled switches with each switch having a single conjunctive guard. We keep this assumption in the remainder.

With this "inhib" constraint at hand, we can now define a constraint called permits. When given the guard of a $\varepsilon$-labeled switch, it defines when the other switches from the same source location can be enabled without contradicting M-Invoke constraints.

**Definition 4.3** (permits constraint)**.** *Let a guard $g := (x = k) \wedge g'$ of a $\varepsilon$-labeled switch defined over a $\varepsilon$-labeled switch $l \rightarrow l'$ such as $x$ is a clock over $\mathbb{T}$, $k$ is a constant in $\mathbb{Q} \cup \{\perp\}$ and $g'$ is any clocks constraint. Let $y \in Y$ the clock that is commonly reset by all the switches whose target location is $l$. We define the following clauses: $S_1 = (x < k)$, $S_2 = (x > k) \wedge (x - y > k)$, $S_3 = (x > k) \wedge (x - y \leq k) \wedge \text{inhib}(g)$, and $S_4 = (x = k) \wedge \text{inhib}(g)$.*

*The constraint* permits$(g)$ *is defined as* $\text{permits}(g) = \bigvee_{i \in \{1,2,3,4\}} S_i$

The four permits clauses $S_i$, with $i \in [1, 4]$, play the following roles. $S_1$ captures the cases where the current clocks valuation $v$ ensures that $v(x)$ is still below $k$. $S_2$ captures the cases where $v$ is above $k$ and the location $l$ has been entered after $(x = k)$ was satisfied. This is checked through the clause $(x - y > k)$. $S_3$ captures the cases where $l$ was entered before $(x = k)$ was satisfied, but $g'_i$ could not be satisfied. In such cases, the switches should not be disabled for the clocks valuations such that $(x > k)$ is satisfied. Finally, $S_4$ captures the cases where $(x = k)$ is satisfied but $g'_i$ is not, hence the switches don't have to be disabled as well. Again considering the timed automaton of Figure 5:

$\text{permits}(g_1) = \underbrace{(x_{T_2} < 5)}_{S_1} \bigvee \underbrace{(x_{T_2} > 5) \wedge (x_{T_2} - y_{l_2} > 5)}_{S_2}$, and

$\text{permits}(g_2) = \underbrace{(x_{T_1} < 10)}_{S_1} \bigvee \underbrace{(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} > 10)}_{S_2}$

$\underbrace{\bigvee (x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} \leq 10) \wedge \underbrace{(x_{T_2} - x_{T_1} \leq -8)}_{\text{inhib}(g_2)}}_{S_3} \bigvee \underbrace{(x_{T_1} = 10) \wedge \underbrace{(x_{T_2} - x_{T_1} \leq -8)}_{\text{inhib}(g_2)}}_{S_4}$. We can now define how the guards of the switches whose source locations offer $\varepsilon$-labeled switches need to be rewritten so as to enforce M-Invoke .

**Definition 4.4** ( M-Invoke enforcement)**.** *Let $l$ be a location of a protocol timed automaton $A$ that offers $n > 0$ $\varepsilon$-labeled switches: $\{e_{\varepsilon_1} = (l, g_{\varepsilon_1}, \varepsilon, r_1, l_1), \cdots, e_{\varepsilon_n} = (l, g_{\varepsilon_n}, \varepsilon, r_n, l_n)\}$*

*The rewriting of the guard of each switch whose source location is $l$ (including the $\varepsilon$-labeled ones) is performed as follows:*

1. *for each location $l$ that offers a $\varepsilon$-labeled switch, augment the reset set of each switch whose target location is $l$ with the clock $y_l \in Y$*
2. *compute* $\{\text{permits}(g_{\varepsilon_1}), \cdots \text{permits}(g_{\varepsilon_n})\}$
3. *rewrite the guard $g$ of each switch $(l, g, a, r, l')$ as*
   (a) *when $a \neq \varepsilon$, $g = \bigwedge_{0 \leq i \leq n} \text{permits}(g_{\varepsilon_i})$*
   (b) *when $a = \varepsilon$ and the switch guard is $g_{\varepsilon_j}$ $(j \in \{1, \cdots, m\})$, $g = \bigwedge_{0 \leq i \neq j \leq n} \text{permits}(g_{\varepsilon_i})$*

As an example, we consider again the protocol timed automaton of Figure 5 that has been fixed to enforce M-Invoke semantics on Figure 6.

## 4.5 Correctness of the mapping

The following theorem states the correctness of the mapping. It shows that a timed protocol and its corresponding timed automaton (produced by the mapping described above) have equivalent behaviors (they both recognize the same set of timed words).

**Theorem 4.5.** *Let $A$ be a protocol timed automaton produced by a mapping of a timed protocol $\mathcal{P}$. Then, $Traces(\mathcal{P}) = \mathcal{L}(\mathcal{A})$.*

This theorem states that the set $Traces(\mathcal{P})$ of *timed traces* that are compliant with a protocol $\mathcal{P}$ is exactly the same as the language $\mathcal{L}(\mathcal{A})$ that is recognized by the corresponding timed automaton $A$. Hence, the mapping proposed in the previous section *preserves the semantics* of the timed protocols.

Proof of Theorem 4.5 is quite straightforward in the particular case where the original timed protocol $\mathcal{P}$ does not contain any implicit transitions. Therefore, the main point to prove Theorem 4.5 is to show that the guard rewritings presented in the previous section capture the M-Invoke constraint semantics correctly. The following lemma shows that the function inhib works as expected, i.e., it can inhibit guards when a $\varepsilon$-labeled switch guard is totally satisfied, and allow them when it is not.

**Lemma 4.6.** *Let a protocol timed automaton $A$ and a location $l \in A$ such that there exists a switch $e = (l, g = (x = k) \wedge g', \varepsilon, r, l')$, with $g' = (x_1 \ \#_1 \ k_1) \wedge \cdots \wedge (x_j \ \#_j \ k_j) \wedge (x_{j+1} - x'_{j+1} \ \#_{j+1} \ k_{j+1}) \wedge \cdots \wedge (x_m - x'_m \ \#_m \ k_m)$ for $m \in \mathbb{N}$ and $j \in \{1, \cdots, m\}$. Then: **(i)** $(\text{inhib}(g) = \texttt{true}) \Longrightarrow (g' = \texttt{false})$, and **(ii)** $(\text{inhib}(g) = \texttt{false}) \Longrightarrow (g' = \texttt{true})$.*

Given a location that has several $\varepsilon$-labeled switches, the following lemma checks that only one of them can ever become satisfied, thus disabling and forcing the transition to another location. To do that, we express the following sanity-check type of boolean implication.

**Lemma 4.7.** *Let a protocol timed automaton $A$ and a location $l \in A$ such that it offers $n > 0$ $\varepsilon$-labeled switches. For any $i \in \{1, \cdots, n\}$, the guard $\widetilde{g_i}$ of the $i$-th $\varepsilon$-labeled switch is of the form $g_i \bigwedge_{1 \leq i \neq j \leq n} \text{permits}(g_j)$. Let $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, n\}$ such that $i \neq j$. Then: $(g_j = \texttt{true}) \Longrightarrow (\text{permits}(g_j) = \texttt{false}) \wedge (\text{permits}(g_i) = \texttt{true})$*

This implication expresses the fact that when a M-Invoke guard is satisfied, then its derived permits constraint is false, and the permits clauses of every other M-Invoke guards are still true. Indeed, if any of the later permits clauses was to be false, then it would mean that its guard would have already been actionable in the past, yet not taken and hence not enforced.

The following lemma states that the mapping of a timed protocol to a protocol timed automaton is correct with respect to M-Invoke constraints.

**Lemma 4.8.** *Let a protocol timed automaton $A$ obtained from a timed protocol $\mathcal{P}$. Every $\varepsilon$-labeled switch $e = (l, g = (x = k) \wedge g', \varepsilon, r, l')$ of $A$ is taken as soon as its guard $g$ is satisfied.*

## 4.6 Characterization of protocol timed automata

The previous sections described how a timed protocol can be mapped to a particular class of timed automata called protocol timed automata. In this section, we leverage the formal framework of timed automata to investigate the computational properties (i.e., decidability and closure properties) of the protocol operators. As explained in previous sections, the operators are very useful to perform compatibility and replaceability analysis.

We start by stating the following result regarding the intersection operator.

**Theorem 4.9.** *The class of protocol timed automata is closed under intersection.*

This theorem is derived from existing results in the literature (e.g., [2]). Indeed, it is known that timed automata with $\varepsilon$ transitions are closed under intersection (and hence also under compatible composition). It is easy to extend this result to protocol timed automata (i.e., to show that the intersection or compatible composition of two protocol timed automata is still a protocol timed automata). The next section describes a procedure that implements such an intersection operator.

Let us now turn our attention to the complementation operator which plays a critical role when it comes to characterizing the protocol difference and subsumption operators. While the case of $\|^{\mathtt{TI}}$ and $\|^{\mathtt{TC}}$ was easy to deal with, the complementation operator poses more challenges. Indeed, in the general case, non-deterministic timed automata extended with $\varepsilon$-labeled switches are not closed under complementation [4]. In our case, since we consider deterministic automata, this negative result is mainly due to the presence of $\varepsilon$-labeled switches. [16] investigated the expressive power of $\varepsilon$-labeled switches and identified cases where $\varepsilon$-labeled switches can be removed without a loss of expressiveness (e.g., case of $\varepsilon$-labeled switches that do not reset clocks). Unfortunately, this result is of no use in our case as the $\varepsilon$-labeled switches that we deal with do not belong to the identified cases. In fact, as stated by the following theorem, $\varepsilon$-labeled switches strictly increase the expressiveness of protocol timed automata and hence they cannot be removed without losing in expressiveness.

**Theorem 4.10.** *$\varepsilon$-labeled switches strictly increase the expressiveness of protocol timed automata.*

The proof of this theorem, based on the notion of *precise actions* introduced in [16], is presented in the appendix. Despite these negative results, we are still able to prove that the class of protocol timed automata is closed under complementation. The cornerstone of the proof is to show that, although protocol timed automata include $\varepsilon$-labeled switches, they still exhibit a *deterministic behavior* which ensures that at each step of an execution, all clock values are solely determined by the input word.

**Lemma 4.11.** *Protocol timed automata behave deterministically: given a protocol timed automaton $A$ and a timed word $w \in \mathcal{L}(A)$, $w$ has exactly one run over $A$.*

This result is a key for deriving the following theorem.

**Theorem 4.12.** *The class of protocol timed automata is closed under complementation.*

A procedure which extends the usual complementation construction to the case of protocol timed automaton is described in the next section. The proof of this theorem, provided in the appendix, is based on the observation that the proposed procedure preserves the M-Invoke constraints in timed automata as well as determinism. Thanks to this, a word that is rejected by a protocol timed automaton $A$ is necessarily accepted by its complement $\overline{A}$.

The closure of protocol timed automata under intersection and complementation allows to derive the following results regarding the timed protocol operators.

**Corollary 4.13.** *Timed protocols are closed under $\|^{\mathtt{TI}}$, $\|^{\mathtt{TC}}$ and $\|^{\mathtt{TD}}$.*

The result on the intersection and parallel composition operators is straightforward since both $\|^{\mathtt{TI}}$ and $\|^{\mathtt{TC}}$ operators are based on the intersection using a different matching of the messages depending on their polarities:

- in the case of $\|^{\mathtt{TI}}$, two messages match when they have the same name and polarity (e.g., $a(+)$ and $a(+)$)

- in the case of $\|^{\mathtt{TC}}$, two messages match when they have the same name but a different polarity (e.g., $a(+)$ and $a(-)$).

The result on the difference operator comes from the closure of protocol timed automata under both intersection and complementation. Indeed, $\mathcal{P}_1 \|^{\mathtt{TD}} \mathcal{P}_2$ is equivalent to $\mathcal{P}_1 \|^{\mathtt{TI}} \overline{\mathcal{P}_2}$, hence timed protocol are also closed under difference.

**Corollary 4.14.** *The timed protocol comparison operators $\sqsubseteq$ and $\equiv$ are decidable.*

This comes from the closure under complementation and intersection as well as from the decidability of the reachability problem [2] (i.e., checking whether $L(A_1) \subseteq L(A_2)$ is equivalent to checking whether $L(A_1 \cap \overline{A_2}) = \emptyset$ or not).

With the results presented above, we have proved that our full set of operators can be implemented by exploiting already-known constructs on timed automata [2] and also by establishing new results regarding the novel class of timed automata that we have identified. This makes it possible to conduct automated analysis for all of the compatibility and replaceability classes on timed protocols.

# 5 Protocol operators algorithms

In this section, we give the procedures that implement the intersection and complementation operators of protocol timed automata. The procedure for the other operators (compatible composition, subsumption and equivalence) can be straightforwardly derived from these ones. The proposed procedures extend the existing constructions given in [2] to maintain closure in protocol timed automata.

## 5.1 Intersection of protocol timed automata

The protocol timed automata intersection procedure extends the classical construction on timed automata [2], which in turns already extends the construction on (untimed) automata [35]. In the following steps, we removed the existing permits clauses from the automata guards as new ones are being computed.

Given two protocol timed automata $A_1 = (\Sigma_1, L_1, L_1^0, L_1^f, X_1 \cup Y_1, E_1)$ and $A_2 = (\Sigma_2, L_2, L_2^0, L_2^f, X_2 \cup Y_2, E_2)$, the intersection $A_3 = A_1 \cap A_2$ (with $A_3 = (\Sigma_3, L_3, L_3^0, L_3^f, X_3 \cup Y_3, E_3)$) is built through the following steps.

1. The locations are $L_3 = L_1 \times L_2$, the initial location is $L_3^0 = (L_1^0, L_2^0)$ and the final locations are $L_3^f = \left\{(l_1, l_2) \mid l_1 \in L_1^f,\ l_2 \in L_2^f\right\}$.
2. Two switches $e_1 = (l_1, g_1, a_1, r_1, l_1') \in A_1$ and $e_2 = (l_2, g_2, a_2, r_2, l_2') \in A_2$ are synchronized if and only if $a_1 = a_2 \neq \varepsilon$, producing a new switch $e_1e_2$ which is added to $A_3$: $e_1e_2 = ((l_1, l_2), g_1 \wedge g_2, a_1, \{x_{e_1e_2}\}, (l_1', l_2'))$ (this introduces a new clock $x_{e_1e_2}$ in $A_3$).
3. $\varepsilon$-labeled switches are first added to $A_3$ with their guard being freed of permits clauses. We consider their guards to be disjunction-free (i.e., a $\varepsilon$-labeled switch whose guard is disjunctive is equivalent to several $\varepsilon$-labeled switches with conjunctive guards). For each pair of $\varepsilon$-labeled switches $e_1 = (l_1, (x_1 = k_1) \wedge g_1, \varepsilon, r_1, l_1') \in A_1$ and $e_2 = (l_2, (x_2 = k_2) \wedge g_2, \varepsilon, r_2, l_2') \in A_2$, we add the following switches to $E_3$:

$$\begin{cases} e_1e_\varepsilon = & ((l_1, l_2), (x_1 = k_1) \wedge g_1 \wedge ((x_2 \neq k_2) \vee \neg g_2), \varepsilon, \{x_{e_1e_\varepsilon}\}, (l_1', l_2)) \\ e_\varepsilon e_2 = & ((l_1, l_2), (x_2 = k_2) \wedge g_2 \wedge ((x_1 \neq k_1) \vee \neg g_1), \varepsilon, \{x_{e_\varepsilon e_2}\}, (l_1, l_2')) \\ e_1e_2 = & ((l_1, l_2), (x_1 = k_1) \wedge (x_2 = k_2) \wedge g_1 \wedge g_2, \varepsilon, \{x_{e_1e_2}\}, (l_1', l_2')) \end{cases}$$

4. With the set of clocks in $A_3$ being $X \cup Y$ as per definition, make sure that for each location $l$ offering at least one $\varepsilon$-labeled switch, a clock $y_l \in Y$ is reset on all of the incoming switches to $l$.
5. For each location $l \in A_3$, compute the permits clauses.
6. The guards in $A_3$ need to be rewritten to refer to the clocks of the switches of $A_3$ as they still refer to those of $A_1$ and $A_2$ at this step. A map is maintained between each clock $x_e$ of $A_1$ or $A_2$ and the set of clocks $\{x_{e,e1}, x_{e,e2}, x_{e3,e}, \cdots\}$ that correspond to the switches $\{(e, e1), (e, e2), \cdots, \}$ that were generated from $e$. Given a guard $g$ of a switch in $A_3$, a clause $(x_e \ \# \ k)$ of $g$ is rewritten as a disjunction $(x_{e,e1} \ \# \ k) \vee (x_{e,e2} \ \# \ k) \vee \cdots$. Diagonal constraint clauses in $g$ are also rewritten in a similar fashion using the mappings of its two clocks.

Compared to the classic timed automata intersection procedure, the protocol timed automata intersection has the following differences.

1. Clocks assignment remains "under control" to match the protocol timed automata requirement of having at most two clocks reset per transition. The classical timed automata intersection construction would simply combine the set of clocks from both input timed automata and merge the clocks in the reset sets of each switch.
2. M-Invoke semantics are enforced in the intersection by computing new permits clauses (the permits clauses of the input timed automata guards are discarded).

### 5.2 Complementation of protocol timed automata

We compute the complement of a protocol timed automaton using the following procedure which is derived from the one for deterministic timed automata as given in [2], with the difference lying in the presence of $\varepsilon$-transitions.

Given a protocol timed automaton $A$, we denote by $A^*$ its *complete* automaton which is build as follows.

1. A location $q$ is added to $A^*$ whose role is to act as a *rejection* location: given any timed word $w$ defined over $\overline{\mathcal{L}(A)}$, the execution of $w$ over $A^*$ goes to the location $q$ as soon as an input symbol yields to a word which is not in $\mathcal{L}(A)$. Hence, any timed word $w$ defined over the alphabet of $A$ has a (unique) execution over $A^*$.
2. For each location $l$ of $A$ (this includes $q$) and for each word $a$ of the alphabet, a transition $e = \left(l, \left(g \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i})\right), a, \{x_e\}, q\right)$ is added where:
   (a) $g$ is defined as the negation of the disjunctions[1] of the guards[2] of the other $a$-labeled transitions from $l$, and
   (b) each $g_{\varepsilon i} = (x_i = k_i) \wedge g'_{\varepsilon i}$ appears in the guard of the $i$-th $\varepsilon$-labeled switch from $l$, given that $l$ offers $n \geq 0$ of such switches.

As in [2], the complement $\overline{A}$ of $A$ is deduced from $A^*$ by inverting the final and the normal locations due to the fact that every timed word $w \in \mathcal{L}(A)$ has a unique run over $A$.

## 6 Implementation and usage

In the sections above, we have exposed the timed business protocols model and analysis techniques. We now turn our attention to the validation of our approach. We first present

[1] e.g., given 2 $a$-labeled switches with guards $g_1$ and $g_2$, $g = \neg(g_1 \vee g_2)$

[2] For each guard, we do not take into account the clauses that are obtained through the permits constraint.

our prototype platform, and then we show a typical usage scenario of the tool to show how it supports service development.

### 6.1 Prototype: the *ServiceMosaic Protocols* project

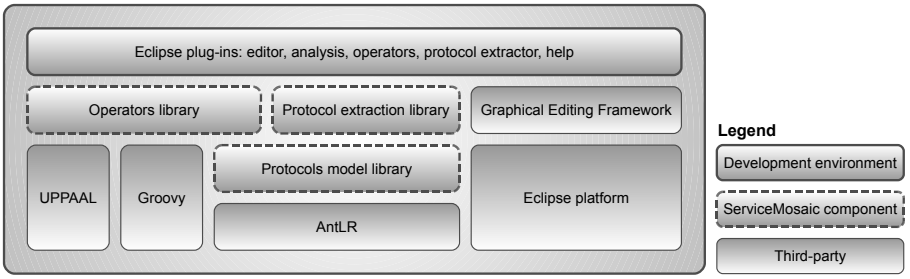



Figure 7: Components of the ServiceMosaic protocol development and analysis prototype.

The concepts discussed in this paper have been implemented in *ServiceMosaic*, a CASE platform for supporting the service development lifecycle that includes facilities for modeling, analyzing, discovering, and adapting web service models [14]. We are releasing most tools at http://servicemosaic.isima.fr/downloads/protocols/ under the terms of the open-source LGPLv3 license. The ServiceMosaic tools are developed for the Java™ platform version 5. Specifically, we created libraries that provide the functionalities of our contributions (e.g., protocol operators) and we integrated them into the Eclipse platform (see http://www.eclipse.org/) as plug-ins (e.g., a plug-in provides the graphical user interface for applying the protocol operators).

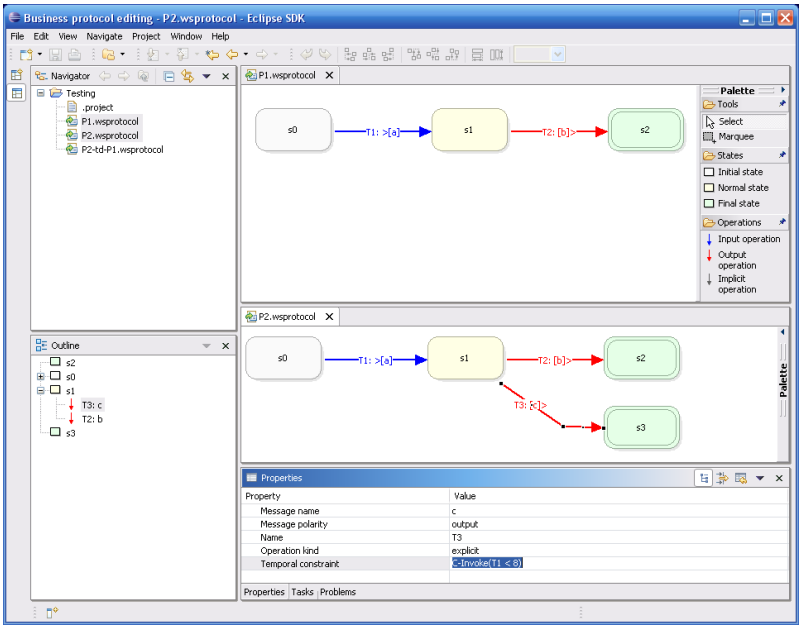



Figure 8: Screenshot of the ServiceMosaic protocol development and analysis prototype.

The components that we have developed for this work are depicted on Figure 7. We have created a library containing the object model of timed protocols. The AntLR parser generator (see http://www.antlr.org/) has been used to parse C-Invoke and M-Invoke constraints from strings representations. We have also created a timed protocol operators

library. All of the operators (intersection, parallel composition, difference, subsumption and equivalence) have been implemented in Java$^{TM}$ . The subsumption and equivalence operators rely on the external UPPAAL timed automata model checker [6], as they require testing for language emptiness. As a requirement of the subsumption operator definition, we have also implemented a complementation operator. The protocol extraction library relies on the protocols model library to extract the web services choreographies from a BPEL process. Figure 8 provides a screenshot of the prototype.

## 6.2 Protocol analysis at work

We now show how the prototype and protocol analysis approach can be used to facilitate service development on the following scenario. The scope of applications of protocol analysis goes however beyond just this example as we will see in the next section. We assume here that a developer is defining a BPEL process, related to the handling of a purchase order, and that the process invokes several services during its execution. The tool will assist the developer in checking if the selected services have a protocol which is fully or partially compatible with the defined BPEL process, will identify which conversations can and cannot be carried out, and will also tackle the case of non compatibility by supporting the development of protocol adapters.

### 6.2.1 BPEL process outline

Consider the BPEL process depicted on Figure 9. It orchestrates four web services to process a purchase order. For the sake of clarity, we have removed the *assign* BPEL instructions from the process diagram, normally required to prepare and reuse the messages exchanged with the involved web services. The first part of the process handles the payment options. If the customer asks for a loan, then the process will make an offer using the *accounting* web service. The customer can then accept or reject it. The asynchronous *pick* BPEL construction defines an alarm that will be fired after 72 hours to discard the process instance if the customer does not reply in time to the loan offer. The second part checks for the ordered goods availability with the warehouse web service. If some goods are not available, they will be ordered. In order to match quality of service requirements, the purchase is canceled if the warehouse does not manage to purchase the missing goods within 48 hours. The third an last part of the process handles the payment and prepares the goods delivery. Finally, the customer is notified that the purchase has successfully completed.

### 6.2.2 Business protocols extraction

Based on this BPEL process definition, we extract the timed protocols that the process supports when interacting with its partner services. To do this, we use our multi-party protocol BPEL extractor, and we then obtain the protocol governing the interaction of the process with each of the partner services by filtering the multi-party protocol based on each service partner link. The extractor, developed as part of the ServiceMosaic project, takes a BPEL process as its input and then outputs a *multi-party protocol*, an extension of a timed protocol where a message is also tagged with the *partner link* of the service which is sending or receiving the message. As such, a multi-party business protocol captures the message *choreography* of a BPEL process orchestrator. First, we identify *protocol extraction patterns* for each type of basic and complex BPEL activity. The extraction starts from the beginning of the process and goes through each activity to apply the protocol

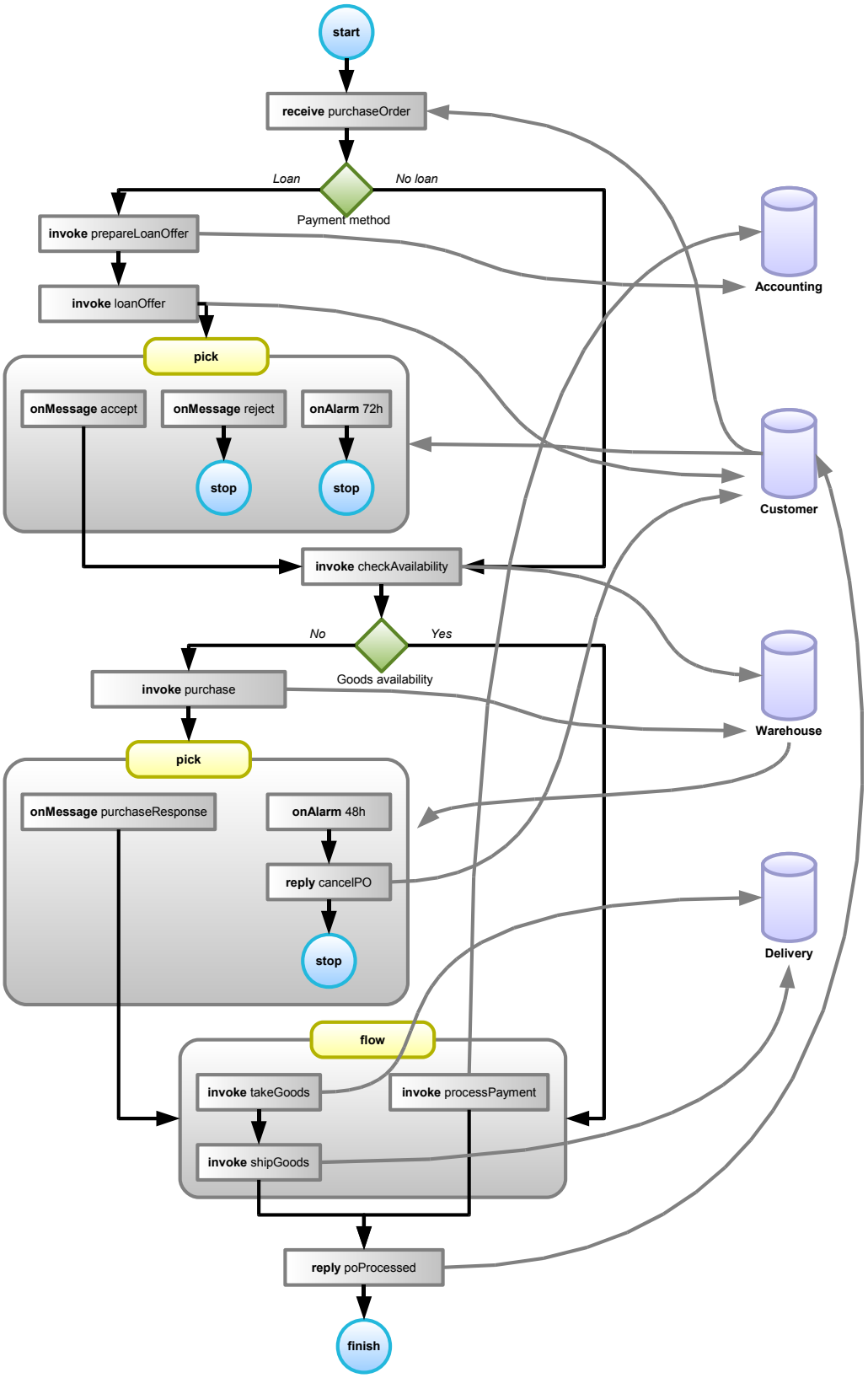



Figure 9: A BPEL process that handles purchase orders.

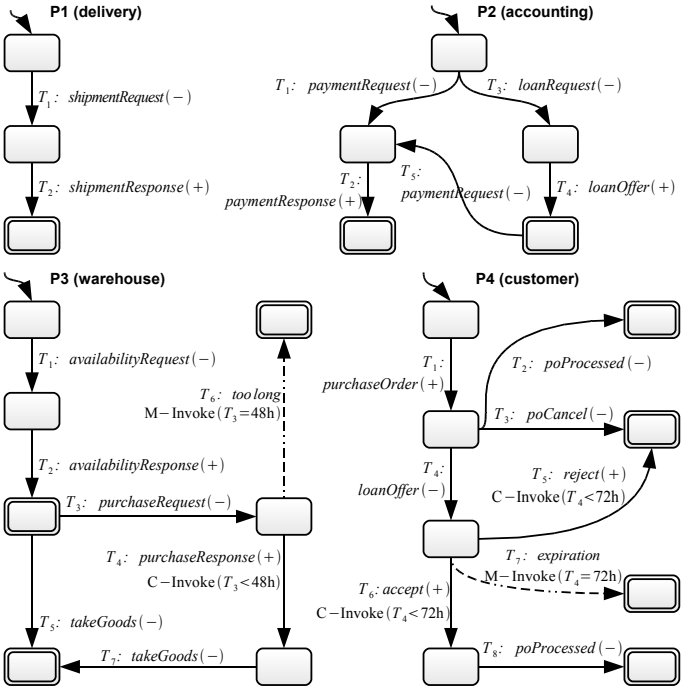



Figure 10: Timed protocols extracted from the BPEL process of Figure 9.

extraction patterns as they are recognized. When a complex activity is encountered (e.g., *if*, *switch*, *while*, *pick*, ...), it is recursively processed on each of its complex activities until basic activities are reached. The obtained protocol fragments are assembled by inverse recursion. For instance, if a *if* activity comprises one *invoke* on each alternative branch, then a protocol fragment is derived from each *invoke*, then they are combined as different branches from the current state in the extraction process.

The protocol which is followed by the process while interacting with a given service (identified by its BPEL *partner link*) can be obtained as follows. The idea is to perform a special form of filtering on the multi-party protocol. In a similar fashion as *projection* for timed automata [4], we replace the messages with $\varepsilon$ on the transitions that are not associated with the partner link of the service that we are interested in. Also, each temporal constraint that refers to a transition which is not from the target partner link is removed. Indeed, they do not make sense in the protocol that we want to obtain since they refer to events that are not "seen" by the orchestrated service. Finally, the service protocol is obtained by removing the $\varepsilon$ transitions using standard techniques on automata [35]. This is possible only because if we mapped to timed automata, there would be no guard nor clock resets on these transitions [31]. While still experimental, we have found out that the protocol extraction operator works well for a large majority of BPEL processes

In this scenario, the resulting protocols are shown in Figure 10. Figure 11 shows instead the protocol of the warehouse service we are planning to use as one of the services invoked by our process. Note that this transformation is not reversible. When generating a protocol, we only care about possible ordering of messages and not about the many details prescribed by a BPEL process (such as why – based on which condition – a certain path is chosen). Nevertheless, we had developed developed techniques for generating service implementation templates in BPEL from protocols definitions [5].

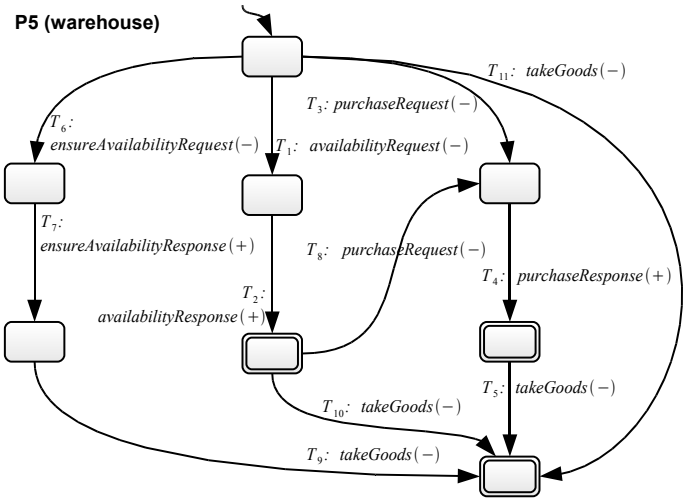



Figure 11: The complete warehouse service protocol.

### 6.2.3 Protocol analysis

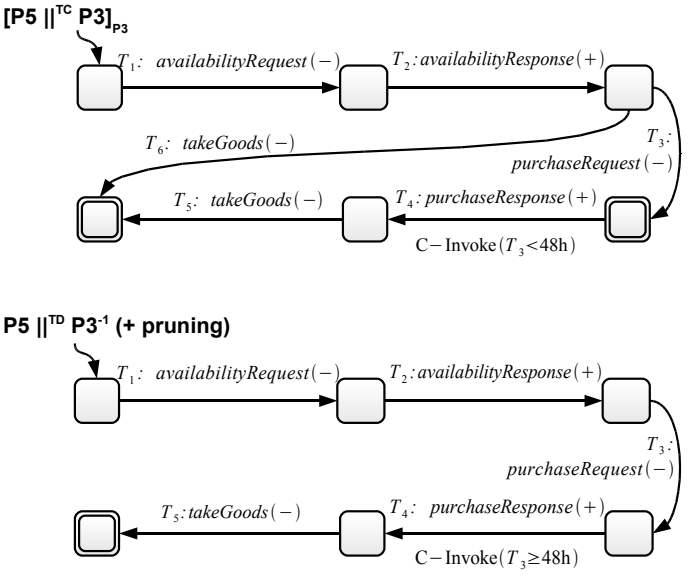



Figure 12: Analysis of the common and differing conversations supported by $P_3$ and $P_5$.

We next apply the protocol analysis operators to assess compatibility between the protocols supported by our process and the protocols of the services we plan to use. For this, we assume that either the protocol or BPEL definition (from which we extract the protocol) of these services is available. Figure 12 shows the results of this analysis for the warehouse service. In particular, the compatible composition operator $P_5 \parallel^{\mathtt{TI}} P_3$ gives the set of the conversations that can occur between protocols $P_3$ and $P_5$. Ideally, we would want this set to be equal to the conversations supported by $P_3$, which means that $P_5$ is fully compatible with $P_3$.

However, in our example, we do not have such luck. In fact we see that the conversations supported by the compatible composition are a subset of those supported by $P_3$. The figure further shows the conversations that are supported by the process but not by

our partner service $P_5$ (which is empty in case of full compatibility), as well as the conversations that the partner supports but that the process does not support. The first of these two combined protocols is obtained by computing the inverse $P_3^{-1}$ of $P_3$ and then the difference $P_3^{-1} \parallel^{\mathtt{TD}} P_5$. The latter is instead computed as $P_5 \parallel^{\mathtt{TD}} P_3^{-1}$. As we will examine later, all these combined protocols will become helpful in examining if and which changes need to be made to the process.

In particular, while the first combined protocol of Figure 12 (compatible composition) tells us what we can do, the second one denotes what our process is prevented from doing when using this partner (hence we call these *prevented interactions*), while the third one denotes conversations that the partner would support, but we are not leveraging due to how we implemented the process. We call these *neglected interactions*.

It is interesting to note that no compatibility problem would have been spotted in the case of business protocols without timing constraints [12]. Indeed, the untimed version of $P_5$ would have supported all of the conversations of the untimed version of $P_3$.

### 6.2.4 Managing partial replaceability scenarios

By looking at the three combined protocols, the developer can assess if the selected service is a good fit or not, and how to handle situations of partial replaceability or of no replaceability. In general, this depends on the specific business purpose of the process. For example, the service I am planning to invoke may not support a *cancelPO* operation, but I may be willing to take the risk and use it anyways even if cancellations are not allowed, for example because it offers cheaper rates. Or, conversely, the selected service supports several forms of payments (accessed via different protocols) but my process can only support one of them, and we may be fine with it as for example our company only issues payments via credit card and not via bank transfers.

Alternatively, we can modify the process definition to adapt it to the service we are using, either to:

1. ensure that our process does not generate conversations our partner cannot understand, or
2. leverage conversations supported by our selected services (e.g., extend our process to support bank transfers).

As another example, in our process, we can remove the *onAlarm 48h* handler of the second *pick* complex activity, so that the process will wait for the *purchaseResponse* message to arrive, thereby removing the problematic temporal constraints in the extracted expected warehouse protocol. However, the process may find itself being put on hold indefinitely if a problem occurs on the warehouse service and it does not send a *purchaseResponse* message back.

Another solution is to generate a protocol adapter [7] to reconcile the differences. It can be done with the ServiceMosaic tools using an aspect-oriented framework [37] where adapters are plugged through *advices* written in BPEL. The adapter is developed as follows. The *pointcut* is triggered when a *purchaseRequest* message is received. The advice is a BPEL process where an alarm starts counting from the reception of the *purchaseRequest* message. If the service does not send a *purchaseResponse* withing the next 48 hours, then the adapter drops it when the warehouse service sends it afterwards. The BPEL engine will have already woken up the process instance by then, and taken action by replying to the client partner link with a *cancelPO* message.

Finally, it should be noted that for most BPEL engines, a message is simply dropped when it cannot be dispatched to any process instance for which it is waiting. An exception

is then usually raised and logged inside the BPEL engine. In this example the adapter would be useful for diminishing the number of internally-thrown exceptions (raising exceptions has a significant performance cost). The choice of developing this adapter should be balanced in light of its development cost compared to the (limited) benefits, as BPEL engines can provide a form of "implicit" adapter in very specific mismatches cases such as this one.

# 7 Discussion

We now provide a discussion that includes related work as well as two limitations of our approach.

## 7.1 Related work

### 7.1.1 Timed automata

Many classes and extensions of timed automata have been studied. Deterministic timed automata [2] are known to be closed under complementation. So are event-clock automata [3], a subclass of deterministic timed automata which have the interesting property that every such indeterministic automaton has an equivalent deterministic automaton. Allowing diagonal constraints ($x - y \# c$) [16] make the model more concise but does not add to the expressiveness. Additive constraints ($x + y \# c$) renders the emptiness checking problem decidable for 1 or 2 clocks, open for 3 clocks and undecidable starting from 4 clocks [15]. Non-standard ($x := 0$) clocks resets make this problem decidable for $x := c$ and $x := x + 1$ (if diagonal constraints are not allowed) but undecidable for $x := x - 1$ [22]. The class of *Robust Timed Automata* allows to recognize events with fuzziness w.r.t. dates [4]. Previous work on timed automata augmented with $\varepsilon$-transition had suggested that the class of protocol timed automata would not have been closed under complementation [23, 16, 31]. While the results presented in [31] still hold in the general case where $\varepsilon$-transitions can reset an arbitrary number of clocks and have complex guards on transitions, we have identified a very specific class for which the traditionally "hard" problems become decidable (closure under complementation and language inclusion). The class of *event-recording timed automata* [3] is a subset of deterministic timed automata. They also form a subset of protocol timed automata.

### 7.1.2 Standardization efforts

Standardization organizations (e.g., W3C, OASIS) have tried to provide specifications for describing the external behavior of web services in terms of both choreographies and orchestrations. They build on top of the existing widely used specifications for the static interfaces of web services (e.g., WSDL, XML-Schema). BPEL, WSCL and WSCI are examples of specifications that feature support for describing services conversations [44], although the last two do not seem to have gained much adoption in practice. Our work is complementary to those specifications rather than competing. Choreography specifications (in WSCI or a WSCL extension) can be derived back and forth from timed protocols. BPEL orchestrations can be processed by our protocol extraction tool for obtaining the expected business protocol of each service that it orchestrates.

### 7.1.3 Models for web services

Research on web services has lead to various models being proposed to describe their behavior and/or implementation for the purpose of analysis and execution. A discussion on modeling web services interactions has been proposed in [25] and is further discussed in [26]. An approach for web services interfaces was defined in [21]. A model with similar goals as timed protocols had been introduced in [18], but the timing constraints defined in the model have not been taken into account. A language for web services choreographies called *Chor* has been proposed in [46] as a simplification of WS-CDL. All of these approaches share many similarities with this work and the base model for business protocols of [12]. Surprisingly, little work has been done on timing abstractions.

### 7.1.4 Verification techniques

Many works have tried to apply in various fields verification techniques such as checking for liveness, the absence of deadlocks or the conformance against specifications. A substantial amount of work has been done in the field of workflow systems [1, 29, 20]. In the case of web services, timed automata have been used in [36, 30, 18]. In the case of [18] the WSTL model had been designed with timing constraints as "first-class citizen", but they have never been leveraged. BPEL-based web services interactions have been analyzed in [33] by the mean of guarded automata with unbounded message queues where the automata synchronization problem is studied in synchronous and asynchronous communications. The same types of verifications can be easily done on protocol timed automata using TCTL, an extension of temporal logics for timed automata, and a model checker such as UPPAAL [6]. Such approaches are complementary to this work. Interestingly, we reused and extended results from the field of formal verification (e.g., timed automata), but not for the purpose of doing "classical" model-checking.

### 7.1.5 Compatibility and replaceability

Software components have some fundamental similarities with web services: they promote good practices such as loose coupling and reuse. Also, they can be remotely accessed over a network. Similar approaches for protocols compatibility and replaceability exist in the area of component-based systems [51, 28]. The importance of being able to check for services compatibility or replaceability has lead to several research works [38, 32, 21]. Surprisingly, these approaches do not cater for timing constraints. They also perform "black or white" analysis. By contrast, our approach is able to provide a more fine-grained type of analysis by identifying the "partial cases" like the partial compatibility or the replaceability with respect to a client protocol. We believe that this flexibility will significantly prove to be useful in practice, as full compatibility or replaceability of business protocols can hardly be reached on the Internet which is an open service deployment environment. The notion of process inheritance has been studied in the domain of workflows [50, 27]. It is similar to protocols replaceability. Different types of inheritance relations are proposed in [50]. They provide some flexibility much like we did with the different classes of protocol replaceability. However, these approaches do not consider temporal abstractions.

### 7.1.6 Model management

The work that has been done in the model management area focuses on manipulating models (e.g., database schemas, XML schemas) and matches between them (e.g., equivalence between 2 database schema) on an equal foot [19]. The matches relationships between

2 models can be used for matching, merging or composition purposes. The models can also be manipulated through various operators like the intersection, the union or the difference. A set of combined static and behavioral matching and merging techniques for statecharts-based specifications have been proposed in [40]. This work has been done in a similar fashion as the approaches for schema matching (including databases and XML) mentioned in [47, 19]. The work presented in this paper is largely inspired what has been done in this research field. Indeed, timed protocols are models and we have defined protocol manipulation operators (composition, difference and intersection) as well as comparison operators (subsumption and equivalence) that define matchings between protocols. The *match* operator of [40] could be used to identify compatibility and replaceability between two (untimed) business protocols of [12]. A clear benefit of this approach would be to detect matches when protocols have different messages names but similar semantics (e.g., *login* and *connect*), as in our case the operators match the messages only on name equality. The matches and mismatches could then be exploited to generate protocol adapters [7, 42]. However, it requires human intervention as the heuristic results may contain missing and invalid matches. Also, it does not cater for timing constraints as they require proper analysis techniques as done on timed automata.

## 7.2 Limitations

### 7.2.1 Constraints with absolute dates

Timed protocol constraints are always expressed relatively to a transition of a given protocol being fired (e.g., C-Invoke($T_1 < 3h$)). Absolute dates cannot be used in constraints (e.g., C-Invoke($T_1 <$ '2007-04-19 14:49:00') or C-Invoke(current_time $<$ '2007-04-19 14:49:00')). Such types of constraints can be found in some specifications such as BPEL [43] where both types of *relative* and *absolute* time expressions can be used. Let us briefly investigate the impact of introducing absolute dates into the model by looking at the involved mechanisms at the timed automata level. Allowing a constraint to compare a clock $x$ to a constant *date* (e.g., $x <$ '2007-04-19 14:49:00') which represents an absolute date requires the following assumptions. (i) $x$ is set to a constant *now* which represents the current date when the automaton execution starts, and (ii) $x$ is always compared to absolute dates, and (iii) $x$ is never reset in the considered automaton.

We claim that making such an extension renders the timed language emptiness checking problem undecidable. The proof can be done by observing that *now* is actually a variable. In timed automata, the clocks are set to a constant value (usually 0) when the execution starts. Here, we would have some special clocks that would be initially set to a value which depends on the current time. Hence, the result of checking for the emptiness of such an extended timed automaton would only hold considering the time at which the checking has been performed (i.e., the results holds at time $t$ but may not hold anymore at time $t + \delta$ with any $\delta \in \mathbb{R}_{\geq 0}$).

This limitation of timed protocols in terms of expressiveness is not a penalty as such constructs are of limited use in practice. In the case of BPEL, timers are mainly used for generating timeout exceptions in asynchronous operations (e.g., the *pick* complex activity). They can be also used for a *wait* activity (e.g., put the process in sleep to enforce legal regulations). In our experience, we have never found a need for expressing absolute dates in BPEL processes. Also, *JBoss JBPM* (see http://www.jboss.com/products/jbpm), a widely used business process management system, offers a workflow language called *jPDL* where time-related constructs are always expressed in a relative manner (i.e., *jPDL* does not allow specifying absolute dates for timers).

### 7.2.2 Message transport communications

Finally, our model is based on the assumption that there are no message transmission delays and losses. This is of course not the case in reality as web services messages are mostly transported over unreliable networks that can have substantial load variations, leading to greatly varying network latencies and even errors. As mentioned in related work, the class of *Robust Timed Automata* [4] is a possible exploration path.

## 8 Conclusion

This paper has revisited the concepts presented in earlier work by providing an extended model for web services business protocols that supports timing abstractions. The level of abstraction that drove the design of this model was developed on the grounds of a study of real-world scenarios related to web services. The model can be leveraged for fine-grained protocol compatibility and replaceability analysis based on a set of protocol manipulation and comparison operators. We showed that the decision problems surrounding their implementation are decidable, thanks to the mapping and the identification of a novel class of timed automata which is closed under complementation and for which the language inclusion problem is decidable despite the presence of $\varepsilon$-transitions with clocks resets. We also presented our initial prototype as part of the ServiceMosaic project and gave a case study.

The results presented in this paper will pave the road for an agile web services composition development and management framework. Briefly, this environment will be centered around protocol repositories. They will be queried for compatibility at development time, allowing the rapid-prototyping of service compositions. In turn, they will be queried for replaceability at runtime to handle needs to substitute a service that becomes unavailable or whose protocol has changed. In both development and runtime environments, the framework will provide means to help at generating adapters.

We believe that modeling and analysis techniques with formal foundations such as the ones that we have presented will help at transforming the development and the maintenance of web services based applications from an "art", requiring a substantial amount of manual interventions, to a model-driven process that is automated to a large extent.

## Acknowledgments

We would like to thank the following people for their help: Cécile Maudelonde-Andrieu (RosettaNet PIPs), Sandrine Rivet (multi-party timed protocols extraction from BPEL) and Kevin Hivernat (protocol repositories). We would also like to thank the anonymous reviewers for their accurate and interesting remarks.

## References

[1] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.

[4] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *SFM*, pages 1–24, 2004.

[5] Karim Baïna, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Model-driven web service development. In *CAiSE*, pages 290–306, 2004.

[6] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In *SFM*, pages 200–236, 2004.

[7] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In *CAiSE*, pages 415–429, 2005.

[8] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. Compatibility and replaceability analysis for timed web service protocols. In *BDA*, 2005.

[9] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. On temporal abstractions of web service protocols. In *CAiSE Short Paper Proceedings*, 2005.

[10] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and Management of Web Services Protocols. In *Proceedings of ER 2004. Shanghai, China*, November 2004.

[11] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 08(1):46–54, 2004.

[12] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing web service protocols. *Data Knowledge. Engineering*, 58(3):327–357, 2006.

[13] Boualem Benatallah, Fabio Casati, Farouk Toumani, and Rachid Hamadi. Conceptual modeling of web service conversations. In *CAiSE*, pages 449–467, 2003.

[14] Boualem Benatallah, Fabio Casati, Farouk Toumani, Julien Ponge, and Hamid Reza Motahari Nezhad. Service mosaic: A model-driven framework for web services life-cycle management. *IEEE Internet Computing*, 10(4):55–63, 2006.

[15] Béatrice Bérard and Catherine Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, 75(1-2):1–7, July 2000.

[16] Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundam. Inform.*, 36(2-3):145–182, 1998.

[17] Daniela Berardi. Automatic composition services: models, techniques and tools, phd thesis. Universita di Roma La Sapienza, Roma, Italy, 2002.

[18] Daniela Berardi, Fabio De Rosa, Luca De Santis, and Massimo Mecella. Finite state automata as conceptual model for e-services. *J. Integr. Des. Process Sci.*, 8(2):105–121, 2004.

[19] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, and Christoph Quix. Industrial-strength schema matching. *SIGMOD Rec.*, 33(4):38–43, 2004.

[20] Claudio Bettini, X. Sean Wang, and Sushil Jajodia. Temporal reasoning in workflow systems. *Distrib. Parallel Databases*, 11(3):269–306, 2002.

[21] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *WWW*, pages 148–159, 2005.

[22] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theor. Comput. Sci.*, 321(2-3):291–345, 2004.

[23] Patricia Bouyer, Serge Haddad, and Pierre-Alain Reynier. Undecidability results for timed automata with silent transitions. Research Report LSV-07-12, Laboratoire Spécification et Vérification, ENS Cachan, France, February 2007.

[24] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In *FORMATS*, pages 112–126, 2005.

[25] Tevfik Bultan. Modeling interactions of web software (invited paper). In *WWV 2006*, 2006.

[26] Tevfik Bultan, Jianwen Su, and Xiang Fu. Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[27] Christoph Bussler. Process inheritance. In *Proceedings of CAiSE'02*, volume 2348 of *Lecture Notes in Computer Science*, pages 701–705. Springer, 2002.

[28] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, Jos M. Troya, and Antonio Vallecillo. Adding roles to corba objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.

[29] Elisabetta De Maria, Angelo Montanari, and Marco Zantoni. An automaton-based approach to the verification of timed workflow schemas. In *TIME*, pages 87–94, 2006.

[30] Gregorio Díaz, María-Emilia Cambronero, Juan José Pardo, Valentin Valero, and Fernando Cuartero. Automatic generation of correct web services choreographies and orchestrations with model checking techniques. In *AICT/ICIW*, page 186, 2006.

[31] Volker Diekert, Paul Gastin, and Antoine Petit. Removing epsilon-transitions in timed automata. In *STACS*, pages 583–594, 1997.

[32] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Leveraging eclipse for integrated model-based engineering of web service compositions. In *ETX*, pages 95–99, 2005.

[33] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.

[34] Claude Girault and Rudiger Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[35] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, November 2000.

[36] Raman Kazhamiakin, Paritosh K. Pandya, and Marco Pistore. Timed modelling and analysis in web service compositions. In *ARES*, pages 840–846, 2006.

[37] Woralak Kongdenfha, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. An aspect-oriented framework for service adaptation. In *ICSOC*, pages 15–26, 2006.

[38] Massimo Mecella, Barbara Pernici, and Paolo Craca. Compatibility of e -services in a cooperative multi-platform environment. In *Procs. of TES '01*, pages 44–57, London, UK, 2001. Springer-Verlag.

[39] Hamid R. Nezhad Motahari, Régis Saint-Paul, Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. Servicemosaic: Interactive analysis and manipulation of service conversations. In *ICDE*, pages 1497–1498, 2007.

[40] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve M. Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *ICSE*, pages 54–64, 2007.

[41] Hamid R. Motahari Nezhad, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. Protocol discovery from imperfect service interaction logs. In *ICDE*, pages 1405–1409, 2007.

[42] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[43] OASIS. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, April 2007.

[44] M. P. Papazoglou and D. Georgakopoulos. Special issue on service oriented computing. *Commun. ACM*, 46(10):24–28, 2003.

[45] Julien Ponge, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Fine-grained compatibility and replaceability analysis of timed web service protocols. In *ER*, pages 599–614, 2007.

[46] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, 2007.

[47] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[48] RosettaNet. RosettaNet PIP Directory. http://www.rosettanet.org/, 1996 – 2008.

[49] Ferucio Laurentiu Tiplea and Geanina Ionela Macovei. E-timed workflow nets. In *SYNASC*, pages 423–429, 2006.

[50] W. van der Aalst. Inheritance of business processes: A journey visiting four notorious problems, 2003.

[51] D.M. Yellin and R.E. Storm. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, March 1997.

# A Proof of Lemma 4.10

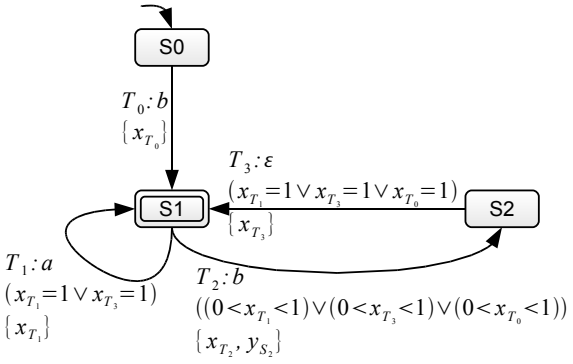



Figure 13: A protocol timed automaton $A$ that cannot be expressed equivalently without $\varepsilon$-transitions.

*Proof.* We need to show that $\varepsilon$-transitions in protocol timed automata cannot always be removed, i.e., there are protocol timed automata for which there doesn't exist equivalent automata without $\varepsilon$-transitions. To do that, we exhibit the protocol timed automaton $A$ depicted on Figure 13 and use the notions of *precise time* and *precise actions* that were introduced in the Theorem 24 of [16] as a tool to identify timed languages that can only be recognized by timed automata featuring $\varepsilon$-transitions. The proof is virtually the same as the one of Corollary 29 in [16].

It is easy to check that $A$ is a protocol timed automaton. $A$ presents 2 $\varepsilon$-transitions lying on directed cycles, hence we don't know if they can be removed using the techniques presented in Section 8 in [16].

Let us now suppose that $\mathcal{L}(A)$ can be recognized by a timed automaton $A'$ without any $\varepsilon$-transition. Note that $A'$ is free of diagonal constraints (e.g., constraints of the form $x - y \# c$). $A'$ can be rendered disjunction-free without any loss of generality (see [16] for techniques and discussion). In order to leverage the Theorem 24 of [16], we define $C_{\max} = 1$ (no constant in the guards of $A$ is larger than 1). Let also $\delta > 0$. $A$ can recognize timed words of the form

$$(b, \delta_1) \cdot (b, \delta_2) \cdots (b, \delta_{d-1}) \cdot (a, d) \cdot (a, d+1) \cdots$$

where $d \in \mathbb{N}$, $d \geq C_{\max}$ and $\delta_i \in (i-1, i) \setminus \delta\mathbb{N}$ for all $0 < i < d$. Let $P$ a path of $A'$ that accepts such a timed word. Given that the $a$-labeled events occur at integer times, their occurrences should be *precise* in $P$. Also, $d \geq C_{\max}$, hence from Theorem 24 of [16], there exist some occurrence of $b$ that should be precise in $P$ which is not possible as $\delta_i \notin \delta\mathbb{N}$ for any $0 < i < d$. Consequently, $\mathcal{L}(A)$ cannot be recognized by a timed automaton without $\varepsilon$-transitions. □

# B Proof of Lemma 4.7

*Proof.* Let us check the implication.

$$(g_j = \texttt{false}) \bigvee (\text{permits}(g_j) = \texttt{false}) \wedge (\text{permits}(g_i) = \texttt{true}) = \texttt{true}$$

is equivalent to

$$(\text{permits}(g_j) = \texttt{true}) \bigvee (\text{permits}(g_j) = \texttt{false}) \wedge (\text{permits}(g_i) = \texttt{true}) = \texttt{true}$$

which reduces to

$$\underbrace{(\text{permits}(g_j) = \texttt{true})}_{\texttt{false as } g_j = \texttt{true}} \bigvee \underbrace{(\text{permits}(g_i) = \texttt{true}) = \texttt{true}}_{\text{permits}(g_i) = \texttt{true}}$$

and the implication is verified. Indeed, $\text{permits}(g_i) = \texttt{true}$, else this would mean that the switch whose guard is $\widetilde{g_i}$ had already been activated. □

# C Proof of Theorem 4.8

*Proof.* Let us compute the cases where $\text{inhib}(g)$ evaluates to `false`. We assume that $y \in Y$ is the clock that is reset on every switch whose target location is $l$. We compute and expand the negation:

$$\begin{aligned}
\neg\,\text{permits}(g) = {} & (x > k) \\
& \wedge ((x \leq k) \vee (x - y \leq k)) \\
& \wedge ((x \leq k) \vee (x - y > k) \vee \neg\,\text{inhib}(g)) \\
& \wedge ((x \neq k) \vee \neg\,\text{inhib}(g))
\end{aligned}$$

we make a first development:

$$
\begin{aligned}
&= ((x = k) \vee (x \geq k) \wedge (x - y \leq k)) \\
&\wedge ((x < k) \vee (x \neq k) \wedge (x - y > k) \\
&\vee (x \neq k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee (x \leq k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee (x - y > k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee \neg \,\mathrm{inhib}(g))
\end{aligned}
$$

and a second one:

$$
\begin{aligned}
&= (x = k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee (x = k) \wedge (x - y > k) \wedge \neg \,\mathrm{inhib}(g) && \texttt{false} \\
&\vee (x > k) \wedge (x - y \leq k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee (x = k) \wedge (x - y \leq k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee (x \geq k) \wedge (x - y \leq k) \wedge \neg \,\mathrm{inhib}(g)
\end{aligned}
$$

by reducing the last 3 disjunctions:

$$
\begin{aligned}
\neg \,\mathrm{permits}(g) &= (x = k) \wedge \neg \,\mathrm{inhib}(g) \\
&\vee (x \geq k) \wedge (x - y \leq k) \wedge \neg \,\mathrm{inhib}(g) \\
&= (x \geq k) \wedge (x - y \leq k) \wedge \neg \,\mathrm{inhib}(g) && (1)
\end{aligned}
$$

(1): $((x = k) = \texttt{true}) \Longrightarrow ((x - y \leq k) = \texttt{true})$.

This means that permits($g$) disables switches when:

1. $(x = k)$ is satisfied as well as $g'$, resulting in the $\varepsilon$-labeled switch whose guard is $g$ to be enabled, or
2. $l$ was entered before $(x = k)$ was satisfied, $g'$ was satisfied when $(x = k)$ was, and the current clocks valuation satisfies $(x \geq k)$, forcing the $\varepsilon$-labeled switch to be taken.

□

# D Proof of Lemma 4.11

*Proof.* Let us consider a location $l$ that offers several switches, including $n > 0$ $\varepsilon$-labeled ones. By considering two switches from $l$, three cases are possible.

1. The switches have both labels that are not $\varepsilon$. By definition their guards are disjoint.
2. One switch $e_i$ ($i \in \{1, \cdots, n\}$) has $\varepsilon$ as its label with a guard

$$(g_i \bigwedge_{1 \leq j \neq i \leq n} \mathrm{permits}(g_j))$$

and the other switch has a label that is not $\varepsilon$ and a guard

$$(g \bigwedge_{1 \leq j \leq n} \mathrm{permits}(g_j))$$

The product of the guards contains a sub-clause $(g_i \wedge \mathrm{permits}(g_i))$ which is false: the guards are disjoint.

3. The two switches have $\varepsilon$ as their label. The product of their guards will make a sub-clause of the following form to appear:

$$(x_i = k_i) \wedge g'_i \wedge \text{permits}(g_j) \wedge (x_j = k_j) \wedge g'_j \wedge \text{permits}(g_i)$$

($i$ and $j$ belong to $\{1, \cdots, n\}$ with $i \neq j$). As $\text{permits}(g_j) \wedge g'_j$ is false, the guards are disjoint.

□

# E Proof of Theorem 4.12

*Proof.* The construction of $A^*$ adds one location $q$ to $A$ as well as one new switch per symbol of the alphabet and per location of $A$ plus $q$. We first show that the construction preserves determinism, and that given an input symbol, it can be recognized for any clocks valuation when the current location doesn't have any $\varepsilon$-labeled switch.

As every new switch guard is defined as the negation of the disjunctions of the guards of the switches from the same label, the intersection of the guards of every switch on the same label for a location $l$ is necessarily false, meaning that those guards are disjoint. In the case where a location $l$ does not offer any $\varepsilon$-labeled switch, it is also easy to check that the disjunction of the guards of the switches having the same label from $l$ is true as in [2].

Let us now consider a location $l$ having $n > 0$ $\varepsilon$-labeled switches $g_{\varepsilon i}$ $(1 \leq i \leq n)$. We also consider any symbol $a$ of the alphabet and the $m > 0$ guards of the $a$-labeled switches from $l$: $\{g_1, \cdots, g_m\}$ (again, given $1 \leq j \leq m$, $g_j$ is considered without its permits constraint clauses). Let us compute the disjunction of the $a$-labeled switches guards:

$$\left(g_1 \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i})\right) \bigvee \cdots \bigvee \left(g_m \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon m})\right)$$

By construction there exists $j \in \{1, \cdots, m\}$ such that $g_j = \neg \left( \bigvee_{1 \leq k \neq j \leq m} g_k \right)$, hence the previous disjunction reduces to:

$$\bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i})$$

which means that $a$ is recognized from $l$ under M-Invoke semantics. However, $\overline{A}$ must also recognize $a$ when this expression evaluates to false, which is clearly not possible from $l$.

Let $v$ be a clocks valuation such that $a$ is to be recognized and

$$v \models \left( \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i}) = \texttt{false} \right)$$

We can make the following remarks:

1. the current location is not $l$ anymore as a $\varepsilon$-labeled switch has been taken for the first clock valuation that stopped satisfying the previous expression, and
2. the current location change through the $\varepsilon$-labeled switch was instantaneous.

Let us call the current location $l'$. By construction, it offers $a$-labeled switches. From the remarks above, the guard of every switch satisfies

$$\neg \left( \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i}) \right)$$

for the clocks valuation $v$.

Consequently, $a$ can always be recognized from $l$ and the locations available through its $\varepsilon$-labeled switches:

$$\left( \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i}) \right) \bigvee \neg \left( \bigwedge_{1 \leq i \leq n} \text{permits}(g_{\varepsilon i}) \right) = \texttt{true}$$

□