N° d'ordre : D.U. 1840
EDSPIC : 402

Université Blaise Pascal – Clermont-Ferrand II
École Doctorale des Sciences pour l'Ingénieur de Clermont-Ferrand

The University of New South Wales, Sydney, Australia
School of Computer Science and Engineering

Thèse

présentée par

Julien PONGE

pour obtenir le grade de

Docteur d'Université

Spécialité : Informatique

Model Based Analysis of Time-aware Web Services Interactions

Thèse dirigée par Farouk TOUMANI / Boualem BENATALLAH
Soutenue publiquement le 1$^{\text{er}}$ Juillet 2008 devant le jury suivant :

Prof. Marie-Christine FAUVET	Présidente
Prof. Schahram DUSTDAR	Rapporteur
Prof. Claude GODART	Rapporteur
Prof. Marlon DUMAS	Examinateur
Prof. Michel SCHNEIDER	Examinateur
Prof. Farouk TOUMANI	Directeur de Thèse (UBP)
Prof. Boualem BENATALLAH	Directeur de Thèse (UNSW)

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 the 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 of availability 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 novel class of *protocol timed automata* (PTA). Specifically, PTA exhibit silent transitions that cannot be removed in general, 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.

Keywords. Web services; timed business protocols; compatiblity and replaceability analysis; choreography; orchestration; business process; service-oriented architectures; timed automata.

Résumé. Les services web gagnent de l'importance en tant que cadre facilitant l'intégration d'applications au sein et en dehors des frontières des entreprises. Il est accepté que la description d'un service ne devrait pas seulement inclure l'interface, mais aussi le protocole métier supporté par le service. Dans le cadre de ce travail, nous avons formalisé la catégorie des protocoles incluant des contraintes de temps (appelés *protocoles temporisés*) et étudié l'impact du temps sur l'analyse de compatibilité et de remplaçabilité.

Nous avons formalisé les contraintes suivantes : les contraintes C-Invoke définissent des fenêtres de disponibilités tandis que les contraintes M-Invoke définissent des délais d'expiration. Nous avons étendu les techniques pour l'analyse de compatibilité et de remplaçabilité entre protocoles temporisés à l'aide d'un mapping préservant la sémantique entre les protocoles temporisés et les automates temporisés, ce qui a défini la classe des *automates temporisés de protocoles* (PTA). Les PTA possèdent des transitions silencieuses qui ne peuvent pas être supprimées en général, et pourtant ils sont fermés par calcul du complément, ce qui rend décidable les différents types d'analyse de compatibilité et de remplaçabilité. Enfin, nous avons mis en oeuvre notre approche dans le cadre du projet ServiceMosaic, une plateforme pour la gestion du cycle de vie des services web.

Mots-clés. Services web; protocoles métiers; compatibilité; remplaçabilité; analyse; chorégraphie; orchestration; processus métier; architectures orientées-services; automates temporisés.

This work was conducted at the LIMOS Laboratory (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes), Université Blaise Pascal, Clermont-Ferrand, France:

LIMOS
Complexe scientifique des Cézeaux
63177 Aubière cedex
France

This work was also conducted under *cotutelle agreements* at the School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia:

School of Computer Science and Engineering
UNSW Kensington Campus
The University of New South Wales
Sydney
NSW 2052
Australia

Acknowledgments

The story behind this PhD thesis started in 2004 while a Master by Research student at the Université Blaise Pascal in Clermont-Ferrand. As I was looking for an internship topic, Michel Schneider pointed me to Farouk Toumani, who would eventually become my PhD adviser. Thank you Michel for that, as Farouk proved to be a very good adviser. During those 4 years working on this topic, he always provided good guidance and support. It has been a great pleasure to work with him, both from a professional and a personal point of view. In turn, Farouk pointed me to Boualem Benatallah, who would become my PhD adviser at the University of New South Wales as I did my PhD *under cotutelle agreements* between the two universities. I enjoyed working with Boualem very much, and his advices turned out to complement nicely those from Farouk.

I would like to thank the people with whom I shared the office room: Yoan Renaud that I first met when I entered the university, Alain Gély and Olivier Coupelon. We have had some great time together, and I am sure that there is still a lot more to come. There is a special mention for a group of 4 students of mine who later became my friends: Pierre Colomb, Olivier Montagner, Olivier Passalacqua and Frédéric Desforges. Of course I have enjoyed working with many other colleagues and friends in Clermont-Ferrand: Ramy Ragab, Marie Agier, Olivier Raynaud, Frédéric Flouvat, Hélène Jaudoin, Stéphanie Chollet, Kevin Hivernat, Yannick Loiseau, Raoul Médina, Béatrice Bourdieu and Françoise Toledo.

I would like to thank my coauthor Fabio Casati for the good times we have had working together, and for his never ending positive attitude. I met a lot of great people during my visits at UNSW. Warm thanks go to my friend Hamid Motahari and his wife Razieh for their kindness and for making my discovery of life in Australia so much easier. Among the people I met there, I would like to thank Halvard Skogsrud and Woralak Kongdenfha. Special thanks go to Mohand-Said Hacid that I first met while in Sydney. We spent some great time at UNSW, and he taught me a lot regarding running training!

Many thanks go to my reviewers and examiners as part of the French side of my PhD defense: Marie-Christine Fauvet, Schahram Dustdar, Claude Godart, Marlon Dumas and Michel Schneider. It has been a great honor to have you all as part of the examination committee!

My warmest thanks naturally go to my family, my parents Evelyne and Jean for having always stood behind me through the good and the bad times. I would probably not have gone so far in my studies without their encouragements and the education that they gave me. My final – and biggest – thanks go to Marie-Anne for her love and steady support during all of these years. Doing this PhD has required some sacrifices and I am deeply grateful to her not only because she accepted them, but because she fully and wholeheartedly endorsed them.

Contents

List of Figures

List of Tables

> “If you find that you’re spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you’re spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.”
>
> Donald Knuth

Part I

Introduction and background

One

Introduction

The introduction of this thesis first presents the context in the field of web services and application integration. We then look at the research issues behind this work, the contributions and finally give an outline of the document.

1.1 Context

Web services are increasingly gaining acceptance as a framework for facilitating application-to-application interactions within and across enterprises (Alonso et al. [4]). Application interoperability has been and still is a difficult issue due to difficulties created by heterogeneous and autonomous systems. Web services provide abstractions and technologies for exposing enterprise applications as services and make them programmatically accessible through standardized interfaces. Indeed, the main benefit they bring to application integration is that of standardization, in terms of description languages, coordination, and interaction protocols (Alonso et al. [4]; Papazoglou and Georgakopoulos [107]; Papazoglou et al. [108]; Papazoglou and van den Heuvel [109]). Standardization at interface definition language (WSDL) and transport protocol (SOAP) has enabled basic interoperability at the messaging layer. Indeed, developers are using SOAP and WSDL to integrate enterprise applications (S. Vinoski [124]).

This is by itself a huge improvement over previous application integration middlewares (e.g., RPC and messaging systems) as no costly adapters have to be developed so that the systems can interoperate (e.g., network protocol entry points and data representation converters).

While much progress has been made toward providing basic interoperability, there is still a lot to be done to simplify service development and interaction. In particular, an important aspect of Web services that affects interoperability is that services are loosely-coupled, that is, they are not developed only to interact with specific clients but are meant to serve the needs of many different (and potentially unknown) clients, possibly developed by different teams or even different companies. Hence, developers of client applications need to be aware of all functional and non-functional aspects of a service to be able to understand if they can interoperate with a service and how to develop clients that can interact correctly with the service. As such, service descriptions are specifications of the syntactic or semantic properties of a service that are made available to potential clients, for example with the purpose of:

1. assisting developers in creating clients that can correctly use and interact with a service, and

2. enabling the selection, either at design time or at runtime, of services that match the clients needs.

Hence, service descriptions need to be richer than “just” descriptions of interfaces as in conventional middleware (e.g, Corba IDL). Specifically, it is commonly accepted that a service description should include not only the interface, but also the *business protocol* supported by the service, i.e., the specification of possible message exchange sequences (conversations) that are supported by the service (Benatallah et al. [18]). Tools supporting service development today are mainly concerned with interoperability at the lower levels of the service stack (e.g. mappings from WSDL to Java and vice versa, or making two SOAP-based systems talk to each other). There is little support for protocol modeling and management.

Protocols can be specified using BPEL (*Business Process Execution Language* (OASIS [102])) or any of the many other formalisms developed for this purpose (e.g., (Benatallah et al. [18, 21]; Daniela Berardi [46])). Providing service descriptions is not in itself sufficient to facilitate development and binding. In addition to descriptions, we need formal methods and software tools for automatically analyzing service descriptions to: (i) identify if interaction between a client and a service is possible, and if it is, (ii) identify which conversations can be carried out between two services, to help developers to check if these include all and only the desired ones, and if it is not, (iii) understand mismatches between protocols and, if possible, create adapters to allow interactions to occur.

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. In (Benatallah et al. [18, 21]; Boualem Benatallah et al. [29]), an approach, a model for business protocols and a framework for protocol-based analysis had been presented. The availability of such analysis concepts and tools is quite powerful in that it allows us to assess compatibility in both top-down and bottom-up development approaches. The *ServiceMosaic project* (Benatallah et al. [22]) aims at developing a model-driven framework for web service life-cycle management. Business protocol management is a key concern in the ServiceMosaic tools set. In addition to protocol design and analysis tools, it also includes facilities for adapting services at the protocol level and for discovering protocol models from service execution logs. As such, the work that we present here is being developed as part of the ServiceMosaic effort.

1.2 Research issues

The present work 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 (Bettini et al. [27]; Maria et al. [87]; Tiplea and Macovei [130]) and even web services (Berardi et al. [25]; Dyaz et al. [52]; Kazhamiakin et al. [78]). There are countless examples of behaviors that involve timing issues in any kind of protocol (Benatallah et al. [18]), from business protocol for web services (e.g., see the *RosettaNet PIPs*), 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). However, most approaches mainly consider time for performing traditional model checking verifications such as liveness (a condition can be satisfied), safety (a condition can never be satisfied) and testing the absence of deadlocks (Berardi et al. [25]; Dyaz et al. [52]; Kazhamiakin et al. [78]).

The work that we present here formalizes the timing constraints of the business protocol model and extends the analysis approach introduced in (Benatallah et al. [18, 21]; Boualem Benatallah et al. [29]). The introduction of time aspects adds significant complexity to the problem compared to the case of "simple" business protocols from (Benatallah et al. [21]). 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. For example, *timed automata*, which are today considered as a standard 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 (Alur and Dill [6]). 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) (Rajeev Alur and P. Madhusudan [118]).

In our case, supporting compatibility and replaceability analysis requires tackling a number of challenges and producing several contributions, which we outline hereafter.

1.3 Contributions

Given the importance of considering time-related properties, we present a set of concepts and techniques, supported by a tool, for performing compatibility and replaceability analysis between timed protocols. The contributions are the following.

Timed protocols model. The first step consists in defining a protocol model, called *timed business protocol* that enhances the business protocol model presented in (Benatallah et al. [18, 21]; Boualem Benatallah et al. [29]) by introducing time-related constraints. We identified two timing constraint primitives that we added to the model for which we give both a syntax and semantics.

1. C-Invoke constraints define time windows during which an action can take place (e.g., receiving a purchase order acknowledgment), and

2. M-Invoke constraints define triggers for implicit actions to take place (e.g., a timeout).

Timed business protocols provide a precise understanding of the external behavior of a service, as one knows exactly which conversations are valid with respect to both the messages ordering and the timing at which those messages are exchanged. Syntax and semantics are required for timed protocols. As timed automata (Alur and Dill [6]) form a widely studied model for capturing timing requirements of systems, a link between timed protocols and timed automata allowed to re-use and derive properties. We will see that the task was not straightforward, especially as M-Invoke constraints are not easy to represent equivalently in timed automata.

Timed protocols analysis. The next step consists in defining analysis concepts for identifying compatibility and replaceability between timed protocols. To do that, we extended the work of (Benatallah et al. [21]) by introducing several flexible classes for both compatibility and replaceability, and by characterizing them using a set of protocol manipulation and comparison operators. By flexible, we mean that those classes cater for more than "black or white" compatibility or replaceability cases like it has traditionally been done for hardware and software components, as the versatile, fast-changing nature of web services requires flexibility for such an approach to be useful.

In our timed protocol operators investigations, we chose to reuse and extend work from timed automata, and from there, we have obtained their decidability. To do that, we defined a semantic-preserving mapping between timed protocols and timed automata, leading to a novel class of timed automata that we called *protocol timed automata.* The task is however not trivial, as:

- protocol timed automata exhibit ε-labeled switches with clock resets that make them strictly more expressive than timed automata (Beatrice Berard et al. [13]), keeping the traditional problems of language inclusion and complementation undecidable (Alur and Dill [6]), and

- M-Invoke constraints semantics are hard to implement in timed automata as we will see in chapter 4.

The study of protocol timed automata allowed for two important theoretical results:

1. the class of protocol timed automata is closed under both intersection and complementation, making it the first such one among the existing classes of timed automata having ε-labeled switches with resets, and

2. protocol operators remain decidable in the case of timed protocols as we derive techniques from timed automata. Had the class of protocol timed automata not been closed under complementation, those results would not have hold true.

ServiceMosaic prototype. We implemented our approach in the context of a larger project called ServiceMosaic (Nezhad et al. [99]). More specifically, we developed:

- a business protocol model library
- a protocol manipulation and comparison operators library that also offers checking for all of the classes of protocol compatibility and replaceability
- a set of plug-ins for the Eclipse platform (see http://www.eclipse.org/) that includes a protocol editor and a visual interface for doing protocol analysis work.

The components developed in this project serve as the basis of many other ServiceMosaic projects and will be published under an open source license in 2008.

The approach that we present is innovative in that it provides a fine-grained analysis of web service interactions, and timed automata are used in a different context of what they have been traditionally used for (i.e., model checking). Last but not least, few works have focused on timing abstractions in web services, and when it is the case, it is mainly for reusing model-checking techniques.

1.4 Outline

This thesis has been divided in the following three parts.

Introduction and background. The present chapter (chapter 1) provided an introduction. Chapter 2 is an overview of web services and their origins in application integration. We also provided succinct materials on the related concepts and technologies. Finally, chapter 3 provides background knowledge on the field of timed automata. It introduces the formalisms, classes and common problems that have been studied by the formal verification analysis community. This is especially interesting in the context of this work, as we make a different usage of timed automata than doing verification tasks.

Protocol modeling and analysis. Chapter 4 introduces the model of timed protocols that supersedes the business protocols of (Benatallah et al. [21]). It gives both the syntax and semantics. It also gives the semantic-preserving mapping to protocol timed automata, and illustrates that implementing M-Invoke constraints in timed automata is everything but a straightforward task. Chapter 5 presents the protocol-based compatibility and replaceability analysis concepts. They are a direct extension of what had been done in (Benatallah et al. [21]) as protocol operators have been upgraded to support timing constraints as well. Finally, chapter 6 studies the properties of protocol operators in the context of protocol timed automata. This is where we obtain their decidability by reusing some results from timed automata and contributing new ones (e.g., complementation of the class of protocol timed automata despite the presence of ε-labeled switches).

Applications and perspectives. Chapter 7 presents the ServiceMosaic project and the tools that we developed as part of a subproject called ServiceMosaic Protocols. Chapter 8 provides one scenario where compatibility and replaceability analysis can be leveraged in the context of a BPEL-based composition of web services. Finally, chapter 9 opens perspectives for future work, and gives hints for applying the contributions of this work well beyond "just" timed protocol analysis.

Two

Web services

This chapter introduces web services and the novelties brought by service-oriented computing for facilitating enterprise integration beyond the traditional boundaries of organizations information systems. Indeed, web services are not “just” an evolution of existing middleware, they also address new integration needs.

The structure of this chapter is the following. We start by a brief introduction to enterprise integration and application architectures. We then present some families of conventional middlewares that have been used for application integration, and we show their limitations in light of new integration challenges, including the need to facilitate integration between business partners. We then present service-oriented computing and the business protocol model and analysis techniques that serve as a foundation of this thesis contributions.

2.1 Enterprise integration

The information system of an organization typically comprises various assets that range from data sources (e.g., relational databases systems) to applications that fulfill different needs (e.g., office tasks suites or business intelligence solutions). Such parts of an information system are either developed “in-house” to meet specific requirements, or they are acquired from third-party vendors (with possible customer-specific adaptations). Those assets rarely live isolated to each other. Indeed, it is often necessary for an application to access the data produced by another one, or even to access some parts of an API[1] to reuse some of its functionalities. Enterprise integration is a strong information technology concern (Hohpe and Woolf [73]). Careful practice is required to foster existing applications and stir their full potential in the larger view of an enterprise information system (Fowler [58]).

Web services, and more generally service-oriented architectures, provide new perspectives for facilitating enterprise integration. They not only break technical barriers (e.g., difficulties in bridging applications from different organizations), they also enhance the architecture of information systems in a distributed, platform and language agnostic manner (e.g., bridging applications written in Cobol, Java and .Net). However, web services should not be viewed as *revolutionary*. They should rather be seen as *evolutionary*, i.e., they are an evolution of enterprise integration practices over the last few decades (Alonso et al. [4]).

This section reviews the architectures found in enterprise applications, and highlight where integration can happen as it next focuses on middlewares[2].

2.1.1 Application architectures

From an architectural point-of-view, it is common to differentiate different *tiers* in an application. This is especially true with the generalization of distributed applications as the different tiers can be physically deployed separately (Alonso et al. [4]). We briefly present the typical tiered architectures while an overview of them is available in Figure 2.1.

[1] Application Programming Interface.

[2] Briefly, the term *middleware* designates software that facilitates the interoperability between heterogeneous systems.

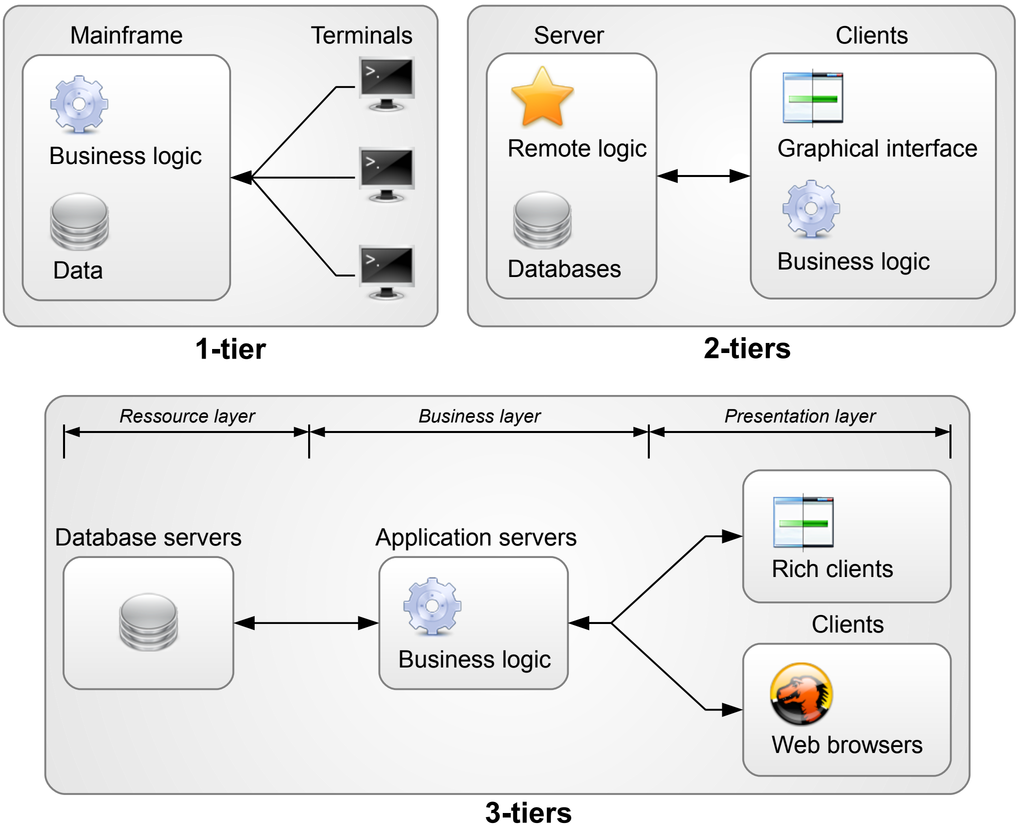


Figure 2.1: Overview of typical 1-tier, 2-tiers and 3-tiers architectures.

1-tier (70's – 80's). This type of architecture is highly reminiscent of the early design of enterprise applications of the 70's and has kept being used throughout many developments of the 80's. Those architectures were relying on applications to be deployed on a single physical machine, called a *mainframe*, while end-users would access it through *terminals* over a network (often a proprietary network solution rather than a standards-based one). This type of architecture turned out to be relatively simple to run with simplified maintenance life-cycles as applications were deployed to a single, central machine. It had however a lot of obvious problems, including the need for expensive hardware upgrades to scale up in the number of concurrent terminals or the single-point of failure syndrome.

2-tiers (90's). This is also called *client / server* in reference to the decomposition of the application units in server-side and client-side units. Those architectures have been primarily pushed forward by the generalization of relational database management systems exposed through network interfaces, thus providing an application neutral way for applications to store and gather data. The database systems are deployed server-side while the remainder of the application (business logic and visual interface) is client-side. There are variants to this: some computation-intensive tasks can be deported to the server as well. Also, some applications can embed client-side databases. A typical example of this is the support of "offline modes" where an application needs to store data when no network connection is available and synchronize it with the server when it becomes available again. Compared to 1-tier architectures, 2-tier applications brought a number of improvements, among which the promotion of standard-based networks as well as normalized data storage and representation systems. Performance-wise, they were also able to introduce clustering of both databases and portions of business logic to handle scalability in the number of clients.

3-tiers and *n*-tiers (end 90's – 2000's). A 3-tiers application comprises the following layers (Hohpe and Woolf [73]).

- The *resources layer* provides an access to databases, files, network interfaces and legacy applications.
- The *business layer* provides the core functionalities of applications, no matter how they are accessed. This layer often relies on component technologies (e.g., EJBs[3], see http://java.sun.com/products/ejb/).
- The *presentation layer* exposes the applications to end-users and applications through a rich set of devices and technologies. In a similar fashion as the model–view–controller paradigm, the very same application can be exposed through different presentation interfaces.

[3] Enterprise Java Beans

This type of architecture started to become widespread with the advent of database-driven web applications. It pushed the generalization of *application servers* (e.g., JavaEE, .Net Framework or LAMP[4] solutions) that embed the business logic, with the database storage being often delegated to another server. In turn, the presentation layer is accessed through web browsers which are light and generic display devices. Those applications can still be accessed from desktop applications, now called *rich clients* by contrast to web browsers being considered as *thin clients*. The term "*n*-tiers" refers to a finer-grained decomposition of an application architecture than 3-tiers. For example in the business layer, a purchase order management component may rely on a remote payment processing component.

Integration placeholders. Enterprise integration can be performed:

- *horizontally*, by integrating assets from the same tier (e.g., integrating data from heterogeneous customer-centric databases), and
- *vertically*, by allowing the components from one tier to access to an upper or lower tier (e.g., a payroll management component accessing a relational database).

Integrating such assets would be easy in a perfect world where data representation and application interfaces would be uniform. The reality is of course very different as an information system present some of the following sources of heterogeneity: platforms (including operating systems, hardware architectures and execution platforms), databases systems and programming languages.

As an organization evolves, it periodically changes its technological standards, resulting in the need to integrate new and legacy systems. For example a warehouse management application may have been developed a few years back using the *Delphi* programming platform while new systems need to be developed in Java. Such legacy systems are often kept "as-is" since rewriting them has significant costs (e.g., development and training

[4] Acronym for Linux, Apache, MySQL and PHP / Perl / Python (this is not strict as other technologies can be used as replacements).

of employees) without necessarily resulting in practical benefits (e.g., the "old" application already fulfills the needs).

Systems that facilitate both horizontal and vertical enterprise applications integration are called *middlewares*. The next section provides an overview of them.

2.1.2 Middlewares

In this section, we review some common middleware families. We also expose the limitations of conventional middlewares. The role of a middleware goes beyond just bridging heterogeneous software artifacts: it hides some of their inherent complexity (much like a *Facade* does in object-oriented programming (Gamma et al. [63])). A middleware can provide some development-time infrastructure (e.g., developer tools) and runtime infrastructure (e.g., deployed execution and monitoring components). We give a few examples of middlewares families (Alonso et al. [4]; Hohpe and Woolf [73]). The platforms that are based on middleware technologies rarely support only a single family. Full-stack application servers such as the ones that provides the JavaEE specification implement much of the following ones.

Data integration. Applications physically store their data in files, databases, relational databases and so on. Several approaches exist, and we illustrated them on Figure 2.2. The first one is to adapt the data from one or more applications so that another one can process it (Figure 2.2(a)). An example would be to extract some data from a spreadsheet document and a relational database and transfer to another application. This involves creating ad-hoc transformation components (e.g., scripts or XSLT transformers). The other approach to data integration is to integrate data sources as heterogeneous, distributed relational databases. In this case there are two ways of achieving it.

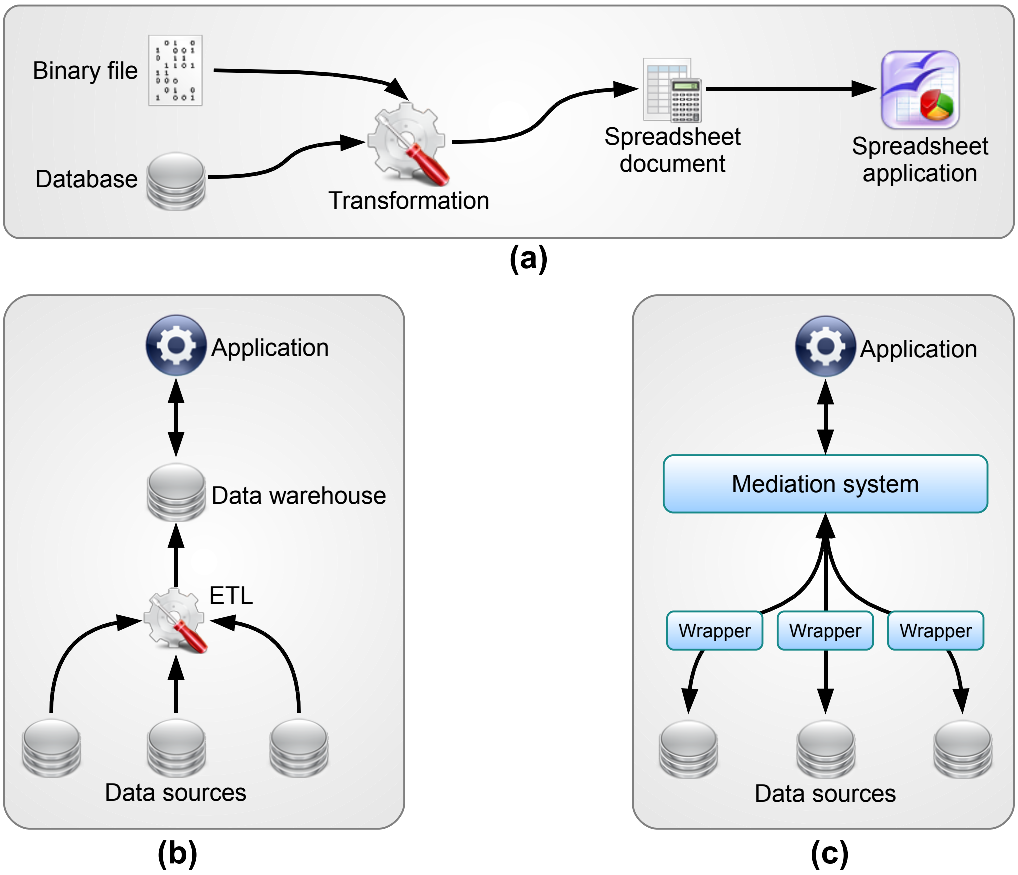


Figure 2.2: Different approaches for data integration.

1. Materialized approaches use ETL[5] tools that pull data from several relational database sources, normalize/transform it, then finally push it to *data-warehouses* (Figure 2.2(b)).

2. Virtual approaches (Figure 2.2(c)) use techniques and concepts such as *global-as-view* and *local-as-view* where the data is exposed by the mean of queries over the sources (see (Lenzerini [82]; Ullman [132]) for an overview and (Chawathe et al. [44]; Goasdoué et al. [65]; Halevy et al. [67]; Levy et al. [83]) for implementation-related discussions).

[5] Extract, Transform and Load

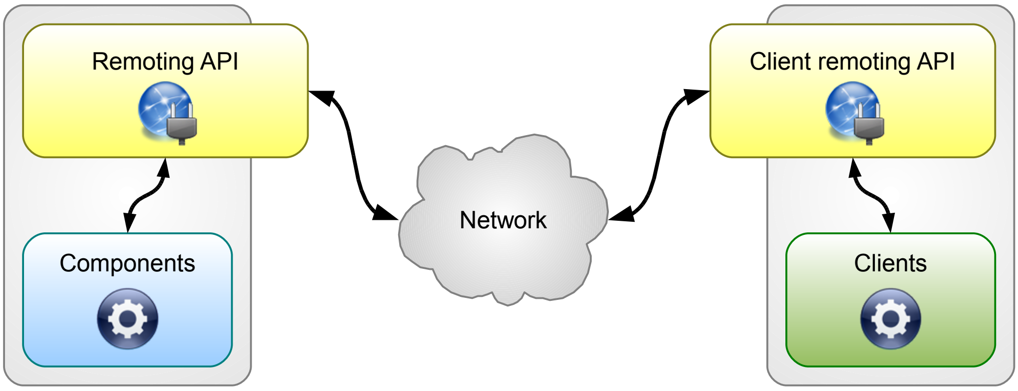


Figure 2.3: RPC-style middleware.

Remote procedure calls (RPC). They allow applications to communicate through regular *procedure / function / method* calls of programming languages. Such a call looks like a normal, local invocation of some application code, but in reality, the actual code that is executed runs on a remote system. To do that, RPC middlewares encode the call parameters in a message that is sent over the network to the remote machine that will run the actual code. Once it has been executed, a response message is sent back to the invoking application with a return value being encoded inside the message. An illustration is provided on Figure 2.3. The preparation of the message (either for performing the call or returning the value) is called *marshaling*. Similarly, the decoding of the message (either on the remote server or the invoker side) is called *unmarshaling*. RCP middlewares have been implemented for a lot of languages and platforms. Examples include distributed components such as EJBs (remote session beans) or Microsoft DCOM.

Message-oriented middlewares (MOM). They allow the communication between applications through asynchronous messages. By contrast to RPC middlewares, the messages do not cary “function call” semantics as they are rather general purpose documents exchanged between the systems. A MOM is responsible for collecting and routing messages. It also ensures their integrity, and that the messages are not being lost. Quality of service

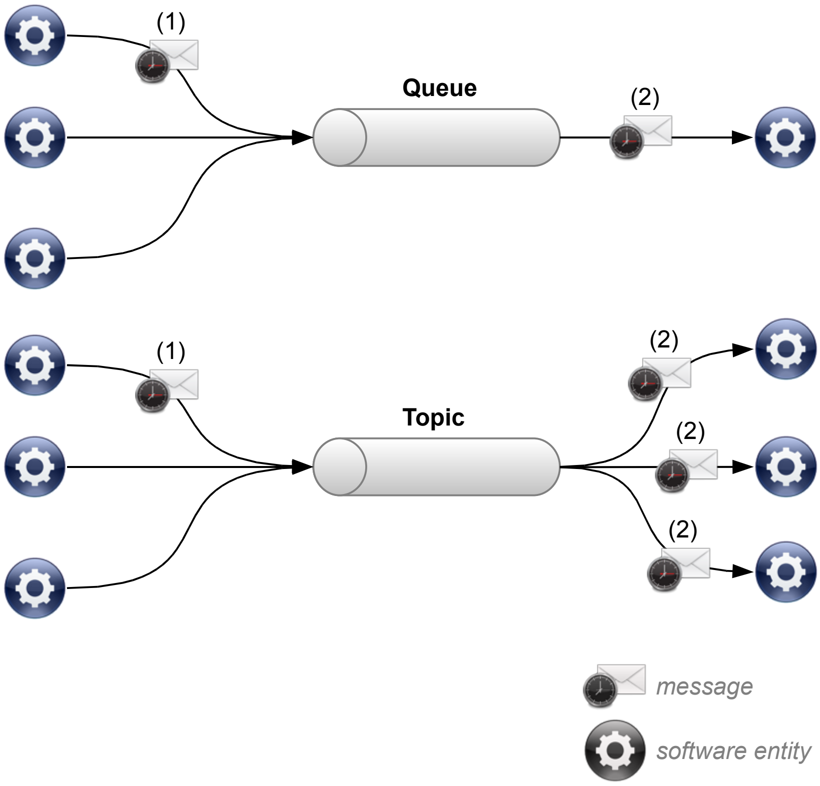


Figure 2.4: Messaging middleware.

requirements are generally available, with priorities and validity expiration dates being the two most common in use. A MOM is also able to transform the message contents. This is useful when a message needs to be routed to an application that does not understand the initial message data format. Two message basic communication patterns are found in MOM (see Figure 2.4): *message queues* provide n-to-1 communication between n message emitters and one receiver while *message topics* allow broadcasting messages between n emitters and m receivers. An example specification of a MOM is the Java Messaging System (JMS).

Object request brokers (ORB). They allow clients to delegate the discovery and binding processes of components that match programmatic interfaces. The broker will select the best component according to the criteria it has been configured for, and will provide the clients with objects without them knowing what their concrete implementations are. To do that, an ORB relies on directories of components. Corba (http://www.corba.org/) is an example of a middleware comprising ORB technology.

Transaction monitors (TP). They ensure the coordination of distributed components as part of transactional processes. Relational database management systems have efficient support for transactions on data manipulation, hence it is often natural to rely on them for transactional behavior in applications. This is however not always possible when different business logic components have to be assembled as each of them may not rely on the same database. Those systems usually implement variants of the three-phases commit protocol (Skeen and Stonebraker [126]). An example TP is the Java Transactional API (JTA).

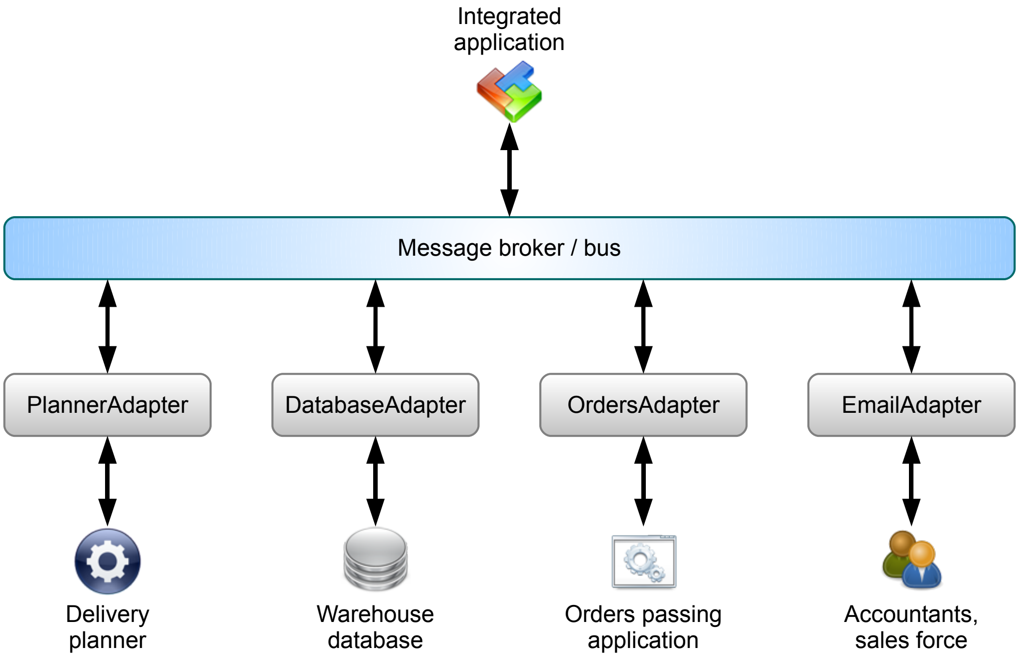


Figure 2.5: Enterprise application integration using a message broker / bus.

Example. We give an example of an enterprise application integration on Figure 2.5 where the business process of the purchase orders management is handled through a set of existing applications. Each application is connected through an adapter to a message-oriented middleware, called a *messages bus*. Each message is routed (and possibly transformed) to other entities (e.g., a message sent by the orders passing application for asking goods availability is delivered to the warehouse database adapter). Each adapter is using diverse forms of middleware to bridge the bus and the application. For example the warehouse database adapter can use an ODBC connection to deal with the underlying database, while the delivery planner adapter may use RPC to communicate with the plannification system.

Limitations of conventional middlewares The "conventional" middlewares that we saw above all share some common limitations. The first one is probably that a lot of adaptation components have to be developed to allow two middleware-based systems to "talk" to each other. Another issue is that conventional middlewares are largely centralized (Alonso et al. [4]), which limits deployment flexibility and again, requires the development of adapters. As an example, given two RPC frameworks, there is often a need for creating an adapter if they come from different vendors. Worse, even a standard specification is not enough for granting interoperability of systems. A good example is the Java Messaging System (JMS) that defines a standard API for MOM middleware in Java. While every JMS-compliant implementation ensures that an application can be ported from one JMS vendor to another one, there is no requirement on the network protocols and data representation formats, meaning that adapters have to be developed to bridge JMS brokers originating from different vendors.

Things get worse when integration needs to be performed across the boundaries of organizations, as network connectivity issues get added to the (already costly) development of adapters. Indeed, crossing companies firewalls is a serious concern as security need to be preserved, and maintaining such exceptions is complex. Also, a company will rarely trust a partner enough for allowing a direct connexion between their information systems: connectors also need to provide a controlled, coarse-grained interface to the internal information system. Another issue is that in such cases, integration is performed in a point-to-point fashion, something that does not scale well as the number of integrated partners grows (Alonso et al. [4]). There is also an inherent lack of flexibility in those approaches as developing adapters and allowing basic connectivity takes time. By contrast, organizations today want to reduce costs, integrate easily with partners and be able to change and outsource some of their assets "painlessly".

Web services, and more generally service-oriented architectures have most benefits of the previous middleware families for facilitating application integration. But better, they also solve the mentioned limitations as we will see now.

2.2 Service-oriented computing

This section presents service-oriented computing. We start by an overview of service-oriented architectures. We then review the existing web service technologies before looking at service descriptions.

2.2.1 Service-oriented architectures

A commonly-accepted definition[6] of web services is the following:

> *Web services are a new breed of Web applications. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes... Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service.*

Service-oriented architectures (SOA) form the new wave of information systems (Papazoglou et al. [108]; Papazoglou and van den Heuvel [109]).

[6] http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/index.html and http://www6.software.ibm.com/developerworks/education/wsbasics/wsbasics-ltr.pdf

The key idea of SOA is to turn the data sources and applications in an enterprise system into distributed units referred to as *services*. Each service is self-described and it is expected to fulfill a very focused set of coarse-grained functionality requirements (e.g., a payroll processing service is not expected to also handle warehouse provisioning). They can then be assembled to realize applications and business processes. Services provide standard-based, platform-neutral interfaces that hide the implementation details. Also, the services should be autonomous with little dependencies to each other. This means that SOA promotes the loose coupling of services that are used for performing application integration of heterogeneous assets. The loose-coupling comes from both the fact that services are autonomous and that they use open standards for their external access (interfaces and data exchange formats).

While approaches with similar goals had been developed before the advent of web services (e.g., Corba, distributed components, ...), they bring some novelty (Alonso et al. [4]). SOA are indeed often associated with web services as the core mean to realize them. Instead of re-inventing new interface specifications and data encoding means, they directly use widespread, ubiquitous standards such as XML or HTTP. The entry ticket to those technologies is rather low compared to previous approaches, which makes it possible to reuse legacy applications, hide their implementation (including the languages and platform choices) and integrate them with other applications also exposed as services. Another significant improvement in terms of application integration is that SOA blur the traditional boundaries of information systems. Indeed, integrating the information systems of different organizations has traditionally been a costly, largely ad-hoc process (Fowler [58]), as information systems rarely rely on the same technological choices (e.g., languages, platforms) and normative choices (e.g., data representation formats, schema). Technologies such as web services naturally cross networks without having to develop network-specific bridges (e.g., virtual private networks) and as such, they make it “natural” to integrate the heterogeneous information systems of different organizations.

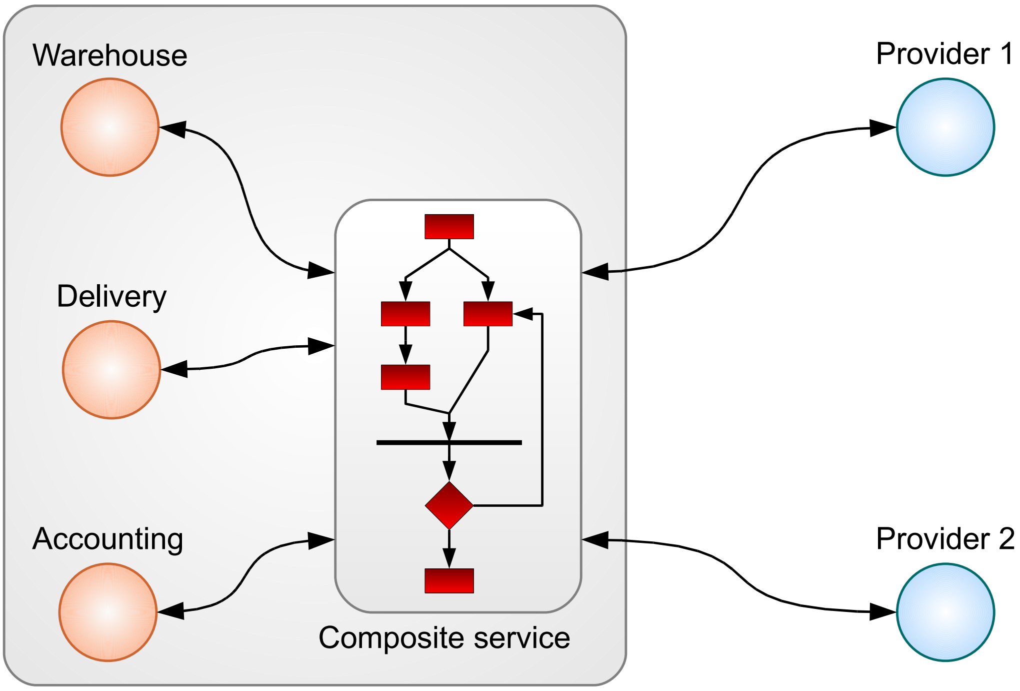


Figure 2.6: A SOA composition scenario.

We give an example of a simple service-oriented architecture on Figure 2.6. Three in-house services are provided: a warehouse management service, a delivery service and an accounting service. Each of them is an interface to larger applications and data sources and may well have been implemented using different technologies. There are also two external services of providers. A composite service is created by using an orchestration described by a BPEL process (which is not detailed on the figure). When a purchase order is received, it first checks with the warehouse for availability. If not, a quote is asked to each provider for provisioning the warehouse with the cheapest offer. Once this is done, goods are removed from the warehouse for delivery and the payment is delegated to the accounting. In this example, it is possible to externalize the accounting of the organization very quickly: all that is needed is that the accounting provider offers a service with a compatible interface (or more realistically with minimal adaptation required). This is by itself a significant improvement over traditional integration across enterprise information systems.

2.2.2 Technologies

We now give an overview of the main web services technologies.

XML-RPC. XML-RPC is the first form of web services to have appeared. Defined in (Dave Winer [48]), it is a small and simple specification for performing RPC-style communications between systems. As the name suggests, the marshaling and unmarshaling of the invocations are done using XML for data representation. The marshaled messages are conveyed using the HTTP protocol (HTTPS can be used for secure point-to-point communications). Both the limited data types and the general minimalism in XML-RPC have facilitated its widespread adoption.

SOAP. SOAP (Martin Gudgin et al. [89]) is a standard protocol defined by the W3C for exchanging XML-based messages between web services and their requesters. It builds on top of widespread existing standards such as XML, URI, HTTP and SMTP. Basically, a SOAP message is an XML document that acts as an envelope for the content being sent. In turn, a SOAP message is meant to be transported over various kind of protocols, the most used being HTTP / HTTPS. While the usual expected alternative is the *Standard Mail Transfer Protocol* (SMTP), real-world applications have been using IIOP (from Corba), XMPP (Jabber open instant messaging) or even JMS (*Java Messaging System*) to route SOAP messages. SOAP messages can be used either to simulate RCP-style communications, or document-style communications which are analogous to asynchronous messaging systems.

REST. The concept of REST[7]-style services first appeared in (Roy Thomas Fielding [123]). It is not a formal specification of a technology for building

[7] REST stands for ***RE****presentational* ***S****tate of* ***T****ransfer.*

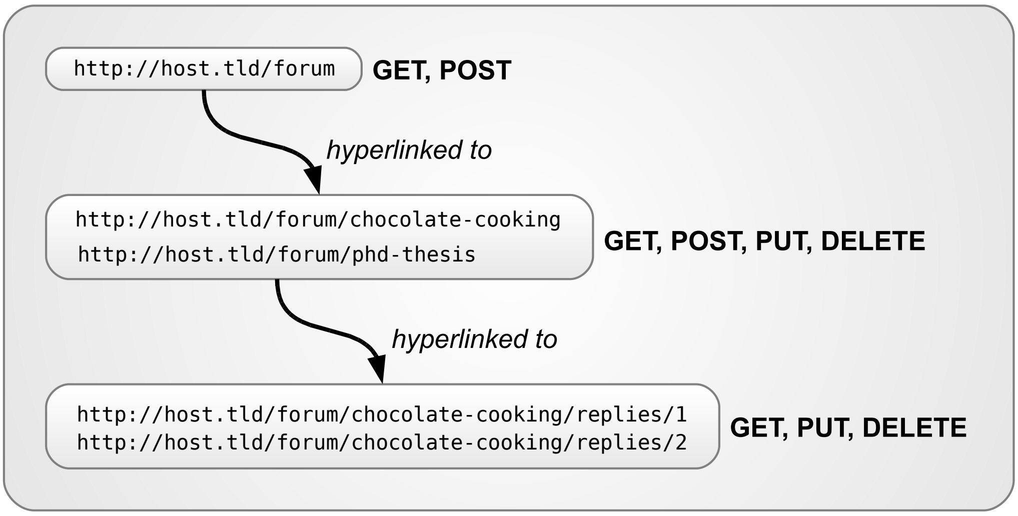

http://host.tld/forum
GET, POST
hyperlinked to
http://host.tld/forum/chocolate-cooking
http://host.tld/forum/phd-thesis
GET, POST, PUT, DELETE
hyperlinked to
http://host.tld/forum/chocolate-cooking/replies/1
http://host.tld/forum/chocolate-cooking/replies/2
GET, PUT, DELETE

Figure 2.7: A *RESTful* forum application.

web services. Rather, it is an architectural style for building web services that emphasizes the semantics of both the HTTP protocol and the *Universal Resource Identifiers* (URIs). The core concept behind REST is that applications expose *resources* that are accessible through URIs over the HTTP protocol. A resource may have several *representations*, i.e., it may use different data encodings (e.g., XML, CSV, plain text) to accommodate different requesters. The resources can be *hyperlinked* with each other and manipulated through a limited set of *verbs*: the HTTP protocol methods (GET, POST, PUT and DELETE, see http://www.w3.org/Protocols/rfc2616/). As such, resources provide a *uniform interface* to applications (Roy Thomas Fielding [123]). We give an example of a *RESTful*[8] forum application on Figure 2.7. Hyperlinked resources can be manipulated by a HTTP software agent (e.g., for fetching message threads or posting a reply).

Syndication feeds. Syndication web services have become increasingly popular with the rise of collaborative online applications such as blogs, wikis and podcasting. They provide resources called *feeds* to notify *subscribers*

[8] Widely-used jargon for touting an application as being designed using the REST philosophy.

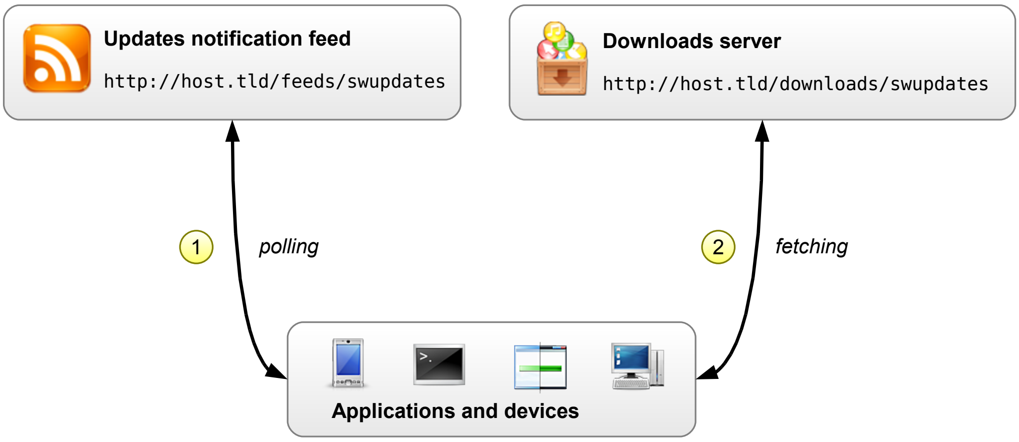

Updates notification feed
http://host.tld/feeds/swupdates
Downloads server
http://host.tld/downloads/swupdates
1
polling
2
fetching
Applications and devices

Figure 2.8: A simple software updates notification and distribution scenario based on syndication.

of events (e.g., a new blog post is available). The service does not directly notifies the subscribers of changes. Rather, it provides a URL-accessible resource which is updated when needed. Subscribers poll the resource for changes on a periodical basis. Two XML-based specifications are currently competing. An example is given on Figure 2.8 where a feed is made available for publishing software update notifications.

2.2.3 Service description and discovery

We now briefly review standards and specifications for describing static web services interfaces and discovering them.

Static interface. The *Web Services Description Language* (WSDL) (Erik Christensen et al. [53]) is an important add-on to SOAP-based web services. It provides a description of the static interface of such a service, much like IDL can be used in component technologies. As such, it specifies the operations supported by the service, the data schemas, and the binding information used to communicate with the service such as the transport protocol being used (ex: HTTP) and the location of the service (in the case of HTTP as a transport protocol, a URL). The messages that can be exchanged between a requester and a service are described using XML Schema (http://www.w3.org/XML/Schema).

A WSDL document can be used to bind to a SOAP-based service in two ways: either client invocation source code can be generated from the WSDL, or the binding can be performed dynamically at runtime. The latter case is especially possible with dynamic and reflexive languages such as *Java* or *Ruby*, by using the *Proxy* design-pattern (Gamma et al. [63]). Once either the static or dynamic binding has been performed, the service can be invoked.

Registries. The *Universal Description, Discovery and Integration (UDDI) protocol* (see http://www.uddi.org/) provides a way for repositories to contain service functional and non-functional descriptions. Ultimately, a client interested in a given discovered service is able to obtain its WSDL description and bind to it after having found it. Another proposal for services repositories can be found in *ebXML* (see http://www.ebxml.org/).

WS-*. Several specifications are gravitating around the SOAP web services stack (Nezhad et al. [96]), often referred to as "WS-*" specifications. Most of them are actually irrelevant with no practical adoption in real-world use-cases (S. Vinoski [124]). Among the very few relevant ones we have:

- WS-Security (http://docs.oasis-open.org/wss/v1.1/), a specification for end-to-end preservation of the messages confidentiality and authenticity

- WS-Addressing (http://www.w3.org/Submission/ws-addressing/), which defines a standard representation for endpoint references and routing schemes (useful for correlation in stateful web services)

- WS-Policy (http://www.w3.org/Submission/WS-Policy/), a standard representation of quality of service requirements (e.g., security, message-encoding optimizations, ...)

- WS-Reliability (http://docs.oasis-open.org/wsrm/ws-reliability/v1.1) is useful for ensuring that messages are properly delivered in contexts that are not point-to-point communications between the service provider and its requesters

- WS-Coordination (http://docs.oasis-open.org/ws-tx/wscoor/2006/06/) and WS-Transaction (http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html) allow the coordination of web services.

The next section presents the model of business protocols (Benatallah et al. [21]) that serves as a foundation of this thesis work as it is suitable for describing the external behavior of a service with respect to the conversations it supports.

2.3 Business protocols

Static interface descriptions such as WSDL are essential for facilitating the use of web-service based technologies. There is however a strong need for abstractions of a higher-level than just "data formats and message names". As such, there is a consensus over the need for also specifying the dynamic interface of services (Papazoglou and Georgakopoulos [107]; Papazoglou et al. [108]; Papazoglou and van den Heuvel [109]). In particular, the externally-observable behavior of a service is important for avoiding invalid conversations between a service and its requesters. For example, a service may offer to add goods to a shopping cart and make a payment. A WSDL document of such a service would give the related message names (e.g., *addGoods* and *pay*) as well as their XML schema. This is clearly insufficient as for instance, paying before adding goods is most often not a valid business process.

This section first recalls the business protocol model and its usefulness. It then presents protocol-based analysis before providing a discussion.

2.3.1 Model

The model of business protocols of (Benatallah et al. [16, 18, 21]; Karim Baina et al. [77]) is the foundation of the work being presented in this thesis. This model captures all of the conversations that a service supports, i.e., the set of ordered message exchanges that are valid (e.g., a purchase order should not be exchanged before an authentication message). As such, it provides an abstraction for capturing a service external behavior, allowing potential requesters to know how to avoid generating invalid conversations, which is by itself a significant enhancement compared to having "just" a WSDL description. The model is based on state-charts as it is commonly accepted as a suitable model for describing behaviors, but alternatives could have been used as well such as Petri-nets or abstract BPEL (the later capture the operations orderings in a BPEL process).

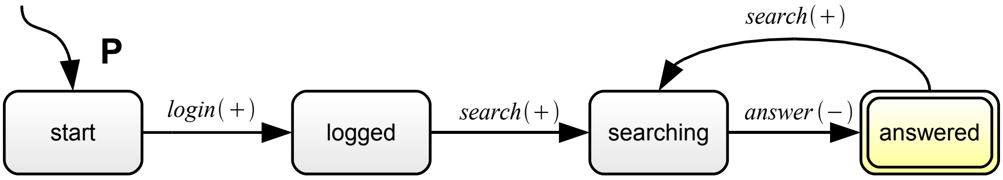


Figure 2.9: A business protocol of a search engine.

As an example, Figure 2.9 shows a business protocol of a search engine. States represent the various stages a service may go through while transitions are triggered when a message is received or sent. There is a unique initial state and many final states. In this protocol, conversations start with the service receiving a `login` message (the polarity (+) is used to indicate that the message is incoming). Following that, a search can be performed and the results can be retrieved through repeatable sequences of `search` – `answer` messages (again, the polarity (−) is used for messages that are emitted by the service). The answered state is final: every conversation that ends in this state is said to be valid. By contrast, an invalid conversation would be `search` · `login` · `answer`. Finally, it should be noted that state names do not have much importance in the model as conversations are about messages being exchanged.

Business protocols can be useful in several analysis contexts, both at development and runtime (Benatallah et al. [16, 18, 21]; Karim Baina et al. [77]). Conformance checking is useful for a service provider to check if the actual implementation of a service meets a specification which can be either provided to clients (e.g., in the documentation), or imposed from standards (e.g., RosettaNet PIPs (RosettaNet [121])). Another one is for a requester to check that its own client protocol is compatible with a service, and if it is not, perform some adaptations so as to accommodate the service protocol to avoid errors. Finally, there are cases were a service needs to be replaced (e.g., failures, business-driven need for changing partners, etc). Replaceability analysis is useful so that a new service can be used as an alternative with minimal impacts on the existing requesters. Especially, it helps in identifying the potential mismatches at the protocol level so that adapters can be created. As such, both compatibility and replaceability analysis can be interesting as part of a service discovery process. In this work, we are interested in protocol-based compatibility and replaceability analysis that we outline hereafter.

2.3.2 Protocol analysis

Business protocols allow to perform fine-grained types of compatibility and replaceability analysis to see if (and how) two services can have interactions based on their protocols (compatibility), or if a service can be replaced transparently by another one for its current requesters (replaceability). Several classes of both compatibility and replaceability have been defined.

- *Partial compatibility* between two protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ implies that at least one conversation can be carried out between two services implementing these protocols.

- A protocol $\mathcal{P}_1$ is *fully compatible* with $\mathcal{P}_2$ if $\mathcal{P}_2$ can support all message exchanges that $\mathcal{P}_1$ can generate (the inverse is not necessarily true).

- *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.

- *Protocol replaceability w.r.t. a client protocol* occurs when a protocol $\mathcal{P}_1$ can replace a protocol $\mathcal{P}_2$ when interacting with a client protocol $\mathcal{P}_c$ if every valid conversation between $\mathcal{P}_2$ and $\mathcal{P}_c$ is also a valid conversation between $\mathcal{P}_1$ and $\mathcal{P}_c$.

- *Protocol replaceability w.r.t. an interaction role* is similar to the previous one. It occurs when a protocol $\mathcal{P}_1$ can replace a protocol $\mathcal{P}_2$ if $\mathcal{P}_1$ behaves like $\mathcal{P}_2$ when $\mathcal{P}_2$ behaves as an interaction role $\mathcal{P}_r$.

Those analysis classes are characterized by using a set of protocol operators.

- The *intersection* operator computes the common conversations that two services support.

- The *parallel composition* operator computes the common interactions that can take place between two services.

- The *difference* operator computes which conversations are supported by a given service, but not by a second one.

As an example, two services are partially compatible as long as their parallel composition yields a non-empty protocol (i.e., at least one conversation is supported). This is the case between the protocol P of Figure 2.9 and P_2 of Figure 2.10. They are however not fully compatible as P_2 can generate some conversations that P cannot support (e.g., P does not support `editPreferences` messages).

In terms of protocol replaceability, the protocol P of Figure 2.9 can be replaced by the protocol P_1 of Figure 2.10. Indeed, there is protocol

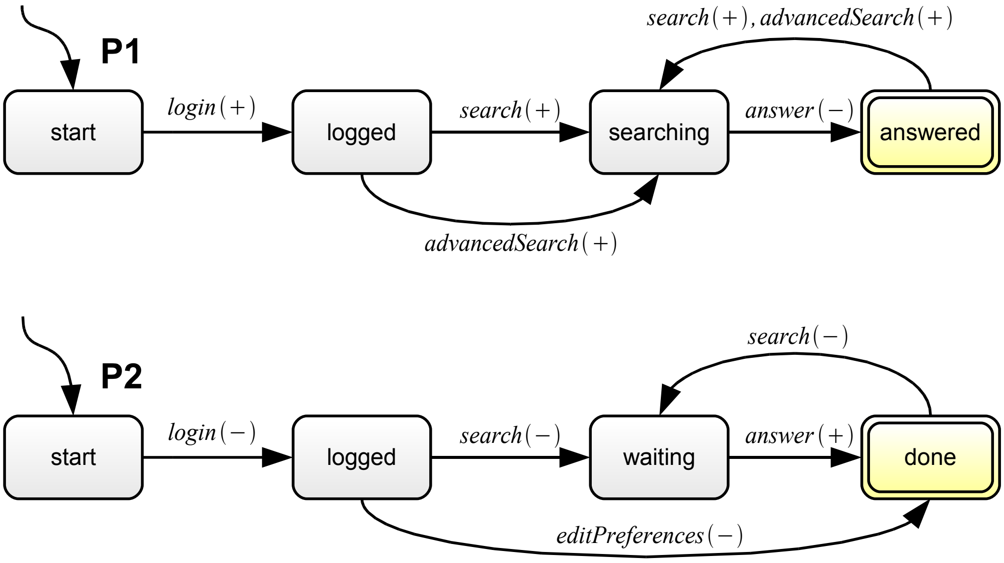


Figure 2.10: Two business protocol for illustrating analysis.

subsumption of P by P_1. The two protocols are however not equivalent as P does not support `advancedSearch` messages: P cannot replace P_1.

2.3.3 Discussion and related work

The following discussion first reviews common model families as business protocols opted for a state-based model. Notions such as orchestration, choreography and models for service abstractions are then briefly reviewed. Finally, research projects for web services life-cycle management are mentionned.

Model families. We consider the following three families: state-based models, activity-based models and rule-based models. State-based models can be found in a variety of contexts, mainly for describing behavioral abstractions of a system. In particular, the UML models family ([103]) contains several of them, acting as different "facets" over the representation of a system depending on the type of information that is to be captured (i.e., the general system behavior, the system behavior in a specific choreography, etc). We briefly recall the most important of them.

- Collaboration diagrams allow to represent the message exchanges between several (distributed) entities. Arrows are drawn between the entities, carrying both a message and number. The number (part of a total order) is used to define the messages chronology of the collaboration that is being modeled.

- Sequence diagrams are similar to the collaboration diagrams except that they use *lifelines* to denote when the object representing the entities are instantiated and discarded. Messages are exchanged between those lifelines, optionally having a number as in the case of collaboration diagrams. Indeed, the message exchange arrows are usually sorted vertically by chronological order.

- Statecharts capture all of the possible states of a given entity. The transitions between states are triggered on events (for example a message is received).

Activity-based models have been used to represent systems in an executable form (e.g, workflow management systems (Aalst [2])). States represent activities, and the transitions are triggered when the activities are finished. Each entity is given a *swimlane*, and an activity belongs to the swimlane of the executing entity. In some of those models, the transitions can carry data flows.

Rule-based systems (Forgy [54, 55]) define behavior through a set of *rules*. A rule is triggered when a given *event* occurs, or when a given *condition* is met. The rule then defines an action which is executed. Rule-based systems have been widely used where the systems behavior need frequent updates from domain experts writing the rules in domain-specific languages.

We chose to base our model upon the state-machine formalism. Indeed, it is commonly used to model the behavior of systems, due to the fact that it is simple and intuitive. Activity-based models are more suitable for creating executable models. Finally, rule-based models are a natural fit for complex decision-making systems where the logic must be frequently updated (e.g., by business analysts, accountants, etc.). They are however less suitable for describing behaviors.

Models for service abstractions. Research on web services has naturally lead to various models being proposed for capturing different types of abstractions. A discussion on modeling web services interactions has been proposed in (Bultan [39]) and is further discussed in (Bultan et al. [41]). An approach for web services interfaces was defined in (Beyer et al. [28]). A model with similar goals as timed protocols had been introduced in (Berardi et al. [25]), but the timing constraints defined in the model have yet to be taken into account. A language for web services choreographies called *Chor* has been proposed in (Qiu et al. [114]) as a simplification of WS-CDL. WS-CDL (see http://www.w3.org/TR/ws-cdl-10/) is the reference specification for choreographies. Tools leveraging WS-CDL for facilitating development of SOA systems are currently emerging (Kang et al. [76]). BPEL also offers *abstract process* for describing the externally observable behavior of a service composition. Lesser-known alternatives to BPEL include YAWL (van der Aalst and ter Hofstede [134]; van der Aalst et al. [135]), a general-purpose workflow language that has support for web services.

All of those approaches share many similarities with this work and the base model for business protocols of (Benatallah et al. [21]). Surprisingly, little work has been done on timing abstractions. Also, those works do not provide a fine-grained range of compatibility and replaceability analysis classes, which makes our work original. Finally, business protocols and WS-CDL are complementary, not competing proposals.

Life-cycle management research projects. The *ServiceMosaic project* (Benatallah et al. [22]) aims at developing a model-driven framework for web service life-cycle management. It has a CASE tools set for supporting the service development life-cycle that includes facilities for modeling, analyzing, discovering and adapting web service models (Benatallah et al. [22]; Nezhad et al. [99]). The work presented in this thesis is part of this project. ServiceMosaic is presented in larger details in chapter 7.

The *Self-Serv* project was the precursor of the larger-scope ServiceMosaic project: (Benatallah et al. [15, 17]; Sheng et al. [125]). The goal of Self-Serv was to allow for rapid composition of web services through a declarative approach. The execution of the obtained composite services was made in peer-to-peer, dynamic environment.

The project called *Astro* is another platform for the life-cycle management of web services (Trainotti et al. [131]). It features several contributions in the area of business requirements and verification, service synthesis and composition, and semantic web services. While it shares some similarities with ServiceMosaic (e.g., adaptation, supporting composition, Eclipse-based implementation), it focuses more on orchestration models (based on BPEL) than choreographies (e.g., the business protocol that a service supports). There is also no model discovery component such as the protocol discovery from execution log tools suite that ServiceMosaic provides (Nezhad et al. [98]). Finally, some works in Astro consider timing abstractions (Kazhamiakin et al. [78]). However, the work that is presented in this paper mainly reuses well-known timed automata model-checking techniques in service-based compositions. As we will see later, our approach makes a novel use of timed automata for obtaining the decidability and closure of protocol operators, not for performing straightforward, traditional model-checking.

While business protocols provide *temporal constraints* (i.e., the messages ordering), they do not provide *timing constraints* (e.g., a payment must be performed at most 48 hours after having purchased). This thesis revisits the concepts of business protocols modeling and analysis and enhances them using timing constraints. The result is a model that supersedes business protocols with added timing constraints and adapted analysis techniques. As we established a link between our extended model and timed automata (Alur and Dill [6]) to obtain theoretical results on the time-aware analysis of protocols, we present them in the next chapter.

Three

Timed automata

We give here some background knowledge on timed automata, a model that has been extensively used in the field of real-time model checking, and that will be useful for the remainder of this work. We begin with an overview of the model before focusing on the common problems (e.g., closure under complementation and reachability analysis). We then review some classes of timed automata that are interesting for this work, before finishing with model checking tools.

This will turn out to be useful later in the remainder of this work. Indeed, our approach for analyzing services business protocols completely depends on the closure under intersection and complementation in timed automata. It also depends on the ability to perform location reachability analysis using a model checking tool.

3.1 Overview

We start with an overview of the model of timed automata, including its semantics. We then turn our attention to the "classic" problems such as closure under intersection or language inclusion.

3.1.1 Model and semantics

Timed automata were introduced in (Alur and Dill [6]) as an extension of classical automata (Hopcroft et al. [74]) to model real-time systems. Some preliminary works appeared in (R. Alur et al. [115]). We start by an informal example before going through the formalization and semantics of the model.

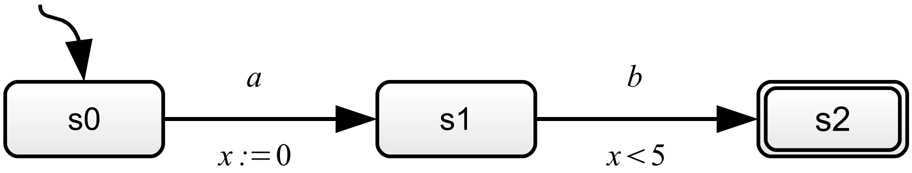


Figure 3.1: A sample timed automaton A.

Extension of automata with clocks. We take as an example the timed automaton A depicted on Figure 3.1. 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. Hence, an (untimed) word $a \cdot b$ can be recognized by A while words such as $a \cdot a$ or $b \cdot a$ cannot. 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 $\mathbb{R}_{\geq 0}$. Of course, a timed automaton can have more than one clock. 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. Clocks can be used in constraints attached to the switches, called guards, and that can enable or disable a switch depending on how those boolean functions over the clocks evaluate. 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.

Time elapses in the locations, while the switches are instantaneous. It is also possible to bound the time spent in the locations by defining clocks constraints called *invariants*, which can be used to force switches to be fired before the invariants conditions become violated. Indeed, it is a requirement that time can always elapse in the locations. Timed automata recognize *timed words* which are sequences in which a non-negative real value is attached to each symbol. For example $w = (a, 0) \cdot (b, 1)$ is a timed word where b has been recognized 1 unit of time after a. More precisely, a timed word over an alphabet Σ is a sequence $(a_0, t_0) \cdot (a_1, t_1) \cdots (a_n, t_n)$ such that $a_i \in \Sigma$, $t_i \in \mathbb{R}_{\geq 0}$ and $t_0 < t_1 < \cdots < t_n$.

Clocks valuations. In what follows, we reuse the same notations as in (Alur and Dill [6]; Rajeev Alur [117]). Given a set of clocks X with their values being in $\mathbb{R}_{\geq 0}$, a clocks valuation v for X is a function $X \longrightarrow \mathbb{R}_{\geq 0}$ that associates to each clock $x \in X$ its value $v(x)$. The set of clocks valuations for X is written $\mathbb{R}^X_{\geq 0}$. The set of clocks constraints over X is $\mathcal{C}(X)$, built using boolean combinations of atomic constraints of the form $x \# c$ with $x \in X$, $\# \in \{=, \neq, <, \leq, >, \geq\}$ and $c \in \mathbb{Q}$. $\mathcal{C}_{\prec}(X)$ is the restrictions of $\mathcal{C}(X)$ where the clocks constraints are of the form $x < c$ or $x \leq c$. 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 complete constraint 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, given a subset of clocks $r \in X$, $v' = [r \leftarrow 0]v$ is the clocks valuation such that $v'(x) = 0$ if $x \in r$ and $v'(x) = v(x)$ if $x \in X \setminus \{r\}$.

Definition 1 (Timed automata). *(Alur and Dill [6]; Rajeev Alur [117])*
A timed automaton A is a tuple $A = (\Sigma, L, L^0, L^f, X, I, E)$ where:

- Σ *is the* alphabet*,*
- L *is a finite set of* locations *(or states),*
- $L^0 \subseteq L$ *is the set of* initial locations*,*
- $L^f \subseteq L$ *is the set of* final locations *(or accepting states),*

- *X is a finite set of clocks,*
- *$I : L \to \mathcal{C}_{\prec}(X)$ associates an* invariant *to each location*
- *$E \subseteq L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ is a finite set of switches (or switches) $e = (l, g, a, r, l') \in E$ from l to l', where g is the guard, r is the set of clocks to be reset and a is the label.*

A timed automaton A recognizes timed words that are sequences of the form $(a_1, t_1) \cdot (a_2, t_2) \cdots (a_n, t_n)$ where given $i \in \{1, 2, \cdots, n\}$, $a_i \in \Sigma$ is a symbol from the alphabet and $t_i \in \mathbb{R}_{\geq 0}$ is the time at which a_i is recognized by A. The set of timed words recognized by a timed automaton A is the timed language denoted as $L(A)$.

Semantics. The semantic of a timed automaton A is defined using an (infinite) timed *labeled transitions system* (LTS). Each state of the LTS is a pair $(l, v) \in L \times \mathbb{R}^X_{\geq 0}$, called a *configuration*, where l is the current location in A and v is a clocks valuation. There are two types of transitions: *action transitions* (labeled with a symbol of Σ) and *time transitions* (labeled with a real-valued duration). More precisely, the semantic of $A = (L, L^0, L^f, X, I, E, \Sigma)$ is given by the LTS $\mathcal{S}_A = (S, s_0, \to, \Sigma)$ where:

- $S = L \times \mathbb{R}^X_{\geq 0}$,
- $s_0 = (l_0, v_0)$ with $l_0 \in L^0$ and $v_0(x) = 0\ \forall x \in X$,
- $\to$ is the transition relation:
  - action transitions: $(l, v) \xrightarrow{a} (l', v')$ if and only if there exists $e = (l, g, a, r, l') \in E$ such that $v \models g$, $v' = [r \leftarrow 0]v$ and $v' \models I(l')$
  - time transitions: if $d \in \mathbb{R}_{\geq 0}$ then $(l, v) \xrightarrow{d} (l, v + d)$ if and only if $v + d \models I(l)$.

The LTS $\mathcal{S}_A$ starts from an initial state made from an initial location and each clock set to 0. Then, either instantaneous action transitions occur (possibly resetting some clocks), or time transitions allow the clocks to grow synchronously. The executions over $\mathcal{S}_A$ match the timed words that can be recognized by A when they start from the initial state of $\mathcal{S}_A$ and they can reach a final state.

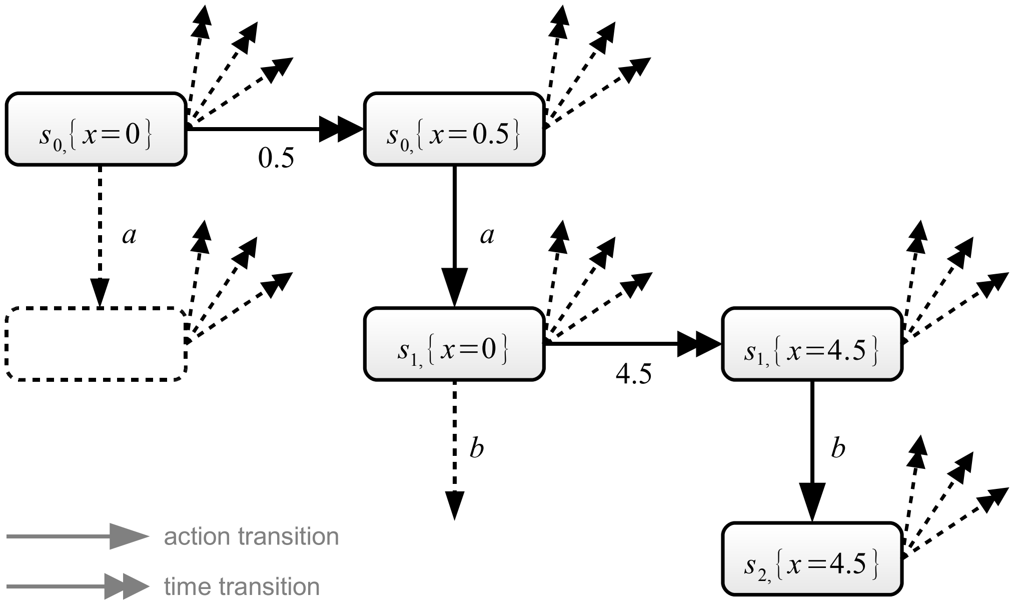


Figure 3.2: The LTS $\mathcal{S}_A$ associated to the timed automaton A of Figure 3.1.

Let us consider again A defined on Figure 3.1 and its semantic LTS $\mathcal{S}_A$ which is depicted on Figure 3.2. $(s_0, 0) \xrightarrow{1} (s_0, 1) \xrightarrow{0.25} (s_0, 1.25) \xrightarrow{a} (s_1, 0) \xrightarrow{0.1} (s_1, 0.1) \xrightarrow{b} (s_2, 0.1)$ is a valid execution over $\mathcal{S}_A$ which recognizes the timed word $(a, 1.25) \cdot (b, 1.35)$ of the timed language $L(A)$.

3.1.2 Classic problems

We now review some classic problems for timed automata. They are summarized in Table 3.1.

Closure properties. Closure of timed automata under the following operators has been studied in (Alur and Dill [6]): union, intersection, comple-

Problem	*Results*
Union	Closed
Intersection	Closed
Projection	Closed
Complementation	Not closed
Language inclusion	Undecidable
Language equivalence	Undecidable
Universality	Undecidable
Language emptiness / reachability analysis	PSPACE-Complete

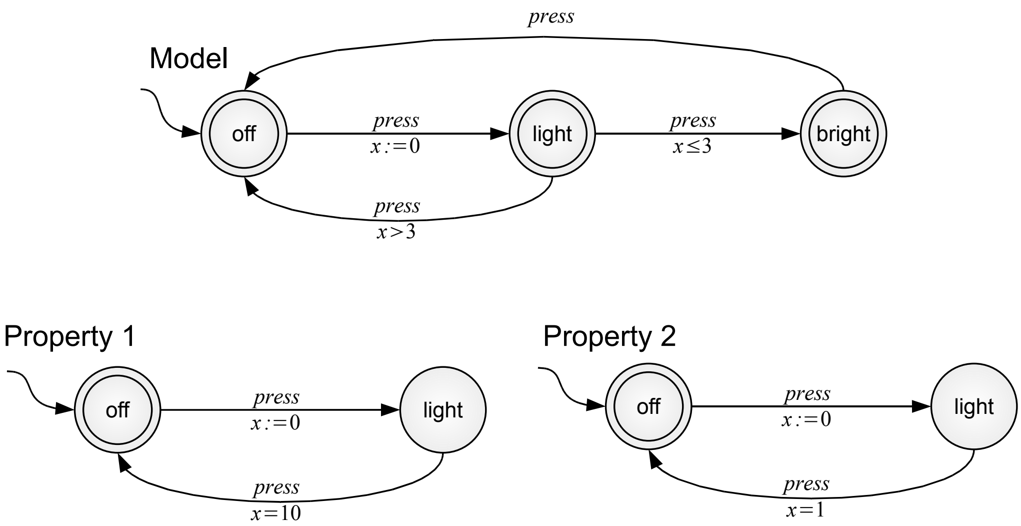
Table 3.1: Classic problems for (general) timed automata (Alur and Dill [6]).

mentation and projection[1]. Union and intersection are based on extensions of the classical procedures on automata (Hopcroft et al. [74]). The closure under union and intersection is relatively easy. Closure under union comes from the fact that timed automata are indeterministic and support more than one location. Intersection is similar. The problematic non-closure under complementation is a consequence of indeterminism, as a timed word may have more than one execution. The proof is based on the observation that a timed word can have an execution ending in a final location and one in a non-final location, making it impossible to construct the complement like for untimed automata.

Timed language inclusion, equivalence and universality. We briefly introduce these three decision problems. Given two timed automata A and A', the *timed language inclusion problem* refers to checking if $\mathcal{L}(A) \subset \mathcal{L}(A')$. Similarly, the *timed language equivalence problem* refers to checking if $\mathcal{L}(A) = \mathcal{L}(A')$. Finally, the *universality problem* refers to checking whether a timed automaton A defined over an alphabet Σ is able to recognize all of the possible timed words over Σ. These problems require the ability to complement timed automata. As an example, checking whether

[1] Projection allows to map a timed language from one to another with a different alphabet (symbols that are not mapped are replaced by ε).

$\mathcal{L}(A) \subset \mathcal{L}(A')$ reduces to checking for the timed language emptiness of the automaton $A' \cap \overline{A}$. Unfortunately, timed automata are not closed under complementation, making these problems undecidable. Checking for the emptiness of a timed language is a PSPACE-Complete problem. This problem is also referred to as the *location reachability* problem (i.e., a timed language is empty if no final location can be reached.



(*model from* http://www.cis.upenn.edu/~alur/Talks/sfm-rt-04.ppt)

Figure 3.3: A model checking example that reduces to state reachability analysis.

Verification through test automata. In this case, the property to verify is expressed as a timed automaton, referred to as a *test timed automaton*. The verification is performed by a reduction to a reachability / emptiness analysis. We give an example on Figure 3.3. The system that is modeled is a light controller. The light is initially turned off, and pressing it will turn it on. However, if the button is pressed twice quickly (within at most 3 units of time), then the light will be brighter. Finally, pressing the button another time will turn the light off.

We express two properties on this system using timed automata. The first one (property one) is used to test whether pressing the button once, then 10 units of time later will turn it off again. This is property that we expect to be satisfied by the modeled system. In turn, we do not expect the second property to be true: when pressed quickly twice, the light should not be turned off.

Model-checking through the mean of test automata is not always the best option, as verification reduces to a timed language inclusion checking problem. While checking for timed language emptiness is decidable (yet PSPACE-Complete), the non-closure under complementation is problematic. We will see later in this chapter that some classes of timed automata do not suffer from this issue. Back to the example of Figure 3.3, this is possible as the automata are deterministic.

3.2 Classes of timed automata

Several classes and extensions of timed automata have been studied. Indeed, the fact that the decision problems seen above are undecidable in the general case motivated such research directions. We summarize here the principal classes and extensions of timed automata that turned out to be useful in the context of this work. Table 3.2 summarizes the results regarding the timed language inclusion and emptiness checking problems, while Table 3.3 details the results on expressiveness. Further pointers can be found in (Bouyer and Laroussinie [32]; Joel Ouaknine and James Worrell [75]; Rajeev Alur [117]; Rajeev Alur and P. Madhusudan [118]; Stavros Tripakis [129]).

Deterministic timed automata. The class of *deterministic timed automata* has been defined in (Alur and Dill [6]). It restricts the definition of timed automata by:

1. allowing only a single initial location, and
2. requiring that given two switches from the same source location having the same input symbol, their guards are disjoint so as to preserve determinism.

As such, the timed automaton of Figure 3.1 is deterministic. This is also true for the timed automata depicted on Figure 3.3. Each timed word that

Class or extension	*Emptiness checking*	*Language inclusion*
Timed automata	PSPACE-Complete	Undecidable
Deterministic timed automata	PSPACE-Complete	Decidable
Event-clock automata	PSPACE-Complete	Decidable
Robust timed automata	PSPACE-Complete	Undecidable
ε-transitions without clocks resets	PSPACE-Complete	Undecidable
ε-transitions with clocks resets	PSPACE-Complete	Undecidable
Diagonal constraints $(x - y \# c)$	PSPACE-Complete	Undecidable
Additive constraints $(x + y \# c)$	Decidable for 1 or 2 clocks, open problem for 3 clocks and undecidable starting from 4 clocks (Bérard and Dufourd [24])	Undecidable
Constraints of the form $x = 2y$	Undecidable (Alur and Dill [6])	Undecidable
Constraints with irrational constants	Undecidable (Miller [93])	Undecidable
Non-standard $(x := 0)$ clocks resets	Decidable for $x := c$, undecidable for $x := x - 1$ and decidable for $x := x + 1$ if diagonal constraints are not allowed (Bouyer et al. [33])	Undecidable

Table 3.2: Emptiness and timed language inclusion checking results for some classes and extensions of timed automata.

is recognized by a deterministic timed automaton has exactly one accepting run over it. Deterministic timed automata form a strict subclass of timed automata (they are less expressive). The problem of the determinization of indeterministic (untimed) automata is decidable (Hopcroft et al. [74]). This is however not the case for timed automata (Stavros Tripakis [129]).

Event-recording timed automata. Event-clock automata, and more particularly *event-recording automata* form an interesting class of timed automata (Rajeev Alur et al. [119]). In this model, clocks are assigned to the input symbols. Each time one of symbol is recognized (e.g., a), its associated clock is reset (e.g., x_a). The time domain $\mathbb{T}$ is the set of positive reals $\mathbb{R}_{\geq 0}$ augmented with the $\perp$ symbol to denote the fact that a given symbol has not been recognized yet (i.e., every clock is set to $\perp$ initially).

Class or extension	*Observations*
Deterministic timed automata	Strictly less expressive than timed automata
Event-clock automata	Indeterministic event-clock automata can always be rendered deterministic while this is not the case for (general) timed automata (Rajeev Alur et al. [119])
Robust timed automata	Robust timed languages are *open* and their expressiveness is not comparable with the one of timed languages (Rajeev Alur and P. Madhusudan [118])
ε-transitions without clocks resets	The silent transitions can be removed (Volker Diekert et al. [136])
ε-transitions with clocks resets	Strictly more expressive than general timed automata: the silent transitions that reset clocks cannot be removed in general (Beatrice Berard et al. [13]; Bouyer et al. [35]; Volker Diekert et al. [136])
Diagonal constraints $(x - y \# c)$	More concise than timed automata, but not more expressive (Beatrice Berard et al. [13])
Additive constraints $(x + y \# c)$	More expressive than timed automata

Table 3.3: Expressiveness of some classes and extensions of timed automata.

The fundamental property in event-recording automata is that the value of clocks only depends on the input word. The consequence of this is that such automata are *determinizable* (i.e., an indeterministic event-recording automaton can always be transformed into a deterministic one that recognizes the same language) and *complementable* (i.e., event-recording automata are closed under complementation). Extensions of event-recording automata can be defined as long as the value of clocks only depends on the input word.

Silent transitions. In the case of untimed automata, they do not add to the expressiveness and they can be removed easily (Hopcroft et al. [74]). This is however not the case with timed automata, and as we will see later in this thesis, this will be a major issue for the decidability of timed protocol operators, as the model of timed protocols exhibits such transitions. Indeed, allowing silent (ε) transitions in timed automata strictly adds to the expressiveness (Beatrice Berard et al. [13]; Bouyer et al. [35]; Volker

Diekert et al. [136]). Complex techniques exist to remove the ε-transitions when they do not reset clocks (Volker Diekert et al. [136]). However, this is not possible in the general case when they do reset clocks, i.e., there exist switches of the form $l \xrightarrow{\varepsilon,g,\{x\}} l'$. Of special interest is the notion of *precise actions* defined in (Beatrice Berard et al. [13]), as it gives a tool for proving that some timed languages cannot be recognized by a timed automaton without ε-labeled switches. We will leverage this tool when studying the expressiveness of timed protocols.

Other classes. Some classes or extensions have different expressiveness levels compared to (general) timed automata. The expressiveness of the class of *Robust timed automata*[2] cannot be compared to the one of timed automata (Rajeev Alur and P. Madhusudan [118]). The classical decidability problems (reachability / emptiness, inclusion) remain unchanged.

The closure and decidability results remain unchanged compared to (general) timed automata in the following cases: constraints of the form $x = 2y$, constraints with irrational constants, and allowing non-standard (i.e., $x := 0$) clocks resets.

In most cases, the emptiness / reachability problem is decidable (yet PSPACE-Complete) despite the language inclusion problem being undecidable because of non-closure under complementation. However, some forms of modified clocks constraints and clocks resets can render it undecidable (Alur and Dill [6]; Beatrice Berard et al. [13]; Bérard and Dufourd [24]; Bouyer et al. [33, 35]; Miller [93]; Volker Diekert et al. [136]).

Other formalisms. Several formalisms have been introduce to capture timing constraints. Among them, a common abstraction is to define delays with minimum and maximum values on timed transition systems: (Henzinger et al. [70]; Lynch and Attiya [86]; Merritt et al. [92]; Pnueli and Harel [112]).

[2] Briefly, a robust timed automaton recognizes timed words with some fuzziness in the event dates as no real world system can be expected to be as precise as timed automata expectations.

Petri nets support extensions with timing requirements (Girault and Valk [64]). *Timed Petri nets* are an extension where a transition can be fired only if its enabling duration is in a certain time window (Merlin [91]). In turn, *Timed Petri nets* associate to each transition a set of time windows (Ramchandani [120]). The tokens that circulate in timed Petri nets are given an age. A transition can then be fired only by tokens whose ages are in one of the time windows of the transition.

Lots of contributions bridging timed automata and timed Petri nets have been made. The work done in (Bouyer et al. [34]) is of most interest as it proves that timed Petri nets are not more expressive than timed automata with respect to the language equivalence. It introduces a class called *Read-arc timed Petri nets* which is language-equivalent to timed automata.

Temporal logics. They have been traditionally used for formal verification purposes. A temporal logic formula expresses some form of *qualitative time* property such as *"when the light is off, pressing the button will turn it on"*. To put it another way, temporal logics allow to specify the order of events. However, it is not possible to express *quantitative time* properties such as *"when the light is off and the button is pressed twice in less than 2 seconds, the light will be bright"*.

The purpose of a temporal logic formula is to check wether there exists a path to a state that satisfies the formula in the considered system. Two branches of temporal logics exist:

1. *linear-time temporal logics* allow reasoning over a single time line, and

2. *branching-time temporal logics* allow reasoning over several time lines.

CTL, which stands for *computational tree logic*, is the traditional branching-time logic (Clarke et al. [45]) while LTL (*linear temporal logic*) is the traditional linear-time logic (Pnueli [111]).

Both CTL and LTL have been extended to support quantitative time. TCTL (*timed computational tree logic*) (Henzinger et al. [71]) extends the branching-time logic CTL. Similarly, MTL (*metric temporal logic*) (Koymans [81]) and TPTL (*timed propositional temporal logic*) (Alur and Henzinger [7, 8]) extend the linear-time temporal logic LTL to support quantitative time. Some fragments of MTL have been defined to seek better decidability results: as MITL (*metric interval logic*) (Alur et al. [12]), Safety-MTL (Ouaknine and Worrell [105]) and coFlat-MTL (Bouyer et al. [36]).

Logic	*Model-checking problem*
TCTL	PSPACE-Complete (Alur et al. [9, 10])
MTL over finite runs	Decidable and non-primitive recursive under the pointwise semantics (Ouaknine and Worrell [104])
	Undecidable under the continuous semantics (Alur et al. [12])
MTL over infinite runs	Undecidable under pointwise semantics (Ouaknine and Worrell [106])
TPTL over finite runs	Undecidable under the pointwise and continuous semantics (Alur and Henzinger [8])
MITL over infinite runs	EXPSPACE-Complete (Alur et al. [12])
Safety-MTL over infinite runs	Decidable but not primitive-recursive under the pointwise semantics (Ouaknine and Worrell [105])
coFlat-MTL over infinite runs	EXPSPACE-Complete under the pointwise semantics (Bouyer et al. [36])

Table 3.4: Results on the problem of model-checking using timed temporal logics.

The results on the problem of model-checking using timed temporal logics are summarized in Table 3.4. The branching-time logic TCTL offers very good decision problem properties while the linear timed temporal logics are much harder, if not decidable. The research on linear logics has been motivated by the fact that they are sometimes more powerful than the branching ones. A detailed survey on timed temporal logics can be found in (Bouyer [31]).

3.3 Software tools

We briefly introduce the general principle of *model checking* that we illustrated on Figure 3.4. It refers to the process of testing *properties* of a system for which a *model* is given. The *model checker* is a “black box” component

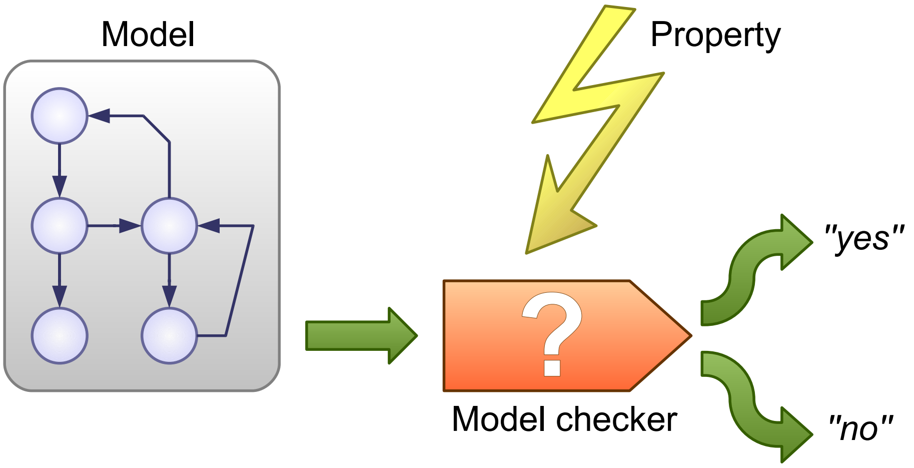


Figure 3.4: Principle of model checking.

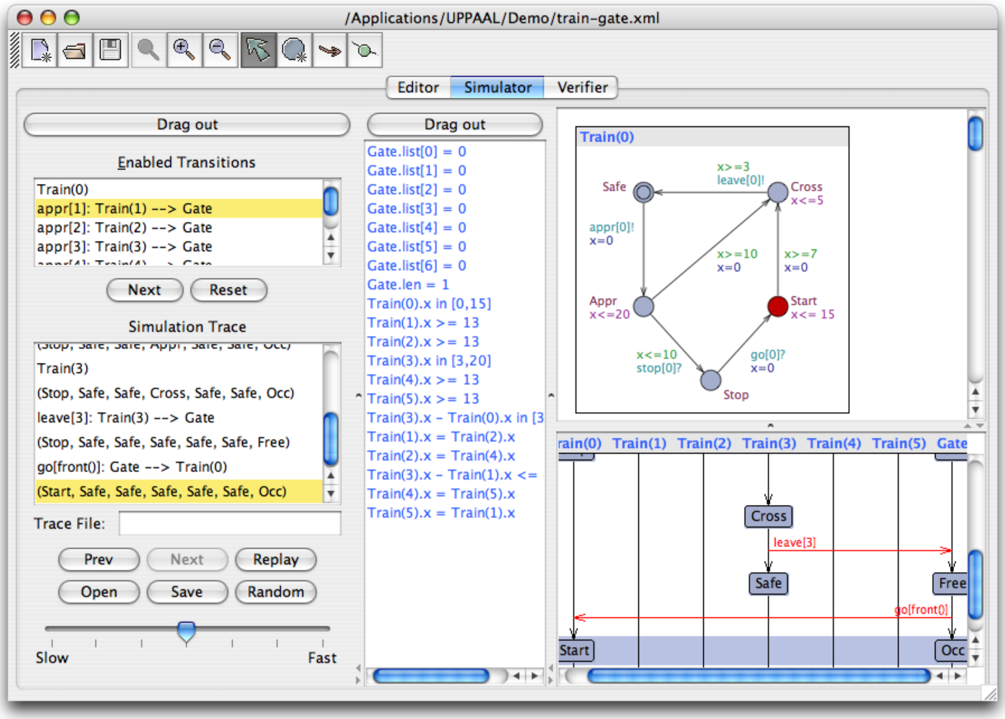
that takes both of them as inputs then outputs whether the property is satisfied or not. Some model checkers can also output a *trace*, mostly to give details of why the property cannot be satisfied (they are sometimes referred as *error traces*).

The types of properties to be checked are usually classified in the following categories, although a model checker does not especially distinguish them. *Reachability properties* check whether a property can possibly be satisfied by the system (e.g., *"the light can be switched off"*). *Safety properties* state that something which is considered as "bad" will never happen in the system (e.g., *"the light cannot stay on for more than 20 minutes"*). Finally, *liveness properties* state that something which is considered as "good" will eventually happen (e.g., *"when the light is off, pressing the button turns it on"*).

Various timed automata model checkers have been proposed. They target different classes of timed automata as a model. They leverage timed temporal logics for expressing properties and query the models. We briefly present the main tools.

Tool	*Model*	*Temporal logic*
Kronos (Bozga et al. [38]; Daws et al. [50])	Timed automata	TCTL
HyTech (Henzinger et al. [72])	Hybrid automata (Alur et al. [5, 11])	ICTL, an extension of TCTL (Alur et al. [11])
UPPAAL (Behrmann et al. [14])	Hybrid extension of timed automata	Subset of TCTL
Tempo (Maria Sorea [88])	Event-recording timed automata	*Event-recording fixpoint logic* defined in (Maria Sorea [88])

Table 3.5: Model checking tools for classes and extensions of timed automata.


/Applications/UPPAAL/Demo/train-gate.xml

Enabled Transitions

Train(0)
appr[1]: Train(1) --> Gate
appr[2]: Train(2) --> Gate
appr[3]: Train(3) --> Gate
appr[4]: Train(4) --> Gate

Simulation Trace

(Stop, Safe, Safe, Appr, Safe, Safe, Occ)
Train(3)
(Stop, Safe, Safe, Cross, Safe, Safe, Occ)
leave[3]: Train(3) --> Gate
(Stop, Safe, Safe, Safe, Safe, Safe, Free)
go[front()]: Gate --> Train(0)
(Start, Safe, Safe, Safe, Safe, Safe, Occ)

Trace File:

```
Gate.list[0] = 0
Gate.list[1] = 0
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 1
Train(0).x in [0,15]
Train(1).x >= 13
Train(2).x >= 13
Train(3).x in [3,20]
Train(4).x >= 13
Train(5).x >= 13
Train(3).x - Train(0).x in [3
Train(1).x = Train(2).x
Train(2).x = Train(4).x
Train(3).x - Train(1).x <=
Train(4).x = Train(5).x
Train(5).x = Train(1).x
```

Figure 3.5: A screenshot of UPPAAL.

The tools are presented in Table 3.3. Not surprisingly, branching timed temporal logics are popular given the decidability of the model checking problem. Tempo is specialized on event-recording timed automata and uses a logic developed on purpose. Kronos is the only tool to use "canonical" timed automata while UPPAAL and HyTech both use hybrid extensions. Briefly, an hybrid system (Alur et al. [5]) features both continuous (e.g., variables in $\mathbb{R}$) and discrete behavior (e.g., variables in $\mathbb{N}$). At the time of the writing of this document, it should be noted that UPPAAL is the only actively developed project. UPPAAL has also managed to emerge from an academic research prototype to a commercial product for model checking. It has been used in several industrial studies (a complete list is available at http://www.it.uu.se/research/group/darts/uppaal/examples.shtml). We give a few references containing such case studies: (Aceto et al. [3]; Bengtsson et al. [23]; Bowman et al. [37]; D'Argenio et al. [47]; David and Yi [49]; Havelund et al. [68, 69]; Lindahl et al. [84]; Lönn and Pettersson [85]).

A screenshot of the UPPAAL simulation environment is available on Figure 3.5. A more detailed presentation of UPPAAL is available in the appendix from page 177.

Part II

Protocol modeling and analysis

Four

Timed protocol modeling

This chapter presents the model of *timed business protocols*. It is a formalization of the business protocol introduced in (Benatallah et al. [21]) and the timing constraints primitives that were mentioned in (Benatallah et al. [16, 18]). By investigating (online) applications to define which type of timing abstraction would turn out to be useful in practice for describing the externally-observable behavior of services, we identified two primitives: C-Invoke constraints for defining time windows of availabilities and M-Invoke constraints for defining implicit actions that occur when some forms of delays expire. The introduction of timing constraints into the model poses significant challenges, especially when it comes to studying the impacts on protocol analysis as we will see in the next chapters. We obtained a characterization of timed protocols and their operators by establishing a link with the field of timed automata (Alur and Dill [6]). This way, we will see that introducing timing constraints in business protocols is a significant challenge as it makes protocol analysis much more complex than in the untimed case. More specifically, we will see that with timed automata, complementation and subsumption are generally undecidable while M-Invoke constraints are hard to express.

The chapter is structured as follows. We first provide a concise discussion on the timing abstractions that we identified. Then, we present the model of timed business protocols that we envisioned as a conceptual model. Next, we explain how timed protocols can be equivalently expressed using *timed automata* (Alur and Dill [6]), a formal framework that has been extensively investigated in formal verification research works (Behrmann et al. [14]; Rajeev Alur and P. Madhusudan [118]). This will later allow us to derive theoretical properties for the model of timed protocols by establishing a semantic-preserving mapping between the two models. Finally, we provide a discussion.

4.1 Timing abstractions

Before designing our web services business protocols model, we started by investigating the timing abstractions that were needed to capture the ones that would be useful in practice. We first present three motivating examples, then introduce the two timing abstraction primitives that we have identified.

4.1.1 Motivating examples

Timing constraints appear in countless scenarios. We give here three examples related to web services and business processes were timing constraints play a critical role.

E-Commerce portals

The sales conditions notices of many E-Commerce portals provide timing 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 involving the effective payment is made. The seller implicitly releases the seat holding and cancels the purchase process after a delay (e.g., 24 hours) without a confirmation or canceling notification from the buyer. There are other examples in the field of E-Commerce examples. 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. They are often also constrained to respect delays when dealing with customers for operations such as the deliveries or the refunding of returned purchases.

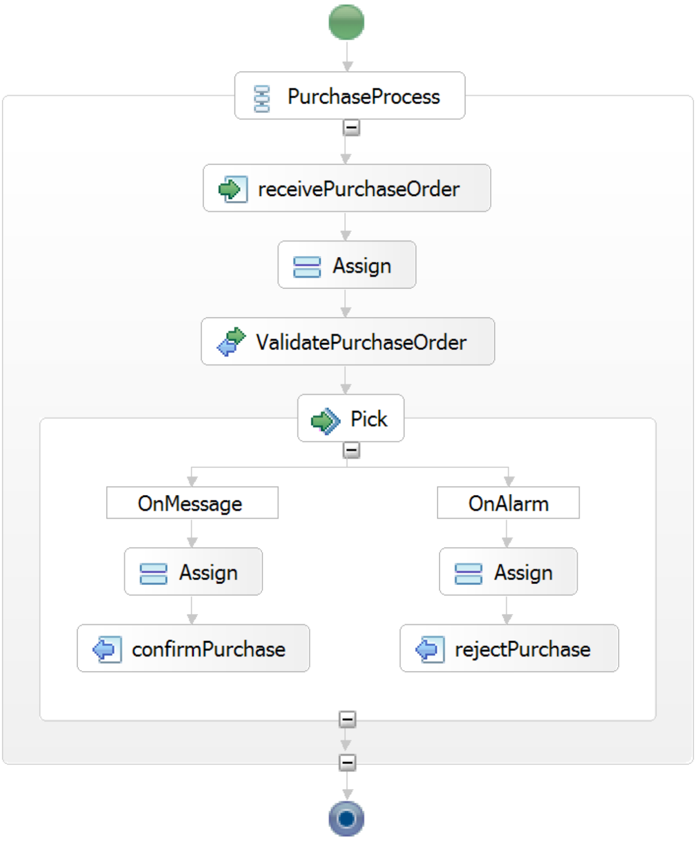


Figure 4.1: A BPEL process that uses an alarm.

BPEL processes

The *Business Process Execution Language* (OASIS [102]) is the major specification in the field of long-running web services orchestrations. A BPEL process is able to achieve a complex business process using a set of web services. Figure 4.1 exhibits a sample purchasing BPEL process. The process starts by receiving a purchase order. A BPEL *Assign* operation is then executed to prepare the data for the invocation of a service providing a *ValidatePurchaseOrder* operation. This operation is used to delegate the processing of the purchase order to a service that will check for various conditions like the ordered items availability, the existence of contracts between the requesting entity and the provider, the validity of the chosen payment option and checking that the requesting entity does not have outstanding debts to the provider. Given that those checkings may take a substantial amount of time (for examples humans have to be involved for some of the checkings), the process uses a *pick* block. It allows to wait for receiving a response that validates the order. If no response has been received after an arbitrary delay such as 48 hours, the process is woken-up to reject the purchase. BPEL actually exhibits two similar time-related constructs: *onAlarm* and *wait*. The former can be used in *pick* blocks, while the later is a normal BPEL operation that will put the process execution on hold for a duration or until a fixed date is reached.

RosettaNet PIPs

RosettaNet (RosettaNet [121]) is an industrial consortium which aims at facilitating the transactions among the supply chains of trading partners using specifications called the *Partner Interface Processes* (PIPs) that thoroughly detail the processes involved in those transactions. The RosettaNet PIPs exhibit time-dependent behaviors. Let us focus on the PIP 3A4 depicted on Figure 4.2. It specifies how a seller and a buyer can process a purchase order. This PIP specifies the following timing constraints:

- the *PurchaseOrderRequestAction* and the *PurchaseOrderConfirmationAction* must be acknowledged within 2 hours
- the reply to the *PurchaseOrderRequestAction* must be sent within 24 hours.

4.1.2 Timing abstraction primitives

The three motivating examples that we gave illustrate the need for capturing timing abstractions into web services and business process models. We did not limit our investigations to these examples in order to identify

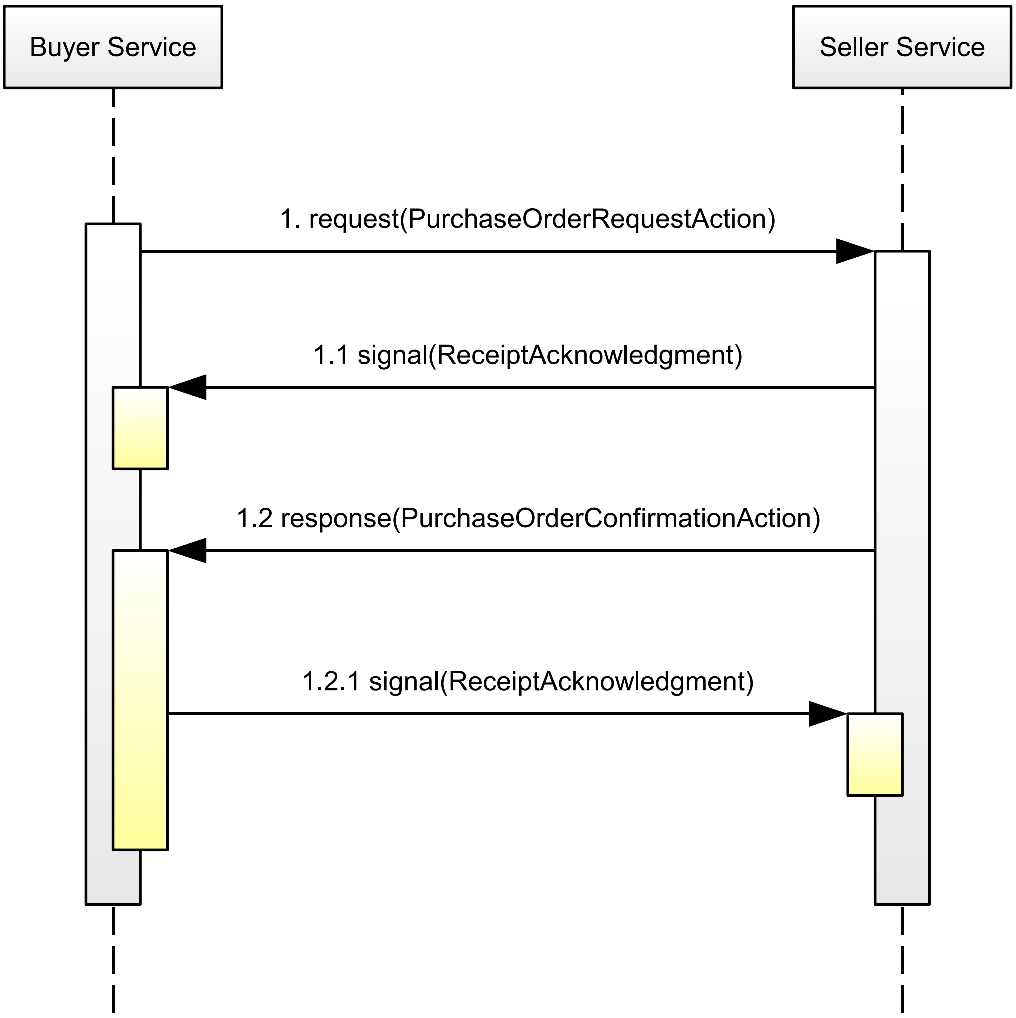

Buyer Service
Seller Service
1. request(PurchaseOrderRequestAction)
1.1 signal(ReceiptAcknowledgment)
1.2 response(PurchaseOrderConfirmationAction)
1.2.1 signal(ReceiptAcknowledgment)

Figure 4.2: Sequence diagram of the RosettaNet PIP 3A4 (Request Purchase Order).

the primitives that would be useful to capture the vast majority of cases appearing in practice. A major difficulty when designing a model is to find a good expressiveness level tradeoff. Indeed, a simplistic model does not capture all of the cases while an overly complex one becomes hard to use for practitioners, and may lead to undecidability problems in algorithms.

The starting point of this work was done in (Benatallah et al. [16]; Karim Baina et al. [77]). Using a similar approach based on the exploration of existing web portals and E-Commerce websites, we found that the following timing primitives would be useful.

Time windows. Many actions need to be available only during certain time windows. For example, a quotation usually has a finite validity, and a purchase cannot be made in the same conditions once it has expired. Another example in the commerce domain is the case of discounts which can be valid only for a limited period of time.

Timeouts. Most software systems support the notion of timeout, for example to avoid locking resources for an unlimited amount of time and to handle partnering components errors. Considering again an example in the commerce domain, a purchase order needs to be paid to be validated. A seller system will usually discard the business transaction after a given delay if the buyer did not actually send the payment.

4.2 Extending business protocols with temporal abstractions

We now present the *timed business protocol* model. We first illustrate it informally, then provide a formalized definition.

4.2.1 Timed business protocols

Our model is an extension of the *business protocol* model (Benatallah et al. [18, 21]) which is built upon the traditional state-machine formalism. Indeed, it is commonly used to model the behavior of systems, due to the fact that it is 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.

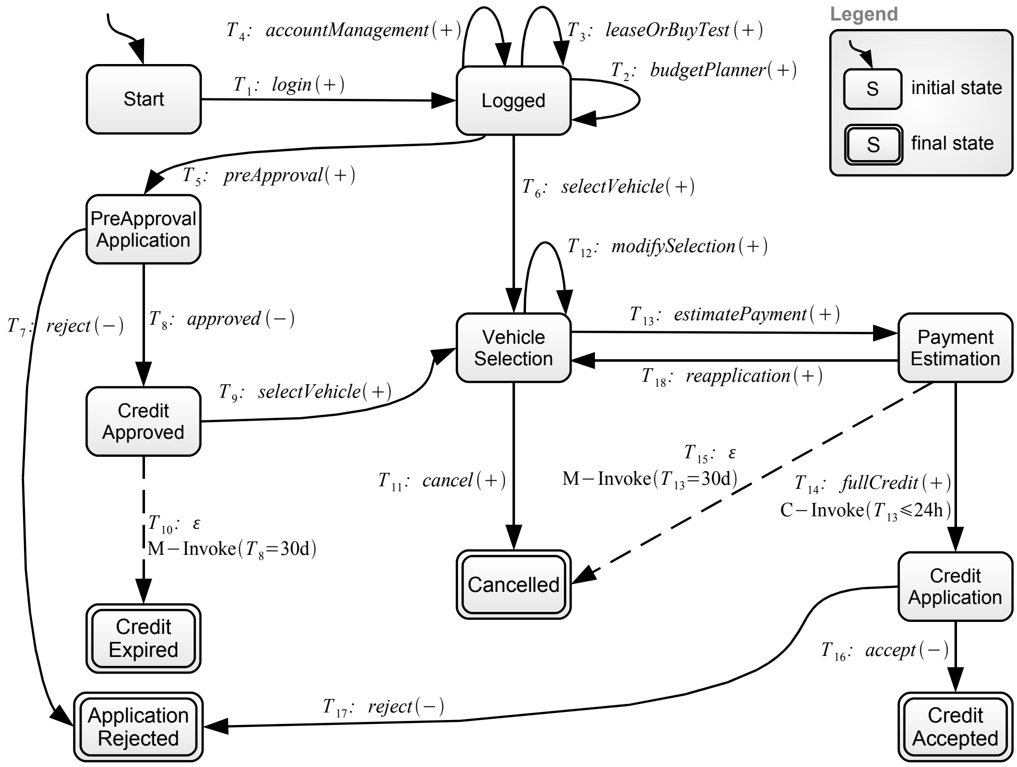


Figure 4.3: A timed protocol of an online financing service.

As an example, the protocol depicted on Figure 4.3 (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 need to be able to determine in which state the service is, else 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). They are not related to message exchanges between the service provider and its requesters. Implicit transitions are analogous to the silent transitions in the automata theory (Hopcroft et al. [74]) and we associate the empty word ε as the label of those transitions. There is however a significant difference compared to "classical" automata: they cannot be removed in the general case as we will see later.

Continuing with the example protocol depicted on Figure 4.3, 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}.

Both C-Invoke and M-Invoke constraints are orthogonal. Indeed, we will see in Theorem 1 that the transitions having M-Invoke constraints strictly increase expressiveness of our model. Naturally, the usefulness of C-Invoke constraints can be questioned as in many cases, a C-Invoke -free construction can be made by using extra states and M-Invoke constraints on implicit transitions. Let us consider the examples of Figure 4.4. The case (a) is simple as the C-Invoke constraint is easily translated using M-Invoke by adding an extra state s'. This is however not true in the general case as the (b) counter example shows as it yields the following problem: $k \in \mathbb{Q}$ cannot be determined as the set of rational numbers is infinite, hence we cannot find a value for k that would be "close-enough" to 3 for ensuring that $a(+)$ is only received when T_1 is exactly equal to 3.

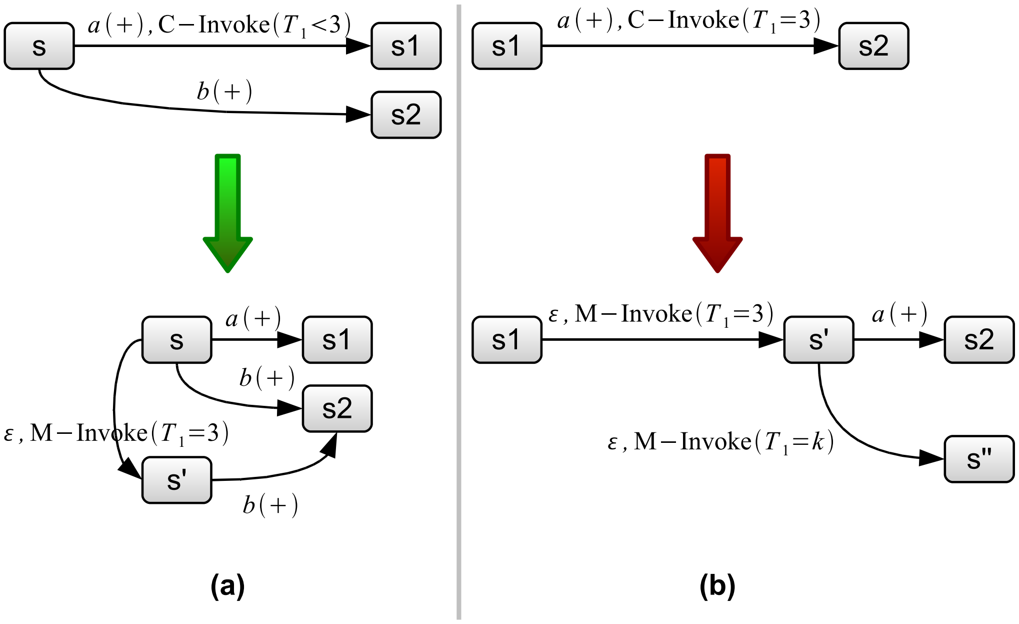


Figure 4.4: Examples for illustrating why both C-Invoke and M-Invoke are necessary.

4.2.2 Formalization

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 $k \in \mathbb{Q} \cup \{\bot\}$, 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 (Benatallah et al. [18, 21]).

Definition 2. *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* ε*. 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 $c_1 \wedge c_2 \equiv$ `false`.

Semantics

Variable interpretation. Defining the timed protocol semantics requires the introduction of the notion of variable valuation. 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

$$\mathcal{V} : \mathcal{X} \to \mathbb{R}_{\geq 0} \cup \{\perp\}$$

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 $\sigma = s_0 \cdot (m_0, t_0) \cdot s_1 \dots s_{n-1}.(m_{n-1}, t_{n-1}) \cdot s_n$ 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 $op \in \{<, \leq, =, \neq, >, \geq\}$:

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

Given a variable valuation $\mathcal{V}$ and a constraint C-Invoke(c) (respectively, M-Invoke(c)), we note 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 (Alur and Dill [6]).

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),

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 $\sigma' = \mathit{Start} \cdot (\mathit{login}(+), 0) \cdot \mathit{Logged} \cdot (\mathit{preApproval}(+), 1) \cdot \mathit{PreApprovalApplication} \cdot (\mathit{approved}(-), 3) \cdot \mathit{CreditApproved} \cdot (\varepsilon, 33) \cdot \mathit{CreditExpired}$ is a correct execution of the financing service protocol depicted on Figure 4.3.

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 σ' of the above service protocol leads to the timed trace $tr(\sigma') = (\mathit{login}(+), 0) \cdot (\mathit{preApproval}(+), 1) \cdot (\mathit{approved}(-), 3) \cdot (\varepsilon, 33)$.

During an execution σ 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 σ', the observable behavior of the financing service is described by the timed conversation $obs(\sigma') = (\mathit{login}(+), 0) \cdot (\mathit{preApproval}(+), 1) \cdot (\mathit{approved}(-), 3)$.

In the following, given a protocol $\mathcal{P}$, we denote by $Tr(\mathcal{P})$ 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 4.3 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 τ is a timed interaction trace of $\mathcal{P}$ and $\mathcal{P}'$ if and only if there exist two timed conversations σ_1 and σ_2 such that:

1. $\sigma_1 \in Tr(\mathcal{P})$ and $\sigma_2 \in Tr(\mathcal{P}')$, and

2. σ_1 is the reverse conversation of σ_2 (i.e., the conversation obtained from σ_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 σ by removing the polarities of the messages.

4.3 From timed protocols to protocol timed automata

The previous section has introduced the model of timed business protocols which is suitable for describing the external behavior of web services, including timing constraints. In turn, the model of timed automata (Alur and Dill [6]) has been extensively studied as an extension of classical automata (Hopcroft et al. [74]) with real-valued clocks and conditions on the transitions. Given the extensive research that has been made on timed automata, we chose to:

1. use timed protocols as a conceptual model, and
2. use timed automata for implementing the behavior of timed protocols, and
3. derive theoretical properties of timed protocols from existing work on timed automata.

To do that, we give a mapping from timed protocols to a new class of timed automata that we identified, called *protocol timed automata*. However, defining such a mapping is not a trivial task as we will see that M-Invoke constraints are not straightforward to implement using timed automata.

This section is structured as follows. We start by an informal, illustrated walk through the challenges of converting a timed protocol to a timed automaton that correctly implements its behavior. We then briefly define the class of protocol timed automata. Then, we formally study the M-Invoke constraints enforcement techniques for timed automata.

4.3.1 Informal overview of the challenges

We will use the timed protocol from Figure 4.5 as a running example throughout this section. It will allow us to illustrate the challenges in creating a timed automaton that behaves like a timed protocol. At first

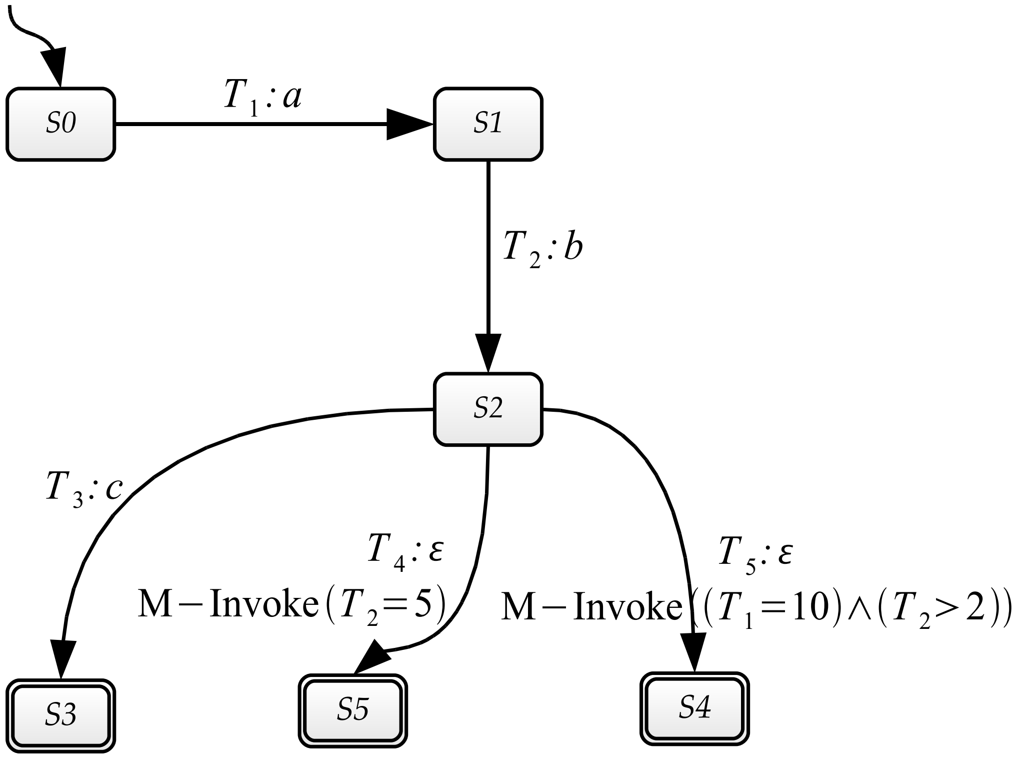


Figure 4.5: A sample timed protocol $\mathcal{P}$ used as a mapping running example.

sight, the translation may look straightforward. However as we will see, preserving the behavior of M-Invoke is surprisingly not an easy task. The translation is performed using a mapping that we now illustrate. The mapping is done in two steps.

The first step to convert a timed protocol $\mathcal{P}$ into a timed automaton A is relatively simple. The mapping of the states of $\mathcal{P}$ to the locations of A is direct (e.g., the state s_0 of $\mathcal{P}$ is mapped to a location l_0). Similarly, each transition in $\mathcal{P}$ is translated as a switch with the message name as its label (e.g., in $\mathcal{P}$, $T_1 : \mathcal{R}(s_0, s_1, a, \texttt{true})$ is translated as a switch $e_1 = (l_0, \texttt{true}, a, r, l_1)$ in A). Also, the initial location and the final locations in A correspond to the initial and final states in $\mathcal{P}$.

The transition identifiers from $\mathcal{P}$ are used to generate clocks in A. In-

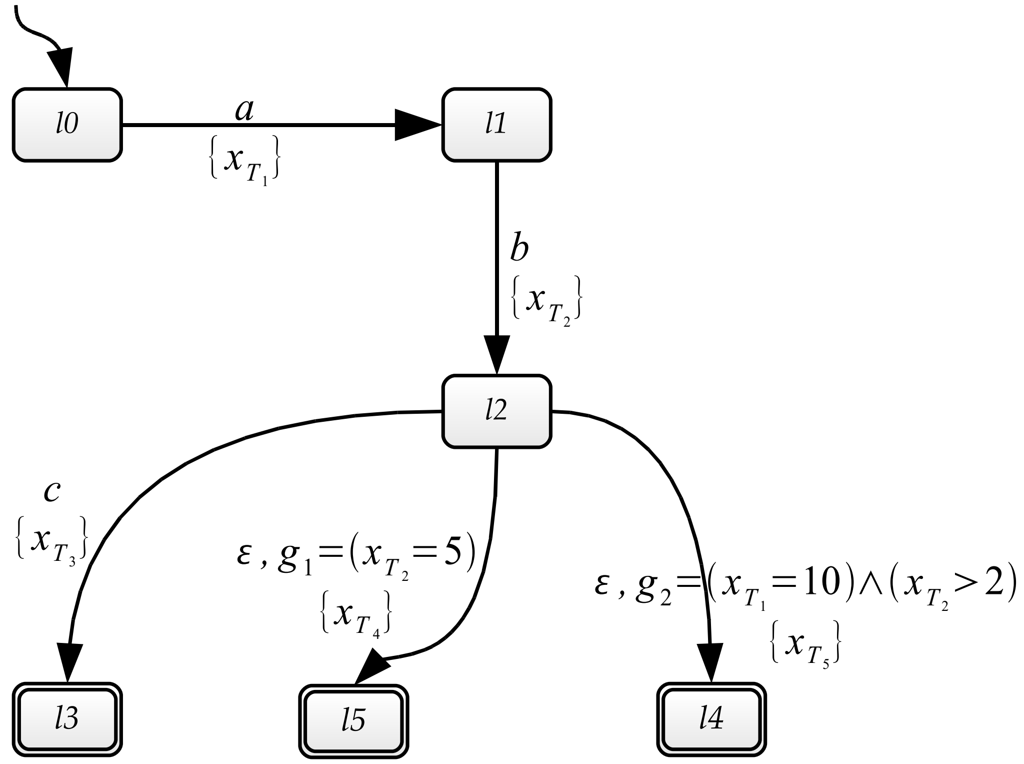


Figure 4.6: A sample timed automaton that does not enforce M-Invoke semantics.

deed, 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 4.6.

Assuming that we mapped implicit transitions as ε-labeled switches with a direct translation of M-Invoke constraints into guards, the following problems arise on the timed automaton of Figure 4.6 when the current location is l_2.

1. **Indeterminism:** g_1 and g_2 from Figure 4.6 can potentially become

satisfied at the same time, and the c-labeled transition can be taken at the same time as either g_1 or g_2 is satisfied.

2. **Semantics of M-Invoke constraints:** if one of the labeled switch guard g_1 or g_2 Figure 4.6 becomes satisfied while in l_2, it is not necessarily taken. Worse, the c-labeled switch can still be activated afterward.

As a consequence, the timed automaton of Figure 4.6 does not respect the semantics of M-Invoke constraints, and they require more investigations to both enforce that ε-labeled switches are taken when their guards are satisfied, and that the other switches become "disabled" as soon as those guards become satisfied. As we will see, indeterminism is relatively easy to solve. However, ensuring that the semantics of M-Invoke constraints are enforced is a more involved task.

Let us have closer look at the various cases when entering a location that offers one ε-labeled switch. We illustrate that again on the location l_2 and the guard g_2 of Figure 4.6. Three cases are possible. The first one is that l_2 is entered before g_2 has been satisfied (i.e., $(x_{T_1} < 10)$ is true). In this case c can be recognized as long as $(x_{T_1} < 10)$. 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 recognized at any time since g_2 is not going to be satisfied for any successive clocks valuation. Finally, if l_2 is entered while $(x_{T_1} < 10)$, for any clocks valuation such that $(x_{T_1} \geq 10)$, c can be recognized if and only if when $(x_{T_1} = 10)$ was true, $(x_{T_2} > 2)$ was false. The generalization to more than one constraint is similar, and will be detailed later.

Clearly, a direct translation of the constraints to guards is not enough to properly represent M-Invoke constructions in timed automata. Because of that, more elaborated constraints need to be added into the guards than a direct translation does. 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 ε-labeled switches (e.g., when l_2 is entered, do we have $(x_{T_2} < 5)$? $(x_{T_2} > 5)$?)

   b) to know if the ε-labeled switches guards will be satisfiable or not (e.g., g_2) when their equality clause is satisfied.

Especially, 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 there is no way the ε-labeled switch whose guard is g_1 can be enabled, and thus it should not disable the c-labeled switch nor the other ε-labeled switch.

In timed automata, the difference between 2 clocks x_1 and x_2 is a constant until either of them is reset as clocks evolve synchronously. Guards allow *diagonal constraints*, i.e., constraints of the form $(x_1 - x_2 \# k)$ in the guards (x_1, x_2 being clocks and $k \in \mathbb{Q} \cup \bot$). We use such diagonal constraints to capture the the clock valuations when locations are entered: for each location offering at least one ε-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 4.6, 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 clocks valuation of x_e when l was entered.

Using this technique to capture the clock valuations when a location is entered, the second step of the mapping can be done by rewriting the guards of every switch whose source location offers ε-labeled switches. The rewriting must take care of allowing normal (i.e., non ε-labeled switches) to recognize input symbols when M-Invoke constraints allow it, and block them otherwise. As we will see later in this chapter, we will introduce a clocks constraint function to capture *when* a given switch is allowed with respect to the guard of a ε-labeled switch.

Going back to the example of Figure 4.6 with a new clock y_{l_2} having been added, observe that g_1 must enable the other switches in the following two cases:

1. $(x_{T_2} < 5)$, and
2. $(x_{T_2} > 5) \wedge (x_{T_2} - y_{l_2} > 5)$

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:

1. $(x_{T_1} < 10)$, and
2. $(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} > 10)$, and
3. $(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} \leq 10) \wedge (x_{T_2} - x_{T_1} \leq -8)$, and
4. $(x_{T_1} = 10) \wedge (x_{T_2} - x_{T_1} \leq -8)$

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.

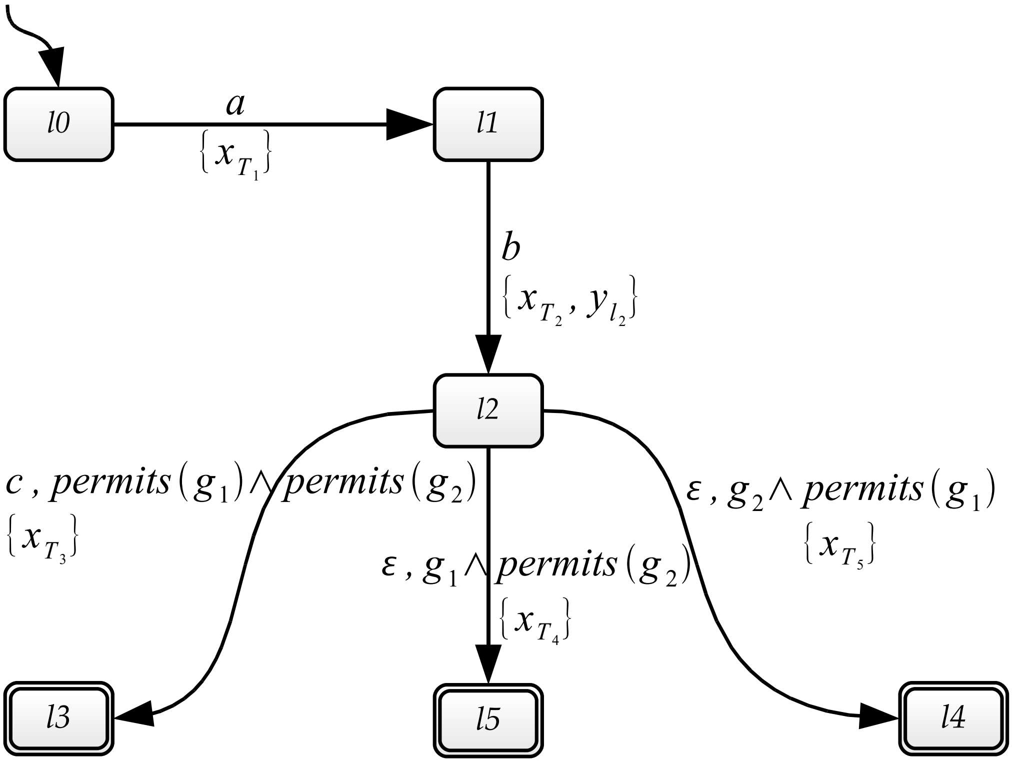

l0
a
$\{x_{T_1}\}$
l1
b
$\{x_{T_2}, y_{l_2}\}$
l2
$c, permits(g_1) \wedge permits(g_2)$
$\{x_{T_3}\}$
$\varepsilon, g_1 \wedge permits(g_2)$
$\{x_{T_4}\}$
$\varepsilon, g_2 \wedge permits(g_1)$
$\{x_{T_5}\}$
l3
l5
l4

Figure 4.7: A sample protocol timed automaton that does enforce M-Invoke semantics.

The correct mapping of $\mathcal{P}$ to A is given on Figure 4.7, where the "permits" functions are just the cases that we mentioned above. For example, permits$(g_1) = (x_{T_2} < 5) \bigvee x_{T_2} > 5) \wedge (x_{T_2} - y_{l_2} > 5)$.

4.3.2 Protocol timed automata

We named as *protocol timed automata* the class of timed automata (Alur and Dill [6]) that we obtain by the mapping from a timed protocol that we illustrated informally in the previous section. They form a novel class of timed automata with the following characteristics:

1. they support the empty word ε as a switch label, and
2. clocks are not freely added and reset on switches, they are instead split into two sets: one associates a unique clock to a unique switch while the second one associates a clock to every location that is the source of a ε-labeled switch, and
3. the guards of ε-labeled switches must have at least one equality clause per possible disjunction (i.e., a clause of the form $(x = k)$ with x being a clock and k a rational constant or $\perp$), and
4. they are deterministic, and
5. the time domain is the set of positive reals $\mathbb{R}_{\geq 0}$ augmented with the special symbol $\perp$[1].

The following definition gives the base of protocol timed automata. However, any automaton that satisfies this definition does not automatically enforce M-Invoke semantics. The techniques for doing that will be given later in this section.

Definition 3 (Protocol timed automata)**.** *Let $\mathbb{T}$ be the time domain defined as $\mathbb{T} = \mathbb{R}_{\geq 0} \cup \{\perp\}$. A* protocol timed automaton *A is a timed automaton $A = (L, L^0, L^f, X \cup Y, E, \Sigma)$ over $\mathbb{T}$ such that $|L^0| = 1$ and the following conditions hold.*

1. *For 2 switches $e = (l, g, a, r, l')$ and $e' = (l, g', a', r', l'')$ having the same source location l, either $a = a'$, or $g \wedge g' \models \texttt{false}$.*
2. *For every switch $e = (l, g, a, r, l')$, either:*
   a) *l' is the source of a ε-labeled switch: $r = \{x_e, y_{l'}\}$ with $x_e \in X$ being the clock attached to e and $y_{l'} \in Y$ being the clock attached to l', or*

[1] This symbol is used to denoted the fact that a clock is still at its initial value and has never been reset.

   b) l' is not the source of a ε-labeled switch: $r = \{x_e\}$ with $x_e \in X$ being the clock attached to e.

It should be noted that the reset set of any switch is composed of 1 or 2 clocks.

3. *For every switch $e = (l, g, a, r, l')$ such that $a = \varepsilon$, g is defined as a disjunction of conjunctive constraints having each at least one clock equality clause: $g = (x_1 = k_1 \wedge g'_1) \vee \cdots \vee (x_n = k_n \wedge g'_n)$ with x_i being any clock, $k_i \in \mathbb{Q} \cup \{\perp\}$ a rational constant or $\perp$, $n \in \mathbb{N}$ and g'_i being any clocks constraint ($i \in \{1, \cdots, n\}$).*

4.3.3 Enforcing M-Invoke constraints in protocol timed automata

We define the M-Invoke semantics so as to enforce that when the guard of a ε-labeled switch becomes satisfied, it is mandatorily taken. Indeed, classical timed automata augmented with ε-labeled switches only make the location change *possible* when a guard is satisfied, not *mandatory*.

Definition 4 (M-Invoke semantics)**.** *Let A be a protocol timed automaton and $e : l \xrightarrow{\varepsilon, g} l'$ a ε-labeled switch of A. Let v be the current clocks valuation such that:*

1. *the current location is l, and*

2. *$v \models g$*

In this case, the execution must immediately move to the location l'.

Under M-Invoke semantics, the following behavior needs to be enforced (examples are taken from Figure 4.6).

1. When a ε-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 ε-labeled switch is entered after the equality clause of its guard has been satisfied, the other switches must not be disabled. If l_2 is entered while $x_{T_2} > 5$, the other switches must not be disabled.

3. When the guard of a ε-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 functions. First, we define a function “inhib” over a ε-labeled transition guard of the form $g = (x = k) \wedge g'$. The role of this function 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 function plays a critical role in enforcing the M-Invoke constraints in protocol timed automata. Then, we will provide a function “permits”, also defined over a ε-labeled transition guard. It defines when it allows other switches from the same source location to become actionable. This function 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 5 (inhib function)**.** *Let a guard $g := (x = k) \wedge g'$ of a ε-labeled switch defined over a ε-labeled switch $l \rightarrow l'$ such as x is a clock over $\mathbb{T}$, k is a constant in $\mathbb{Q} \cup \{\bot\}$ 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 $i \in \{1, \cdots, m\}$: $x_i, x'_i \in X \cup Y$, $k_i \in \mathbb{Q} \cup \{\bot\}$ and $\#_i$ is any comparison operator).*

We define the function 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 4.6:

$$\begin{cases} \text{inhib}(g_1) = \texttt{false} \\ \text{inhib}(g_2) = (x_{T_2} - x_{T_1} \leq -8) \end{cases}$$

Without loss of generality, we chose to reduce the ε-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 ε-labeled switches with each switch having a single conjunctive guard. We keep this assumption in the remainder.

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

Definition 6 (permits function)**.** *Let a guard $g := (x = k) \wedge g'$ of a ε-labeled switch defined over a ε-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:*

1. $S_1 = (x < k)$
2. $S_2 = (x > k) \wedge (x - y > k)$
3. $S_3 = (x > k) \wedge (x - y \leq k) \wedge \text{inhib}(g)$
4. $S_4 = (x = k) \wedge \text{inhib}(g)$

The function permits(g) *is defined as*

$$\text{permits}(g) = \bigvee_{i \in \{1,2,3,4\}} S_i$$

The permits function disjunctive clauses 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 4.6:

$$\begin{cases} \text{permits}(g_1) & = & \underbrace{(x_{T_2} < 5)}_{S_1} \\ & \bigvee & \underbrace{(x_{T_2} > 5) \wedge (x_{T_2} - y_{l_2} > 5)}_{S_2} \\ \text{permits}(g_2) & = & \underbrace{(x_{T_1} < 10)}_{S_1} \\ & \bigvee & \underbrace{(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} > 10)}_{S_2} \\ & \bigvee & \underbrace{(x_{T_1} > 10) \wedge (x_{T_1} - y_{l_2} \leq 10) \wedge \underbrace{(x_{T_2} - x_{T_1} \leq -8)}_{\text{inhib}(g_2)}}_{S_3} \\ & \bigvee & \underbrace{(x_{T_1} = 10) \wedge \underbrace{(x_{T_2} - x_{T_1} \leq -8)}_{\text{inhib}(g_2)}}_{S_4} \end{cases}$$

We can now define how the guards of the switches whose source locations offer ε-labeled switches need to be rewritten so as to enforce M-Invoke .

Procedure 1 (M-Invoke enforcement)**.** *Let l be a location of a protocol timed automaton A that offers $n > 0$ ε-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 ε-labeled ones) is performed as follows:

1. *for each location l that offers a ε-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_{ε_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 4.6 that has been fixed to enforce M-Invoke semantics on Figure 4.7.

4.3.4 Theoretical results

We now study the expressiveness of protocol timed automata, then prove that the mapping from timed protocols to timed automata is correct.

Expressiveness

Theorem 1. *ε-transitions strictly increase the expressiveness of protocol timed automata.*

Protocol timed automata are strictly more expressive than (general) timed automata because of their ε-labeled switches. The proof, given on page 167, is based on the notion of *precise actions* and is the same as Corollary 29 of (Beatrice Berard et al. [13]). The idea is that the protocol timed automaton of Figure 4.8 cannot be expressed equivalently by a timed automaton without ε-labeled switches.

Correctness of the mapping

We now give some results showing that the guards rewritings above are correct with respect to the M-Invoke constraint semantics. Indeed, the correctness of the mapping from timed protocols to protocol timed automata

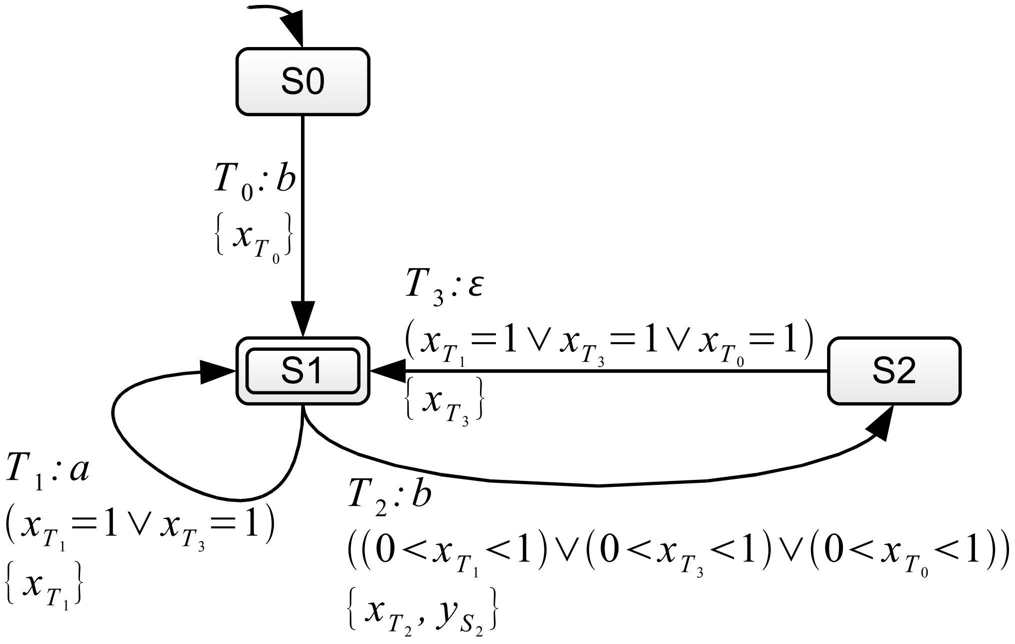

S0
$T_0 : b$
$\{x_{T_0}\}$
$T_3 : \varepsilon$
$(x_{T_1} = 1 \vee x_{T_3} = 1 \vee x_{T_0} = 1)$
$\{x_{T_3}\}$
S1
S2
$T_1 : a$
$(x_{T_1} = 1 \vee x_{T_3} = 1)$
$\{x_{T_1}\}$
$T_2 : b$
$((0 < x_{T_1} < 1) \vee (0 < x_{T_3} < 1) \vee (0 < x_{T_0} < 1))$
$\{x_{T_2}, y_{S_2}\}$

Figure 4.8: A protocol timed automaton A that cannot be expressed equivalently without ε-transitions.

relies on the rewriting to correctly enforce M-Invoke constraints semantics. The following lemma shows that the function inhib works as expected, i.e., it can inhibit guards when a ε-labeled switch guard is totally satisfied, and allow them when it is not. The proof is given on page 168.

Lemma 1. *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:*

1. $(\mathrm{inhib}(g) = \mathtt{true}) \Longrightarrow (g' = \mathtt{false})$
2. $(\mathrm{inhib}(g) = \mathtt{false}) \Longrightarrow (g' = \mathtt{true})$

Given a location that has several ε-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. A proof is given on page 169.

Lemma 2. *Let a protocol timed automaton A and a location $l \in A$ such that it offers $n > 0$ ε-labeled switches. For any $i \in \{1, \cdots, n\}$, the guard $\widetilde{g_i}$ of the i-th ε-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 theorem states that the mapping of a timed protocol to a protocol timed automaton is correct with respect to M-Invoke constraints. The proof is given on page 170.

Theorem 2. *Let a protocol timed automaton A obtained from a timed protocol $\mathcal{P}$ on which Procedure 1 has been applied. Every ε-labeled switch $e = (l, g = (x = k) \wedge g', \varepsilon, r, l')$ of A is taken as soon as its guard g is satisfied.*

Finally, the following lemma states that protocol timed automata are deterministic. The proof is given on page 85.

Lemma 3. *Protocol timed automata are deterministic: given a protocol timed automaton A and a timed word $w \in \mathcal{L}(A)$, w has exactly one run over A.*

4.4 Discussion

The following discussion aims at first positioning protocol timed automata with respect to event-clock automata as defined in (Rajeev Alur et al. [119]). We then discuss two potential limitations in our model. The first is the absence of support for absolute time constants (e.g., supporting constraints of the form ($x <$ `2008-05-10:20:00:00`)). The second one is the assumption that we do not take into account network transport delays and losses.

4.4.1 Relationship to event-clock automata

The class of protocol timed automata can be seen as an extension of event-recording automata (Rajeev Alur et al. [119]). In fact, their design has been made with them in mind. Yet, any protocol timed automaton is not necessarily a event-recording automaton. One straightforward similarity between the two classes is the time domain $\mathbb{T} = \mathbb{R}_{\geq 0} \cup \{\perp\}$ and that clocks are assigned and reset in a restricted fashion. There are however several differences:

- protocol timed automata support ε-labeled switches with clock resets, and
- some particular switches may reset two clocks (i.e., the ones whose target location offers a ε-labeled switch), and
- one subset of the clocks assigns them to switches (i.e., X), not input symbols, while the complement subset assigns them to locations (Y).

However as we will see in the next chapters, protocol timed automata are fully deterministic, and they have the property that the value of clocks only depends of the input words, despite ε-labeled switches (at first sight, one would think that they necessarily introduce indeterminism).

The class of event-recording automata is however a subset of protocol timed automata. Indeed, let us consider an event-recording automaton: it can be translated directly to a protocol timed automaton A as:

1. there is no ε-labeled switch, and
2. for every input symbol a and any guard clause of the form $(x_a \# k)$ ($\#$ is any operator and k a constant), it can be rewritten as $(x_{e1} \# k) \vee \cdots \vee (x_{en} \# k)$ where $\{x_{e1}, \cdots, x_{en}\}$ is the set of switches in A where a is the input symbol.

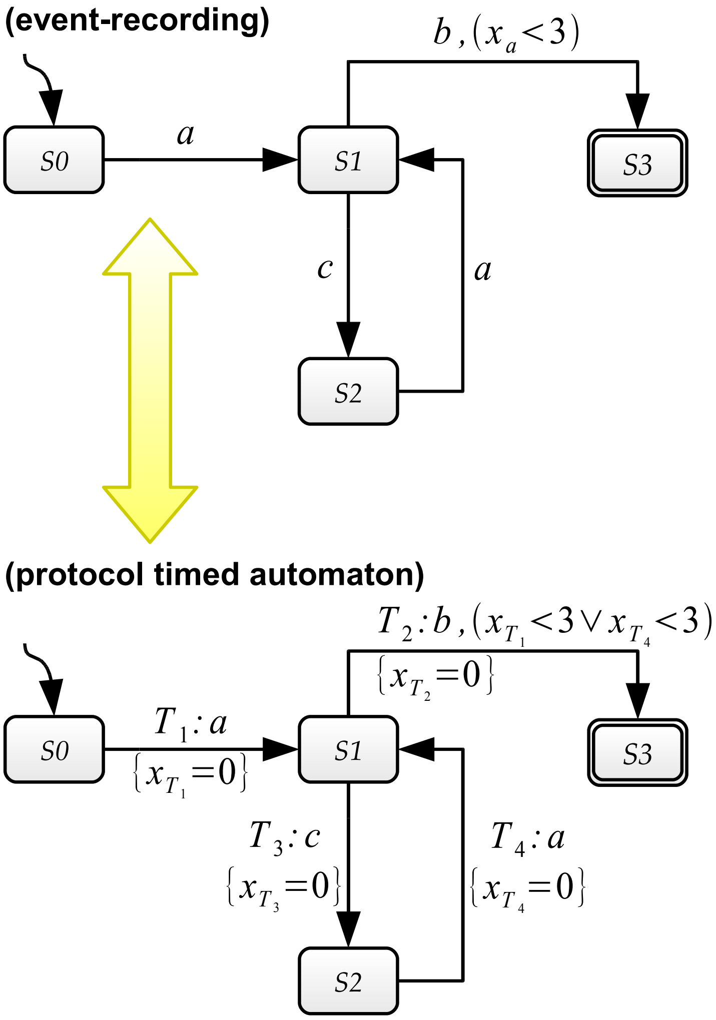


Figure 4.9: A sample event-recording automaton viewed as a protocol timed automaton.

We give an example on Figure 4.9 of a event-recording timed automaton encoded as a protocol timed automaton.

4.4.2 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 (OASIS [102]) 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.

1. x is set to a constant *now* which represents the current date when the automaton execution starts, and
2. x is always compared to absolute dates, and
3. 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). 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).

4.4.3 Message transport communications

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. The class of *Robust Timed Automata* (Rajeev Alur and P. Madhusudan [118]) is a possible exploration path for solving those issues. Briefly, a robust timed automaton recognizes timed words with some fuzziness in the event dates as no real world system can be expected to be as precise as timed automata expectations. The classical decidability problems (reachability / emptiness, inclusion) remain unchanged for this class. However, the expressiveness of such automata cannot be compared to the one of timed automata (Rajeev Alur and P. Madhusudan [118]). That is why we guess that using them as a formal framework for timed protocols probably requires substantial investigations from a theoretical point of view.

Five

Protocol analysis

Timed protocols can be used to model the external behaviors of web services interfaces by capturing the messages choreographies as well as the timing constraints that are put on them. We focus now on the analysis of timed protocols given the following two dimensions: compatibility and replaceability. To do that, we introduce for each type a set of analysis flexible classes. By flexible, we mean that those classes cater for more than "black or white" compatibility or replaceability cases like it has been traditionally done for hardware and software components. For example one may look for compatibility or replaceability only for a limited subset of a provider or requester protocol specification.

This chapter is structured as follows. We first present compatibility then replaceability analysis before turning our attention to a set of timed protocol operators that can be combined to characterize those classes. The approach that is presented here extends the one from (Benatallah et al. [21]) with timing constraints being added to the business protocol model.

5.1 Classes of protocol-based analysis

We start this chapter by presenting compatibility then replaceability analysis.

5.1.1 Protocol-level compatibility

Compatibility analysis is concerned with verifying whether two services can interoperate. 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.

1. Partial compatibility between two protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ implies that at least one conversation can be carried out between two services implementing these protocols.

2. A protocol $\mathcal{P}_1$ is *fully compatible* with $\mathcal{P}_2$ if $\mathcal{P}_2$ can support all message exchanges that $\mathcal{P}_1$ can generate (the inverse is not necessarily true).

Ideally, if we have developed a service S characterized by protocol $\mathcal{P}$, at binding time we will want to look for services that have a protocol with which $\mathcal{P}$ is fully compatible, so that every message exchange that our service can generate is understood by our partner.

We illustrate compatibility analysis with respect to timing constraints and its challenges on the examples below. Let us consider the protocols $\mathcal{P}$ and $\mathcal{P}'$ depicted on Figure 5.1. Without the time constraints, we can observe that $\mathcal{P}$ is fully compatible with $\mathcal{P}'$: $a \cdot b \cdot c$ and $a \cdot b \cdot d$ are valid interaction traces of the untimed versions of $\mathcal{P}$ and $\mathcal{P}'$. However, due to the C-Invoke constraints specified on transition T_3 of each protocol, $\mathcal{P}$ and $\mathcal{P}'$ cannot interact correctly. Indeed, $\mathcal{P}$ supports timed conversations of the form $(a(-), 0) \cdot (b(+), t) \cdot (c(+), t')$ (with $t' < t+5$), while $\mathcal{P}'$ supports timed conversations of the form $(a(+), 0) \cdot (b(-), t) \cdot (c(-), t')$ (with $t' > t + 10$). Hence, these two protocols cannot interact correctly since $\mathcal{P}'$ will always send a message c later than $\mathcal{P}$ allows it to be received. Therefore, two protocols must agree on the ordering of the exchanged messages as well as on

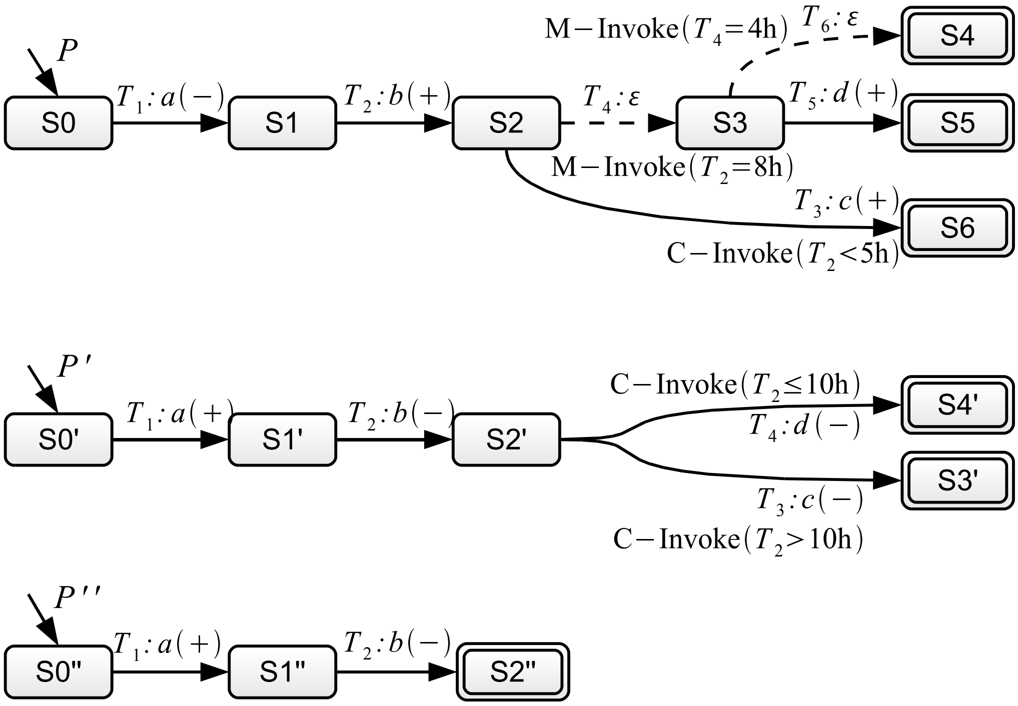


Figure 5.1: Three protocols for illustrating protocol analysis.

the time constraints to be compatible.

Let us now consider the protocols $\mathcal{P}$ and $\mathcal{P}''$ on Figure 5.1. When interacting according to the timed interaction trace $(a, 0) \cdot (b, t)$, $\mathcal{P}$ moves to a non-final state s_2 while $\mathcal{P}''$ moves to a final state s_2'', ending the conversation from its side. However, due the presence of the implicit transitions T_4 and T_5, $\mathcal{P}$ is able to terminate correctly its execution by moving automatically to the final state s_4: it waits at state s_2 for 8 hours, then moves to s_3 where it waits for 4 hours before finally moving to s_4 which is a final state. Therefore, $\mathcal{P}$ and $\mathcal{P}''$ can interact correctly following the interaction trace $(a, 0) \cdot (b, t)$.

The last example shows that the implicit transitions have an impact on the identification of the final states, naturally impacting the compatibility analysis. We consider again $\mathcal{P}$ and $\mathcal{P}'$ from Figure 5.1. After exchanging the messages a and b, the two protocols move to the states s_2 and s'_2 respectively. If we consider the explicit operations that are available from these two states, s'_2 provides an operation $d(-)$ while s_2 does not enable any invocation of a d message receiving operation. As a consequence, focusing the compatibility checking only on these two states is not enough. Indeed, the presence of the implicit transition T_4 in $\mathcal{P}$ moves automatically a service conversation to the state s_3 after 8 hours from when d can be received. Hence, $\mathcal{P}$ and $\mathcal{P}'$ can interact correctly by following timed interactions traces of the form $(a, 0) \cdot (b, t) \cdot (c, t')$, with $t + 8 < t' \leq t + 10$ (i.e., if the message d is sent between 8 and 10 hours after the message b).

5.1.2 Protocol-level replaceability

Replaceability analysis identifies whether a service can acts 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, for example, 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.

1. *Protocol equivalence* occurs when two protocols support exactly the same conversations.

2. *Protocol subsumption* occurs when a protocol supports at least all of the conversations of another one.

3. *Protocol replaceability w.r.t. a client protocol* occurs when a protocol $\mathcal{P}_1$ can replace a protocol $\mathcal{P}_2$ when interacting with a client protocol $\mathcal{P}_c$ if every valid conversation between $\mathcal{P}_2$ and $\mathcal{P}_c$ is also a valid conversation between $\mathcal{P}_1$ and $\mathcal{P}_c$. 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.

4. *Protocol replaceability w.r.t. an interaction role* is similar to the previous one. It occurs when a protocol $\mathcal{P}_1$ can replace a protocol $\mathcal{P}_2$ if $\mathcal{P}_1$ behaves like $\mathcal{P}_2$ when $\mathcal{P}_2$ behaves as an interaction role $\mathcal{P}_r$.

For all of the above classes, we can distinguish between full and partial replaceability. Full replaceability is as defined above. Partial replaceability is when there is replaceability but only for some conversations. As an example, we have a partial replaceability with respect to a client protocol when a protocol $\mathcal{P}_1$ can replace another protocol $\mathcal{P}_2$ in at least one conversation that can occur with $\mathcal{P}_c$.

To illustrate replaceability, let us consider a protocol $\mathcal{P}_1$ obtained from the protocol $\mathcal{P}''$ on Figure 5.1 by reversing the polarities of the messages. Such a protocol can be replaced by $\mathcal{P}$ on Figure 5.1. Indeed, the only timed conversation supported by $\mathcal{P}_1$ are of the form $(a(-), 0) \cdot (b(-), t)$ (with $t > 0$) and such conversations are also supported by $\mathcal{P}$. The opposite is however not true because for example $\mathcal{P}$ may support some conversations that contain the messages c or d, while $\mathcal{P}_1$ does not. Interestingly, we can observe that $\mathcal{P}_1$ can replace $\mathcal{P}$ when interacting with $\mathcal{P}''$: the only timed conversations of $\mathcal{P}$ that are understood by $\mathcal{P}''$ are of the form $(a(-), 0) \cdot (b(-), t)$ (with $t > 0$) and which are also supported by $\mathcal{P}_1$.

The next section presents a set of timed protocol operators that can be used to completely characterize the above compatibility and replaceabilty classes.

5.2 Protocol operators

We split the set of protocol operators in two categories: *manipulation* and *comparison* operators. The former category allows to compute protocols capturing properties regarding a pair of protocols, for example to compute a protocol that captures all of the common timed conversations of two protocols. The later category allows to compare two protocols for subsumption ($\sqsubseteq$) and equivalence ($\equiv$). We informally describe the protocol manipulation operators below, while their formal semantics are presented in Table 5.1.

Definition 7 (Timed protocol operators)**.**

Parallel composition *($\|^{\texttt{TC}}$) 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 *is used to project the polarity of one protocol on the parallel composition of two protocols, and is denoted as $[\mathcal{P}_1 \|^{\texttt{TC}} \mathcal{P}_2]_{\mathcal{P}_1}$.*

Intersection *($\|^{\texttt{TI}}$) takes two input timed protocols and returns one timed protocol that captures the timed conversations that they have in common.*

Difference *($\|^{\texttt{TD}}$) 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.*

Subsumption *($\sqsubseteq$) tests whether one protocol supports all of the timed conversations of another protocol (i.e., $\mathcal{P} \sqsubseteq \mathcal{P}'$ if and only if $Tr(\mathcal{P}) \subseteq Tr(\mathcal{P}')$).*

Equivalence *($\equiv$) checks whether two timed protocols support exactly the same set of timed conversations (i.e., $\mathcal{P} \equiv \mathcal{P}'$ if and only if $Tr(\mathcal{P}) = Tr(\mathcal{P}')$.*

To illustrate these operators, Figure 5.2 shows three simple timed protocols $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ as well as some results when applying operators on them. For example, the protocol $\mathcal{P}_1 \|^{\texttt{TI}} \mathcal{P}_3$ captures the timed conversations that are commonly supported by both $\mathcal{P}_1$ and $\mathcal{P}_3$: $\mathcal{P}_1$ does not support receiving a message c, hence it does not appear in $\mathcal{P}_1 \|^{\texttt{TI}} \mathcal{P}_3$. Similarly $\mathcal{P}_1$ can only

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

Table 5.1: Protocol manipulation operators semantics.

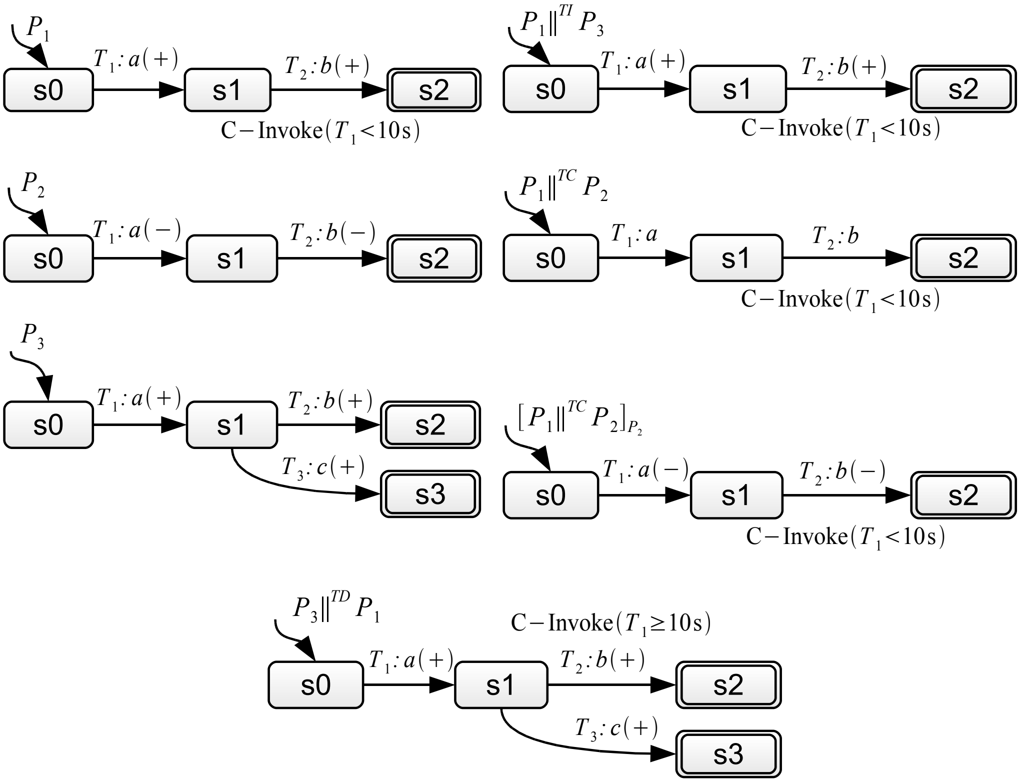


Figure 5.2: Three timed protocols $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{P}_3$ and some resulting protocols when using protocol manipulation operators.

receive a b message within the 10 seconds that follow the reception of a a message. Another example is the protocol $\mathcal{P}_3 \parallel^{\mathtt{TD}} \mathcal{P}_1$ that captures all of the conversations that $\mathcal{P}_3$ supports but that $\mathcal{P}_1$ doesn't support. This is why the C-Invoke constraint of T_2 in $\mathcal{P}_3 \parallel^{\mathtt{TD}} \mathcal{P}_1$ is the negation of the one of T_2 in $\mathcal{P}_1$ as $\mathcal{P}_3$ does not carry a C-Invoke constraint on its transition T_2. Similarly, $\mathcal{P}_3$ supports receiving c messages while $\mathcal{P}_1$ does not. Finally, the protocol $\mathcal{P}_3$ on Figure 5.2 subsumes the protocol $\mathcal{P}_1$.

5.3 Characterizing the compatibility and replaceability classes

Based on the operators introduced above, the following lemma gives the necessary and sufficient conditions to identify the compatibility level between two timed protocols.

Lemma 4. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed business protocols.*

1. *$\mathcal{P}_1$ and $\mathcal{P}_2$ are* partially compatible *iff $\mathcal{P}_1 \parallel^{\mathtt{TC}} \mathcal{P}_2$ is not an empty protocol (i.e., $Tr(\mathcal{P}_1 \parallel^{\mathtt{TC}} \mathcal{P}_2) \neq \emptyset$)).*
2. *$\mathcal{P}_1$ and $\mathcal{P}_2$ are* fully compatible *iff $[\mathcal{P}_1 \parallel^{\mathtt{TC}} \mathcal{P}_2]_{\mathcal{P}_1} \equiv \mathcal{P}_1$.*

Indeed, if a timed interaction protocol resulting from a compatible composition of two input protocols is not empty, it means that there is at least one timed interaction trace between the input protocols, and hence these protocols are compatible. Otherwise, the input protocols are not compatible. Regarding the second item of the lemma, the projection $[\mathcal{P}_1 \parallel^{\mathtt{TC}} \mathcal{P}_2]_{\mathcal{P}_1}$ describes all the timed conversations of $\mathcal{P}_1$ that are also understood by $\mathcal{P}_2$. As a consequence, if such a set of conversations contains all the ones of $\mathcal{P}_1$, it implies that $\mathcal{P}_1$ is fully compatible with $\mathcal{P}_2$. Otherwise, $\mathcal{P}_1$ is not fully compatible with $\mathcal{P}_2$ since there is at least one timed conversation of $\mathcal{P}_1$ which cannot be understood by $\mathcal{P}_2$.

The following lemma characterizes the replaceability levels of two given protocols using the operators introduced in the previous section.

Lemma 5. *Let $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_C$ and $\mathcal{P}_R$ be three timed business protocols.*

1. $\mathcal{P}_1$ *can* replace $\mathcal{P}_2$ *iff* $\mathcal{P}_2 \sqsubseteq \mathcal{P}_1$.

2. $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *are* equivalent *w.r.t. replaceability iff* $\mathcal{P}_1 \equiv \mathcal{P}_2$.

3. $\mathcal{P}_1$ *can replace* $\mathcal{P}_2$ *w.r.t. a* client protocol $\mathcal{P}_C$ *iff* $[\mathcal{P}_C \|^{\texttt{TC}} \mathcal{P}_2]_{\mathcal{P}_2} \sqsubseteq \mathcal{P}_1$ *(or equivalently iff* $\mathcal{P}_C \|^{\texttt{TC}} (\mathcal{P}_2 \|^{\texttt{TD}} \mathcal{P}_1)$ *is an empty protocol).*

4. $\mathcal{P}_1$ *can replace* $\mathcal{P}_2$ *w.r.t. a* role $\mathcal{P}_R$ *iff* $(\mathcal{P}_R \|^{\texttt{TI}} \mathcal{P}_2) \sqsubseteq \mathcal{P}_1$.

The characterization of the subsumption and equivalence w.r.t. replaceability is immediate. Regarding the third class of replaceability, the lemma defines that if all of the timed conversations of a protocol $\mathcal{P}_1$ that are also understood by a protocol $\mathcal{P}_C$ (given by the projection of $\mathcal{P}_1$ on compatible composition of $\mathcal{P}_1$ and $\mathcal{P}_C$) are compliant with a protocol $\mathcal{P}_2$, then $\mathcal{P}_2$ can replace $\mathcal{P}_1$ when interacting with $\mathcal{P}_C$. Finally, the last item of the lemma defines that if the common timed conversations of $\mathcal{P}_2$ and $\mathcal{P}_R$ (given by the intersection of $\mathcal{P}_2$ and $\mathcal{P}_R$) are compliant with $\mathcal{P}_1$ then $\mathcal{P}_1$ can replace $\mathcal{P}_2$ when $\mathcal{P}_2$ behave as $\mathcal{P}_R$.

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

Table 5.2: Characterization of the compatibility and replaceability classes.

We have summarized the operators-based characterization of the compatibility and replaceability classes in Table 5.2.

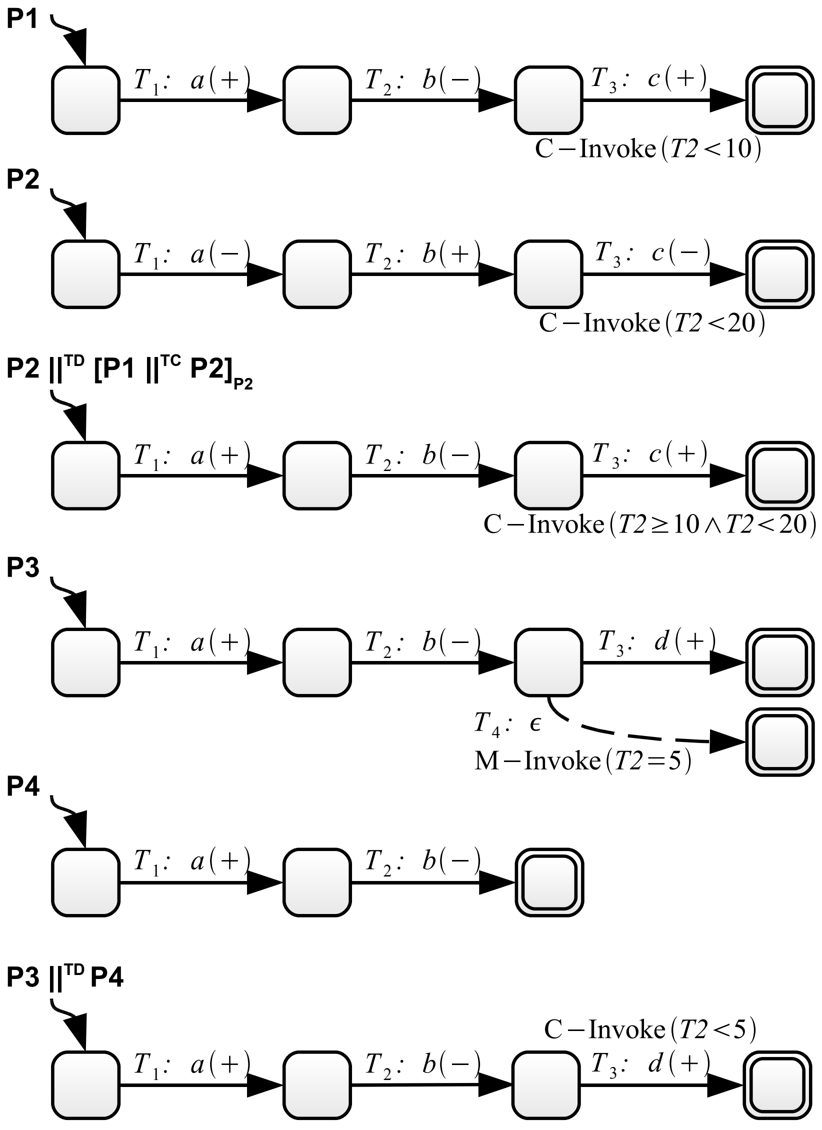


Figure 5.3: Compatibility and replaceability analysis.

We give examples of operators-based compatibility and replaceability analysis on Figure 5.3. $\mathcal{P}_1$ and $\mathcal{P}_2$ are only partially compatible, as

$$\left[\mathcal{P}_1 \parallel^{\texttt{TC}} \mathcal{P}_2\right]_{\mathcal{P}_2} \not\equiv \mathcal{P}_2$$

By using the difference operator to compute $\mathcal{P}_2 \parallel^{\texttt{TD}} [\mathcal{P}_1 \parallel^{\texttt{TC}} \mathcal{P}_2]_{\mathcal{P}_2}$, we get the set of conversations that are supported by $\mathcal{P}_2$ but not by $\mathcal{P}_1$ which yield to a partial compatibility ($\mathcal{P}_1$ does not support receiving a c message after 10 units of time). $\mathcal{P}_4$ can be replaced by $\mathcal{P}_3$ as it supports all of the conversations that $\mathcal{P}_4$ supports: $\mathcal{P}_3 \sqsubseteq \mathcal{P}_4$. The converse is however not true as illustrated by $\mathcal{P}_3 \parallel^{\texttt{TD}} \mathcal{P}_4$: $\mathcal{P}_3$ cannot handle d messages.

5.4 Discussion

We briefly review the related work and discuss the limitations regarding the message-level matching that is performed in our approach.

5.4.1 Related work

Verification techniques

Many works in various fields have applied 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 (Aalst [2]; Bettini et al. [27]; Maria et al. [87]). In the case of web services, timed automata have been used in (Dyaz et al. [52]; Kazhamiakin et al. [78]). In (Berardi et al. [25]) the introduced WSTL model had been designed with timing constraints as “first-class citizen”. The analysis techniques presented in this paper did not leverage the timing constraints though, and while this had been mentioned as future work, it hasn’t yet been done.

BPEL-based web services interactions have been analyzed in (Fu et al. [62]) by the mean of guarded automata with unbounded message queues where the automata synchronizablility problem is studied in synchronous and asynchronous communications. Formal verification of service compositions is the target of several works: (Bultan et al. [40]; Foster et al. [56]; Gu et al. [66]; Rouached et al. [122]). The same types of verifications can be performed on protocol timed automata using TCTL, an extension of temporal logics for timed automata, and a model checker such as UPPAAL (Behrmann et al. [14]).

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 replace-ability exist in the area of component-based systems (Canal et al. [43]; Yellin and Storm [137]).

The importance of being able to check for services compatibility or replace-ability has lead to several research works (Beyer et al. [28]; Foster et al. [57]; Mecella et al. [90]). 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 replace-ability with respect to a client protocol. We believe that this flexibility will significantly prove to be useful in practice, as full compatibility or replace-ability 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 (Bussler [42]; van der Aalst [133]). It is similar to protocols replaceability. Different types of inheritance relations are proposed in (van der Aalst [133]). They provide some flexibility much like we did with the different classes of protocol replaceability. However, these approaches do not consider temporal abstractions.

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 (Bernstein et al. [26]). 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 (Nejati et al. [95]). This work has been done in a similar fashion as the approaches for schema matching (including databases and XML) mentioned in (Bernstein et al. [26]; Rahm and Bernstein [116]).

The work presented in this thesis shares some analogies with what has been done in this research field. Indeed, timed protocol is a model for which there exist protocol manipulation operators (composition, difference and intersection) as well as comparison operators (subsumption and equivalence).

5.4.2 Message matching

Our approach is mainly “syntactic” in the sense that when considering two messages named `login`, we assume that both their schema and semantics are identical. This is of course not always the case in reality. As far as the schema is concerned, a better way for comparing two messages in the protocol operators would be to also look at their definitions (e.g., in XML Schema, grabbed from the services WSDL specifications). To do that, we could reuse schema matching algorithms as studied in (Bernstein et al. [26]; Rahm and Bernstein [116]) or the approach presented in (Dong et al. [51]) which is more specific to web services interfaces. In some cases an adapter could be generated at the static interface level (messages schema and operations).

Another limit of the way that the messages matching is performed is that 2 messages `login` and `logUserIn` are considered to be different, although they could have exactly the same schema and/or semantics (or both could be “close” enough for easy adaptation). A potential solution would be to integrate the *match* operator presented in (Nejati et al. [95]) with our compatibility and replaceability analysis techniques as it addresses such type of issues. The operator uses a heuristic that requires human intervention for identifying missing and invalid matches. If such an approach turned out to be useful, the identified message matches and mismatches could then be exploited to generate protocol adapters (Benatallah et al. [19]; Motahari et al. [94]).

Six

Properties of protocol operators

In the previous chapters, we have both introduced the model of *timed protocols* and presented our approach for analyzing the pairwise compatibility or replaceability of two timed protocol instances. The set of flexible protocol compatibility and replaceability analysis classes that we presented can each be characterized by combining timed protocol operators. Yet, the decidability and closure properties of those operators remain to be studied ($\|^{\mathtt{TI}}$, $\|^{\mathtt{TC}}$, $\|^{\mathtt{TD}}$, $\sqsubseteq$ and $\equiv$). To do that, we study those issues on protocol timed automata by reusing and adapting existing work from the theory of timed automata. As we will see, the case of $\|^{\mathtt{TI}}$ and $\|^{\mathtt{TC}}$ is easy while the remaining operators pose more challenges. Indeed, they all rely on the ability to complement protocol timed automata, which is difficult as they have ε-labeled switches that cannot be removed in the general case. One strong theoretical contribution of this work is given here through the closure under complementation of protocol timed automata. This is the first class of timed automata with ε-labeled switches where this is possible.

This chapter is divided in two sections. The first one works at the protocol timed automata level and studies the closure of this class under intersection, then complementation. The second section leverages the results on protocol timed automata and translates them to timed protocols, leading to the decidability and closure properties of the timed protocol operators

$\|^{\mathtt{TI}}$, $\|^{\mathtt{TC}}$, $\|^{\mathtt{TD}}$, $\sqsubseteq$ and $\equiv$.

6.1 Results in protocol timed automata

We first study the intersection of protocol timed automata, then complementation. For each operator, the structure of the sections is identical: we give a procedure, an example and a theorem for the closure under the considered operator.

6.1.1 Intersection of protocol timed automata

The protocol timed automata intersection procedure extends the classical construction on timed automata (Alur and Dill [6]), which in turns already extends the construction on (untimed) automata (Hopcroft et al. [74]). We start by giving the procedure followed by an example. Then, we introduce a theorem for the closure of protocol timed automata under intersection.

Procedure 2. *Given two protocol timed automata $A_1 = (L_1, L_1^0, L_1^f, X_1 \cup Y_1, E_1, \Sigma_1)$ and $A_2 = (L_2, L_2^0, L_2^f, X_2 \cup Y_2, E_2, \Sigma_2)$, the intersection $A_3 = A_1 \cap A_2$ (with $A_3 = (L_3, L_3^0, L_3^f, X_3 \cup Y_3, E_3, \Sigma_3)$) 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. *ε-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 ε-labeled witch whose guard is disjunctive is equivalent to several ε-labeled switches with conjunctive guards). For each pair of ε-labeled*

switches
$$\begin{cases} e_1 = (l_1, (x_1 = k_1) \wedge g_1, \varepsilon, r_1, l'_1) \in A_1 \\ e_2 = (l_2, (x_2 = k_2) \wedge g_2, \varepsilon, r_2, l'_2) \in A_2 \end{cases}$$

we add the following switches to E_3:

$$\begin{cases} e_1 e_\varepsilon = ((l_1, l_2), (x_1 = k_1) \wedge g_1 \wedge ((x_2 \neq k_2) \vee \neg g_2), \varepsilon, \{x_{e_1 e_\varepsilon}\}, (l'_1, l_2)) \\ e_\varepsilon e_2 = ((l_1, l_2), (x_2 = k_2) \wedge g_2 \wedge ((x_1 \neq k_1) \vee \neg g_1), \varepsilon, \{x_{e_\varepsilon e_2}\}, (l_1, l'_2)) \\ e_1 e_2 = ((l_1, l_2), (x_1 = k_1) \wedge (x_2 = k_2) \wedge g_1 \wedge g_2, \varepsilon, \{x_{e_1 e_2}\}, (l'_1, l'_2)) \end{cases}$$

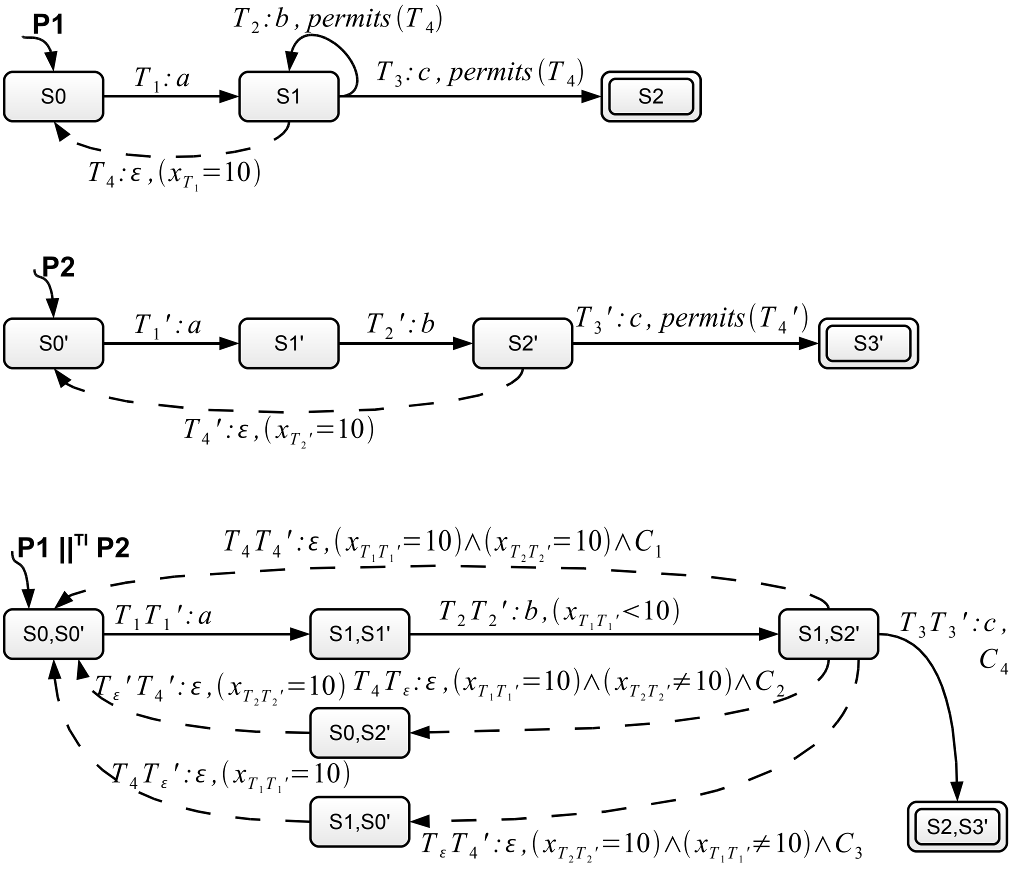
4. *With the set of clocks in A_3 being $X \cup Y$ as per definition, make sure that for each location l offering at least one ε-labeled switch, a clock $y_l \in Y$ is reset on all of the incoming switches to l.*

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 refere 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.*

Figure 6.1 gives an example of two protocol timed automata P_1 and P_2 as well as their intersection $P_1 \parallel^{\texttt{TI}} P_2$. We deliberately abused notation for the sake of clarity by referring to transition identifiers in the permits functions instead of referring to guards.

Please note that in this procedure we remove the existing permits clauses in the guards as new ones are computed. An extra step when actually implementing this procedure in a programming language would be to prune the locations that are not reachable from the initial location and the *"deadlock"* locations, i.e., the non-final locations l such that they don't provide any outgoing switch. The guards rewriting step is necessary because a


P1
$T_2: b$, $permits(T_4)$
S0
$T_1: a$
S1
$T_3: c$, $permits(T_4)$
S2
$T_4: \varepsilon, (x_{T_1} = 10)$
P2
S0'
$T_1': a$
S1'
$T_2': b$
S2'
$T_3': c$, $permits(T_4')$
S3'
$T_4': \varepsilon, (x_{T_2'} = 10)$
P1 ||TI P2
$T_4 T_4': \varepsilon, (x_{T_1 T_1'} = 10) \wedge (x_{T_2 T_2'} = 10) \wedge C_1$
S0,S0'
$T_1 T_1': a$
S1,S1'
$T_2 T_2': b, (x_{T_1 T_1'} < 10)$
S1,S2'
$T_3 T_3': c,$
C_4
$T_\varepsilon' T_4': \varepsilon, (x_{T_2 T_2'} = 10)$
$T_4 T_\varepsilon: \varepsilon, (x_{T_1 T_1'} = 10) \wedge (x_{T_2 T_2'} \neq 10) \wedge C_2$
S0,S2'
$T_4 T_\varepsilon': \varepsilon, (x_{T_1 T_1'} = 10)$
S1,S0'
$T_\varepsilon T_4': \varepsilon, (x_{T_2 T_2'} = 10) \wedge (x_{T_1 T_1'} \neq 10) \wedge C_3$
S2,S3'

$C_1 = permits(T_4 T_\varepsilon) \wedge permits(T_\varepsilon T_4)$
$C_2 = permits(T_4 T_4') \wedge permits(T_\varepsilon T_4)$
$C_3 = permits(T_4 T_4') \wedge permits(T_4 T_\varepsilon)$
$C_4 = permits(T_4 T_4') \wedge permits(T_\varepsilon T_4) \wedge permits(T_4 T_\varepsilon)$

Figure 6.1: Two protocol timed automata P_1 and P_2 as well as their intersection $P_1 \parallel^{\mathtt{TI}} P_2$.

switch of one of the input protocol timed automata may yield more than one switch in the resulting one. An example is the T_4 switch on P_1 from Figure 6.1 as it generates T_4T_4', T_4T_ε, $T_\varepsilon T_4'$, T_4T_ε' and $T_\varepsilon' T_4'$ in $P_1 \parallel^{\texttt{TI}} P_2$.

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. For example on Figure 6.1, T_2T_2' would reset the clock assigned to T_2 and the one assigned to T_2'.

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).

Protocol timed automata are closed under intersection (e.g., $A_3 = A_1 \cap A_2$ still belongs to the class).

Theorem 3. *The class of protocol timed automata is closed under intersection.*

The proof of the previous theorem is given on page 172.

6.1.2 Complementation of protocol timed automata

While protocol timed automata intersection is useful for characterizing timed protocol intersection and parallel composition, the complementation plays a critical role when it comes to characterizing the protocol difference and subsumption operators. Moreover, complementation of timed automata has traditionally been a difficult problem. Indeed, few classes of timed automata are closed under complementation (Rajeev Alur and P. Madhusudan

[118]).

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 (Alur and Dill [6]), with the difference lying in the presence of ε-transitions.

Procedure 3. *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 ε-labeled switch from l, given that l offers $n \geq 0$ of such switches.*

As in (Alur and Dill [6]), 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.

[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 function.

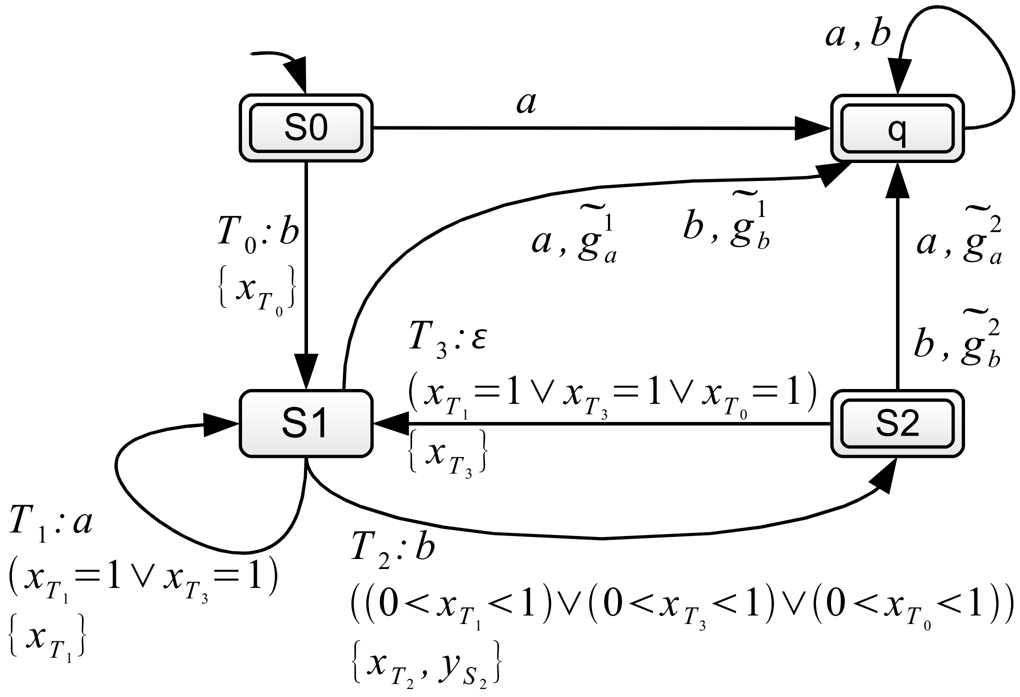


Figure 6.2: The complement of the protocol timed automaton of Figure 4.8 (with drawing shortcuts).

As an example, we give on Figure 6.2 the complement of the protocol timed automaton of Figure 4.8. For the sake of clarity, we took some shortcuts in the drawing by combining the switches to q that have been added from the same source location. The added guards are as follows:

- $\widetilde{g_a^1} = (x_{T_1} \neq 1) \wedge (x_{T_3} \neq 1)$
- $\widetilde{g_b^1} = (x_{T_1} = 0 \vee x_{T_1} \geq 1) \wedge (x_{T_3} = 0 \vee x_{T_3} \geq 1) \wedge (x_{T_0} = 0 \vee x_{T_0} \geq 1)$
- $\widetilde{g_a^2} = \widetilde{g_b^2} =$
  $((x_{T_1} < 1) \vee ((x_{T_1} > 1) \wedge (x_{T_1} - y_{S_2} > 1))) \bigwedge$
  $((x_{T_3} < 1) \vee ((x_{T_3} > 1) \wedge (x_{T_3} - y_{S_2} > 1))) \bigwedge$
  $((x_{T_0} < 1) \vee ((x_{T_0} > 1) \wedge (x_{T_0} - y_{S_2} > 1)))$

As mentioned in the next theorem (a proof is given on page 173), protocol timed automata are closed under complementation. This is a very interesting result, not only because it allows us to properly implement a timed protocol operator such as the difference, but also because the introduction of ε-labeled switches strictly increases expressiveness over (indeterministic) timed automata. Interestingly, previous results would not have suggested that protocol timed automata would be closed under complementation.

Theorem 4. *The class of protocol timed automata are closed under complementation.*

The proof is given on page 173.

6.2 Results for timed protocol operators

The closure properties of protocol timed automata under intersection and complementation allows to derive the following theoretical results for timed protocol operators. The result on the intersection and parallel composition operators is straightforward.

Corollary 1. *Timed protocols are closed under* $\|^{\mathtt{TI}}$, $\|^{\mathtt{TC}}$ *and* $\|^{\mathtt{TD}}$.

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 derives 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 2. *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 (Alur and Dill [6]). Checking if $L(A_1) \subseteq L(A_2)$ is equivalent to checking whether $L(A_1 \cap \overline{A_2}) = \emptyset$ or not. A technique for checking emptiness of protocol timed automata using the UPPAAL model checker is discussed in the appendix at page 183.

Part III

Applications and perspectives

Seven

The ServiceMosaic Protocols project

This chapter presents the implementation of the contributions contained in this thesis work. We first present *ServiceMosaic*, the umbrella project in which this work has been made. We then focus on the components that we have implemented for modeling and analyzing timed business protocols.

7.1 ServiceMosaic

This section presents the ServiceMosaic project. We first give an outlook of the general project architecture, then provide some technical details regarding how the various projects that compose ServiceMosaic are made.

7.1.1 Project overview

ServiceMosaic[1] is an international project for research in the context of web services. It currently acts as a bridge between several research groups:

- the SOC Group at the University of New South Wales, Sydney, Australia

- the APIS Research Group at Université Blaise Pascal, Clermont-Ferrand, France

[1] See http://servicemosaic.isima.fr/

- the BD-RCR Group at Université Claude Bernard, Lyon, France
- the group of Fabio Casati at the University of Trento, Italy.

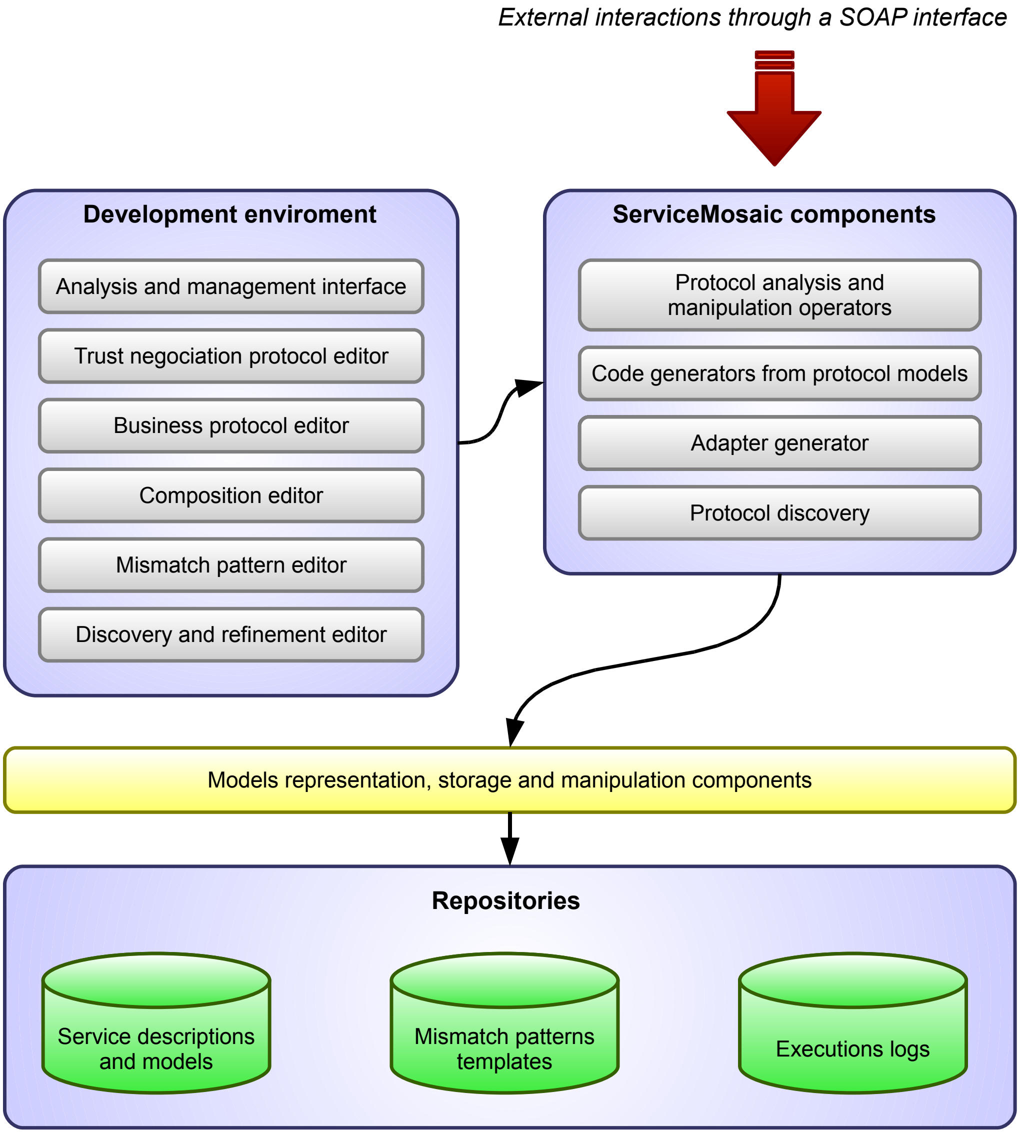

External interactions through a SOAP interface
Development enviroment
Analysis and management interface
Trust negociation protocol editor
Business protocol editor
Composition editor
Mismatch pattern editor
Discovery and refinement editor
ServiceMosaic components
Protocol analysis and manipulation operators
Code generators from protocol models
Adapter generator
Protocol discovery
Models representation, storage and manipulation components
Repositories
Service descriptions and models
Mismatch patterns templates
Executions logs

Figure 7.1: Architecture overview of the ServiceMosaic platform.

ServiceMosaic is a CASE platform for supporting the service development life-cycle that includes facilities for modeling, analyzing, discovering and adapting web service models (Benatallah et al. [22]; Nezhad et al. [99]).

The architecture, depicted on Figure 7.1, comprises the following components.

Models and manipulation components support representing, storing an manipulating service descriptions and protocols. We provide basic manipulation operations of model elements such as protocols as core libraries that shield higher-level components from the details of their physical representations (e.g., plain files, XML, relational databases and so on).

Analysis and management components include operators for protocol compatibility and replaceability analysis (Benatallah et al. [21]), a code generator that produces BPEL skeletons from protocols (Benatallah et al. [18]), a code generator that produces BPEL templates for implementing the adapters (Benatallah et al. [19]; Nezhad et al. [97]), and protocol discovery from service execution logs (Nezhad et al. [98]).

The development environment provides visual exploration for modifying, analyzing and managing model elements. For example, it offers editors for business protocols, trust negotiation protocols (Skogsrud et al. [127, 128]) and orchestration models.

Finally, the ServiceMosaic components can be accessed through a set of programmatic SOAP web service interfaces. Each service is a simple thin wrapper on top of the components application programming interfaces.

7.1.2 Technical overview

The tools that we develop use the Java^{TM} platform version 5[2]. Most developments are being made using the Java^{TM} programming language, but given the recent interest in dynamic languages that run on the Java^{TM} platform (e.g., Groovy, Ruby, Python, etc), we are allowing their use where useful. Indeed, the language facilities that are provided by some of those languages

[2] See http://java.sun.com/j2se/1.5.0/

(e.g., functional programming inspired constructs such as closures[3]) allow for writing code that is arguably more concise and easier to read and maintain.

For every project, the development approach is the following:

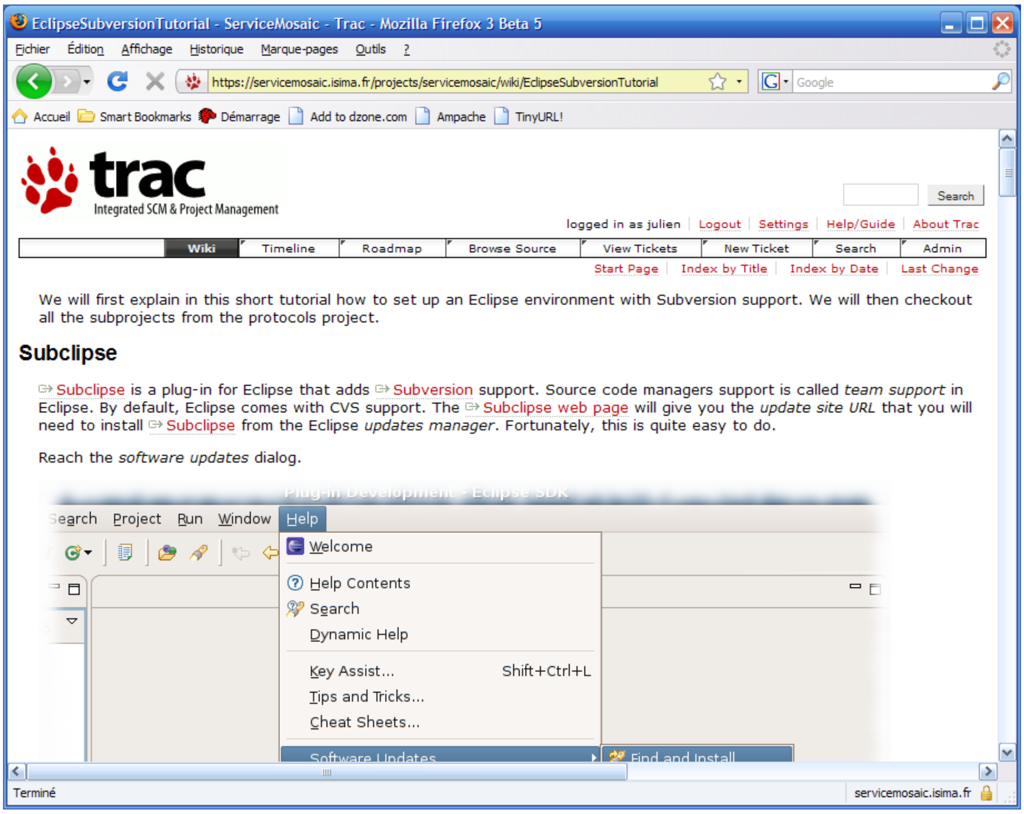
1. develop functionalities as standalone, reusable libraries that can be used in console, desktop, web or service-based tools, and
2. expose the functionalities in development tools as plug-ins for the Eclipse platform[4].

The choice of Eclipse as a development tools platform is justified by the following reasons. First of all, it is an open-source platform whose goal is to explicitly integrate tools from various vendors inside the same environment. As such, it provides several APIs and frameworks for building applications that can seamlessly integrate with third-party ones. The benefits for a research project are numerous.

1. The implementation work can be focused on the sole contribution of each work, not on less critical details such as providing common dialog boxes, a preferences support framework, or integrated user assistance.
2. The integration with new developments from inside the ServiceMosaic project or third-parties is vastly facilitated since a common foundation is being used. In most cases, the plug-ins that bring the contributions can simply be assembled inside an Eclipse-based environment without any modification having to be made.
3. The Eclipse ecosystem is broad with a mixture of open-source and commercial offerings (see http://www.eclipse.org/membership/exploreMembership.php). As such, it is a positive factor for facilitating the dissemination of the work that we conduct.

[3] Martin Fowler gives a good concise presentation of closures at http://martinfowler.com/bliki/Closure.html

[4] See http://www.eclipse.org/


EclipseSubversionTutorial - ServiceMosaic - Trac - Mozilla Firefox 3 Beta 5
Fichier Édition Affichage Historique Marque-pages Outils ?
https://servicemosaic.isima.fr/projects/servicemosaic/wiki/EclipseSubversionTutorial
Google
Accueil Smart Bookmarks Démarrage Add to dzone.com Ampache TinyURL!
trac
Integrated SCM & Project Management
Search
logged in as julien | Logout | Settings | Help/Guide | About Trac
Wiki Timeline Roadmap Browse Source View Tickets New Ticket Search Admin
Start Page | Index by Title | Index by Date | Last Change
Plug-in Development - Eclipse SDK
Search Project Run Window Help
Welcome
Help Contents
Search
Dynamic Help
Key Assist... Shift+Ctrl+L
Tips and Tricks...
Cheat Sheets...
Software Updates
Find and Install
Terminé
servicemosaic.isima.fr

We will first explain in this short tutorial how to set up an Eclipse environment with Subversion support. We will then checkout all the subprojects from the protocols project.

Subclipse

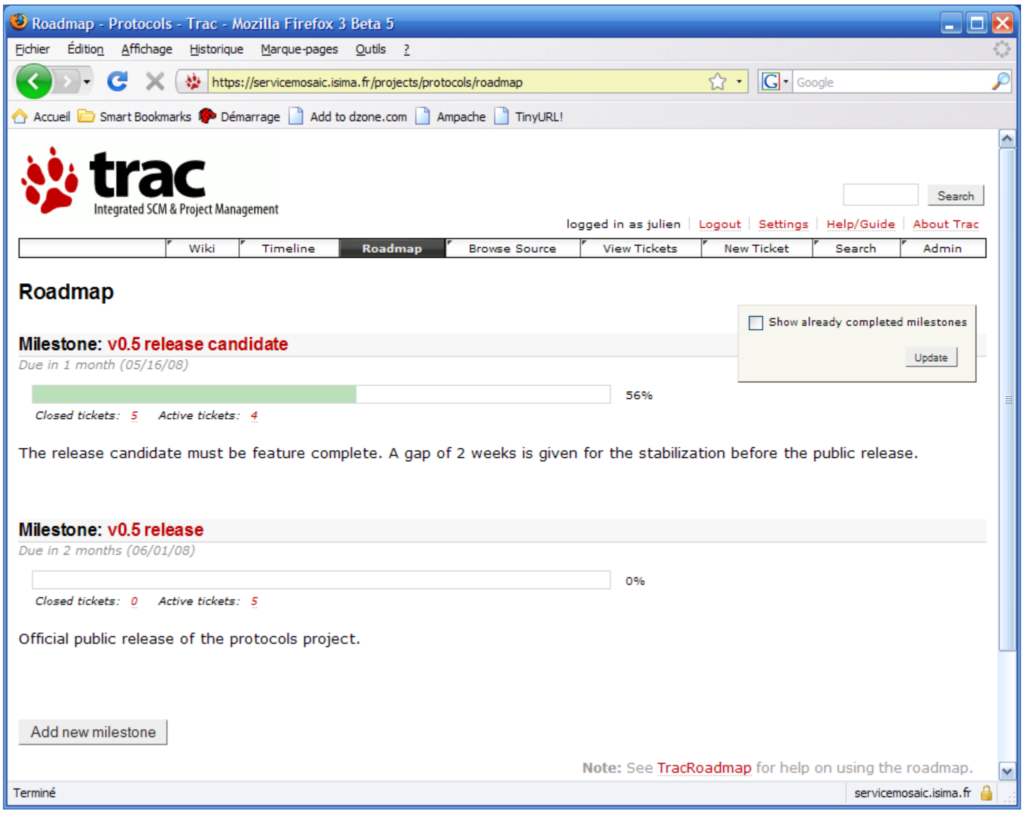
Subclipse is a plug-in for Eclipse that adds Subversion support. Source code managers support is called *team support* in Eclipse. By default, Eclipse comes with CVS support. The Subclipse web page will give you the *update site URL* that you will need to install Subclipse from the Eclipse *updates manager*. Fortunately, this is quite easy to do.

Reach the *software updates* dialog.

Figure 7.2: Trac wiki view.

The platform being developed across different institutions around the world, we have deployed a server at http://servicemosaic.isima.fr/ with collaborative services:

- the source code management and versioning is done using Subversion (see http://subversion.tigris.org/), allowing for distributed, concurrent modifications and sharing of the source code bases
- each project has an instance of Trac (see http://trac.edgewall.org/), a web-based application for managing projects that offers a wiki (see Figure 7.2), a source code browser (integrated with Subversion, see Figure 7.4), a roadmap (see Figure 7.3) and an issues manager (defects, tasks, enhancements, see Figure 7.5).
- mailing-lists and public / private download areas are available


Roadmap - Protocols - Trac - Mozilla Firefox 3 Beta 5
Fichier Édition Affichage Historique Marque-pages Outils ?
https://servicemosaic.isima.fr/projects/protocols/roadmap
Google
Accueil Smart Bookmarks Démarrage Add to dzone.com Ampache TinyURL!
trac
Integrated SCM & Project Management
Search
logged in as julien | Logout | Settings | Help/Guide | About Trac
Wiki Timeline Roadmap Browse Source View Tickets New Ticket Search Admin
Terminé
servicemosaic.isima.fr

Roadmap

□ Show already completed milestones

Update

Milestone: v0.5 release candidate

Due in 1 month (05/16/08)

56%

Closed tickets: 5 *Active tickets:* 4

The release candidate must be feature complete. A gap of 2 weeks is given for the stabilization before the public release.

Milestone: v0.5 release

Due in 2 months (06/01/08)

0%

Closed tickets: 0 *Active tickets:* 5

Official public release of the protocols project.

Add new milestone

Note: See TracRoadmap for help on using the roadmap.

Figure 7.3: Trac roadmap view.

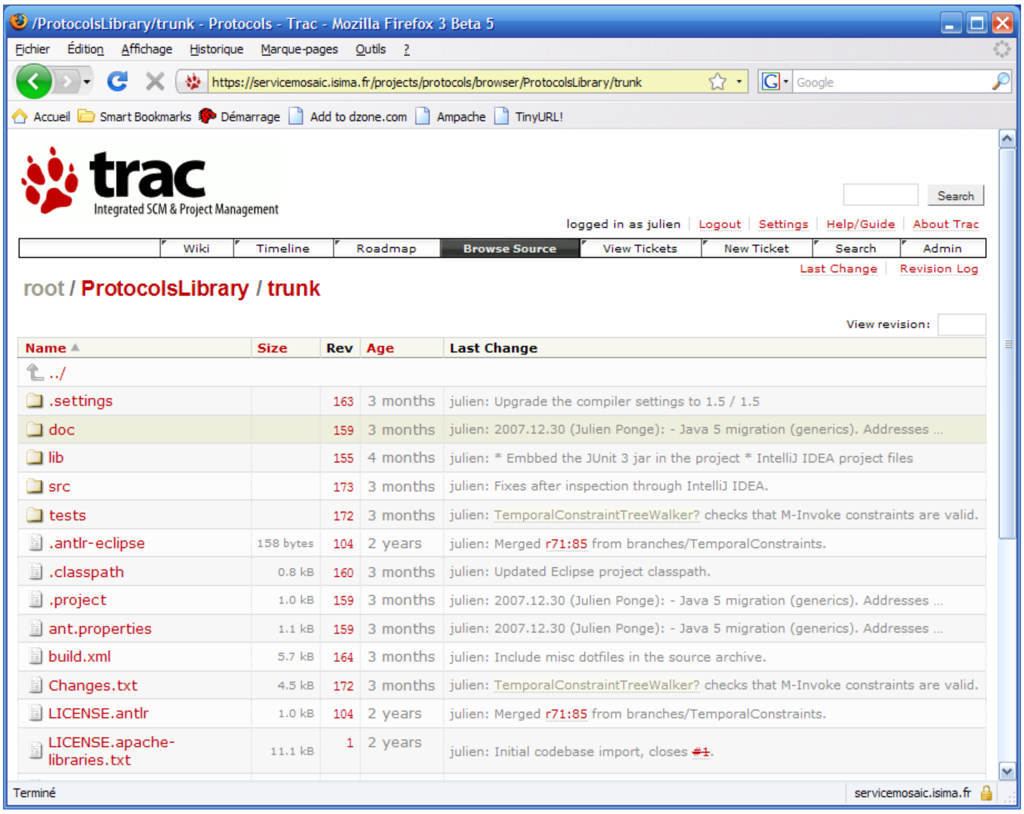
- a JavaEE 5 compliant application server[5] is available for the deployment of server-side applications, including web applications and web services.

A number of projects implemented as part of the ServiceMosaic platform are set to be progressively released to the public over time.

7.2 Prototype: the ServiceMosaic Protocols project

The work presented in this thesis has been implemented as part of a subproject of ServiceMosaic called *ServiceMosaic Protocols*. It groups the libraries

[5] In our case, we chose to run Glassfish from Sun Microsystems: http://glassfish.org/.



/ProtocolsLibrary/trunk - Protocols - Trac - Mozilla Firefox 3 Beta 5

https://servicemosaic.isima.fr/projects/protocols/browser/ProtocolsLibrary/trunk

root / ProtocolsLibrary / trunk

Name ▲	Size	Rev	Age	Last Change
../				
.settings		163	3 months	julien: Upgrade the compiler settings to 1.5 / 1.5
doc		159	3 months	julien: 2007.12.30 (Julien Ponge): - Java 5 migration (generics). Addresses ...
lib		155	4 months	julien: * Embbed the JUnit 3 jar in the project * IntelliJ IDEA project files
src		173	3 months	julien: Fixes after inspection through IntelliJ IDEA.
tests		172	3 months	julien: TemporalConstraintTreeWalker? checks that M-Invoke constraints are valid.
.antlr-eclipse	158 bytes	104	2 years	julien: Merged r71:85 from branches/TemporalConstraints.
.classpath	0.8 kB	160	3 months	julien: Updated Eclipse project classpath.
.project	1.0 kB	159	3 months	julien: 2007.12.30 (Julien Ponge): - Java 5 migration (generics). Addresses ...
ant.properties	1.1 kB	159	3 months	julien: 2007.12.30 (Julien Ponge): - Java 5 migration (generics). Addresses ...
build.xml	5.7 kB	164	3 months	julien: Include misc dotfiles in the source archive.
Changes.txt	4.5 kB	172	3 months	julien: TemporalConstraintTreeWalker? checks that M-Invoke constraints are valid.
LICENSE.antlr	1.0 kB	104	2 years	julien: Merged r71:85 from branches/TemporalConstraints.
LICENSE.apache-libraries.txt	11.1 kB	1	2 years	julien: Initial codebase import, closes ~~#1~~.

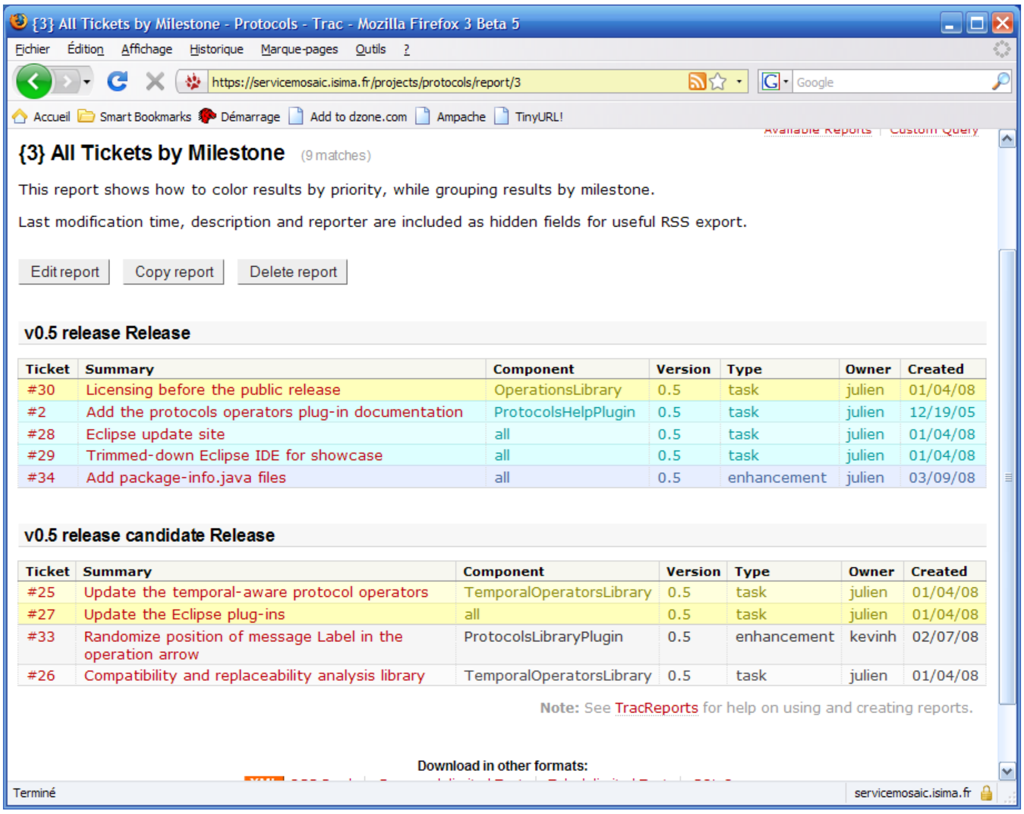
Figure 7.4: Trac source browser view.

and development tools that are related to business protocol modeling, analysis and management. Figure 7.6 shows a screenshot of the prototype development environment which is based on the Eclipse platform. It features two business protocol being edited, including timing constraints.

7.2.1 Components

The following components have been developed as part of the ServiceMosaic Protocols project, as depicted on Figure 7.7.

A business protocols model library has been developed. It contains an object-oriented model for representing and manipulating business protocols. While abstracting from the physical representation of protocols (e.g, files, databases, ...), it contains a simple XML persistence



{3} All Tickets by Milestone (9 matches)

This report shows how to color results by priority, while grouping results by milestone.

Last modification time, description and reporter are included as hidden fields for useful RSS export.

v0.5 release Release

Ticket	Summary	Component	Version	Type	Owner	Created
#30	Licensing before the public release	OperationsLibrary	0.5	task	julien	01/04/08
#2	Add the protocols operators plug-in documentation	ProtocolsHelpPlugin	0.5	task	julien	12/19/05
#28	Eclipse update site	all	0.5	task	julien	01/04/08
#29	Trimmed-down Eclipse IDE for showcase	all	0.5	task	julien	01/04/08
#34	Add package-info.java files	all	0.5	enhancement	julien	03/09/08

v0.5 release candidate Release

Ticket	Summary	Component	Version	Type	Owner	Created
#25	Update the temporal-aware protocol operators	TemporalOperatorsLibrary	0.5	task	julien	01/04/08
#27	Update the Eclipse plug-ins	all	0.5	task	julien	01/04/08
#33	Randomize position of message Label in the operation arrow	ProtocolsLibraryPlugin	0.5	enhancement	kevinh	02/07/08
#26	Compatibility and replaceability analysis library	TemporalOperatorsLibrary	0.5	task	julien	01/04/08

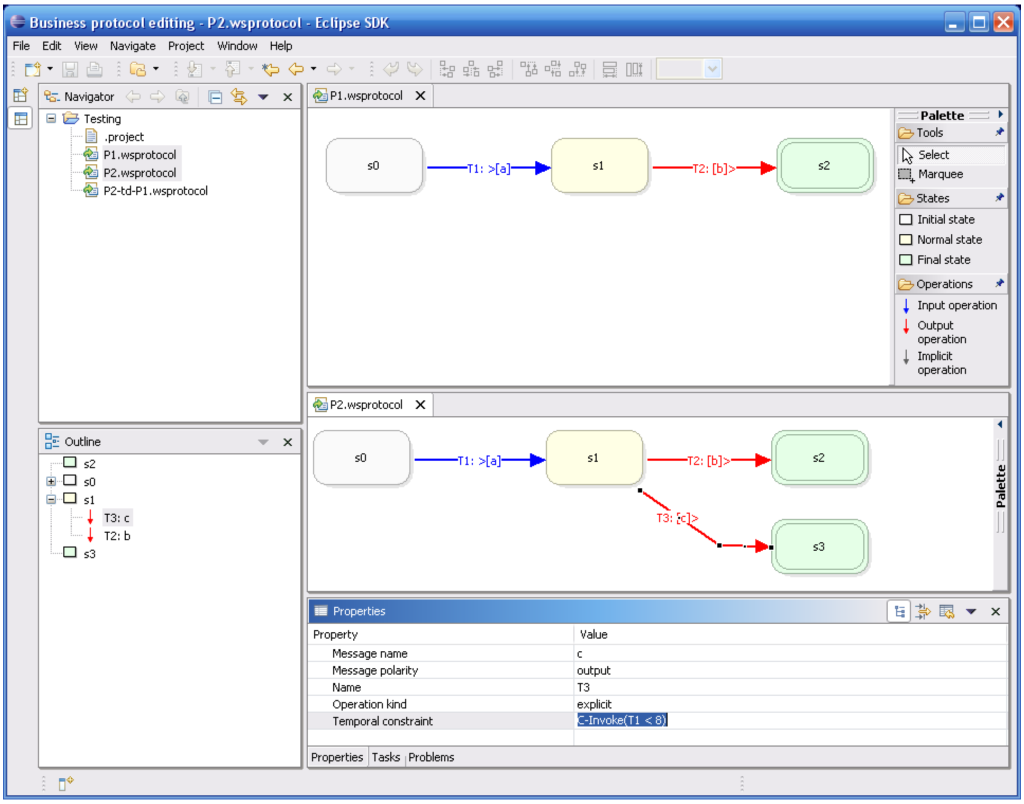
Note: See TracReports for help on using and creating reports.

Download in other formats:

Figure 7.5: Trac issues management view.

class that allows protocols to be stored and read from various XML sources (e.g., files, XML databases, network interface, ...). Temporal constraints are explicitly supported by the library. They are stored as plain strings in the model that are attached to the transitions of a protocol. However, a complete object model is part of the library for manipulating temporal constraints (e.g., programmatically constructing an in-memory constraint, computing the conjunction of two constraints, renaming variables, computing the negation of a constraint, ...). We use a parser written using AntLR[6] (Parr and Quong [110]) for obtaining the object model that corresponds to a constraint in a plain string and vice-versa. As such, the protocol model library is self-contained.

[6] AntLR is actively developed at http://www.antlr.org/.



Property	Value
Message name	c
Message polarity	output
Name	T3
Operation kind	explicit
Temporal constraint	C-Invoke(T1 < 8)

Figure 7.6: Screenshot of the ServiceMosaic Protocols prototype.

A protocol manipulation and comparison operators library has been developed using the Groovy language (Koenig et al. [79]). It implements the complete range of business protocol operators (i.e., $\|^{\mathrm{TI}}$, $\|^{\mathrm{TC}}$, $\|^{\mathrm{TD}}$, $\sqsubseteq$, $\equiv$). The operators can be used "as-is", or a facade class can be used for directly assessing compatibility or replaceability between two protocols. In fact, the facade class does nothing but invokes the manipulation and comparison operators as described in Table 5.2. Emptiness checking is required by some operators. It is done using the UPPAAL model checker using the technique that is presented from page 183.

A set of Eclipse plug-ins have been developed, based on the libraries mentioned above and the Eclipse platform. The business protocol editor is based on a set of customized SWT/JFace figures (e.g., the shape

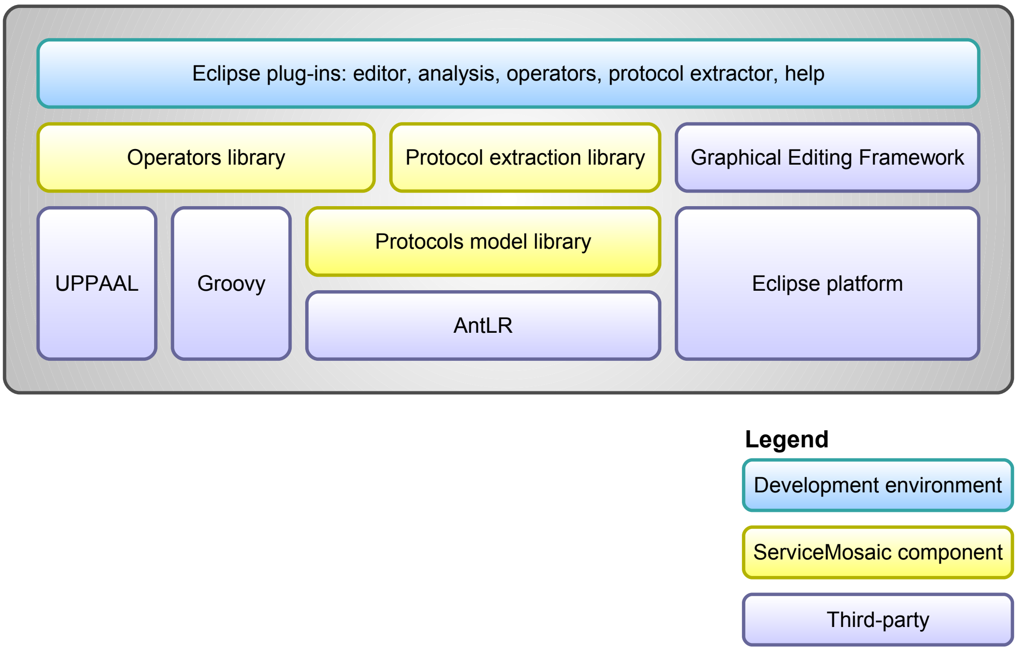


Figure 7.7: Architecture of the ServiceMosaic Protocols prototype.

of states in a protocol) and the Eclipse Graphical Editing Framework (see http://www.eclipse.org/gef/). It can take advantage of WSDL documents for ensuring that only valid messages are being put into protocols by designers. The analysis components provide a visual interface for either invoking the manipulation and comparison operators, or directly checking for compatibility and replaceabilty of protocols. An embedded help support is available for providing assistance.

The figure also depicts a *protocol extraction* component. This companion component having been developed by an engineering intern of the APIS Research Group[7], and since it is still very experimental at this stage, it will be presented separately in the following section.

To make the libraries easier to use, we adopted the use of *fluent interfaces* (Fowler [60]). Briefly, such "fluent" interfaces tend to use techniques such as making method calls chainable. This arguably makes the code

[7] See http://apis.isima.fr/.

sometimes easier to read, and in many cases, fluent interfaces can be enough for making an *internal domain-specific language* (Fowler [59]; Freeman and Pryce [61]).

The components developed in this project have served as the basis of many other ServiceMosaic projects. For example, the protocols model library serves as the standard ServiceMosaic library for developing components that manipulate protocols. Also, the protocol editor has been used in derivative works such as the protocol discovery and refinement tool (Nezhad et al. [98]).

The ServiceMosaic Protocols project are staged to be released in 2008 under the terms of the *GNU Lesser General Public License version 3*[8], a moderate open-source license that facilitates dissemination of the work while enforcing modifications of the code to be released under the same licensing conditions.

7.2.2 Protocol extraction

We now outline a *protocol extraction operator*, developed externally as part of the ServiceMosaic project, that takes a BPEL process as input and outputs a *multi-party protocol*, which is basically 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. Hence, a multi-party business protocol captures the global message *choreography* of a BPEL process.

To extract the multi-party protocol we proceed as follows. 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 extraction patterns as such they are recognized. When a complex activity is encountered (e.g., *if*, *switch*, *while*, *pick*, ...), it is recursively processed on each of its complex

[8] See http://www.gnu.org/licenses/lgpl.html.

activities until basic activities are reached. Hence, 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.

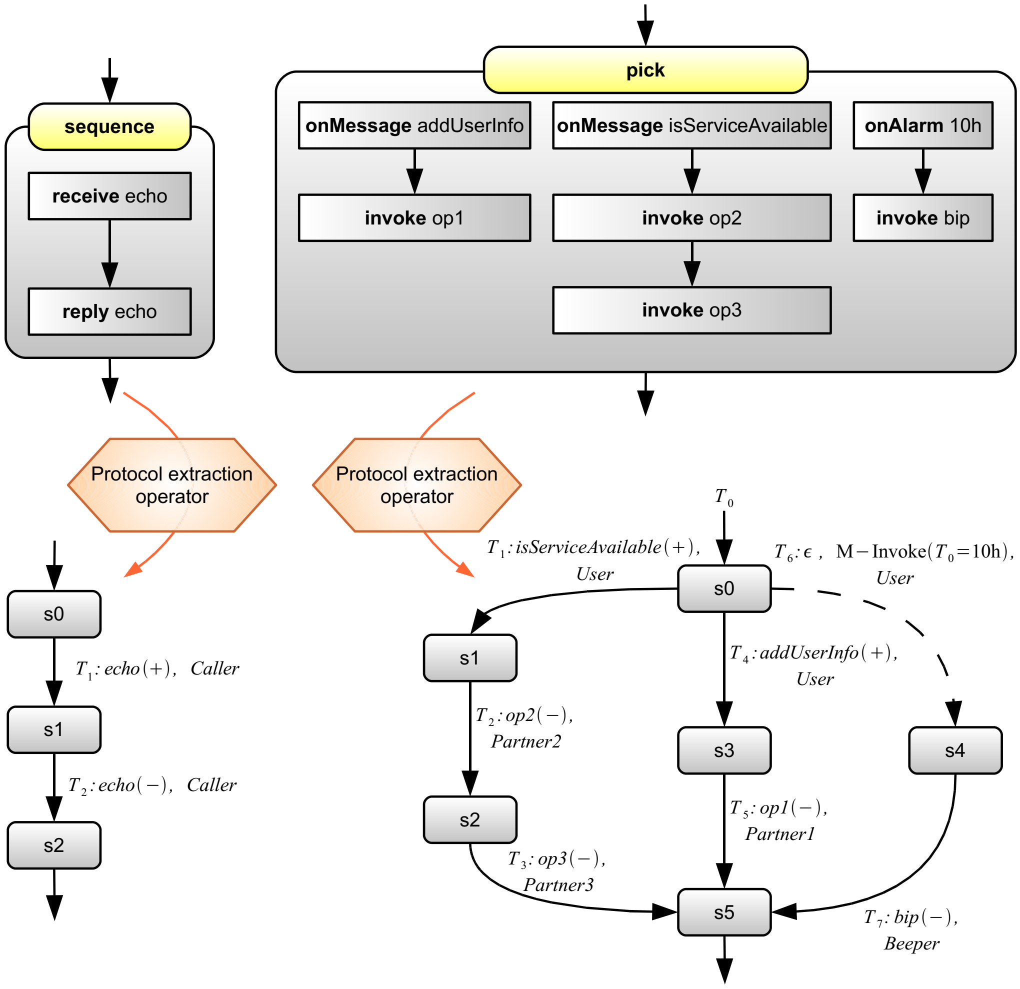


Figure 7.8: Extraction of multi-party protocols fragments from BPEL.

We give two examples of BPEL patterns extractions on Figure 7.8. The first example shows a simple *sequence* activity that consists of a *receive* activity followed by a *reply*. What is done by the sequence is simply the echo-ing of a message. This is translated using two transitions (one for each message). Because a multi-party protocol is extracted, each transition is also tagged with the partner link (e.g. *Caller*) and the BPEL activity (e.g. *reply*).

The second example shows a *pick* complex activity involving two message handlers and an alarm handler. Each message or alarm handler leads to a branch for the message choreography that corresponds to its activities. For example on the *addUserInfo* message handler, there is a $op1$ invocation on the *Partner1* partner link. The *onAlarm* handler leads to an implicit transition featuring a M-Invoke constraint. Temporal informations can be extracted from either *onAlarm* handlers or *wait* activities. Finally, all branches join in the state s_5 which corresponds to the end of the *pick* activity.

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 (Karim Baina et al. [77]).

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 (Rajeev Alur and P. Madhusudan [118]), we replace the messages with ε 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 ε transitions using standard techniques on automata (Hopcroft et al. [74]). This is possible only because if we mapped to timed automata, there would be no guard nor clock resets on these transitions (Volker Diekert et al. [136]).

In our experiments, we have found out that the protocol extraction operator works well for a large majority of BPEL processes. As mentioned previously, this protocol extraction operator is absolutely not part of this thesis work contributions. It will require further investigations, including formalization, a proper theoretical study and implementation work.

Eight

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 chapter. 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.

8.1 BPEL process outline

Consider the BPEL process depicted on Figure 8.1. 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

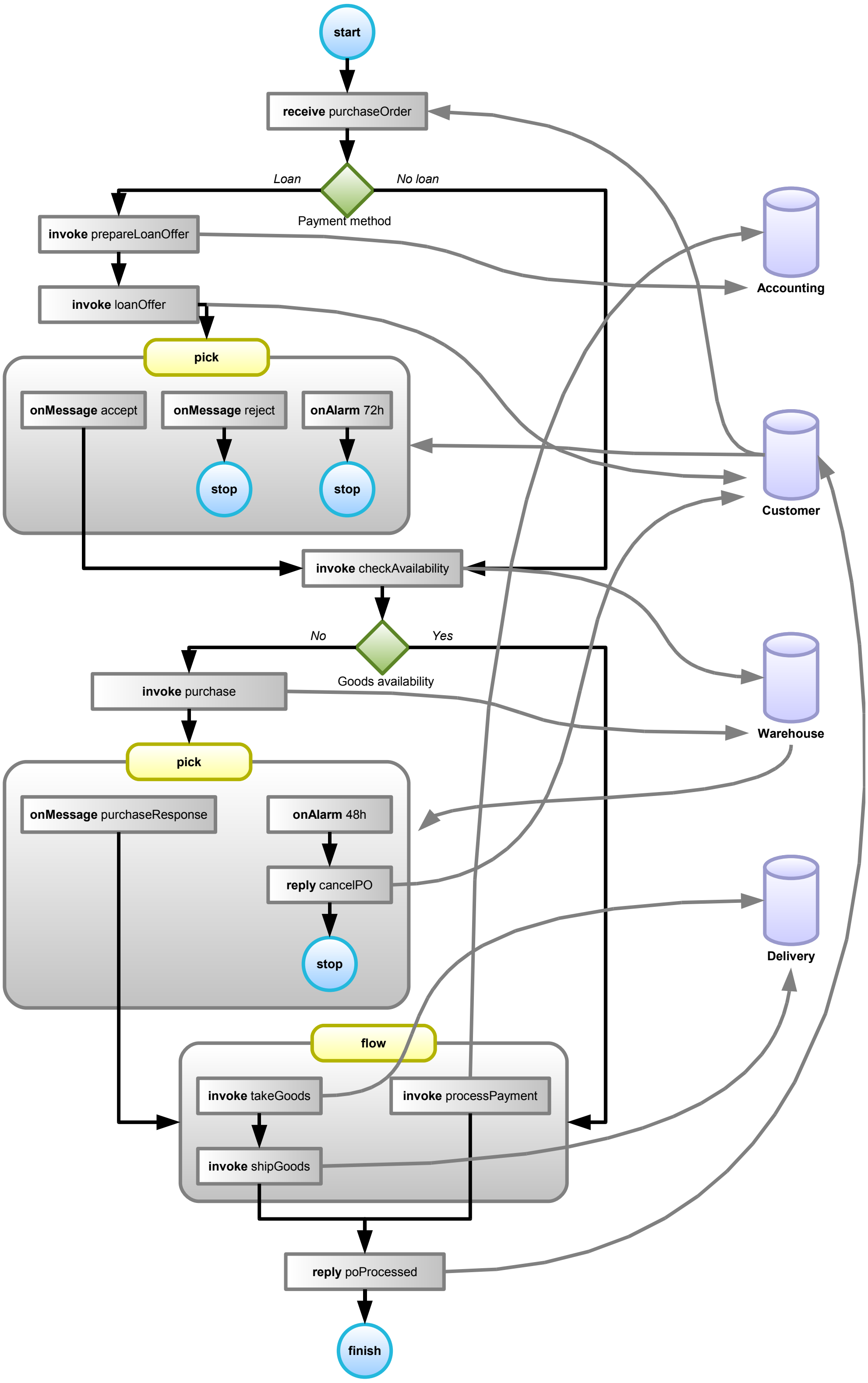


Figure 8.1: Simplified view of a BPEL process that handles purchase orders.

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.

8.2 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 resulting protocols are shown in Figure 8.2. Figure 8.3 shows instead the protocol of the warehouse service we are planning to use as one of the services invoked by our process.

8.3 Protocol analysis

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 8.4) 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.

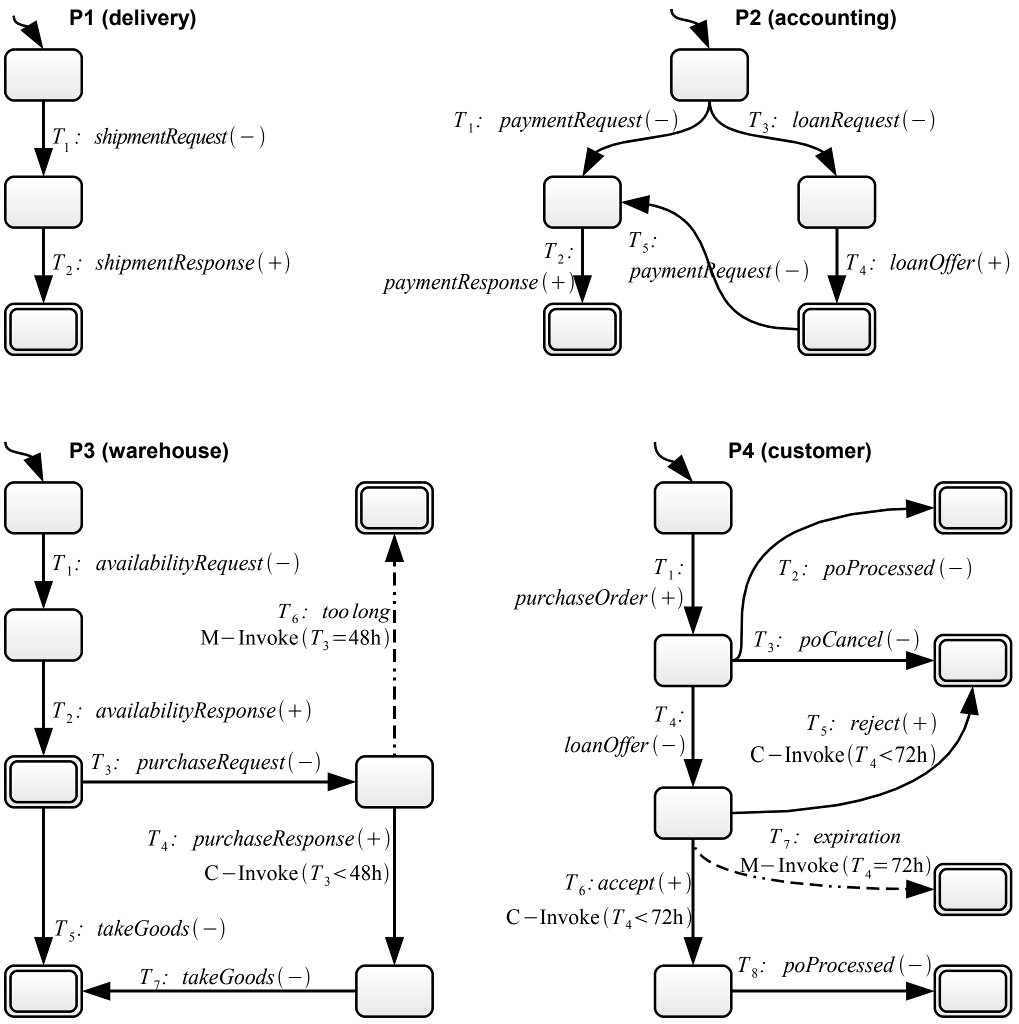


Figure 8.2: Timed protocols extracted from the BPEL process of Figure 8.1.

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 protocol is obtained by computing the inverse P_3' of P_3 and then the difference $P_3^{-1} \|^{\mathtt{TD}} P_5$. The latter is instead computed as $P_5 \|^{\mathtt{TD}} P_3^{-1}$. As

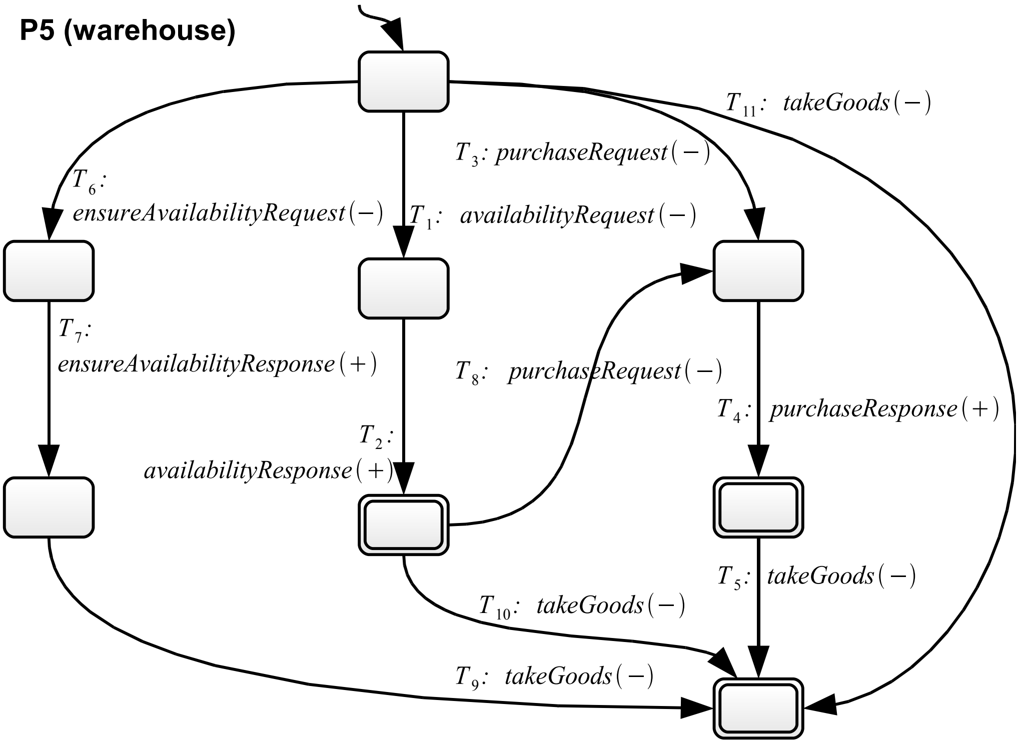


Figure 8.3: The complete warehouse service protocol.

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 8.4 (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 (Benatallah et al. [21]). Indeed, the untimed version of P_5 would have supported all of the conversations of the untimed version of P_3.

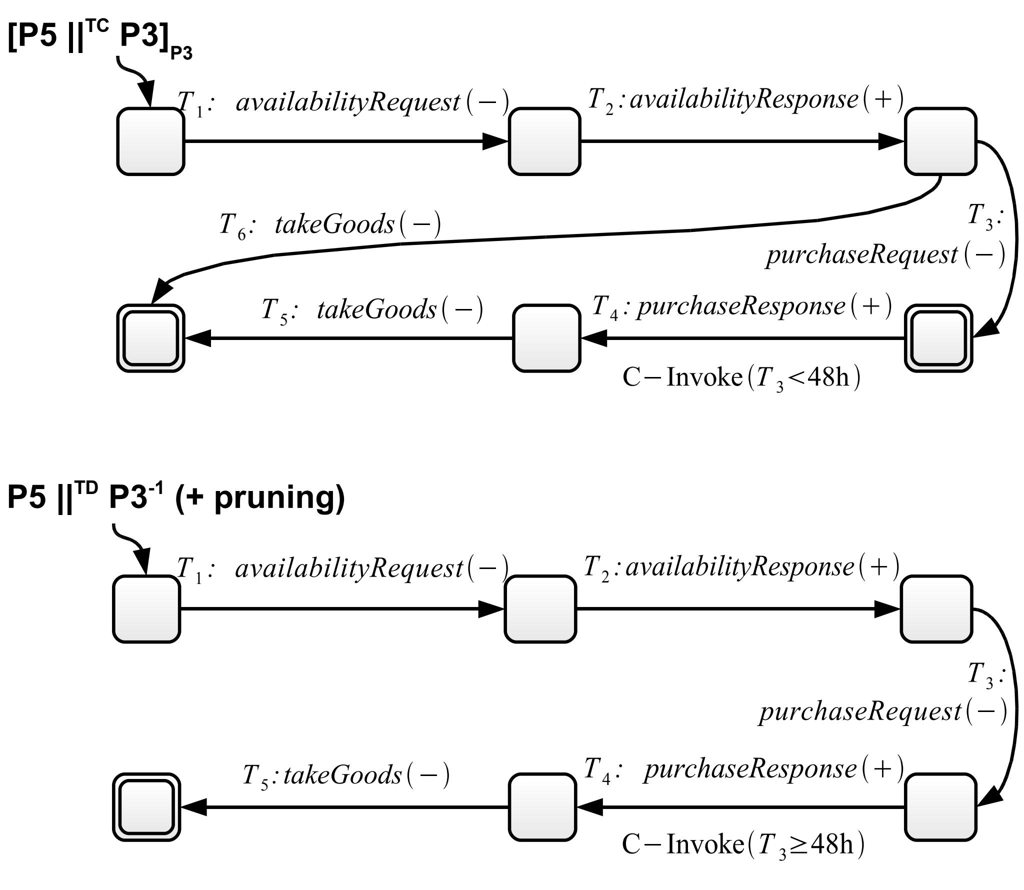


Figure 8.4: Analysis of the common and differing conversations supported by P_3 and P_5.

8.4 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 (Benatallah et al. [19]) to reconcile the differences. It can be done with the ServiceMosaic tools using an aspect-oriented framework (Kongdenfha et al. [80]) where adapters are plugged through *advices* written in BPEL. The adapter is be 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.

Nine

Conclusion and perspectives

We conclude this document by summarizing the contributions. We then provide research and application perspectives beyond this work. Finally, we recall the publications on which this work is based.

9.1 Summary

This work has revisited, formalized and further extended the concepts presented in (Benatallah et al. [16, 18, 21]; Karim Baina et al. [77]) 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, all of this despite the major issue of having ε-labeled switches with clocks resets. Another issue was with the mapping from timed protocols to timed automata itself, as M-Invoke constraints are not easy to represent and enforce in timed automata. We also presented our initial prototype as part of the ServiceMosaic project and gave an application case study.

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.

9.2 Perspectives beyond protocol analysis

The concepts and techniques presented in this thesis focused on both a model for describing service protocols that includes timing constraints, and on compatibility / replaceability analysis between two such protocol instances. While already being innovative by themselves, we believe that those contributions can play a significant role when leveraged in the following contexts.

Protocol discovery and querying. It would be interesting to have protocol repositories as part of service-oriented infrastructures. Such repositories would contain timed protocols for each referenced service. They could either be based on existing repository infrastructures (e.g., UDDI and ebXML), or be standalone (e.g., given a service, point to its WSDL and timed protocol). Users and applications could then leverage them by performing queries based on protocols. Given a protocol, a repository could be queried for services whose protocols would be compatible (or replaceable).

There are several challenges linked to this type of application. The first one is to provide a large-scale, efficient physical representation for timed protocols (e.g., XML files, relational databases or XML databases). The second one is to provide an efficient indexing technique for retrieval based on compatibility and replaceability. While the timed protocol operators allow to assess either compatibility or replaceabilty between two protocols, it is clearly not advisable to take the input protocol from a query and test each protocol from the repository.

Some exploratory work has begun in the APIS Research Group, Clermont-Ferrand. The first results show promising outcomes in terms of compatibility and replaceability based services retrieval.

Web services testing. Providing automated testing at different levels (e.g., unit-testing or functional testing) is critical in today applications. This is becoming even more true in the case of service-oriented computing, as often one has no control on the services it uses. A service may respect a given protocol today, and an upgrade performed tomorrow may introduce a small change that breaks some of its clients. Worse, the changes can happen without any notification having been sent, as a service may not know all of its requesters. Both the model of timed protocols and compatibility/replaceability analysis have a significant role to play for improving web services testing practices. Indeed, protocol-based analysis can be used to detect incompatible protocol updates, while the model by itself can be used for generating conversations in the test cases.

Providing the basic infrastructure for running tests is not a big concern as extending existing tools is relatively easy (e.g., JUnit and TestNG are common testing frameworks[1]). The bigger challenges reside in generating test cases for a given service. Ideally, the set of generated test cases would provide full test coverage in a minimal number of cases. There are two research problems here:

1. generating conversations that would each form the skeleton of a test case, and
2. inject both meaningful and erroneous data in messages.

In terms of conversations generation, one promising technique would be to start from timed protocols and reuse work on automated tests generation in timed automata (Nielsen and Skou [100, 101]).

Runtime stateful support. There is today little support in existing tools and frameworks for stateful web services. In many cases, support

[1] See http://junit.org/ and http://testng.org/.

for stateful interactions has to be implemented manually. This means that developers need to cater with correlation and state management instead of “just” creating their service interfaces.

We propose to leverage the timed protocols model to facilitate the creation of stateful web services. The core idea is to keep service interfaces development simple by taking out such cross-cutting concerns out of the actual implementation code. The framework would be based on:

1. a correlation component that intercepts messages to correlate them with service requester instances, and
2. a conversation controller that checks if the intercepted message does not violate the service protocol, and
3. a dispatcher that forwards messages to the actual service implementation with correlation and state information having been attached transparently to the message context.

Timed protocol discovery and adaptation techniques. Existing work in ServiceMosaic projects tackled the two different problems of protocol discovery and adaptation (Benatallah et al. [19]; Motahari et al. [94]; Nezhad et al. [98]). The protocol model being used was the untimed one from (Benatallah et al. [21]). As such, a natural perspective is to extend it for timed protocols.

“Agile” composition. Today, developing an application by composing existing services (e.g., in BPEL) is arguably a very static process. We envision “agile” frameworks based on some of the components above for facilitating the development and the maintenance of service-based compositions. In this perspective, timed protocols play a critical role (both the model and compatibility / replaceability analysis). The following frameworks would be part of this.

- A development framework that would support the development of a composition using BPEL. It would allow developing the composition without specifying some or all of the services to be involved. Then, it would be able to query a protocols repository for compatible services, meaning that it would allow for *rapid prototyping* by testing different combinations of services. Also, the framework would deal with the assembly details so that developers can almost "drag and drop" services in BPEL processes. Finally, it would also assist in developing adapters when services protocols would not be completely compatible with the BPEL process. Conversely, it would assist in adapting the BPEL process itself if the developer does not want to create adapters.

- A runtime framework would assist when changes get necessary. Indeed, a given service may have to be replaced, either because of unavailability or simply because the composition developers have a compelling reason to do so. In this regard, we can note that having test cases and running them periodically could be useful for rapidly spotting composition breakages. The framework would query a repository for replaceability to find a new service. Again, it would assist in developing adapters or changing the BPEL process definition so as to respect business protocols. In particular, such a framework would make it easier to cope with failures by reducing replacement costs and delays.

9.3 Publications

Parts and preliminary versions of this work have been published.

We started in (Benatallah et al. [20]; Boualem Benatallah et al. [30]) with a simple extension of business protocols that featured only M-Invoke constraints. We had introduced protocol operator algorithms tailored for the model, but it suffered from expressiveness problems. While many C-Invoke constraints could be encoded using M-Invoke -based constructions, many complex ones could not. Also, the M-Invoke constraints always referred to the last action, which is a strong limitation by itself. Finally, the model suffered a states explosion problem when encoding C-Invoke constraints using M-Invoke primitives. The timed protocol model presented here subsumes this initial work.

The model evolved up to a new model presented in (Ponge et al. [113]) which features complex C-Invoke and M-Invoke constraints. It also introduces the reuse of the framework of timed automata for deriving the timed protocol operator properties. The model presented in this thesis contains a few tweaks that make it more expressive with respect to the M-Invoke constraints.

The approach has been presented as part of the larger ServiceMosaic project in (Benatallah et al. [22]). A demonstration featuring the ServiceMosaic tools was given in (Nezhad et al. [99]).

International refereed conferences

- Julien Ponge, Farouk Toumani, Boualem Benatallah and Fabio Casati. **Fine-grained Compatibility and Replaceability Analysis of Timed Web Service Protocols**. *In the 26th International Conference on Conceptual Modeling (ER).* Auckland, New Zealand. November 2007.

- Boualem Benatallah, Fabio Casati, Julien Ponge and Farouk Toumani. **On Temporal Abstractions of Web Services Protocols**. *Proceedings of CAiSE Forum 2005.* Porto, Portugal. June 2005.

National refereed conferences

- Boualem Benatallah, Fabio Casati, Julien Ponge, Farouk Toumani. **Compatibility and replaceability analysis for timed web service protocols**. *In BDA 2005.* Saint-Malo, France. October 2005.

Refereed journals

- 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, vol. 10, no. 4, pp. 55-63. July/August 2006.

Refereed workshops and demonstrations

- Hamid Motahari, Regis Saint-Paul, Boualem Benatallah, Fabio Casati, Julien Ponge and Farouk Toumani. **ServiceMosaic: Interactive Analysis and Manipulations of Service Conversations**. *In International Conference on Data Engineering (ICDE'07).* Istanbul, Turkey. April 2007.

- Julien Ponge, **A New Model For Web Services Timed Business Protocols**. *Atelier "Conception des systemes d'information et services Web"- Conférence INFORSID. Hammamet.* Tunisia, May 2006.

- Julien Ponge, **Modeling and Analysing Web Services Protocols**. *In IBM PhD Student Symposium at ICSOC 2005.* Amsterdam, The Netherlands, December 2005.

Bibliography

[1] *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey*, 2007. IEEE.

[2] 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.

[3] Luca Aceto, no Augusto Burgue and Kim Guldstrand Larsen. Model checking via reachability testing for timed automata. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 263–280, London, UK, 1998. Springer-Verlag. ISBN 3-540-64356-7.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures, and Applications.* Springer Verlag, 2004.

[5] R. Alur, T. Henzinger, and H. Wong-Toi. Symbolic analysis of hybrid systems. In *Proc. 37-th IEEE Conference on Decision and Control, 1997.*, 1997.

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

[7] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *FOCS*, pages 164–169. IEEE, 1989.

[8] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–204, 1994.

[9] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS*, pages 414–425. IEEE Computer Society, 1990.

[10] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993.

[11] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, London, UK, 1993. Springer-Verlag. ISBN 3-540-57318-6.

[12] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.

[13] Beatrice Berard, Volker Diekert, Paul Gastin, and Antoine Petit. Characterization of the expressive power of silent transitions in timed automata. Technical report, LIAFA Jussieu, 1999.

[14] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[15] Boualem Benatallah, Marlon Dumas, and Zakaria Maamar. Definition and execution of composite web services: The self-serv project. *IEEE Data Eng. Bull.*, 25(4):47–52, 2002.

[16] Boualem Benatallah, Fabio Casati, Farouk Toumani, and Rachid Hamadi. Conceptual Modeling of Web Service Conversations. In *Procs of CAiSE'03*, volume 2681 of *LNCS*, pages 449–467, Klagenfurt, Austria, 2003. Springer.

[17] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[18] 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. ISSN 1089-7801. doi: http://doi.ieeecomputersociety.org/10.1109/MIC.2004.1260703.

[19] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In Oscar Pastor and João Falcão e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 415–429. Springer, 2005. ISBN 3-540-26095-1.

[20] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. On temporal abstractions of web service protocols. In Orlando Belo, Johann Eder, João Falcão e Cunha, and Oscar Pastor, editors, *CAiSE Short Paper Proceedings*, volume 161 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[21] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing web service protocols. *Data Knowledge. Engineering*, 58(3):327–357, 2006. ISSN 0169-023X. doi: http://dx.doi.org/10.1016/j.datak.2005.07.006.

[22] 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. ISSN 1089-7801. doi: http://doi.ieeecomputersociety.org/10.1109/MIC.2006.87.

[23] Johan Bengtsson, W. O. David Griffioen, Kaare J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In *CAV '96: Proceedings of the 8th International Conference*

on Computer Aided Verification, pages 244–256, London, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5.

[24] Béatrice Bérard and Catherine Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, 75(1-2):1–7, July 2000. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/BerDuf-IPL2000.ps.gz.

[25] Daniela Berardi, Fabio De Rosa, Luca De Santis, and Massimo Mecella. Finite State Automata as Conceptual Model for e-Services, 2003.

[26] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, and Christoph Quix. Industrial-strength schema matching. *SIGMOD Rec.*, 33(4):38–43, 2004. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/1041410.1041417.

[27] Claudio Bettini, X. Sean Wang, and Sushil Jajodia. Temporal reasoning in workflow systems. *Distrib. Parallel Databases*, 11(3): 269–306, 2002. ISSN 0926-8782. doi: http://dx.doi.org/10.1023/A:1014048800604.

[28] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-046-9. doi: http://doi.acm.org/10.1145/1060745.1060770.

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

[30] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. Compatibility and replaceability analysis for timed web service protocols. In *Proceedings of BDA 2005, Saint-Malo, France*, October 2005.

[31] Patricia Bouyer. Model-cheking timed temporal logics. In Carlos Areces and Stéphane Demri, editors, *Proceedings of the 4th Workshop on Methods for Modalities (M4M-5)*, Electronic Notes in Theoretical Computer Science, Cachan, France, 2008. Elsevier Science Publishers. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/bouyer-M4M5.pdf. To appear.

[32] Patricia Bouyer and François Laroussinie. Vérification par automates temporisés. In Nicolas Navet, editor, *Systèmes temps-réel 1 : techniques de description et de vérification*, pages 121–150. Hermès, June 2006. URL http://www.lavoisier.fr/fr/livres/index.asp?texte=2746213030&select=isbn&from=Hermes.

[33] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theor. Comput. Sci.*, 321(2-3): 291–345, 2004. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2004.04.003.

[34] Patricia Bouyer, Serge Haddad, and Pierre-Alain Reynier. Timed Petri nets and timed automata: On the discriminating power of Zeno sequences. In Michele Buglesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP'06) — Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 420–431, Venice, Italy, July 2006. Springer. doi: 10.1007/11787006_36. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BHR-icalp06.pdf.

[35] 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. URL http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2007-12.pdf. 22 pages.

[36] Patricia Bouyer, Nicolas Markey, Joël Ouaknine, and James Worrell. The cost of punctuality. In *LICS*, pages 109–120. IEEE Computer Society, 2007.

[37] Howard Bowman, Giorgio Faconti, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. Automatic verification of a lip-synchronisation protocol using uppaal. *Formal Aspects of Computing*, 10(5-6):550–575, 1998.

[38] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 546–550, London, UK, 1998. Springer-Verlag. ISBN 3-540-64608-6.

[39] Tevfik Bultan. Modeling Interactions of Web Software (invited paper). In *Proceedings of the Second International Workshop on Automated Specification and Verification of Web Systems (WWV 2006)*, 2006.

[40] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW 2003, Budapest, Hungary*, pages 403–410. ACM, May 2003.

[41] Tevfik Bultan, Jianwen Su, and Xiang Fu. Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):18–25, 2006. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/MIC.2006.1.

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

[43] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, Josï¿½ M. Troya, and Antonio Vallecillo. Adding roles to corba objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2003.1183935.

[44] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The tsimmis project: Integration of heterogeneous information sources. In *IPSJ*, pages 7–18, 1994.

[45] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[46] Daniela Berardi. Automatic Composition Services: Models, Techniques and Tools, PhD thesis, 2002.

[47] Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 416–431, London, UK, 1997. Springer-Verlag. ISBN 3-540-62790-1.

[48] Dave Winer. XML-RPC Specification. http://www.xmlrpc.com/, 1998.

[49] Alexandre David and Wang Yi. Modelling and analysis of a commercial field bus protocol. In *Proceedings of the 12th Euromicro Conference on Real Time Systems*, pages 165–172. IEEE Computer Society, 2000. ISBN 0-7695-0734-4.

[50] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc. ISBN 3-540-61155-X.

[51] Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity Search for Web Services. In *VLDB*, pages 372–383, 2004.

[52] Gregorio Dyaz, M. Emilia Cambronero, Juan J. Pardo, Valentin Valero, and Fernando Cuartero. Automatic generation of correct web services choreographies and orchestrations with model checking techniques. In *AICT-ICIW '06: Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, page 186, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2522-9. doi: http://dx.doi.org/E2A74A0F-588E-4CA4-9E39-169928FAF43A.

[53] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) – W3C Note. http://www.w3.org/TR/wsdl, March 2001.

[54] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[55] Charles Lanny Forgy. *On the efficient implementation of production systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979.

[56] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. *ase*, 0:152, 2003. ISSN 1527-1366. doi: http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240303.

[57] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Leveraging eclipse for integrated model-based engineering of web service compositions. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 95–99, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-000-0. doi: http://doi.acm.org/10.1145/1117696.1117716.

[58] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321127420.

[59] Martin Fowler. Domain specific language. http://martinfowler.com/bliki/DomainSpecificLanguage.html, February 2004.

[60] Martin Fowler. Fluent interface. http://martinfowler.com/bliki/FluentInterface.html, December 2005.

[61] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 855–865. ACM, 2006. ISBN 1-59593-491-X.

[62] 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. ISBN 1-58113-844-X. doi: http://doi.acm.org/10.1145/988672.988756.

[63] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[64] 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. ISBN 3540412174.

[65] François Goasdoué, Véronique Lattès, and Marie-Christine Rousset. The use of carin language and algorithms for information integration: The picsel system. *Int. J. Cooperative Inf. Syst.*, 9(4):383–401, 2000.

[66] Zhifeng Gu, Juanzi Li, Jie Tang, Bin Xu, and Ruobo Huang. Verification of web service conversations specified in wscl. *Compsac*, 2: 432–437, 2007. ISSN 0730-3157. doi: http://doi.ieeecomputersociety.org/10.1109/COMPSAC.2007.235.

[67] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.

[68] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 2, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8268-X.

[69] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. In *ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, pages 277–298, London, UK, 1999. Springer-Verlag. ISBN 3-540-66010-0.

[70] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for timed transition systems. *Inf. Comput.*, 112 (2):273–337, 1994.

[71] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1994.1045.

[72] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 460–463, London, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6.

[73] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321200683.

[74] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, November 2000. ISBN 0201441241. URL http://www.amazon.fr/exec/obidos/ASIN/0201441241/citeulike04-21.

[75] Joel Ouaknine and James Worrell. On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap. In *Proceedings of LICS 04*. IEEE Computer Society Press, 2004.

[76] Zuling Kang, Hongbing Wang, and Patrick C. K. Hung. Ws-cdl+: An extended ws-cdl execution engine for web service collaboration. In *ICWS*, pages 928–935. IEEE Computer Society, 2007.

[77] Karim Baina, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Model-Driven Web Service Development. *Proceedings of CAiSE 2004, Riga, Latvia*, June 2004.

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

[79] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007. ISBN 1932394842.

[80] Woralak Kongdenfha, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. An aspect-oriented framework for service adaptation. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2006. ISBN 3-540-68147-7.

[81] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[82] Maurizio Lenzerini. Data integration: A theoretical perspective. In Lucian Popa, editor, *PODS*, pages 233–246. ACM, 2002.

[83] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. The world wide web as a collection of views: Query processing in the information manifold. In *VIEWS*, pages 43–55, 1996.

[84] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4th Workshop on*

Tools and Algorithms for the Construction and Analysis of Systems, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer–Verlag, March 1998.

[85] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

[86] Nancy A. Lynch and Hagit Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6(2):121–139, 1992.

[87] Elisabetta De Maria, Angelo Montanari, and Marco Zantoni. An automaton-based approach to the verification of timed workflow schemas. In *TIME '06: Proceedings of the Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, pages 87–94, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2617-9. doi: http://dx.doi.org/10.1109/TIME.2006.6.

[88] Maria Sorea. Tempo: A Model Checker for Event-Recording Automata. Technical report, SRI International, 7 November 2001.

[89] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, http://www.w3.org/TR/soap12-part1/, June 2003.

[90] Massimo Mecella, Barbara Pernici, and Paolo Craca. Compatibility of e -services in a cooperative multi-platform environment. In *TES '01: Proceedings of the Second International Workshop on Technologies for E-Services*, pages 44–57, London, UK, 2001. Springer-Verlag. ISBN 3-540-42565-9.

[91] Philip Meir Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, Irvine, 1974.

[92] Michael Merritt, Francesmary Modugno, and Marc R. Tuttle. Time-constrained automata (extended abstract). In Jos C. M. Baeten and

Jan Friso Groote, editors, *CONCUR*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1991. ISBN 3-540-54430-5.

[93] Joseph S. Miller. Decidability and complexity results for timed automata and semi-linear hybrid automata. In Nancy A. Lynch and Bruce H. Krogh, editors, *HSCC*, volume 1790 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 2000. ISBN 3-540-67259-1.

[94] Hamid Motahari, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. To appear in proceedings of the 16th International World Wide Web Conference (WWW2007), Alberta, Canada, May 2007.

[95] Shiva Nejati, Mehrdad Sabetzadeh, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. To appear in the proceedings of ICSE'07, May 2007.

[96] Hamid R. Motahari Nezhad, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web services interoperability specifications. *Computer*, 39(5):24–32, 2006. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.181.

[97] Hamid R. Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 993–1002. ACM, 2007. ISBN 978-1-59593-654-7.

[98] Hamid R. Motahari Nezhad, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. Protocol discovery from imperfect service interaction logs. In *ICDE* DBL [1], pages 1405–1409.

[99] Hamid R. Motahari Nezhad, Régis Saint-Paul, Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. Servicemosaic:

Interactive analysis and manipulation of service conversations. In *ICDE* DBL [1], pages 1497–1498.

[100] Brian Nielsen and Arne Skou. Automated test generation from timed automata. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2001. ISBN 3-540-41865-2.

[101] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *STTT*, 5(1):59–77, 2003.

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

[103] Object Management Group Inc. (OMG). Omg unified modeling language (omg uml), infrastructure, v2.1.2. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/, November 2007.

[104] Joël Ouaknine and James Worrell. On the decidability and complexity of metric temporal logic over finite words. *CoRR*, abs/cs/0702120, 2007.

[105] Joël Ouaknine and James Worrell. On the decidability of metric temporal logic. In *LICS*, pages 188–197. IEEE Computer Society, 2005. ISBN 0-7695-2266-1.

[106] Joël Ouaknine and James Worrell. On metric temporal logic and faulty turing machines. In Luca Aceto and Anna Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2006. ISBN 3-540-33045-3.

[107] M. P. Papazoglou and D. Georgakopoulos. Special issue on service oriented computing. *Commun. ACM*, 46(10):24–28, 2003. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/944217.944233.

[108] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and

research challenges. *Computer*, 40(11):38–45, 2007. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2007.400.

[109] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB Journal*, 16(3):389–415, 2007.

[110] Terence John Parr and Russell W. Quong. Antlr: A predicated- *LL(k)* parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[111] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[112] Amir Pnueli and Eyal Harel. Applications of temporal logic to the specification of real-time systems. In Mathai Joseph, editor, *FTRTFT*, volume 331 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1988. ISBN 3-540-50302-1.

[113] Julien Ponge, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Fine-grained compatibility and replaceability analysis of timed web service protocols. In Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *ER*, volume 4801 of *Lecture Notes in Computer Science*, pages 599–614. Springer, 2007. ISBN 978-3-540-75562-3.

[114] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the Theoretical Foundation of Choreography. To appear in WWW2007, May 2007.

[115] R. Alur, C. Courcoubetis, and T.A. Henzinger. The observational power of clocks. In *Proceedings of the Fifth Conference on Concurrency Theory*, number 836 in LNCS, pages 162–177, 1994.

[116] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s007780100057.

[117] Rajeev Alur. Timed Automata. NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems, 1998.

[118] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *4th Intl. School on Formal Methods for Computer, Communication, and Software Systems: Real Time*, 2004.

[119] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. In *Proceedings of the Sixth Conference on Computer-Aided Verification*, number 818 in LNCS, 1994.

[120] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* Cambridge, Mass.: MIT, Dept. Electrical Engineering, PhD Thesis, 1974.

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

[122] Mohsen Rouached, Olivier Perrin, and Claude Godart. Towards formal verification of web service composition. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 257–273. Springer, 2006. ISBN 3-540-38901-6.

[123] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[124] S. Vinoski. W WS-Nonexistent Standards. *IEEE Internet Computing*, 8(6):94–96, 2004.

[125] Quan Z. Sheng, Boualem Benatallah, Marlon Dumas, and Eileen Oi-Yan Mak. Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. In *VLDB*, pages 1051–1054. Morgan Kaufmann, 2002.

[126] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Software Eng.*, 9(3): 219–228, 1983.

[127] Halvard Skogsrud, Boualem Benatallah, and Fabio Casati. Model-driven trust negotiation for web services. *IEEE Internet Computing*, 7(6):45–52, 2003.

[128] Halvard Skogsrud, Boualem Benatallah, Fabio Casati, and Manh Q. Dinh. Trust-serv: A lightweight trust negotiation service. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 1329–1332. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

[129] Stavros Tripakis. Folk theorems on the determinization and minimization of timed automata. In *FORMATS*, pages 182–188. Springer, 2003.

[130] Ferucio Laurentiu Tiplea and Geanina Ionela Macovei. E-timed workflow nets. *synasc*, 0:423–429, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/SYNASC.2006.33.

[131] Michele Trainotti, Marco Pistore, Gaetano Calabrese, Gabriele Zacco, Gigi Lucchese, Fabio Barbon, Piergiorgio Bertoli, and Paolo Traverso. Astro: Supporting composition and execution of web services. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 495–501. Springer, 2005. ISBN 3-540-30817-2.

[132] Jeffrey D. Ullman. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210, 2000.

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

[134] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[135] Wil M. P. van der Aalst, Lachlan Aldred, Marlon Dumas, and Arthur H. M. ter Hofstede. Design and implementation of the yawl system. In Anne Persson and Janis Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2004. ISBN 3-540-22151-4.

[136] Volker Diekert, Paul Gastin, and Antoine Petit. Removing ε-transitions in timed automata. In *STACS'97*, volume 1200 of *Lecture Notes in Computer Science*, pages 583–594, Lübeck, Germany, February 1997. Springer. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/DGP-stacs97.ps.

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

Part IV

Appendix

One

Proofs

A.1 Proof of Theorem 1

Proof. We need to show that ε-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 ε-transitions. To do that, we exhibit the protocol timed automaton A depicted on Figure 4.8 and use the notions of *precise time* and *precise actions* that were introduced in the Theorem 24 of (Beatrice Berard et al. [13]) as a tool to identify timed languages that can only be recognized by timed automata featuring ε-transitions. The proof is virtually the same as the one of Corollary 29 in (Beatrice Berard et al. [13]).

It is easy to check that A is a protocol timed automaton. A presents 2 ε-transitions lying on directed cycles, hence we don't know if they can be removed using the techniques presented in Section 8 in (Beatrice Berard et al. [13]).

Let us now suppose that $\mathcal{L}(A)$ can be recognized by a timed automaton A' without any ε-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 (Beatrice Berard et al. [13]) for techniques and discussion). In order to leverage the Theorem 24 of (Beatrice Berard et al. [13]), 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 $d \in \mathbb{N}$, $d \geq C_{\text{max}}$ 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, $d \geq C_{\text{max}}$, hence from Theorem 24 of (Beatrice Berard et al. [13]), 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 ε-transitions. □

A.2 Proof of Lemma 1

Proof. $\text{inhib}(g) = (x_1 - x\ \texttt{not}(\#_1)\ k_1 - k) \vee \cdots \vee (x_j - x\ \texttt{not}(\#_j)\ k_j - k) \vee (x_{j+1} - x'_{j+1}\ \texttt{not}(\#_{j+1})\ k_{j+1}) \vee \cdots \vee (x_m - x'_m\ \texttt{not}(\#_m)\ k_m)$

For every $i \in \{1, \cdots, j\}$, note that $(x_i - x\ \texttt{not}(\#_i)\ k_i - k)$ continuously evaluates to either `true` or `false` while the current location is l. Also, note that when $(x = k)$ is satisfied,

$$(x_i - x\ \texttt{not}(\#_i)\ k_i - k) = (x_i\ \texttt{not}(\#_i)\ k_i)$$

which is the exact negation of the clause $(x_i\ \#_i\ k_i) \in g'$.

For every $i \in \{j+1, \cdots, m\}$, $(x_i - x'_i\ \texttt{not}(\#_i)\ k_i)$ is the negation of the clause $(x_i - x'_i\ \#_i\ k_i) \in g'$.

Consequently, when any clause of g' is `false`, $\text{inhib}(g) = \texttt{true}$ and, conversely, when every clause of g' is `true` then $\text{inhib}(g) = \texttt{false}$. □

A.3 Proof of Lemma 2

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

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

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

A.4 Proof of Theorem 2

Proof. Let us compute the cases where 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:

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

and a second one:

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

by reducing the last 3 disjunctions:

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

(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 ε-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 ε-labeled switch to be taken.

□

A.5 Proof of Lemma 3

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

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

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

   and the other switch has a label that is not ε and a guard

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

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

3. The two switches have ε 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 permits$(g_j) \wedge g'_j$ is false, the guards are disjoint.

□

A.6 Proof of Theorem 3

Proof. The proposed constructions yields a timed automaton that qualifies as a protocol timed automaton with respect to Definition 3. Let us have a look at the specificities of protocol timed automata that required an extension of the construction compared to regular timed automata (Alur and Dill [6]).

The introduction of ε-labeled switches in the resulting automaton preserves determinism. Indeed, given a pair of those switches, the step 3 of the construction makes sure that either both switches are triggered at the same time, or one will be triggered before the other one. Without the addition of a third ε-labeled switch that corresponds to both being triggered at the same time, and without the addition of inequality clauses in the original switches, there could be indeterminism, thus effectively breaking closure under intersection.

Finally, the guards rewriting step keeps them sound with respect to their original clocks. Indeed, consider any clock x_e from either of the input automata and its mapped clocks set $\{x_{e,e1}, x_{e,e2}, \cdots, x_{e,en}\}$ ($n > 0$) in the resulting automaton. Then per construction:

- x_e is reset implies that for any $i \in \{1, 2, \cdots, n\}$, $x_{e,ei}$ is also reset, and
- $i \in \{1, 2, \cdots, n\}$, $x_{e,ei}$ is reset implies that x_e is also reset.

Consequently, the guard of both the input and resulting automata exhibit the same behaviors since the clocks that they refer to have synchronous resets. □

A.7 Proof of Theorem 4

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 ε-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 ε-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 (Alur and Dill [6]).

Let us now consider a location l having $n > 0$ ε-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 function 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 ε-labeled switch has been taken for the first clock valuation that stopped satisfying the previous expression, and

2. the current location change through the ε-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 ε-labeled switches:

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

□

Two

Semantics of protocol timed automata

The semantics of protocol timed automata over an infinite labeled transition system $\mathcal{S}_A$ are borrowed from those of event-clock automata (Rajeev Alur et al. [119]), another class of timed automata defined over $\mathbb{T} = \mathbb{R}_{\geq 0} \cup \{\perp\}$.

Definition 8 (Protocol timed automata semantics)**.** *The semantic of a protocol timed automaton $A = (L, L^0, L^f, X \cup Y, E, \Sigma)$ is given by the (infinite) LTS $\mathcal{S}_A = (S, s_0, \rightarrow, \Sigma)$ where:*

- *$S = L \times \mathbb{T}^X$ is the set of states (l, v) (also called* configurations*) comprising each a location and a clocks valuation,*
- *$s_0 = (l_0, v_0)$ with $l_0 \in L^0$ and $v_0(x) =\perp \ \forall x \in X$,*
- *$\rightarrow$ is the transition relation:*
  - *action transitions: $(l, v) \xrightarrow{a} (l', v')$ if and only if there exists $e = (l, g, a, r, l') \in E$ such that $v \models g$, $v' = [r \leftarrow 0]v$*
  - *time transitions: $\xrightarrow{d} (l, v')$ for $d \in \mathbb{R}_{\geq 0}$ if $\forall x \in X$:*
    - *if $v(x) \neq\perp$ then $v'(x) = v(x) + d$, else*
    - *if $v(x) =\perp$ then $v'(x) =\perp$.*

A timed word w is recognized by a protocol timed automaton A if there exists a run over $\mathcal{S}_A$: $(l_0, v_0) \xrightarrow{d_0} (l_0, v_1) \xrightarrow{a} (l_1, v_1) \xrightarrow{\cdots} \cdots \xrightarrow{\cdots} (l_n, v_n)$

such that $l_n \in L^F$.

Reachability analysis is the key to most of the analysis techniques on timed automata. The formal grounds for this reside in the ability to group "similar" configurations that each comprise the current location and a clocks valuation. Those groups, called regions, are built using an equivalence relation between the configurations that represent the states of the (infinite) labeled transition system associated to a timed automaton (Alur and Dill [6]). We adapt an equivalence relation for protocol timed automata.

We define an equivalence relation between the states of $\mathcal{S}_A$, denoted as $\cong$. We denote the integer part of a real-valued number $r \in \mathbb{R}_{\geq 0}$ as $\lfloor r \rfloor$ and its fractional part as $\{r\}$ (e.g., $\lfloor 2.55 \rfloor = 2$ and $\{2.55\} = 0.55$).

Definition 9. *Two configurations* (l, v) *and* (l', v') *are* equivalent*, denoted as* $(l, v) \cong (l', v')$ *if* $l = l'$ *and:*

1. $\forall x \in X$*:*
   - *if* $v(x) = \perp$ *then* $v'(x) = \perp$*, else*
   - $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ *or both* $v(x) > c_x$ *and* $v'(x) > c_x$
2. $\forall x, y \in X$ *such as* $v(x) \leq c_x$ *and* $v(y) \leq c_y$*, then* $\{v(x)\} \leq \{v(y)\}$ *if and only if* $\{v'(x)\} \leq \{v'(y)\}$
3. $\forall x \in X$ *such as* $v(x) \leq c_x$*,* $\{v(x)\} = 0$ *if and only if* $\{v(x)\} = 0$*.*

Three

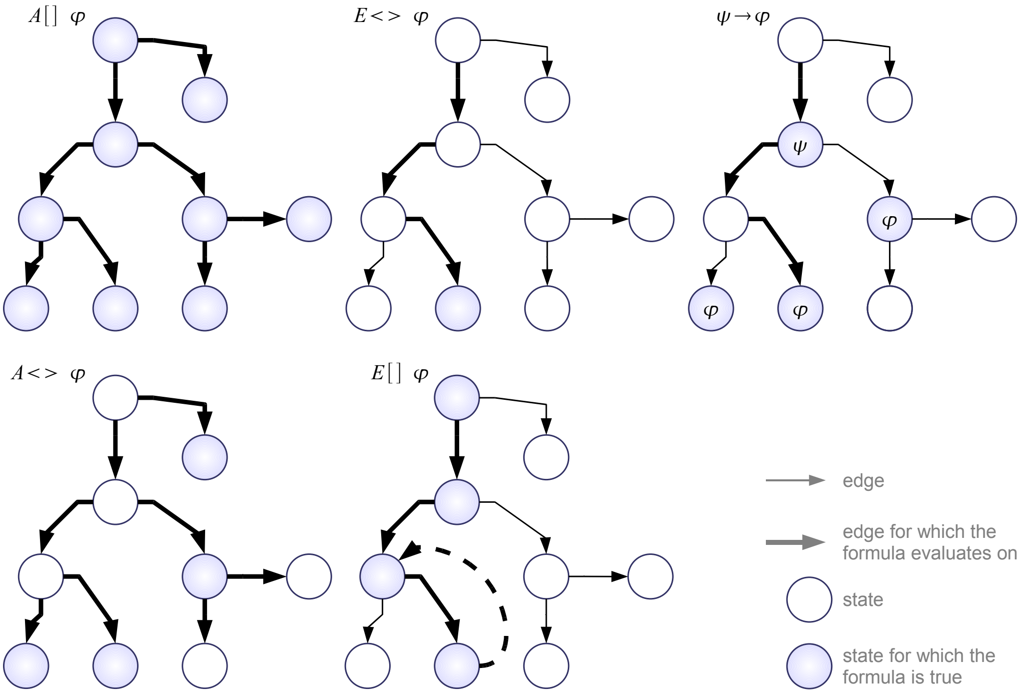
An overview of UPPAAL

We present here the model used in UPPAAL which is based on timed automata. Then, we present the query language of UPPAAL before finishing with the tools that it provides.

C.1 UPPAAL model and query language

UPPAAL uses an hybrid extension of timed automata as a model and a subset of TCTL as a query language to express properties. Briefly, an hybrid system (Alur et al. [5]) features both continuous (e.g., variables in $\mathbb{R}$) and discrete behavior (e.g., variables in $\mathbb{N}$). This allows mixing discrete variables such as booleans or integers with clocks in the models that are processed by UPPAAL. Those discrete variables can be used in guards, invariants and be reset on switches just like a clock would be. Actually, a timed automaton can be viewed as a linear hybrid automaton whose continuous variables evolve at a constant rate (Alur et al. [5]). Further pointers on hybrid automata can be found in (Alur et al. [11]; Henzinger et al. [72]; Miller [93]).

We summarize here the types of formula that can be defined for queries in UPPAAL as detailed in (Behrmann et al. [14]). We will use Figure C.1 to illustrate them. To begin with, we need *state formula*. Such a formula can be made from clock constraints (e.g., $(x < 3.75) \wedge (y \geq 5)$) and tests



(source: (Behrmann et al. [14]))

Figure C.1: The path formulae that UPPAAL supports.

that check if a system is in a given location or not (e.g., $P.l$ is true when the system P is currently in the l location). By combining them using boolean conjunctions and disjunctions, it is possible to define complex state formulae such as $\varphi = (x < 3.75) \wedge (y \geq 5) \wedge P.l$. There is also a special state formula `deadlock` that is true when neither the current extended state (l, v) offers an outgoing action transition nor there exists a time successor (l, v') that does so. Using state formulae as a basis, the path formulae that can be expressed have been summarized in Table C.1.

Going back to the examples of verifications that we expressed as *property 1* and *property 2* on Figure 3.3, the properties can be checked by UPPAAL using the following formulae:

- *property 1*: $E\Diamond(\mathsf{Model.off} \wedge \mathsf{x} = 10)$ (the location $\mathsf{Model.off}$ is reach-

Formula	*Semantic*	*Analysis*
$E\Diamond\varphi$	There exists a path to a state (l, v) that verifies φ.	Reachability
$A\Box\varphi$	Every reachable state (l, v) satisfies φ.	Safety
$E\Box\varphi$	There exists a path to a state (l, v) with no outgoing transition that satisfies φ, or there exists an infinite path where φ is satisfied by all states at some point.	Safety
$A\Diamond\varphi$	For every possible path, there exists a state (l, v) such that φ is satisfied.	Liveness
$\varphi \rightarrow \psi$, or equivalently $A\Box(\varphi \Rightarrow A\Diamond\psi)$	Given a state (l_1, v_1) satisfying φ, every path starting from it contains a state (l_2, v_2) such that ψ is satisfied.	Liveness

Table C.1: Semantics of the path formulae supported by UPPAAL (a subset of TCTL).

able while the clock x has a valuation of 10)

- *property 2*: $E\Diamond(\mathsf{Model.off} \land \mathsf{x} = 1)$ (there is a path to a state in the LTS of Model such that the current location is Model.off and the current valuation of the clock x is 1).

UPPAAL evaluates the first property to `true` and the second one to `false`. Please note that "undesirable" properties (i.e., the ones that should normally evaluate to `false`) are expressed *positively* in UPPAAL (as opposed to *negatively* using a boolean negation operator).

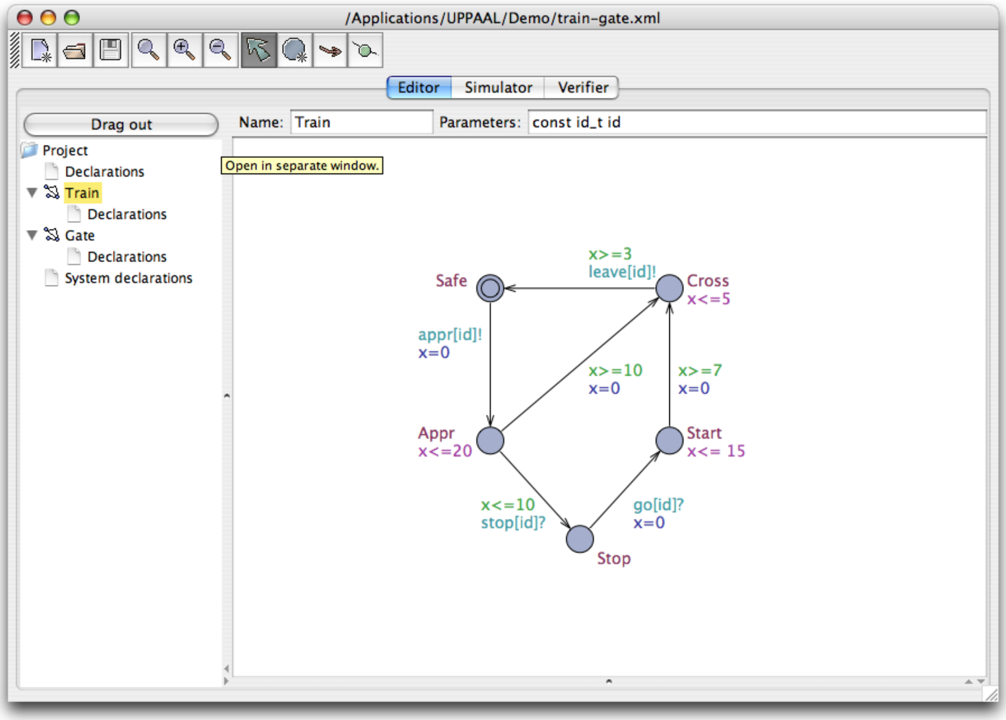
UPPAAL supports a subset of TCTL. which provides more path formulae. Also, TCTL allows nested path formulae, something that UPPAAL does not support.

C.2 UPPAAL tools

UPPAAL comprises two main components:

1. a command-line model checker called *verifyta*, written in C and ported to Unix variants (Linux, *BSD), Windows and Mac OS X, and

2. a graphical user interface (GUI) written in Java.



/Applications/UPPAAL/Demo/train-gate.xml

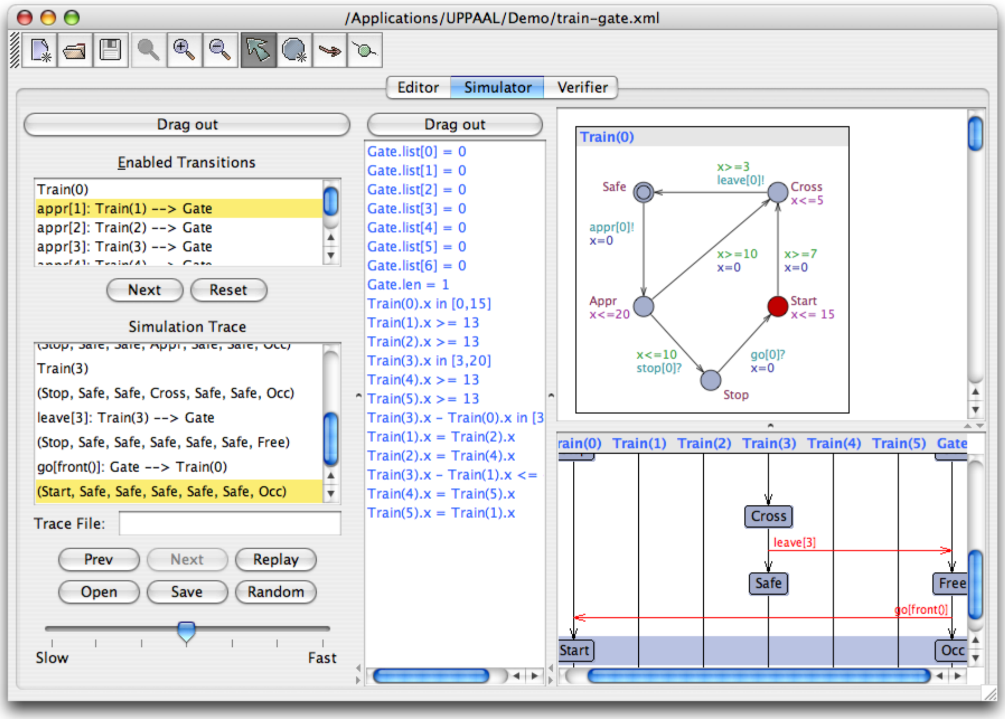
Name: Train

Parameters: const id_t id

Figure C.2: UPPAAL: modeling environment.

The UPPAAL GUI (see Figure C.2) is a comprehensive environment for modeling, simulating and verifying systems represented as timed automata. The editor allows to define a set of (usually interacting) systems (e.g., the gate, barrier controller and train systems of the examples found in (Alur and Dill [6]) or the light controller of Figure 3.3).

The simulator component (see Figure C.3) allows users to simulate the execution of the systems to see which switches are taken, what are the clocks valuation ranges at each step and so on. The simulation can be run automatically by the tool: when several switches can be enabled from a given location, the next switch is selected randomly. Otherwise, the user can select which switch should be taken next at each step. In the later case, the simulator acts as a kind of step-by-step debugger as found for traditional programming using languages such as C/C++ or Java.



Enabled Transitions

```
Train(0)
appr[1]: Train(1) --> Gate
appr[2]: Train(2) --> Gate
appr[3]: Train(3) --> Gate
appr[4]: Train(4) --> Gate
```

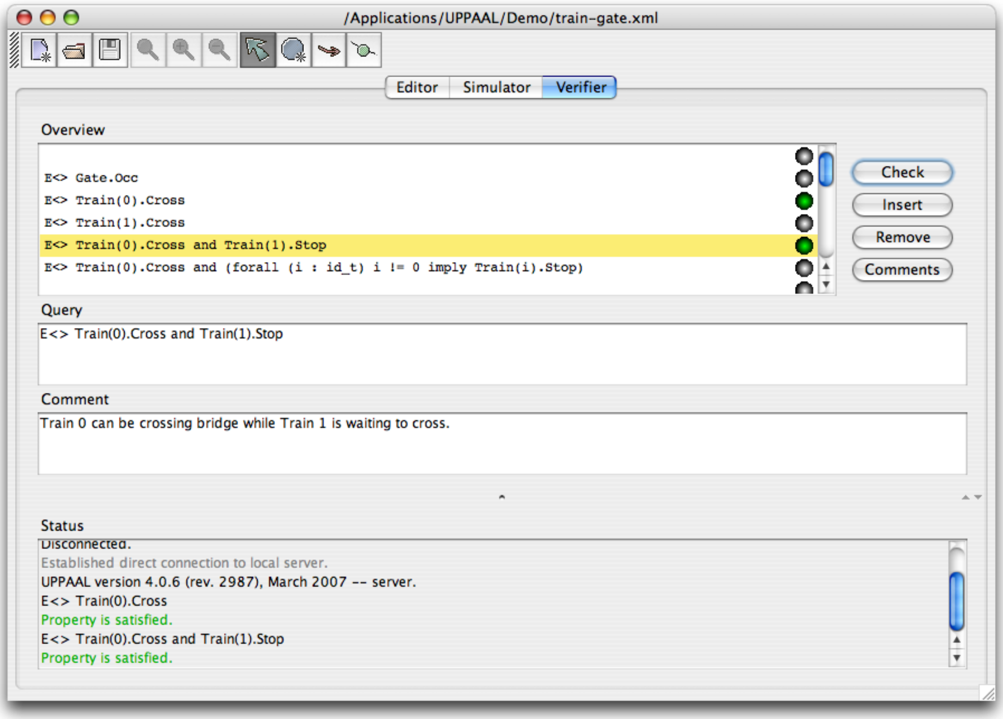
Simulation Trace

```
(Stop, Safe, Safe, Appr, Safe, Safe, Occ)
Train(3)
(Stop, Safe, Safe, Cross, Safe, Safe, Occ)
leave[3]: Train(3) --> Gate
(Stop, Safe, Safe, Safe, Safe, Safe, Free)
go[front()]: Gate --> Train(0)
(Start, Safe, Safe, Safe, Safe, Safe, Occ)
```

```
Gate.list[0] = 0
Gate.list[1] = 0
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 1
Train(0).x in [0,15]
Train(1).x >= 13
Train(2).x >= 13
Train(3).x in [3,20]
Train(4).x >= 13
Train(5).x >= 13
Train(3).x - Train(0).x in [3
Train(1).x = Train(2).x
Train(2).x = Train(4).x
Train(3).x - Train(1).x <=
Train(4).x = Train(5).x
Train(5).x = Train(1).x
```

Figure C.3: UPPAAL: simulation environment.

Finally, the verification component (see Figure C.4) allows to perform model checking tasks by encoding requests in (timed) temporal logics. In fact, this component is directly using the command-line *verifyta* tool to perform the verifications.



Overview

```
E<> Gate.Occ
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)
```

Query

E<> Train(0).Cross and Train(1).Stop

Comment

Train 0 can be crossing bridge while Train 1 is waiting to cross.

Status

```
Disconnected.
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2987), March 2007 -- server.
E<> Train(0).Cross
Property is satisfied.
E<> Train(0).Cross and Train(1).Stop
Property is satisfied.
```

Figure C.4: UPPAAL: verification environment.

Four

UPPAAL and protocol timed automata

As we will see hereafter, checking emptiness on protocol timed automata with UPPAAL is not straightforward. We first explain the issues in converting a protocol timed automaton into an automaton that UPPAAL can process. We then propose a procedure for performing a semantic-preserving conversion. Finally, we illustrate it on the protocol P_2' of Figure D.1.

D.1 Conversion issues

We use UPPAAL for verifying whether a timed protocol is empty or not, i.e., whether it supports at least a conversation or not. To do that, we consider each final location of a protocol timed automaton and leverage UPPAAL to verify if it can be reached from its initial location or not (i.e., there exists a timed word whose execution over the protocol timed automaton ends in the considered final location). The language that is recognized by a protocol timed automaton is not empty as long as one final location can be reached.

This checking is however not straightforward. Indeed, a protocol timed automaton cannot be directly given to UPPAAL:

1. clocks in UPPAAL are defined over $\mathbb{R}_{\geq 0}$ whereas protocol timed au-

tomata are defined over $\mathbb{R}_{\geq 0} \cup \{\perp\}$, and

2. in protocol timed automata, the clocks are initially set to $\perp$ and keep this value until they have been reset for the first time while in classical timed automata (and hence for UPPAAL) they start from 0 and immediately start their progression, and

3. UPPAAL expects the guards to be conjunctions of atomic clock constraints (e.g., $x \# k$) and diagonal constraints (e.g., $x - y \# k$).

D.2 Conversion technique

These differences can be overcome as UPPAAL does not work on *strict* timed automata, but rather uses an hybrid model where non-clock variables can be defined, reset on switches and used in guards (Behrmann et al. [14]). We propose the following procedure for checking protocol timed automata emptiness using UPPAAL. It has been implemented as part of the ServiceMosaic Protocols project presented earlier.

Procedure 4. *Let us consider a protocol timed automaton A. The first step is to generate a UPPAAL model* `Process` *as follows.*

1. *Remove the disjunctions in the guards of A:*

   a) *obtain the abstract syntax tree (AST) of each constraint, then*

   b) *use an AST visitor (Gamma et al. [63]): for each disjunction node, split into two switches (the disjunction node is replaced by the left-hand node in one switch guard, and by the right-hand node in the other switch guard)*

   c) *repeat until no AST contains a disjunction.*

2. *Assign a unique boolean variable b_x (initially set to* `false`*) to each switch that is set to* `true` *in the switch reset. This encodes $\perp$.*

3. *Using an AST visitor, for each constraint clause $(x \# k)$ where $k \neq \perp$, rewrite as $(x \# k) \wedge (b_x = \texttt{true})$ (note that it does not make sense to*

write a diagonal constraint comparing to $\bot$*). When* $k = \bot$*, rewrite it as* $(x \# k) \wedge (b_x = \texttt{false})$*.*

Finally, to check for emptiness: for each final location l, the corresponding TCTL property to be passed to UPPAAL is `E<> Process.l` *and the procedure stops at the first final location l such that the property is met, else the the timed language is empty.*

We can point out the following remarks.

- A procedure can be derived from this one to verify event-recording timed automata using UPPAAL.
- The procedure generates an automaton which is not a protocol timed automaton.
- Removing disjunctions from guards had already been discussed in (Beatrice Berard et al. [13]) – in fact the technique is the same.

D.3 Sample conversion and emptiness checking

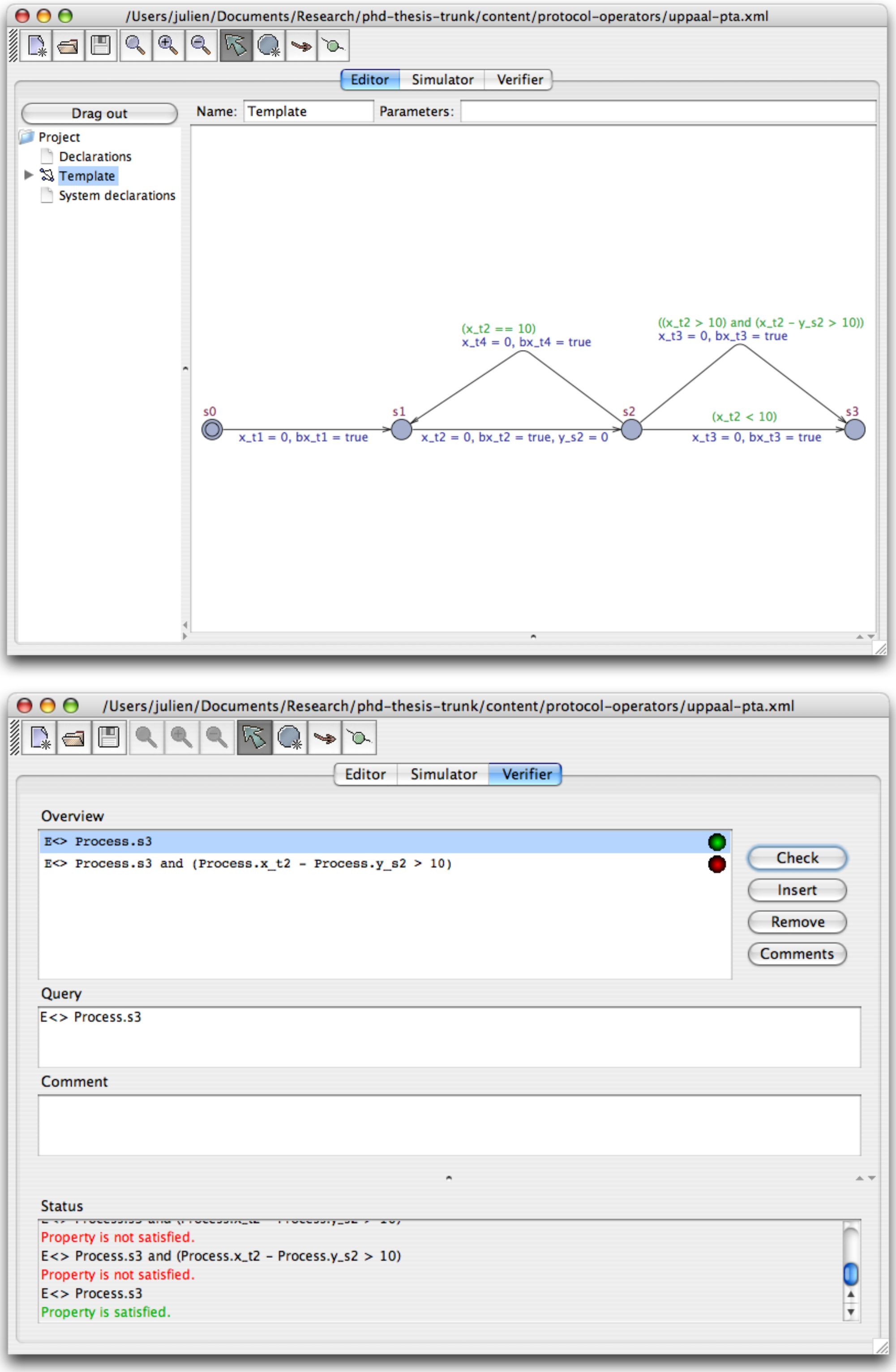
Let us consider the protocol timed automaton P_2 of Figure 6.1. The conversion to a UPPAAL automaton is illustrated on Figure D.1 along with the emptiness checking. As we can see, the conversion involves adding an extra switch between S_2' and S_3' as the permits clause is disjunctive and UPPAAL forbids disjunctions in guards.

The emptiness checking is performed using the following query:

$$E\Diamond \; \texttt{Process.s3}$$

On the figure, we have also added another query:

$$E\Diamond \; \texttt{Process.s3 and } (\texttt{Process.x\_t2} - \texttt{Process.y\_s2} > 10)$$



/Users/julien/Documents/Research/phd-thesis-trunk/content/protocol-operators/uppaal-pta.xml

Name: Template

Parameters:

/Users/julien/Documents/Research/phd-thesis-trunk/content/protocol-operators/uppaal-pta.xml

Overview

```
E<> Process.s3
E<> Process.s3 and (Process.x_t2 - Process.y_s2 > 10)
```

Query

```
E<> Process.s3
```

Comment

Status

```
Property is not satisfied.
E<> Process.s3 and (Process.x_t2 - Process.y_s2 > 10)
Property is not satisfied.
E<> Process.s3
Property is satisfied.
```

Figure D.1: Emptiness checking of protocol timed automata with UPPAAL.

that checks that the ε-labeled switch is enforced. Indeed, due to the configuration of this automaton, the added switch with a guard `((x_t2 > 10) and (x_t2 - y_s2 > 10))` cannot be taken under M-Invoke semantics.

Here is the corresponding UPPAAL automaton XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN"
"http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd">
<nta>
  <declaration></declaration>
  <template>
    <name x="5" y="5">Template</name>
    <declaration>
      clock x_t1;
      clock x_t2;
      clock x_t3;
      clock x_t4;
      clock y_s2;
      bool bx_t1 = false;
      bool bx_t2 = false;
      bool bx_t3 = false;
      bool bx_t4 = false;
    </declaration>
    <location id="id0" x="224" y="-320">
      <name x="214" y="-350">s3</name>
    </location>
    <location id="id1" x="-40" y="-320">
      <name x="-50" y="-350">s2</name>
    </location>
    <location id="id2" x="-312" y="-320">
      <name x="-322" y="-350">s1</name>
    </location>
    <location id="id3" x="-536" y="-320">
      <name x="-546" y="-350">s0</name>
    </location>
    <init ref="id3"/>
    <transition>
      <source ref="id1"/>
```

```
   <target ref="id0"/>
   <label kind="guard" x="-8" y="-456">
    ((x_t2 &gt; 10) and (x_t2 - y_s2 &gt; 10))</label>
   <label kind="assignment" x="32" y="-320">
    x_t3 = 0, bx_t3 = true</label>
   <nail x="88" y="-424"/>
  </transition>
  <transition>
   <source ref="id1"/>
   <target ref="id2"/>
   <label kind="guard" x="-240" y="-448">
    (x_t2 == 10)</label>
   <label kind="assignment" x="-240" y="-432">
    x_t4 = 0, bx_t4 = true</label>
   <nail x="-168" y="-416"/>
  </transition>
  <transition>
   <source ref="id1"/>
   <target ref="id0"/>
   <label kind="guard" x="56" y="-344">(x_t2 &lt; 10)</label>
   <label kind="assignment" x="-8" y="-440">
    x_t3 = 0, bx_t3 = true</label>
  </transition>
  <transition>
   <source ref="id2"/>
   <target ref="id1"/>
   <label kind="assignment" x="-288" y="-320">
    x_t2 = 0, bx_t2 = true, y_s2 = 0</label>
  </transition>
  <transition>
   <source ref="id3"/>
   <target ref="id2"/>
   <label kind="assignment" x="-504" y="-320">
    x_t1 = 0, bx_t1 = true</label>
  </transition>
 </template>
 <system>
  Process = Template();
  system Process;
  </system>
</nta>
```

Here is also the UPPAAL queries file:

```
E<> Process.s3
E<> Process.s3 and (Process.x_t2 - Process.y_s2 > 10)
```

Notice regarding the production of this document

This document was produced with a LaTeX-based tools chain and the memoir class by Peter Wilson which is available from CTAN at http://www.ctan.org/tex-archive/macros/latex/contrib/memoir/.

Screen captures of various softwares appear in several places of this document. Those images were captured by the author for the sole purpose of illustration.

The figures were produced using the OpenOffice.org Draw software which is available from http://www.openoffice.org/. Some graphic elements used in the figures originate from the Crystal Project and the Tango Desktop Project. They are used in due respect to their (permissive) licenses agreements. Refer to http://everaldo.com/crystal/ and http://tango.freedesktop.org/Tango_Desktop_Project for more informations.