PhD-FSTC-2016-45
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 24/10/2016 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

WEI DOU
Born on 5 June 1987 in Xianyang (Shaanxi, China)

A MODEL-DRIVEN APPROACH TO OFFLINE TRACE CHECKING OF TEMPORAL PROPERTIES

Abstract

Offline trace checking is a procedure for evaluating requirements over a log of events produced by a system. The goal of this thesis is to present a practical and scalable solution for the offline checking of the temporal requirements of a system, which can be used in contexts where model-driven engineering is already a practice, where temporal specifications should be written in a domain-specific language not requiring a strong mathematical background, and where relying on standards and industry-strength tools for property checking is a fundamental prerequisite.

The main contributions of this thesis are: i) the *TemPsy* (Temporal Properties made easy) language, a pattern-based domain-specific language for the specification of temporal properties; ii) a model-driven trace checking procedure, which relies on an optimized mapping of temporal requirements written in *TemPsy* into Object Constraint Language (OCL) constraints on a conceptual model of execution traces; iii) a model-driven approach to violation information collection, which relies on the evaluation of OCL queries on an instance of the trace model; iv) three publicly-available tools: 1) TEMPSY-CHECK and 2) TEMPSY-REPORT, implementing, respectively, the trace checking and violation information collection procedures; 3) an interactive visualization tool for navigating and analyzing the violation information collected by TEMPSY-REPORT; v) an evaluation of the scalability of TEMPSY-CHECK and TEMPSY-REPORT, when applied to the verification of real properties.

The proposed approaches have been applied to and evaluated on a case study developed in collaboration with a public service organization, active in the domain of business process modeling for eGovernment. The experimental results show that TEMPSY-CHECK is able to analyze traces with one million events in about two seconds, and TEMPSY-REPORT can collect violation information from such large traces in less than ten seconds; both tools scale linearly with respect to the length of the trace.

Acknowledgements

This thesis is the culmination of four years of my research at the Software Verification and Validation lab. I would like to express my gratitude to these people who helped me with my work and my life during this journey.

I would like to first thank my supervisor Prof. Dr.-Ing. Lionel Briand for his continuous support of my research, for his vision, motivation, and immense knowledge. I would also like to thank my co-supervisor Dr. Domenico Bianculli for his unceasing advice on my research and his assistance for writing high-quality research papers. Without their guidance, I would not have enjoyed and finished my research.

I would like to thank our public service partner CTIE for supporting my research and providing time, data, and feedback for the evaluation of my research.

I would like to express my gratitude to the other committee members: Dr. Radu State, Dr. Mehrdad Sabetzadeh, Prof. Dr.-Ing. Cesare Pautasso, and Prof Dr.-Ing. Carlo Alberto Furia, for their valuable time and remarks on my work.

My thanks also go to my officemates: Ameni Ben Fadhel, Ines Hajri, Salma Messaoudi, Bing Liu, Ha Thanh Le, Sadeeq Jan, and Julian Thome, who often supported me and made a pleasant work environment. I would also like to thank Chetan Arora and Ashkan Kalantari, for their generous help, in particular at the beginning of my life in Luxembourg.

Finally, I want to express my gratitude to my loved mother, father, and sister for their endless support and encouragement from far away.

Contents

List of Figures

List of Tables

Acronyms

BPMN	Business Process Model and Notation
CSV	Comma Separated Values
CTIE	Centre des technologies de l'information de l'Etat; Luxembourg national center for information technology
CTL	Computation Tree Logic
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
LTL	Linear Temporal Logic
MDE	Model-Driven Engineering
MTL	Metric Temporal Logic
OCL	Object Constraint Language
OMG	Object Management Group
UML	Unified Modeling Language
XMI	XML Metadata Interchange

Chapter 1

Introduction

1.1 Motivation

Modern enterprise information systems are often designed and built using the principles and technologies of business process modeling, based on business process languages like Business Process Model and Notation (BPMN) [OMG, 2011a]. Recently, the design and implementation of business processes have started leveraging Model-Driven Engineering (MDE) methodologies [Brambilla et al., 2010] and code generation techniques. For example, our public service partner CTIE (Centre des technologies de l'information de l'Etat; Luxembourg national center for information technology[1]), from which we draw the main motivation of this thesis and our case study, has developed in-house a model-driven methodology for designing eGovernment business processes.

These business processes are usually very complex and are realized as compositions of services provided by different administrations, and third-party suppliers. They act as the "glue" to orchestrate different information systems, possibly by many different organizations, in an effort to foster cooperation of various administrations. Designing and operating effective and efficient processes to drive e-service delivery is one of the most challenging tasks for public administrations. The correct enactment of business processes is of utmost importance to guarantee reliable digital solutions to citizens and enterprises, as well as to foster an effective cooperation of the various public administrations in a state.

From a more general standpoint, in information systems, the correct enactment of a business process can be ensured [Baresi et al., 2007] by:

1) precisely specifying its requirements;
2) using a verification technique to check the compliance of the business process with respect to its requirements;
3) reporting useful and clear information when a verification procedure finds a violation of the business process requirements.

[1]`www.ctie.public.lu`.

Regarding the specification of requirements of business processes, the analysis of the requirements of various applications developed as business processes by our partner revealed that the majority of these requirements could be expressed as temporal properties, enriched with timing information. Temporal properties are qualitative about the occurrence of an event or the order in which multiple events should occur. For example, the requirement "If a card has been registered as lost, a new card should be produced and issued" specifies the order in which the events should occur one after another. In addition to temporal properties, timing information are quantitative about exact time or time distance. For example, the requirement "A new card should be produced at least two days before its issuance" specifies the time distance between two sequential events. Temporal and timing properties have been widely studied in the context of concurrent, real-time critical systems [Dwyer et al., 1999] and, more recently, also in other domains like service-based applications [Bianculli et al., 2012, Li et al., 2005, Simmonds et al., 2009, Kallel et al., 2009] and automotive [Post et al., 2012]. There have been several proposals to formally specify these properties; many of these proposals rely on some temporal logic, either the classic Linear Temporal Logic (LTL) or Computation Tree Logic (CTL), or more specialized versions like SOLOIST [Bianculli et al., 2013], MFOTL [Basin et al., 2008], CTL-FO$^+$ [Hallé et al., 2009]). However, these specification approaches require strong theoretical and mathematical background, which are rarely found among practitioners.

To partially mitigate this problem, researchers have proposed catalogues of *property specification patterns* [Autili et al., 2015, Bianculli et al., 2012, Gruhn and Laue, 2006, Konrad and Cheng, 2005, Dwyer et al., 1999], which collect generalized, proven solutions for expressing recurrent, common types of specifications. In some cases, catalogues include a restricted natural language grammar front-end to express the patterns, and a mapping of the semantics of (restricted) natural language constructs to temporal logic formalisms; this mapping can be automated with tools like PSPWizard [Lumpe et al., 2011].

From the MDE side of specification languages there are Object Constraint Language (OCL) [OMG, 2012] and the Unified Modeling Language (UML) profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [OMG, 2011b]. Although also based on mathematical foundations such as first-order logic and set theory, OCL includes many helper functions—to keep the constraints compact—and navigation expressions that reflect the structure of class diagrams (conceptual models)—to help with writing expressions that look more alike to program code. These features made OCL the de-facto constraints specification language in MDE practice and an international standard [OMG, 2012], which is supported by mature constraint checking technology, such as the constraint/query evaluator included in Eclipse OCL [Eclipse, 2015a]. However, OCL does not support natively the specification of temporal constraints in an intuitive fashion. To overcome this limitation, several temporal extensions of OCL have been proposed in the literature [Conrad and Turowski, 2001, Lavazza et al., 2003, Ziemann and Gogolla, 2003, Bill et al., 2014, Flake and Mueller, 2004, Soden and Eichler, 2009]; however, these extensions include temporal logic operators and thus intrinsically inherit the limitations of other specification approaches based on temporal logic. Other temporal extensions of OCL, such as [Küster-Filipe and Anderson, 2006, Flake and Müller, 2003, Robinson, 2008, Kanso and Taha, 2014], explicitly support property specification patterns. Nevertheless, these pattern-based temporal extensions of OCL have limited expressiveness. For example, based on our analysis of a case study in eGovernment systems, none of the current pattern-based temporal extensions of OCL could support a property like "If the physical information of the card requester is collected within three days after the second approval notification, the card will be produced and then issued to the requester", which contains a reference to a specific occurrence of an event ("after the second approval notification . . . ") as well as an explicit temporal distance from an event (". . . within three days. . . "). MARTE defines foundations for model-based descriptions of real-time and embedded systems. Though MARTE provides Value Specification Language (VSL) and clock handling facilities for specifying temporal and timing properties of a system, the specification language does not support property specification patterns as first-class objects.

As for the second step towards the correct enactment of business processes, the compliance of a business process with respect to its requirements can be checked with different verification techniques, such as model checking [Fu et al., 2004, Bianculli et al., 2007], run-time monitoring [Simmonds et al., 2009, Kallel et al., 2009, Baresi et al., 2009, Raimondi et al., 2008], and offline trace checking [Bersani et al., 2016]; in this thesis we focus on the latter. Offline trace checking, also called *trace validation* [Mrad et al., 2013] or *history checking* [Felder and Morzenti, 1994], is a procedure for evaluating requirements (usually specified in a temporal logic) over a log of recorded events produced by a system. Traces can be produced at run time by a proper monitoring/logging infrastructure, and made available at the end of a business process execution to perform offline trace checking. Offline trace checking complements verification activities performed before the deployment of a system, by allowing for the post-mortem analysis of actual behaviors emerged at run time and recorded on a log. These behaviors include the ones of the business process as well as those derived from the interaction of the business process with the various third-parties (e.g., other administrations, suppliers) involved in the execution of the process itself. Offline trace checking is thus also a way to check whether third-party providers fulfill their guarantees and to assess how they interact with the rest of the parties involved in the business process. The tool support in terms of trace checking of temporal logic, however, is limited and often based on prototypes that do not scale for industrial applications.

Regarding the capability of reporting, to business analysts and engineers, useful and clear information upon discovering a violation of the business process requirements, in this thesis we focus on how to provide such information after performing offline trace checking. In general, a violation reporting procedure can be decomposed into two steps: 1) collecting information about the violations (e.g., event(s) that triggered the violations) and 2) presentation to the users of this information (either textually or graphically). State-of-the art tools for offline trace checking provide limited violation information (e.g., either a simple boolean answer or the event upon which the violation was discovered). However, this information might not be enough, for example because the event upon which the violation was discovered is not necessarily the event responsible for the violation, or because a requirement could be violated in different ways (e.g., because of the order of or the time distance between events). Furthermore, the current tools provide only textual output, which is cumbersome to navigate when many violations are found in a trace.

1.2 Research Contribution

The goal of this thesis is to present a practical and scalable solution for the offline checking of the temporal requirements of a business process, which is expected to be advantageous in contexts where the following requirements hold:

R1) when analysts do not have adequate skills to make use of temporal logic, an *alternative* domain-specific language should be provided to facilitate the specification of business process requirements;

R2) to be viable in the long term, any solution shall rely on standard and stable MDE technology for checking the compliance of a business process to the application requirements;

R3) any solution shall be scalable, such that a trace with millions of events could be processed within seconds;

R4) if a trace is found violating some of its requirements, any solution should provide useful and clear information to understand how the requirements are violated.

This goal is motivated by specific requirements from our partner in the context of business process models for eGovernment systems. Nevertheless, we believe, based on experience, that these requirements can be generalized to other contexts in which analysts cannot handle the mathematical background required by temporal logic and solutions have to be engineered by using MDE technologies already in place in the targeted development environment.

To achieve the above objectives, the thesis will make the following contributions:

i) the *TemPsy* (Temporal Properties made easy) language, a pattern-based domain-specific language for the specification of temporal properties;

ii) a model-driven trace checking procedure, which relies on a mapping — optimized to minimize the execution time — of temporal requirements written in *TemPsy* into OCL constraints on a conceptual model of execution traces;

iii) a model-driven approach to violation information collection, which relies on the evaluation of OCL queries on an instance of the trace model;

iv) three publicly-available tools: 1) TEMPSY-CHECK and 2) TEMPSY-REPORT, implementing, respectively, the trace checking and violation information collection procedures; 3) an interactive visualization tool for navigating and analyzing the violation information collected by TEMPSY-REPORT;

v) an evaluation of the scalability of TEMPSY-CHECK and TEMPSY-REPORT, when applied to the verification of real properties derived from a case study of our public service partner.

TemPsy is a domain-specific language for the specification of temporal properties based on the catalogue of property specification patterns defined by Dwyer et al. [Dwyer et al., 1999] (with some extensions). To fulfill requirement R1 above, based on the discussions with our partner business analysts, we decided that the language should have the following features: be as close to natural language as possible, make no use of mathematical constructs, and support the commonly understood concepts used in the specification of requirements in the domain of business process modeling. Regarding the latter feature, we analyzed the requirements specifications of our industrial case study, to understand the type of specifications written (in natural language) by business analysts and to characterize them in terms of the property specification patterns in [Dwyer et al., 1999] (with some extensions). The relevant concepts and patterns found through this analysis drove the design of *TemPsy*, which resulted in a language sporting a syntax close to natural language, with all the constructs required to express the property specification patterns found in our case study, and a precise semantics expressed in terms of linear temporal traces. By design, *TemPsy* does not aim at being as expressive as a full-fledged temporal logic. Instead, its goal is to make as easy as possible the specification of the temporal requirements of business processes, by supporting—in an intuitive way—only the constructs needed to express temporal requirements commonly found in business process applications. *TemPsy* has received positive feedback from our partner, which has deemed it as suitable communication mechanism to express the requirements specifications of business processes. Our partner has integrated *TemPsy* into the SoftwareAG ARIS modeling tool [Software AG, 2014], and its analysts have started using it to annotate business process models with *TemPsy* specifications. In this thesis, we show the application of *TemPsy* for the specification of an excerpt of a business process extracted from the case study developed with our partner.

Both the procedures for offline trace checking and violation information collection fulfill requirement R2 above since they follow a model-driven approach, based on industry-strength OCL tools. More specifically, the two procedures rely on a generic conceptual model of system execution traces: the offline trace checking procedure leverages a mapping of *TemPsy* properties into OCL *constraints* defined over this trace model, while the violation information collection procedure uses various OCL *queries* defined on the same trace model to analyze violations. The mappings are supported by two different sets of auxiliary OCL functions and optimized based on the structure of the targeted *TemPsy* property, in order to achieve better performance. In the thesis, we show how the problems of checking a *TemPsy* property over a trace and collecting *TemPsy* violation information from a trace are respectively reduced to the evaluation of semantically-equivalent OCL constraints and queries on the corresponding instance of the trace model.

To show the fulfillment of requirement R3 above, we have conducted an extensive evaluation of the scalability of the two model-driven procedures. For the offline trace checking procedure, we assessed the relationship among the checking time, the structural properties of a trace (e.g., length, distribution of events), and the type of property to check. We evaluated the scalability of our TEMPSY-CHECK tool on 38 properties extracted from our case study, on traces with length ranging from 100K to 1M. We have also compared the performance of TEMPSY-CHECK with a state-of-the-art alternative technology, selected from the participants to the "offline monitoring" track of the first international Competition on Software for Runtime Verification [Bartocci et al., 2014] (CSRV 2014). The experimental results show that TEMPSY-CHECK can analyze very large traces (with one million events) in about two seconds and that it scales linearly with respect to the length of the trace to check. The results also show that TEMPSY-CHECK compares favorably with the state of the art. For the violation information collection procedure, we assessed the relationship among the execution time, the number of violations, the violation type, the structural properties of a trace (e.g., length, distribution of violations and events), and the type of property. The results show that TEMPSY-REPORT is able to collect violation information from large traces (with one million events) in less than ten seconds. The TEMPSY-REPORT tool scales linearly with respect to the length of the trace and keeps approximately constant performance as the number of violations increases.

TEMPSY-CHECK and TEMPSY-REPORT have been implemented and are publicly available with the artifacts used in the evaluation at `http://weidou.github.io/TemPsy-Check` and `http:`

`//weidou.github.io/TemPsy-Report` respectively.

To fulfill R4, we have developed an interactive visualization tool, which is publicly available at `http://weidou.github.io/TemPsy-Violation-Visualization`), to present the violation information collected by TEMPSY-REPORT in a graphical way, for a better understanding of the violations.

1.3 Dissemination

The research work we performed during the PhD program has lead to the following publications:

Published papers

- Dou, W., Bianculli, D., and Briand, L. (2014b). OCLR: a more expressive, pattern-based temporal extension of OCL. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 51–66. Springer

  This paper is the basis for Chapter 2. It presents an early version of the *TemPsy* language.

- Dou, W., Bianculli, D., and Briand, L. (2014c). Revisiting model-driven engineering for run-time verification of business processes. In *Proc. SAM 2014*, volume 8769 of *LNCS*, pages 190–197. Springer

  This paper describes our long-term vision and presents the research roadmap for adopting MDE techniques in the context of run-time verification of business processes. The roadmap guided our research for the development of the *TemPsy* language and of the offline trace checking procedure.

Unpublished report

- Dou, W., Bianculli, D., and Briand, L. (2014a). A model-based approach to offline trace checking of temporal properties with OCL. Technical Report TR-SnT-2014-5, SnT Centre - University of Luxembourg

  This report is the basis for Chapter 3. It presents TEMPSY-CHECK and an evaluation of its scalability.

1.4 Organization of the Dissertation

The rest of the thesis is structured as follows. Chapter 2 describes the syntax, informal and formal semantics of the *TemPsy* language. An application of *TemPsy* is shown in a case study in the domain of eGovernment. In Chapter 3, we present TEMPSY-CHECK, the model-driven procedure for offline trace checking of *TemPsy* properties and the evaluation of the scalability of the TEMPSY-CHECK tool. Chapter 4 presents TEMPSY-REPORT, the model-driven procedure for collecting violation information, and the visualization tool designed for understanding the violations collected by TEMPSY-REPORT; the chapter also reports on the evaluation of the scalability of the TEMPSY-REPORT tool.

Chapter 5 discusses related work. Chapter 6 concludes the thesis, providing directions for future work.

Chapter 2

The *TemPsy* Language

As discussed in Chapter 1, the ultimate goal of this thesis is to present a practical and scalable solution for the offline checking of the temporal requirements of a system with respect to a business process model, motivated by real and specific requirements in eGovernment systems. In this section we present the first step to achieve this goal, which is represented by the definition of the *TemPsy* language for the specification of temporal requirements of business processes, which will then be checked on an execution trace using the procedure described in Chapter 3.

2.1 Background: Property Specification Patterns

A *pattern* represents a reusable solution for a recurrent problem [Alexander et al., 1977]. Though initially proposed in the context of architecture [Alexander et al., 1977], this concept has been adopted also in different sub-domains of software engineering, including software design, with design patterns [Gamma et al., 1995], and formal verification, with *property specification patterns* [Autili et al., 2015].

Property specification patterns have been initially proposed by Dwyer et al. [Dwyer et al., 1999] in the late ‘90s in the context of formal verification, as a means to express recurring properties in a generalized form, which could be formalized in different specification languages, such as temporal logic. The goal of property specification patterns is to facilitate the writing of formal specifications, which can then be used with formal verification tools (e.g., model checkers).

Several catalogues of property specification patterns have been proposed in the literature [Grunske, 2008, Bianculli et al., 2012, Gruhn and Laue, 2006, Konrad and Cheng, 2005, Dwyer et al., 1999]. In the rest of this section we provide a brief overview of the catalogue of property specification patterns by Dwyer et al. [Dwyer et al., 1999], which have been included (with some extensions) in the definition of the *TemPsy* language.

This catalogue[1] contains nine parametrizable *patterns*, representing high-level abstractions of formal specifications, and five *scopes*, which indicate the portions of a system execution in which a certain pattern should hold. In the following, we use the letters W, X, Y, and Z, to denote events or states of a system execution The five scopes, depicted in Figure 2.1, are:

[1]A detailed description is available at `http://patterns.projects.cis.ksu.edu`.

Globally. This scope corresponds to the entire system execution (i.e., the entire trace).

Before. It identifies a portion of a trace up to a certain boundary.

After. It identifies a portion of a trace starting from a certain boundary.

Between-And. It identifies portion(s) of a trace delimited by two boundaries.

After-Until. This scope is similar to *Between-and*, with the difference that each identified segment extends to the right in case the event defined by the second boundary does not occur.

The nine patterns are:

Absence. It describes a portion of a system's execution that is free of certain events or states, as in "it is never the case that X holds".

Universality. It describes a portion of a system's execution that contains only states that have a desired property, as in "it is always the case that X holds".

Existence. It describes a portion of a system's execution that contains an instance of certain events or states, as in "X eventually holds".

Bounded existence. It describes a portion of a system's execution that contains at most a specified number of instances of a designated state transition or event, as in "it is always the case that event X occurs at most 2 times".

Precedence. It describes relationships between a pair of events (or states), where the occurrence of the first is a necessary pre-condition for an occurrence of the second, as in "it is always the case that if X holds, then Y previously held".

Response. It describes cause-effect relationships between a pair of events (or states), where an occurrence of the first must be followed by an occurrence of the second, as in "it is always the case that if X holds, then Y eventually holds".

Response chains. It is a generalization of the response pattern, as it describes relationships between *sequences* of individual states (or events), as in "it is always the case that if W holds, and is succeeded by X, then Z eventually holds after Y".

Precedence chains. It is a generalization of the precedence pattern, as it describes relationships between *sequences* of individual states (or events), as in "it is always the case that if X holds, then Y previously held and was preceded by X".

Constrained chain patterns. It describes a variant of response and precedence chain patterns that restricts user specified events from occurring between pairs of states (or events) in the chain sequences. This pattern has not been included in the definition of *TemPsy*.

Absence, Universality, Existence and Bounded Existence belong to the *Occurrence* category, while Precedence, Response, and Chains belong to the *Order* category.

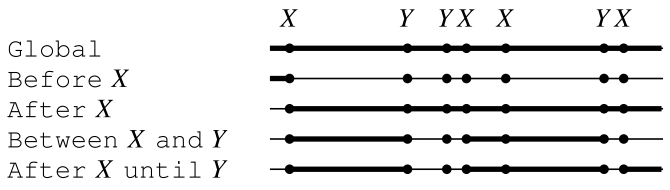


Figure 2.1: Scopes in the catalogue of property specification patterns in [Dwyer et al., 1999]

2.2 Definition of *TemPsy*

2.2.1 Eliciting the requirements of the language

The design of *TemPsy* has been driven by the analysis of the requirements of various applications developed as business processes by CTIE. We analyzed several applications and scrutinized the requirements specifications associated with all use cases and business process descriptions.

This analysis revealed that the vast majority of these requirements could be expressed as temporal properties, enriched with timing information. More specifically, we were able to recast most of specifications written in natural language using the system of property specification patterns of Dwyer et al. [Dwyer et al., 1999]. In some cases, we extended the original definitions proposed in [Dwyer et al., 1999] to match the specifications. For example, we extended the definitions of scopes to support references to a specific occurrence of an event (not only the first one as in [Dwyer et al., 1999]), as in the requirement "event *A* shall occur before the *second* occurrence of event *X*". Another variant of this type of scope boundary that we found is the one with requirements on the distance between events, such as "event *A* shall occur *five time units before the second* occurrence of event *X*". In other cases, the requirements specifications had to be expressed in terms of some real-time specification patterns [Konrad and Cheng, 2005, Gruhn and Laue, 2006], which quantitatively define distance among events and durations of events.

2.2.2 Design

The analysis of the requirements specifications mentioned above made us ponder over the design of the specification language for expressing them.

The intrinsic temporal nature of the requirements specifications we found, including also constraints on the distance between events, could have suggested to follow the direction of building on some (metric) temporal logic. However, this decision would have not allowed us to fulfill requirement R1 (see Chapter 1). One of the motivations behind this requirement is that specification languages based on temporal logic require a certain mathematical knowledge that is not common among practitioners.

Another design option would have been to consider the specification languages defined in the MDE community, namely temporal extensions of OCL, such as [Conrad and Turowski, 2001, Lavazza et al., 2003, Ziemann and Gogolla, 2003, Bill et al., 2014, Flake and Mueller, 2004, Küster-Filipe and Anderson, 2006, Soden and Eichler, 2009, Flake and Müller, 2003, Robinson, 2008, Kanso and Taha, 2014]. However, these temporal extensions either include temporal logic operators—thus intrinsically inheriting the limitations of other specification approaches based on temporal logic, and not fulfilling requirement R1—or are pattern-based but have limited expressiveness. For example, none of the pattern-based OCL temporal extensions can express a property like "If the physical information of the card requester is collected within three days after the second approval notification, the card will be produced and then issued to the requester", which contains a reference to a specific occurrence of an event in a scope boundary, as well as an explicit temporal distance from the scope boundary event.

Based on the discussions with business analysts, and keeping in mind the goal of fulfilling requirement R1 above, we decided that *TemPsy* should have the following features: be as close to natural language as possible, make no use of mathematical constructs, and support the commonly-understood concepts (i.e., property specification patterns) used in the specification of requirements in the domain of business process modeling.

We designed *TemPsy* as a language sporting a syntax close to natural language, with all the constructs required to express the property specification patterns found in the business process applications developed by our partner, and a precise semantics expressed in terms of linear temporal traces. *TemPsy* supports all the patterns and scopes defined in [Dwyer et al., 1999], with the following extensions:

- The possibility, in the definition of a scope boundary, to refer to a specific occurrence of an event, as in "before the second occurrence of event $X\dots$". In the original definition of the pattern systems, boundaries of scopes refer implicitly to the first occurrence of an event.
- The possibility to indicate a time distance with respect to a scope boundary, as in "at least two time units before the n-th occurrence of event $X\dots$".
- Support for expressing time distance between events occurrences in the precedence and response patterns as well as in their chain versions, for expressing properties such as "event B should occur in response to event A within 2 time units".
- Additional variants for the bounded existence and absence patterns.

2.2.3 Syntax

The syntax of *TemPsy* is shown in Figure 2.2: non-terminals are enclosed in angle brackets, terminals are enclosed in single quotes, optional elements are enclosed in brackets, the character '+' indicates one or more occurrences of an element, the character '*' indicates zero or more occurrences of an element.

A ⟨*TemPsyBlock*⟩ comprises a set of conjuncted ⟨*TemPsyExpression*⟩s. Each *TemPsy* expression starts with an optional '`temporal`' keyword plus an alphanumeric identifier, followed by a ⟨*Scope*⟩ and a ⟨*Pattern*⟩. The keywords indicating the five ⟨*Scope*⟩s identify univocally the corresponding scopes from [Dwyer et al., 1999] (see Section 2.1). As for the ⟨*Pattern*⟩s, '`always`' corresponds to universality, '`eventually`' to existence, '`never`' to absence, '`preceding`' to precedence and precedence chain, '`responding`' to response and response chain.

The definitions of ⟨*Scope*⟩s and ⟨*Pattern*⟩s refer to the concept of ⟨*Event*⟩. We assume that an ⟨*Event*⟩ is represented by an alphanumeric string, to match the event strings logged in the execution trace on which the properties specified in *TemPsy* are meant to be checked. ⟨*Scope*⟩s contain

```
⟨TemPsyBlock⟩         ::= ⟨TemPsyExpression⟩+
⟨TemPsyExpression⟩    ::= [‘temporal’ ⟨Id⟩ ‘:’]
                          ⟨Scope⟩ ⟨Pattern⟩
⟨Scope⟩               ::= ‘globally’
                        | ‘before’ ⟨Boundary1⟩
                        | ‘after’ ⟨Boundary1⟩
                        | ‘between’ ⟨Boundary2⟩
                          ‘and’ ⟨Boundary2⟩
                        | ‘after’ ⟨Boundary2⟩
                          ‘until’ ⟨Boundary2⟩
⟨Pattern⟩             ::= ‘always’ ⟨Event⟩
                        | ‘eventually’ ⟨RepeatableEventExp⟩
                        | ‘never’ [‘exactly’ ⟨Int⟩] ⟨Event⟩
                        | ⟨EventChainExp⟩ ‘preceding’
                          [⟨TimeDistanceExp⟩] ⟨EventChainExp⟩
                        | ⟨EventChainExp⟩ ‘responding’
                          [⟨TimeDistanceExp⟩] ⟨EventChainExp⟩
⟨Boundary1⟩           ::= [⟨Int⟩] ⟨Event⟩ [⟨TimeDistanceExp⟩]
⟨Boundary2⟩           ::= [⟨Int⟩] ⟨Event⟩ [‘at least’ ⟨Int⟩ ‘tu’]
⟨EventChainExp⟩       ::= ⟨Event⟩
                          (‘,’ [‘#’ ⟨TimeDistanceExp⟩] ⟨Event⟩)*
⟨TimeDistanceExp⟩     ::= ⟨ComparingOp⟩ ⟨Int⟩ ‘tu’
⟨RepeatableEventExp⟩ ::= [⟨ComparingOp⟩ ⟨Int⟩] ⟨Event⟩
⟨ComparingOp⟩         ::= ‘at least’ | ‘at most’ | ‘exactly’
⟨Event⟩               ::= ⟨Id⟩
⟨Id⟩                  ::= ⟨IdStartChar⟩ ⟨IdChar⟩*
                        | ⟨Id⟩ (⟨IdConnector⟩ ⟨Id⟩)*
⟨IdStartChar⟩         ::= [A-Z] | ‘_’ | [a-z]
⟨IdChar⟩              ::= ⟨IdStartChar⟩ | [0-9]
⟨IdConnector⟩         ::= ‘.’ | ‘::’
⟨Int⟩                 ::= [1-9] ([0-9])*
```

Figure 2.2: Syntax of *TemPsy*

boundaries (expressed with ⟨*Boundary1*⟩ or ⟨*Boundary2*⟩) that denote a specific occurrence of an event as a boundary, possibly with a time distance; notice that ⟨*Boundary2*⟩ represents a syntactic restriction of ⟨*Boundary1*⟩. Chains of events, used in precedence and response patterns, are defined as ⟨*EventChainExp*⟩, which denotes a comma-separated list of events, possibly with a time distance (⟨*TimeDistanceExp*⟩) between each pair of events (denoted with the ‘#‘ symbol). Time distances are expressed with an integer value, followed by the ‘`tu`’ keyword, which represents a generic time unit[2]

[2]The current version of *TemPsy* supports only one (generic) time unit. We refer the reader to [Furia et al., 2012, Section 9.6], for an approach to combine different time units within the same set of requirements.

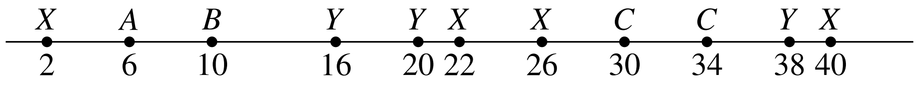


Figure 2.3: An event trace on which to evaluate the properties described in Section 2.2.4; events are above the line, timestamps below

(i.e., any denomination of time).

2.2.4 *TemPsy* at Work

We now present some examples of properties that can be expressed with *TemPsy*, in order to provide the reader with a high-level, intuitive understanding of the language. We consider the execution trace shown in Figure 2.3 and for each property[3] indicate whether it is violated or not by the trace. First, we define the properties in English:

p1) "Event C shall happen 8 time units after the second occurrence of event X." (satisfied)

p2) "Event A shall happen within 30 time units after the first occurrence of event X." (satisfied)

p3) "Event C shall eventually happen after at least 3 time units since the first occurrence of event X; and it shall happen before event Y if the latter happens." (violated because event C occurs after event Y)

p4) "After the second occurrence of event X, event C shall eventually happen exactly twice." (satisfied)

p5) "Event C shall happen at least once between every first occurrence of event X and the next event Y; the time interval between event X and the first occurrence of event C shall be at least 5 time units." (violated because event C does not occur between the first segment delimited by event X on the left and event Y on the right)

p6) "Event B shall happen at least 3 time units before the first occurrence of event Y." (satisfied)

p7) "Before the first occurrence of event Y, once event X occurs, event A shall happen followed by event B; the time interval between X and A shall be at least 3 time units." (satisfied)

The corresponding *TemPsy* expressions are shown below:

- `temporal p1:  after 2` *X* `exactly 8 tu eventually` *C*
- `temporal p2:  after` *X* `at most 30 tu eventually` *A*
- `temporal p3:  after 1` *X* `at least 3 tu until` *Y* `eventually` *C*
- `temporal p4:  after 2` *X* `eventually exactly 2` *C*

[3]These properties are given as an example and should be considered individually, rather than together as a set; they do not correspond to the specification of a real system.

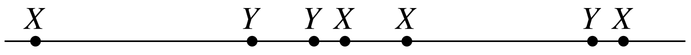


Figure 2.4: A sample trace for the description of scopes

- `temporal p5:` `between` *X* `at least 5 tu and` *Y* `eventually at least 1` *C*
- `temporal p6:` `before` *Y* `at least 3 tu eventually` *B*
- `temporal p7:` `before` *Y A*`,` *B* `responding at least 3 tu` *X*

2.3 Informal Semantics

In this section we present the informal semantics of the scopes and the patterns supported in *TemPsy* expressions; they correspond to non-terminals ⟨*Scope*⟩ and ⟨*Pattern*⟩, respectively. In the following, symbols A,B,C,D,X,Y,Z represent strings that can be derived from non-terminal ⟨*Event*⟩; '`m`', '`m1`', '`m2`', '`n`', '`n1`', and '`n2`' are integers derived from the non-terminal ⟨*Int*⟩; '`tu`' stands for "time unit(s)". The complete definition of the formal semantics of *TemPsy* can be found in Section 2.4.

2.3.1 Scopes

For the description of scopes, we refer to the trace of events depicted in Figure 2.4; to avoid cluttering, the figure does not show the events not used in the explanations. We use symbols X and Y as shorthands for events that can be derived from the non-terminal ⟨*Event*⟩.

Globally. This scope corresponds to the entire trace shown in Figure 2.4.

Before. The general template for this scope in *TemPsy* is "`before` [m] *X* [⟨*ComparingOp*⟩ `n tu`]"; it can be expanded in four forms: 1) "`before` *X*", 2) "`before` *X* ⟨*ComparingOp*⟩ `n tu`", 3) "`before m` *X*", 4) "`before m` *X* ⟨*ComparingOp*⟩ `n tu`". The first two forms are convenient shorthands for the third and fourth ones, respectively, with $\mathtt{m} = 1$. The form "`before m` *X*" selects the portion of the trace up to the *m*-th occurrence of event *X*; see, for example, the top row in Figure 2.5a, where the interval from the origin of the trace up to the third occurrence of *X* is highlighted with a thick line. The form "`before m` *X* ⟨*ComparingOp*⟩ `n tu`" has three variants, depending on the possible expansions of non-terminal ⟨*ComparingOp*⟩:

- "`before m` *X* `at least n tu`" identifies the scope from the origin of the trace up to *n* time units before the *m*-th occurrence of *X*;
- "`before m` *X* `at most n tu`" identifies the scope starting at *n* time units before the *m*-th occurrence of *X* and bounded to the right by the *m*-th occurrence of *X*;
- "`before m` *X* `exactly n tu`" pinpoints the time instant at *n* time units before the *m*-th occurrence of *X*.

Examples of the first two variants of scopes are shown with thick segments in the second and third rows of Figure 2.5a; for the last variant, see the last row of Figure 2.5a, where the time instant selected by the scope is enclosed with a circle. In all examples, we have `m`=3 and `n`=2.

After. It has a dual semantics with respect to the *before* scope. We provide an intuition of its semantics using Figure 2.5b.

Between-And. The general template for this scope in *TemPsy* is "`between` [`m1`] *X* [`at least n1 tu`] `and` [`m2`] *Y* [`at least n2 tu`]"; it can be expanded in four forms:

- "`between` m_1 *X* [`at least` n_1 `tu`] `and` m_2 *Y* [`at least` n_2 `tu`]";
- "`between` *X* [`at least` n_1 `tu`] `and` m_2 *Y* [`at least` n_2 `tu`]";
- "`between` m_1 *X* [`at least` n_1 `tu`] `and` *Y* [`at least` n_2 `tu`]";
- "`between` *X* [`at least` n_1 `tu`] `and` *Y* [`at least` n_2 `tu`]".

The first form is the most general: it selects the single segment of the trace delimited by the m_1-th occurrence of event *X* and the m_2-th occurrence of event *Y* happening after the m_1-th occurrence of *X*. The second and third forms are shorthands for the first one, with `m1`=1 and `m2`=1, respectively. The fourth form is the closest to the original definition in [Dwyer et al., 1999], since it selects all the segments in the trace delimited by the boundaries. In this regard, notice the difference with respect to the expression "`between 1` *X* `and 1` *Y*", which selects the segment delimited by the first occurrence of *X* and the first occurrence of *Y* after *X*. In all forms it is possible to use the expression `at least n tu` when defining boundaries, with the same meaning described for the scope *before*. Four examples of the *Between-and* scope are shown in Figure 2.5c.

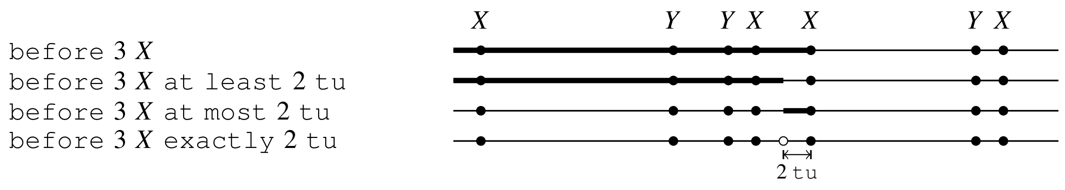
After-Until. This scope is similar to *Between-and*, with the difference that each identified segment extends to the right in case the event defined by the second boundary does not occur; this peculiarity can be noticed in the first two rows of Figure 2.5d (also by comparing them with the corresponding ones in Figure 2.5c), as well as in the last row.

Note that all scopes are open on the bounds delimited by the boundary events themselves, i.e., in general[4], the *before* scope is closed on the left bound and open on the right bound; the *after* scope is open on the left bound, and closed on the right bound; the *between-and* scope is open on both bounds; the *after-until* scope is open on both bounds when the right boundary event occurs, or is open on the left and closed on the right when the right boundary event does not occur.

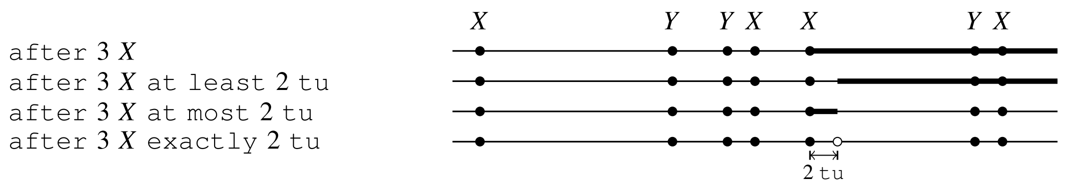
2.3.2 Patterns

TemPsy supports eight of the nine patterns defined in [Dwyer et al., 1999]. Their semantics has been already briefly explained in Section 2.1; below we only highlight the semantics for the patterns that have been extended upon inclusion in *TemPsy*.

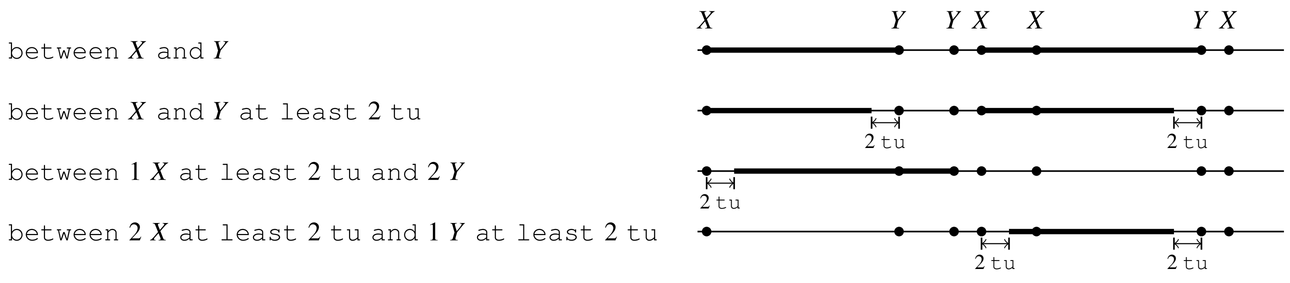
[4]The scopes that contain constraints on time distance from the boundary events (with "`at least`" and "`exactly`") are closed on the bounds



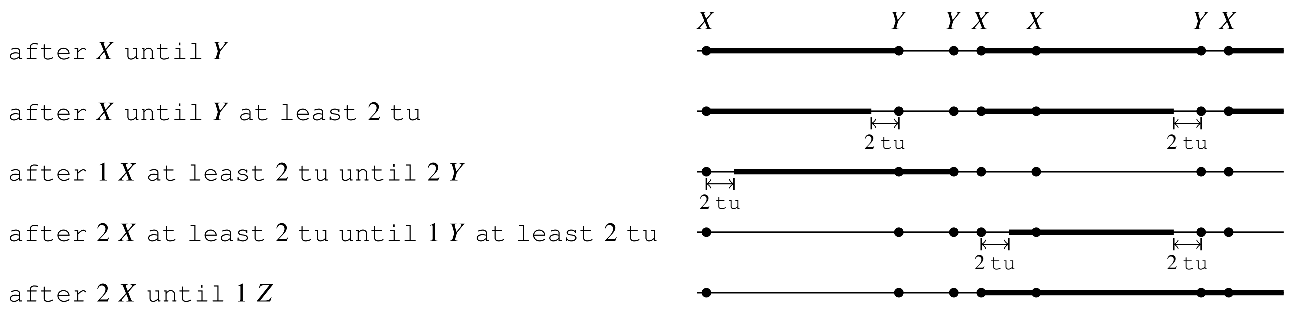
(a) Scope: *before*



(b) Scope: *after*


X Y Y X X Y X
between X and Y
between X and Y at least 2 tu
2 tu
2 tu
between 1 X at least 2 tu and 2 Y
2 tu
between 2 X at least 2 tu and 1 Y at least 2 tu
2 tu
2 tu

(c) Scope: *between-and*


X Y Y X X Y X
after X until Y
after X until Y at least 2 tu
2 tu
2 tu
after 1 X at least 2 tu until 2 Y
2 tu
after 2 X at least 2 tu until 1 Y at least 2 tu
2 tu
2 tu
after 2 X until 1 Z

(d) Scope: *after-until*

Figure 2.5: Examples of *TemPsy* scopes

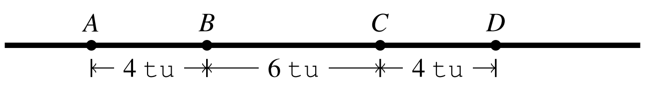


Figure 2.6: Example trace for illustrating the precedence and response patterns

Existence. This pattern comes in four forms:

- "`eventually` *A*" indicates that event *A* will eventually happen at least once;
- "`eventually at least m` *A*" indicates that event *A* will eventually happen at least *m* times;
- "`eventually at most m` *A*" indicates that event *A* will eventually happen at most *m* times;.

- “`eventually exactly m` *A*” indicates that event *A* will eventually happen exactly *m* times.

The last three forms are variants of the *bounded existence* pattern, a subclass [Autili et al., 2015] of the *existence* one.

Absence. In addition to stating that a certain event *never* occurs in the given scope, *TemPsy* makes also possible to specify that a specific number of occurrences of the same event should not happen, as in “`never exactly 2` *X*”, which indicates that *X* should never occur exactly twice.

Precedence. This pattern (also available in the variant called *precedence chain*) indicates the precondition relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the second event (respectively, block) depends on the occurrence of the first event (respectively, block). Based on this original definition, we added support for timing information to enable expressing the time distance between two adjacent events. The semantics can be explained using the following example and the event trace in Figure 2.6; the expression “*A* `preceding at most 10 tu` *B*, `#at least 5 tu` *C*” indicates that the event *A* is the precondition of the block “*B* followed by *C*”, that the time distance between *A* and *B* should be at most 10 time units, and the time distance (expressed using the `#` symbol) between events *B* and *C* should be at least 5 time units. Here, *A* (left-hand side of ‘`preceding`’) represents the first block of the chain, while the expression “*B*, `#at least 5 tu` *C*” represents the second block (right-hand side of ‘`preceding`’).

Response. This pattern (also available in the variant called *response chain*) specifies the cause-effect relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the first event (respectively, first block) leads to the occurrence of the second event (respectively, second block). Similarly to the previous pattern, we added support for timing information to enable expressing the time distance between two adjacent events. The semantics can be explained using the following example and the event trace in Figure 2.6; the expression “*C*, *D* `responding at most 10 tu` *A*, `#at least 5 tu` *B*” specifies that two successive events *A* and *B* stimulate the sequential occurrence of *C* and *D*, the time interval between *A* and *B* should be at least 5 time units, and the time interval between *B* (second element of the first block) and *C* (first element of the second block) should be at most 10 time units. This property is violated by the example in Figure 2.6, because the time distance between *A* and *B* is only 4 time units.

2.4 Formal Semantics

This section presents the formal semantics of *TemPsy*, using the concept of temporal linear traces.

2.4.1 Events and Trace

Event. An atomic event e is an element of the set Σ, which contains all the symbolic strings corresponding to operations recorded in a trace or log.

Trace. An n-length *trace* λ is a finite sequence of atomic events $(e_0, \dots, e_{n-1})$, where e_0 is its starting event and n is the length. The universal set of all the sub-traces of λ is denoted as Λ.

We assume that each event in a trace is timestamped and that there is a function $\tau : \mathbb{N} \rightarrow \mathbb{N}$, which returns the timestamp $\tau(i)$ at which the event in position i of the trace occurred. The timestamp is a natural number and represents the absolute value of time with respect to the time unit defined for the system. Given a trace λ we assume that the sequence of timestamps $\tau(0), \tau(1), \ldots, \tau(n-1)$ is strictly monotonic, i.e., $\tau(i) < \tau(i+1)$ for all i, with $0 \leq i \leq n-2$.

We now introduce some notations used in the rest of the section. Given an n-length trace λ,

- $\lambda(i)$ denotes the atomic event at position i in the trace, with $0 \leq i \leq n-1$;
- $td(i, j)$ denotes the time distance between $\lambda(i)$ and $\lambda(j)$ and is defined as $td(i, j) \equiv \tau(j) - \tau(i)$, with $0 \leq i \leq j \leq n-1$;
- $\lambda(i : j)$ denotes the sub-trace (also referred to as trace segment) of λ from $\lambda(i)$ to $\lambda(j)$ including both bounds, with $0 \leq i \leq j \leq n-1$.
- $\#(\lambda, i, j, e)$ denotes the number of occurrences of event e in the sub-trace $\lambda(i : j)$ of λ.

2.4.2 Temporal expressions

In the following definitions, let e, e_1, e_2 be atomic events; n be the length of a trace; b, d be positive natural numbers denoting time distances; a, c denote the specific occurrence of a scope boundary event and range over $\{0, \ldots, n-1\}$ if defined or be equal to $\{\perp\}$ if undefined; $\alpha, \alpha', \beta', \gamma, \theta, \theta', \eta, \eta'$ be auxiliary variables ranging over $\{0, \ldots, n-1\}$.

Scope. Let s be a scope defined by the non-terminal $\langle Scope \rangle$ in the grammar in Figure 2.2. The semantics of s is to derive a set of sub-traces from an n-length trace $\lambda \in \Lambda$, which is defined by the function $\phi_{[s]} : \Lambda \rightarrow 2^{\Lambda}$ as follows:

globally: $\phi_{[\texttt{globally}]}(\lambda) = \{\lambda\}$
before:

- $\phi_{[\texttt{before } a\ e]}(\lambda) = \Big\{\lambda(0 : \theta - 1) \mid \theta \geq 1, \lambda(\theta) = e, \#(\lambda, 0, \theta, e) = m\Big\}$
- $\phi_{[\texttt{before } a\ e \texttt{ at least } b \texttt{ tu}]}(\lambda) = \Big\{\lambda(0 : \theta') \mid \lambda(\theta) = e, \theta' = \max(\{\gamma \mid td(\gamma, \theta) \geq b\}), \#(\lambda, 0, \theta, e) = m\Big\}$
- $\phi_{[\texttt{before } a\ e \texttt{ at most } b \texttt{ tu}]}(\lambda) = \Big\{\lambda(\theta' : \theta - 1) \mid \lambda(\theta) = e, \theta' = \max(\{\gamma \mid td(\gamma, \theta) \geq b\}), \#(\lambda, 0, \theta, e) = m\Big\}$
- $\phi_{[\texttt{before } a\ e \texttt{ exactly } b \texttt{ tu}]}(\lambda) = \Big\{\lambda(\theta' : \theta') \mid \lambda(\theta) = e, \theta' = \max(\{\gamma \mid td(\gamma, \theta) \geq b\}), \#(\lambda, 0, \theta, e) = m\Big\}$

where $m = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$

after:

- $\phi_{[\texttt{after}\ a\ e]}(\lambda) = \left\{\lambda(\theta+1:n-1) \mid \theta \leq n-2, \lambda(\theta) = e, \#(\lambda,0,\theta,e) = m\right\}$
- $\phi_{[\texttt{after}\ a\ e\ \texttt{at least}\ b\ \texttt{tu}]}(\lambda) = \left\{\lambda(\theta':n-1) \mid \lambda(\theta) = e, \theta' = \min(\{\gamma \mid td(\theta,\gamma) \geq b\}), \#(\lambda,0,\theta,e) = m\right\}$
- $\phi_{[\texttt{after}\ a\ e\ \texttt{at most}\ b\ \texttt{tu}]}(\lambda) = \left\{\lambda(\theta+1:\theta') \mid \lambda(\theta) = e, \theta' = \min(\{\gamma \mid td(\theta,\gamma) \geq b\}), \#(\lambda,0,\theta,e) = m\right\}$
- $\phi_{[\texttt{after}\ a\ e\ \texttt{exactly}\ b\ \texttt{tu}]}(\lambda) = \left\{\lambda(\theta':\theta') \mid \lambda(\theta) = e, \theta' = \min(\{\gamma \mid td(\theta,\gamma) \geq b\}), \#(\lambda,0,\theta,e) = m\right\}$

where $m = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$

between-and:

- $\phi_{[\texttt{between}\ e_1\ \texttt{and}\ e_2]}(\lambda) = \left\{\lambda(\alpha_k+1:\beta_k-1) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1\right\}$
- $\phi_{[\texttt{between}\ e_1\ \texttt{and}\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \left\{\lambda(\alpha_k+1:\beta'_k) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \beta'_k = \max(\{\gamma \mid td(\gamma,\beta_k) \geq d\})\right\}$
- $\phi_{[\texttt{between}\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ e_2]}(\lambda) = \left\{\lambda(\alpha'_k:\beta_k-1) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \alpha'_k = \min(\{\gamma \mid td(\alpha_k,\gamma) \geq b\})\right\}$
- $\phi_{[\texttt{between}\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \left\{\lambda(\alpha'_k:\beta'_k) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \alpha'_k = \min(\{\gamma \mid td(\alpha_k,\gamma) \geq b\}), \beta'_k = \max(\{\gamma \mid td(\gamma,\beta_k) \geq d\}\right\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{and}\ c\ e_2]}(\lambda) = \left\{\lambda(\alpha+1:\beta-1) \mid \lambda(\alpha) = e_1, \#(\lambda,0,\alpha,e_1) = x, \lambda(\beta) = e_2, \#(\lambda,\alpha+1,\beta,e_2) = y\right\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{and}\ c\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \left\{\lambda(\alpha+1:\beta') \mid \lambda(\alpha) = e_1, \#(\lambda,0,\alpha,e_1) = x, \lambda(\beta) = e_2, \#(\lambda,\alpha+1,\beta,e_2) = y, \beta' = \max(\{\gamma \mid td(\gamma,\beta) \geq d\}\right\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ c\ e_2]}(\lambda) = \left\{\lambda(\alpha':\beta-1) \mid \lambda(\alpha) = e_1, \#(\lambda,0,\alpha,e_1) = x, \lambda(\beta) = e_2, \#(\lambda,\alpha+1,\beta,e_2) = y, \alpha' = \min(\{\gamma \mid td(\alpha,\gamma) \geq b\}\right\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ c\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \left\{\lambda(\alpha':\beta') \mid \lambda(\alpha) = e_1, \#(\lambda,0,\alpha,e_1) = x, \lambda(\beta) = e_2, \#(\lambda,\alpha+1,\beta,e_2) = y, \alpha' = \min(\{\gamma \mid td(\alpha,\gamma) \geq b\}, \beta' = \max(\{\gamma \mid td(\gamma,\beta) \geq d\}\right\}$

where $x = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$ and $y = \begin{cases} 1, & \text{if } c = \perp \\ c, & \text{else}\ . \end{cases}$

after-until:

- $\phi_{[\texttt{after } e_1 \texttt{ until } e_2]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ and } e_2]}(\lambda) \cup \left\{\lambda(\eta+1 : n-1) \mid \eta = \min(\{\gamma \mid \gamma \leq n-2, \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\right\}$
- $\phi_{[\texttt{after } e_1 \texttt{ until } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ and } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \left\{\lambda(\eta+1 : n-1) \mid \eta = \min(\{\gamma \mid \gamma \leq n-2, \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\right\}$
- $\phi_{[\texttt{after } e_1 \texttt{ at least } b \texttt{ tu until } e_2]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ at least } b \texttt{ tu and } e_2]}(\lambda) \cup \left\{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta,\gamma) \geq b\}), \eta = \min(\{\gamma \mid \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\right\}$
- $\phi_{[\texttt{after } e_1 \texttt{ at least } b \texttt{ tu until } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) =$ $\phi_{[\texttt{between } e_1 \texttt{ at least } b \texttt{ tu and } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \left\{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta, \gamma) \geq b\}), \eta = \min(\{\gamma \mid \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\right\}$
- $\phi_{[\texttt{after } a\, e_1 \texttt{ until } c\, e_2]}(\lambda) = \phi_{[\texttt{between } a\, e_1 \texttt{ and } c\, e_2]}(\lambda) \cup \left\{\lambda(\eta+1 : n-1) \mid \eta \leq n-2, \lambda(\eta) = e_1, \#(\lambda,0,\eta,e_1) = x, \#(\lambda,\eta+1,n-1,e_2) < y\right\}$
- $\phi_{[\texttt{after } a\, e_1 \texttt{ until } c\, e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \phi_{[\texttt{between } a\, e_1 \texttt{ and } c\, e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \left\{\lambda(\eta+1 : n-1) \mid \eta \leq n-2, \lambda(\eta) = e_1, \#(\lambda,0,\eta,e_1) = x, \#(\lambda,\eta+1,n-1,e_2) < y\right\}$
- $\phi_{[\texttt{after } a\, e_1 \texttt{ at least } b \texttt{ tu until } c\, e_2]}(\lambda) = \phi_{[\texttt{between } a\, e_1 \texttt{ at least } b \texttt{ tu and } c\, e_2]}(\lambda) \cup \left\{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta,\gamma) \geq b\}), \lambda(\eta) = e_1, \#(\lambda,0,\eta,e_1) = x, \#(\lambda,\eta+1, n-1,e_2) < y\right\}$
- $\phi_{[\texttt{after } a\, e_1 \texttt{ at least } b \texttt{ tu until } c\, e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) =$ $\phi_{[\texttt{between } a\, e_1 \texttt{ at least } b \texttt{ tu and } c\, e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \left\{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta,\gamma) \geq b\}), \lambda(\eta) = e_1, \#(\lambda,0,\eta,e_1) = x, \#(\lambda,\eta+1,n-1,e_2) < y\right\}$

where $x = \begin{cases} 1, & \text{if } a = \bot \\ a, & \text{else} \end{cases}$ and $y = \begin{cases} 1, & \text{if } c = \bot \\ c, & \text{else} \end{cases}$.

EventChain. An *EventChain* is a chain of *Events* occurring in sequence, with an optional quantification of the time distance between each pair of adjacent elements. An m-length EventChain ($m \geq 1$) is denoted as $e_0, t_1, e_1, \ldots, t_{m-1}, e_{m-1}$. The symbol t_i (with $1 \leq i \leq m-1$) represents the time distance between e_{i-1} and e_i (if defined) and has the form $t_i = \# \bowtie_i \delta_i$ `tu` with $\delta_i \in \mathbb{N}^+$ and $\bowtie_i \in$ {`at least, at most, exactly`}; when t_i is undefined we use the notation $t_i = \bot$. Function $len(EC)$ returns the length m of an m-length EventChain EC.

Event and EventChain matching function. Let λ be an n-length trace, EC be an m-length EventChain ($1 \leq m \leq n$). The matching function *match* returns true if there is an occurrence of an event (or of an EventChain) in a certain position of the trace. For a 1-length EventChain $EC = e$, i.e., a single event, we have $match(\lambda, EC, i) = true$, with $i, 0 \leq i \leq n-1$, if $\lambda(i) = e$. For an event chain $EC = e_0, t_1, e_1, \ldots, t_{m-1}, e_{m-1}$, we have $match(\lambda, EC, i) = true$, with $i, 0 \leq i \leq n-m$, if there exist $i_0, i_1, \ldots, i_{m-1} \in \{0, \ldots, n-1\}$, such that $i_0 = i$, $i_k = i_{k-1} + 1, 1 \leq k \leq m-1$, $\lambda(i_0) = e_0, \lambda(i_1) =$

$e_1,\ldots,\lambda(i_{m-1}) = e_{m-1}$ and for all $j, 1 \leq j \leq m-1$, such that $t_j \neq \bot$, we have:

$$\begin{cases} td(i_{j-1},i_j) \geq \delta_j \text{ if } \bowtie_j = \texttt{at least}; \\ td(i_{j-1},i_j) \leq \delta_j \text{ if } \bowtie_j = \texttt{at most}; \\ td(i_{j-1},i_j) = \delta_j \text{ if } \bowtie_j = \texttt{exactly}. \end{cases}$$

For an events chain $EC = e_0,t_1,e_1,\ldots,t_{m-1},e_{m-1}$ we also define two auxiliary functions $\mathit{first}(\lambda,EC,i)$ and $\mathit{last}(\lambda,EC,i)$, which return, respectively, the timestamp of the first and the last event of EC when the chain is matched in position i of the trace λ.

In the following definitions, let EC_1, EC_2 be event chains.

Pattern. Let p be a pattern defined by the non-terminal $\langle \mathit{Pattern} \rangle$ in the grammar in Figure 2.2. The semantics of p is to determine whether the pattern holds on an n-length trace $\lambda \in \Lambda$, which is defined by the function $\psi_{[p]} : \Lambda \to \{\mathit{true},\mathit{false}\}$ as follows:

universality: $\psi_{[\texttt{always}\ e]}(\lambda) \Leftrightarrow \forall i, 0 \leq i \leq n-1, \lambda(i) = e$
absence:

- $\psi_{[\texttt{never}\ e]}(\lambda) \Leftrightarrow \forall i, 0 \leq i \leq n-1, \lambda(i) \neq e$
- $\psi_{[\texttt{never exactly}\ m\ e]}(\lambda) \Leftrightarrow \#(\lambda,0,n-1,e) \neq m$

existence:

- $\psi_{[\texttt{eventually}\ e]}(\lambda) \Leftrightarrow \exists i, 0 \leq i \leq n-1, \lambda(i) = e$
- $\psi_{[\texttt{eventually} \bowtie m\ e]}(\lambda) \Leftrightarrow \#(\lambda,0,n-1,e) \triangle\, m$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \texttt{at least}; \\ \leq, & \text{if } \bowtie = \texttt{at most}; \\ =, & \text{if } \bowtie = \texttt{exactly}. \end{cases}$

precedence:

- $\psi_{[EC_1\ \texttt{preceding}\ EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, \mathit{match}(\lambda,EC_2,i) \Rightarrow$
  $\exists j, 0 \leq j \leq i - \mathit{len}(EC_1), \mathit{match}(\lambda,EC_1,j)$
- $\psi_{[EC_1\ \texttt{preceding} \bowtie b\ \texttt{tu}\ EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, \mathit{match}(\lambda,EC_2,i) \Rightarrow$
  $\exists j, 0 \leq j \leq i - \mathit{len}(EC_1), \mathit{match}(\lambda,EC_1,j)$ and $(\mathit{first}(\lambda,EC_2,i) - \mathit{last}(\lambda,EC_1,j)) \triangle\, b$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \texttt{at least}; \\ \leq, & \text{if } \bowtie = \texttt{at most}; \\ =, & \text{if } \bowtie = \texttt{exactly}. \end{cases}$

response:

- $\psi_{[EC_1\ \texttt{responding}\ EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, \mathit{match}(\lambda,EC_2,i) \Rightarrow \exists j, i + \mathit{len}(EC_2) \leq j \leq n-1, \mathit{match}(\lambda,EC_1,j)$
- $\psi_{[EC_1\ \texttt{responding} \bowtie b\ \texttt{tu}\ EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, \mathit{match}(\lambda,EC_2,i) \Rightarrow \exists j, i + \mathit{len}(EC_2) \leq j \leq n-1, \mathit{match}(\lambda,EC_1,j)$ and

$$(\mathit{first}(\lambda, EC_1, i) - \mathit{last}(\lambda, EC_2, j)) \triangle\, b$$
$$\text{where } \triangle = \begin{cases} \geq, & \text{if } \bowtie = \texttt{at least}; \\ \leq, & \text{if } \bowtie = \texttt{at most}; \\ =, & \text{if } \bowtie = \texttt{exactly}. \end{cases}$$

Temporal Expression. The semantics over a trace λ of a temporal expression derived from the non-terminal $\langle \mathit{TemPsyExpression} \rangle$ containing a scope s and a pattern p, represented as a pair $\langle s, p \rangle$, is defined as: $\lambda \models \langle s, p \rangle \Leftrightarrow \forall \lambda' \in \phi_{[s]}(\lambda),\ \psi_{[p]}(\lambda')$.

2.5 Expressivity

As discussed earlier, the main goal of *TemPsy* is to make as easy as possible the specification of the temporal requirements of business processes, by supporting—in an intuitive way—only the constructs needed to express temporal requirements commonly found in business process applications. Hence, by design, *TemPsy* does not aim at being as expressive as a full-fledged temporal logic.

More precisely, *TemPsy* can specify *only* the expressions resulting from the combination of one of the five supported scopes (and their variants) with one of the eight supported patterns (and their variants). For each of these expressions, it is possible to write a formula with the same meaning in a full-fledged temporal logic like MTL [Koymans, 1990] (see, for example, the syntax-directed translation of property specification patterns, targeting MTL, proposed in [Autili et al., 2015]). On the other hand, all the MTL formulae that do not correspond to one of the ⟨scope, pattern⟩ combinations *cannot* be expressed in *TemPsy*.

In our context, this limitation turns out to be more theoretical than practical, since we were able to express in *TemPsy* all the requirements of the business processes of our case study. Nevertheless, as part of future work, we plan to assess the expressivity of *TemPsy* by applying it for the specification of business processes in other application domains.

2.6 Applying *TemPsy*

In this section we report on the application of *TemPsy* for the specification of a business process extracted from the case study developed with our partner. After illustrating the conceptual and behavioral models of some fragments of the business process application, we present some requirements specifications associated with these business process fragments and show how these specifications can be expressed in *TemPsy*. We also discuss the adoption and use of *TemPsy* by our partner.

Notice that the case study description has been sanitized, for the purpose of not disclosing confidential information, and simplified, to obtain a model at the minimum level of detail required to illustrate and express the requirement specifications.

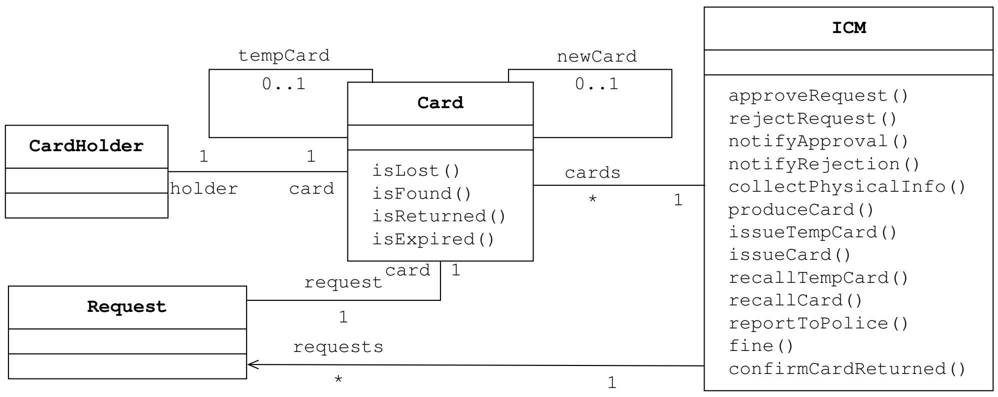
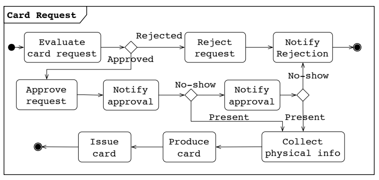
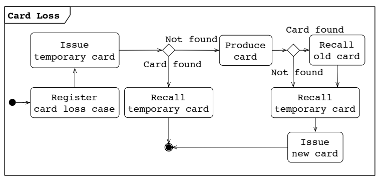


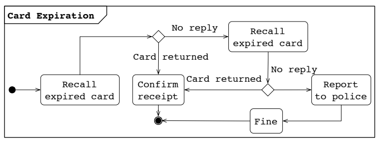
Figure 2.7: Conceptual model of the ICM business process



(a) *card request* fragment



(b) *card loss* fragment



(c) *card expiration* fragment

Figure 2.8: Activity diagrams of three fragments of the ICM business process

2.6.1 Business process models

We consider the *Identity Card Management (ICM)* business process, which is in charge of issuing and managing the ID cards of the diplomatic personnel of the country. Its conceptual model is shown in Figure 2.7, while three activity diagrams corresponding to process fragments are sketched in Figure 2.8.

The conceptual model includes the `ICM` class, which manages `Card`s and `Requests` (for new cards). The `ICM` class has methods that deal with approval/rejection of card requests, card production and issuance, and card loss/expiration. Class `Card` has methods to query about the state of the card, which can be lost, found, expired, or returned (to the administration).

The activity diagram in Figure 2.8a shows the business process fragment for processing a card request. Once a request for a card is submitted to the *ICM* system, it is evaluated and then either approved or rejected. Afterwards, a notification letter of approval or rejection is sent to the requester. Upon approval, the requester is asked to provide her physical information (e.g., hair and eye color, height) to the *ICM* system. In case this information is not provided, a second notification is sent; if the requester does not show up after two notifications, the request is then rejected and the requester notified about it. If the requester provides her information, the *ICM* system requests the production of the physical card, which is then issued to the requester.

The business process fragment executed in case of card loss is depicted in Figure 2.8b. The *ICM* system first registers the card loss case and issues a temporary card to the card holder. If the lost card is found before the production of a new one, the *ICM* system recalls the temporary card. After the production of a new card, the *ICM* system will recall the temporary card and issue the new one. If the lost card is found after the production of the new one but before the recall of the temporary one, the *ICM* system will recall the old card before recalling the temporary one.

The activity diagram in Figure 2.8c corresponds to the business process fragment executed in case of card expiration. When a card expires, the *ICM* system sends the card holder a letter to recall the card. If the card is returned, a confirmation receipt is then sent to the card holder; otherwise, another recall letter is sent to her. If, after two notification letters, the card holder has not returned the card yet, the *ICM* system reports the case to the police and the card holder will be fined.

2.6.2 Requirement specifications

We now list some requirements specifications associated with the three fragments of the *ICM* business process, and show how they can be expressed in *TemPsy*. These nine specifications (three for each business process fragment) have been selected out of the 47 available for the *ICM* application. Notice that these specifications have been written by the business analysts of our partner, who have domain knowledge, and represent realistic properties being used in practice.

Card Request:

R1 Once a card request is approved, the requester is notified within three days; this notification has to occur before the production of the card is started.

R2 The requester has to show up for the collection of her physical information within five days from the first notification.

R3 If the physical information of the requester is collected within three days after the second approval notification, the card will be produced and then issued to the requester.

These requirements specifications can be expressed in *TemPsy* as follows:

```
 1 temporal R1:
 2   before ICM.issueCard
 3   ICM.notifyApproval
 4   responding at most 3*24*3600 tu
 5   ICM.approveRequest
 6 temporal R2:
 7   after 1 ICM.notifyApproval
 8   at most 5*24*3600 tu
 9   eventually ICM.collectPhysicalInfo
10 temporal R3:
11   after 2 ICM.notifyApproval
12   at most 3*24*3600 tu
13   ICM.collectPhysicalInfo
14   preceding
15   ICM.produceCard, ICM.issueCard
```

Property R1 is expressed in lines 1–5. The *before* scope is delimited by the event `ICM.issueCard`. The *response* pattern is bounded (time units are expressed in seconds) and requires the notification to the requester (`ICM.notifyApproval`) to happen in response to the action of approving the request (`ICM.approveRequest`). Property R2 (lines 6–9) combines an *after* scope with an *existence* pattern. In R3, the *after* scope (line 11) is bounded by the second occurrence of `ICM.notifyApproval`; this scope is associated with a *precedence chain* pattern, where `ICM.collectPhysicalInfo` represents the first block and the events chain `ICM.produceCard`, `ICM.issueCard`, the second block.

Card Loss:

L1 If a card is reported as lost, a temporary card will be issued to the card holder within one day, and will be recalled in ten days after the issuance.

L2 After a card has been registered as lost, a new card should be produced at least two days before its issuance.

L3 If the lost card is found after the production of a new card, the old card and the temporary one should be recalled within three days.

These requirements specifications can be expressed in *TemPsy* as follows:

```
temporal L1:
  after Card.isLost
  at most 24*3600 tu
  ICM.recallTempCard
  responding at most 10*24*3600 tu
  ICM.issueTempCard
temporal L2:
  after Card.isLost
  ICM.produceCard
  preceding at least 2*24*3600 tu
  ICM.issueCard
temporal L3:
  after Card.isLost
  until ICM.issueCard
  ICM.recallCard,
  ICM.recallTempCard,
  responding at most 3*24*3600 tu
  ICM.produceCard,
  Card.isFound
```

Property L1 contains an *after* scope and a *response* pattern, where the scope boundary contains a time constraint, and the pattern also restricts the time distance between the issuance of a temporary card (`ICM.issueTempCard`) and the corresponding card recall event (`ICM.recallTempCard`). Property L3 combines an *after-until* scope with a *precedence chain* pattern, where the first block corresponds to the events chain `ICM.recallCard`, `ICM.recallTempCard`, and the second block corresponds to the events chain `ICM.produceCard`, `Card.isFound`.

Card Expiration:

E1 Once a card expires, the holder is notified to return the card at most twice.

E2 In case the expired card has not been returned after five days from the second notification to the holder, the latter will be fined after the case will be reported to the police.

E3 Once a card is returned, the holder will receive a confirmation within one day.

These requirements specifications can be expressed in *TemPsy* as follows:

```
temporal E1:
  after Card.isExpired
  until Card.isReturned
  eventually at most 2 ICM.recallCard
temporal E2:
  after 2 ICM.recallCard
  at least 5*24*3600 tu
  until Card.isReturned
  ICM.fine
  responding
  ICm.reportToPolice
temporal E3:
  globally
  ICM.confirmCardReturned
  responding at most 24*3600 tu
  Card.isReturned
```

Property E1 uses an *after-until* scope, where the left boundary event corresponds to the expiration of the card (`Card.isExpired`) and the right boundary event corresponds to the return of the card (`Card.isReturned`). A *bounded existence* pattern is used to specify the maximum amount of notifications (`ICM.recallCard`) that can occur. In property E2 we use an *after-until* scope combined with the keyword '`at least`' for the first boundary, to delimit the period during which the card holder will be fined once the expiration case is reported to the police (`ICM.reportToPolice`). Property E3 states an invariant of the system (using the *globally* scope) for the *response* pattern that correlates the return of the card (`Card.isReturned`) to the confirmation to the holder (`ICM.confirmCardReturned`).

2.6.3 Adoption of *TemPsy* by our partner

Our partner has adopted *TemPsy* as the specification language for expressing the requirements of its business process models. *TemPsy* specifications have provided business analysts with a means to reason and formalize business process requirements, and have replaced informal specifications written in natural language. Our partner has also developed, for internal use, a graphical version of *TemPsy*, which has been integrated into the SoftwareAG ARIS modeling tool [Software AG, 2014], as part of the Prometa business process modeling framework[5]; although the illustration of the graphical notation for *TemPsy* is out of the scope of this thesis, we provide an example of it in Figure 2.9.

[5]`https://joinup.ec.europa.eu/community/nifo/case/prometa-organisational-interoperability-framework-eservice-design-luxemburg`.

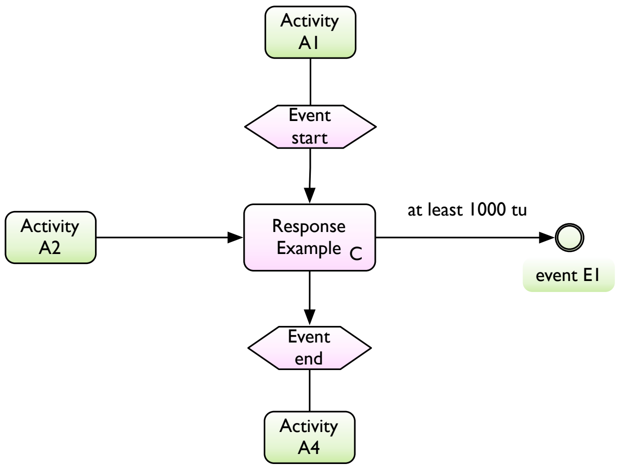


Figure 2.9: Example of the graphical notation for *TemPsy*

Table 2.1: Distribution of requirements from the *ICM* business process in terms of the combination of scopes and patterns

scope+pattern	# of requirements
globally+universality	1
globally+absence	1
globally+existence	2
globally+precedence	4
globally+response	4
before+absence	1
before+existence	2
before+precedence	3
before+response	2
after+universality	1
after+absence	1
after+existence	4
after+precedence	3
after+response	2
between-and+universality	2
between-and+absence	2
between-and+existence	1
between-and+precedence	1
between-and+response	1
after-until+universality	1
after-until+absence	1
after-until+existence	4
after-until+precedence	1
after-until+response	2

In terms of expressiveness, we recall that *TemPsy* has been designed based on the analysis of the structure of the requirements specifications written by our partner. Hence, all the requirements of the case study presented in the previous section could be expressed with *TemPsy*. Table 2.1 shows the distribution of the 47 requirements of the *ICM* business process, in terms of the combination of scopes and patterns.

Chapter 3

Model-driven Offline Trace Checking of Temporal Properties

The idea at the basis of our model-driven trace checking approach (TEMPSY-CHECK) is to *reduce* the problem of checking a *TemPsy* property ρ over a trace λ, to the problem of evaluating an OCL constraint (semantically equivalent to ρ) on an instance of a conceptual model for execution traces (equivalent to λ).

This reduction allows us to rely on standard and stable MDE technology to perform offline trace checking. Indeed, standard OCL checkers, such as Eclipse OCL [Eclipse, 2015a], can be used to evaluate OCL constraints on model instances. The use of a model-driven approach and of standard technologies fulfills requirement R2 stated in Chapter 1, and enables us to provide a practical and scalable solution for trace checking of temporal properties, which is also viable in the long term.

The rest of this chapter is organized as follows. In Section 3.1, we introduce the conceptual model we have defined to represent execution traces. In Section 3.2, we provide an overview of our approach and show how *TemPsy* properties (decomposed in scopes and patterns) can be expressed as OCL constraints on the conceptual model in Sections 3.3 and 3.4. In Section 3.5, we demonstrate with an example the application of the trace checking procedure of and in Section 3.6 we provide some notes about the implementation of the approach our TEMPSY-CHECK tool. We conclude the chapter with the evaluation of the scalability of the tool.

3.1 Conceptual Model for Execution Traces

The definition of a conceptual model for execution traces is a key element of our approach, since the transformation of *TemPsy* properties into efficiently checkable *OCL constraints defined on such model* is a key strategy for us to achieve scalability.

We propose a simple and yet generic model of system execution traces; it can be extended (by enriching the type of event) depending on the actual type of system (e.g., business process, access control framework) and the type of properties to check. The model, depicted in Figure 3.1 with a UML class diagram, contains a `Trace`, which is composed of a sequence of `TraceElements`, accessed through the association `traceElements`. Each `TraceElement` contains an attribute `event` of

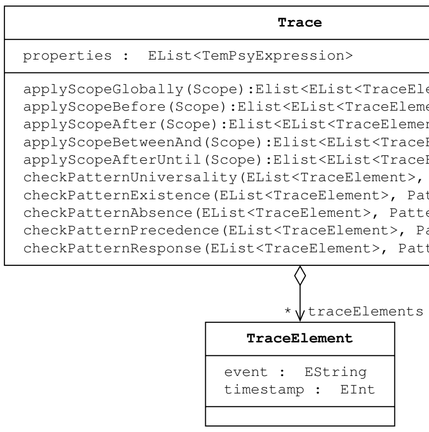

Trace
properties : EList<TemPsyExpression>
applyScopeGlobally(Scope):Elist<EList<TraceElement>>
applyScopeBefore(Scope):Elist<EList<TraceElement>>
applyScopeAfter(Scope):Elist<EList<TraceElement>>
applyScopeBetweenAnd(Scope):Elist<EList<TraceElement>>
applyScopeAfterUntil(Scope):Elist<EList<TraceElement>>
checkPatternUniversality(EList<TraceElement>, Pattern):Boolean
checkPatternExistence(EList<TraceElement>, Pattern):Boolean
checkPatternAbsence(EList<TraceElement>, Pattern):Boolean
checkPatternPrecedence(EList<TraceElement>, Pattern):Boolean
checkPatternResponse(EList<TraceElement>, Pattern):Boolean
* traceElements
TraceElement
event : EString
timestamp : EInt

Figure 3.1: Conceptual model for execution traces

type string, which represents the actual event recorded in the trace, and an attribute `timestamp` of type integer, which indicates the time at which the event occurred. Class `Trace` contains also an attribute `properties`, which is a collection of `TemPsyExpressions`[1], representing the properties to be checked on the trace.

We have defined some side-effect-free operations in OCL for the `Trace` class; these operations consist of two types of functions. The first type, of the form `applyScope*S*`, are named after the different types of scope (e.g., `applyScopeBefore`, `applyScopeBetweenAnd`) and return segment(s) of a trace (i.e., sub-traces) as determined by the parameters of the scope provided in input. The second type, of the form `checkPattern*P*`, are named after the different types of pattern (e.g., `checkPatternExistence`, `checkPatternPrecedence`) and check whether the pattern provided in input as the second parameter holds on the sub-trace(s) represented by the first parameter.

3.2 Overview of the Approach

Our approach for model-driven trace checking is sketched in Figure 3.2: parallelogram shapes correspond to input/output artifacts, while rectangles correspond to steps in the approach. The two inputs are represented by a log, corresponding to the trace one wants to check, and by a set of *TemPsy* properties. The log file is read and converted (step 1a) to an instance of the class `trace` in the trace model shown in Figure 3.1. The *TemPsy* properties are parsed and converted (step 1b) to instances of class `TemPsyExpression`.

The key step (#2 in the figure) of our approach is to evaluate an OCL invariant on the trace instance. The checking of this invariant, which can be done using standard OCL checking tools, is semantically equivalent to performing trace checking of the *TemPsy* properties provided in input.

We have defined this invariant on the `Trace` class, as shown in Figure 3.3. For every *TemPsy* property provided in input (and referenced in the trace instance as the attribute `self.properties`, line 2), the invariant evaluates a boolean function, which conceptually corresponds to applying the se-

[1]Class `TemPsyExpression` belongs to the meta-model of the language (not shown here for space reasons) and represents objects corresponding to the non-terminal ⟨*TemPsyExpression*⟩ of the grammar shown in Figure 2.2.

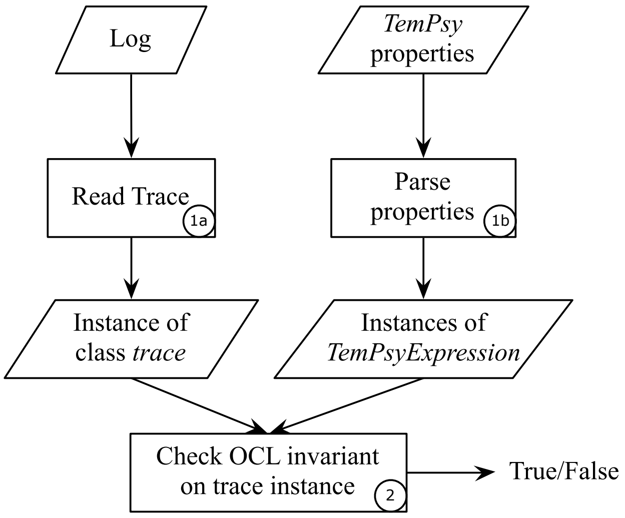


Figure 3.2: Overview of the approach to offline trace checking of *TemPsy* properties

mantics of the pattern used in the property (accessed through the expression `property.pattern`) on a set of sub-traces, as defined by the scope used in the property (accessed through the expression `property.scope`).

More specifically, the body of the invariant expression is a multi-way branch (defined through a sequence of `if` statements), which selects a certain branch based on the specific scope type used within the property. Within the body of a branch, first a function of the form `applyScope*S*` is called. This function takes the scope used in the property as input and returns a collection of sub-traces, as defined by the scope semantics. Afterwards, the invariant invokes a function of the form `checkPattern*P*`, which checks whether the pattern used in the property holds on each sub-trace.

For instance, let us assume that the type of the scope of the *TemPsy* property provided in input is *globally* and that the type of the pattern used in the property is *response*. As shown in line 5, the function `applyScopeGlobally` is invoked to compute the sub-trace(s) defined by the `scope` parameter; the return value of this function is assigned to variable `subtraces`. The branch indicated on line 15 is then taken, which results in the evaluation of the boolean function `checkPatternResponse` on all the elements[2] of `subtraces`, to check whether the input parameter `pattern` holds on each sub-trace.

In Sections 3.3 and 3.4, we describe the `applyScope*S*` and `checkPattern*P*` functions respectively; to ease legibility and conciseness, all the code snippets presented in these subsections are written using pseudocode.

3.3 OCL Functions for Scopes

In this section we illustrate the OCL functions that are used to apply a scope definition on a trace. We show the pseudocode of functions `applyScopeBefore`, `applyScopeAfter`, `applyScope`

[2]In the case of scope *globally*, only the variable `subtraces` will contain, by definition, only one trace.

```
context Trace
inv: self.properties->forAll(property:TemPsy::TemPsyExpression |
  let scope:TemPsy::Scope = property.scope, pattern:TemPsy::Pattern =
     property.pattern in
  if scope.type = TemPsy::ScopeType::GLOBALLY then
    let subtraces:Sequence(OrderedSet(TraceElement)) =
       applyScopeGlobally(scope) in
    if pattern.type = TemPsy::PatternType::UNIVERSALITY then
      subtraces->forAll(subtrace | checkPatternUniversality(subtrace,
         pattern))
    else if pattern.type = TemPsy::PatternType::EXISTENCE then
      subtraces->forAll(subtrace |  checkPatternExistence(subtrace,
         pattern))
    else if pattern.type = TemPsy::PatternType::ABSENCE then
      subtraces->forAll(subtrace | checkPatternAbsence(subtrace,
         pattern))
    else if pattern.type = TemPsy::PatternType::PRECEDENCE then
      subtraces->forAll(subtrace | checkPatternPrecedence(subtrace,
         pattern))
    else if pattern.type = TemPsy::PatternType::RESPONSE then
      subtraces->forAll(subtrace | checkPatternResponse(subtrace,
         pattern))
    endif endif endif endif endif
  else if scope.type = TemPsy::ScopeType::BEFORE then
    ...
  else if scope.type = TemPsy::ScopeType::AFTER then
    ...
  else if scope.type = TemPsy::ScopeType::BETWEENAND then
    ...
  else if scope.type = TemPsy:ScopeType::AFTERUNTIL then
    ...
  endif endif endif endif endif)
```

Figure 3.3: OCL invariant for checking *TemPsy* properties on a trace

`BetweenAnd`, and `applyScopeAfterUntil` corresponding to the *before*, *after*, *between-and*, and *after-until* scopes. These functions take as input an object representing a scope in *TemPsy* and yield one or more segments of the trace (i.e., sub-trace(s)), as determined by the semantics of the scope.

3.3.1 Before

The definition of function `applyScopeBefore` is shown in Algorithm 1. The input parameter *scope* is an instance of the *before* scope, and the output is a list that contains the trace segments as determined by the structure of *scope*. We assume the parameter *scope* to have the form "`before [m]` *X* `[op n tu]`" (see Section 2.3), in which `op` stands for the comparison operator (i.e., "`at least`", "`at most`", or "`exactly`") used in the constraint that defines the time distance from the scope boundary event *X*.

Algorithm 1: applyScopeBefore

```
Input: scope : an instance of the before scope structured as "before [m] X [op n tu]"
Output: result : a list containing the trace segment as determined by the parameters of scope
1  $X \leftarrow$ event name of the scope boundary
2  $m \leftarrow$ index of the specific occurrence of event $X$
3  $op \leftarrow$ comparison operator of the constraint on time distance
4  $n \leftarrow$ time distance from the $m$-th occurrence of $X$
5  $result \leftarrow [], segment \leftarrow []$
6  if $m =$ null then $m \leftarrow 1$
7  $t \leftarrow$ timestamp of the $m$-th occurrence of event $X$
8  if $t \neq$ null then
9      switch $op$ do
10         case "at least" do
11             $segment \leftarrow$ trace elements with timestamp $t'$ satisfying $t' \leq t - n$
12         case "at most" do
13             $segment \leftarrow$ trace elements with timestamp $t'$ satisfying $t - n \leq t' < t$
14         case "exactly" do
15             $segment \leftarrow$ trace elements with timestamp equal to $t - n$
16         otherwise do
17             $segment \leftarrow$ trace elements with timestamp $t'$ satisfying $t' < t$
18 $result$.append($segment$)
19 return $result$
```

The function starts by reading the parameters *X*, *m*, *op*, and *n* from the instance of the *before* scope (lines 1–4). In addition, we define and initialize to an empty list both variable *result* (to store the output value) and the auxiliary variable *segment* (for collecting intermediate trace elements). If the parameter *m* is omitted in the scope definition, variable *m* is replaced with the value 1 (line 6), according to the default semantics of the *before* scope. We then assign to variable *t* the timestamp of the *m*-th occurrence of event *X* in the trace (line 7). If *t* is defined, it means that the *m*-th occurrence of the event has been found in the trace. Lines 9–17 select a segment from the trace, based on the value of *op*. For example, when *op* is "`at least`", line 11 selects all the trace elements that occur at least *n* time unit(s) before the *m*-th occurrence of event *X*. If no time distance constraint is specified in the *scope* (line 17), the function selects the trace segment starting at the beginning of the trace and ending at the *m*-th occurrence of event *X*. The function ends by adding the *segment* selected from the trace to the output variable *result*.

3.3.2 After

The definition of function `applyScopeAfter` is shown in Algorithm 2. The input parameter *scope* is an instance of the *after* scope, and the output is a list that contains the trace segments as determined by the structure of *scope*. We assume the parameter *scope* to have the form "`after [m] X [op n tu]`" (see Section 2.3), in which `op` stands for the comparison operator (i.e., "`at least`", "`at most`", or "`exactly`") used in the constraint that defines the time distance from the scope boundary event *X*.

Algorithm 2: applyScopeAfter

```
Input: scope : an instance of the after scope structured as "after [$m$] $X$ [$op$ $n$ tu]"
Output: result : a list containing the trace segment as determined by the parameters of scope
1  $X \leftarrow$ event name of the scope boundary
2  $m \leftarrow$ index of the specific occurrence of event $X$
3  $op \leftarrow$ comparison operator of the constraint on time distance
4  $n \leftarrow$ time distance from the $m$-th occurrence of $X$
5  $result \leftarrow [], segment \leftarrow []$
6  if $m =$ null then $m \leftarrow 1$
7  $t \leftarrow$ timestamp of the $m$-th occurrence of event $X$
8  if $t \neq$ null then
9      switch $op$ do
10         case "at least" do
11             $segment \leftarrow$ trace elements with timestamp $t'$ satisfying $t' \geq t + n$
12         case "at most" do
13             $segment \leftarrow$ trace elements with timestamp $t'$ satisfying $t < t' \leq t + n$
14         case "exactly" do
15             $segment \leftarrow$ trace elements with timestamp equal to $t + n$
16         otherwise do
17             $segment \leftarrow$ trace elements with timestamp $t'$ satisfying $t' > t$
18 $result$.append($segment$)
19 return $result$
```

The function starts by reading the parameters *X*, *m*, *op*, and *n* from the instance of the *after* scope (lines 1–4). In addition, we define and initialize to an empty list both variable *result* (to store the output value) and the auxiliary variable *segment* (for collecting intermediate trace elements). If the parameter *m* is omitted in the scope definition, variable *m* is replaced with the value 1 (line 6), according to the default semantics of the *after* scope. We then assign to variable *t* the timestamp of the *m*-th occurrence of event *X* in the trace (line 7). If *t* is defined, it means that the *m*-th occurrence of the event has been found in the trace. Lines 9–17 select a segment from the trace, based on the value of *op*. For example, when *op* is "`at least`", line 11 selects all the trace elements that occur at least *n* time unit(s) after the *m*-th occurrence of event *X*. If no time distance constraint is specified in the *scope* (line 17), the function selects the trace elements after the *m*-th occurrence of event *X*. The function ends by adding the *segment* selected from the trace to the output variable *result*.

3.3.3 Between-and

Algorithm 3 presents the definition of the function `applyScopeBetweenAnd`. This function takes as input an object representing an instance of the *between-and* scope and returns a lists of trace segments. We assume the parameter *scope* to have the form "`between [m1]` *X* `[at least n1 tu] and [m2]` *Y* `[at least n2 tu]`".

Function `applyBetweenAnd` starts by reading the parameters from the instance of the *between-and* scope (lines 1–6): variables *X* and *Y* correspond to the event names of the left and right scope

Algorithm 3: applyScopeBetweenAnd

```
Input: scope : an instance of the between-and scope structured as "between [$m1$] $X$ [at
       least $n1$ tu] and [$m2$] $Y$ [at least $n2$ tu]"
Output: result : a list of trace segments, as determined by the parameters of scope
$X \leftarrow$ event name of the left boundary
$Y \leftarrow$ event name of the right boundary
$m1 \leftarrow$ index of the specific occurrence of event $X$
$m2 \leftarrow$ index of the specific occurrence of event $Y$
$n1 \leftarrow$ lower bound of the time distance from the $m1$-th occurrence of event $X$
$n2 \leftarrow$ lower bound of the time distance from the $m2$-th occurrence of event $Y$
result $\leftarrow$ []
if $m1$ = null && $m2$ = null then
    result $\leftarrow$ applyOriginalBetweenAnd($X$, $n1$, $Y$, $n2$)
else
    if $m1$ = null then $m1 \leftarrow 1$
    if $m2$ = null then $m2 \leftarrow 1$
    result.append(applySpecialBetweenAnd($m1$, $X$, $n1$, $m2$, $Y$, $n2$))
```

boundaries; *m1* and *m2* represent the (optional) index of the specific occurrence of event *X* and event *Y* referred to in the scope definition; *n1* and *n2* are the (optional) lower bounds on the time distances from the two scope boundaries. Optional parameters are initialized to null if they are not defined. The output variable *result* is initialized to an empty list.

If both *m1* and *m2* are not defined, we compute the return value by calling the auxiliary function `applyOriginalBetweenAnd` (line 9), which retrieves all the trace segments delimited by the two boundary events (taking into account the distances from the boundaries, if defined). Otherwise, if either *m1* or *m2* is undefined, we compute the return value by calling the auxiliary function `applySpecialBetweenAnd` (line 13), which retrieves only one trace segment, as determined by the specific occurrences of the boundary events and by the time distance from the scope boundaries (if defined). Notice that in the latter case we consider as boundary the first occurrence of event *X* or *Y* (see assignments at lines 11–12).

Function `applyOriginalBetweenAnd` is shown in Algorithm 4. It takes in input the parameters *X*, *Y*, *n1*, *n2* of a *between-end* scope of the form "`between` X `[at least` $n1$ `tu] and` Y `[at least` $n2$ `tu]`" and returns a list of the trace segments determined by the scope semantics. The function goes through all the elements of the list and identifies all the segments delimited by the events *X* and *Y*, taking into account the parameters for the time distance from the scope boundaries.

Besides the output variable *result*, we define an integer tuple (i_1, t_1) to keep track of the starting point of a trace segment. More precisely, element i_1 refers to the index of the trace element that comes after the left bound of the segment (characterized by an occurrence of event *X*), while element t_1 points to the instant that is *n1* time units after the occurrence of the left bound of the segment. The tuple (i_2, t_2) is defined in a similar way, to keep track of the end point of a trace segment (characterized by an occurrence of event *Y*).

Algorithm 4: applyOriginalBetweenAnd

```
Input: strings $X, Y$ and integers $n1, n2$ ($n1 = 0$, $n2 = 0$ by default), i.e., the parameters of a
       between-and scope structured as "between $X$ [at least $n1$ tu] and $Y$
       [at least $n2$ tu]"
Output: result : a list of trace segments, as determined by the parameters of the scope
1  $result \leftarrow []$
2  $index \leftarrow 0$
3  $(i_1, t_1) \leftarrow (0,0)$
4  $(i_2, t_2) \leftarrow (0,0)$
5  for $elem \in self.traceElements$ do
6      $index \leftarrow index + 1$
7      $e \leftarrow elem.event$
8      $t \leftarrow elem.timestamp$
9      if $i_1 = 0$ then
10         if $e = X$ then
11             $(i_1, t_1) \leftarrow (index + 1, t + n1)$
12     else if $e = Y$ && $index > i_1$ then
13         $(i_2, t_2) \leftarrow (index - 1, t - n2)$
14         $segment \leftarrow self.traceElements[i_1 .. i_2]$
15         if $n1 \neq 0 \,||\, n2 \neq 0$ then
16             $segment \leftarrow$ trace elements in $segment$ with timestamps $t'$ satisfying $t_1 \leq t' \leq t_2$
17         $result$.append($segment$)
18         $(i_1, t_1) \leftarrow (0,0)$
19         $(i_2, t_2) \leftarrow (0,0)$
20 return $result$
```

At each iteration of the loop (lines 5–19), for each element of the trace, the function first increments the variable *index* and assigns the event of the trace element to variable *e* as well as its timestamp to variable *t* (lines 6–8). Within the loop, a value of i_1 equal to 0 means that the left bound of the segment has not been found yet. When the current event matches *X* (line 10), i_1 is assigned the next index of current event; t_1 is assigned the value of the timestamp of current event incremented by *n1* time units (line 11). When variable i_1 is different than 0, it means that the left boundary has been found while the right boundary has not been found yet. In this case, the function keeps scanning the remaining trace elements until it finds an occurrence of event *Y*. If the current event matches *Y* and if the current index is more than i_1 (line 12), i_2 is assigned the previous index of current event; t_2 is assigned the value of the timestamp of current event decremented by *n2* time units (line 13). At this point, the function extracts a trace segment comprised between indexes i_1 and i_2 (line 14), whose trace elements have a timestamp comprised between t_1 and t_2 (line 16). This segment is added to the output variable *result* and then the tuples (i_1, t_1) and (i_2, t_2) are reset (for the next loop iteration).

Function `applySpecialBetweenAnd`, as shown in Algorithm 5, is defined similarly to function `applyOriginalBetweenAnd`, but is extended with two additional parameters *m1* and *m2*, referring to the specific index of the occurrence of each of the two boundary events. This function identifies a single segment of the trace between the *m1*-th occurrence of event *X* and the *m2*-th oc-

Algorithm 5: applySpecialBetweenAnd

```
Input: strings $X, Y$ and integers $m1, n1, m2, n2$ ($n1 = 0, n2 = 0$ by default), i.e., the parameters
       of a between-and scope structured as "between $m1$ $X$ [at least $n1$ tu]
       and $m2$ $Y$ [at least $n2$ tu]"
Output: result : a trace segment, as determined by the parameters of the scope
result ← []
index ← 0
$(i_1, t_1) \leftarrow (0,0)$
$(i_2, t_2) \leftarrow (0,0)$
count ← 0
for elem ∈ self.traceElements do
    index ← index + 1
    e ← elem.event
    t ← elem.timestamp
    if $i_1 = 0$ && $e = X$ then
        count ← count + 1
        if count = $m1$ then
            $(i_1, t_1) \leftarrow (index + 1, t + n1)$
            count ← 0
    else if $i_2 = 0$ && $e = Y$ then
        count ← count + 1
        if count = $m2$ then
            $(i_2, t_2) \leftarrow (index - 1, t - n2)$
            break
if $i_1 > 0$ && $i_1 \leq i_2$ then
    result ← self.traceElements$[i_1 .. i_2]$
    if $n1 \neq 0 \parallel n2 \neq 0$ then
        result ← trace elements with timestamps $t'$ satisfying $t_1 \leq t' \leq t_2$
return result
```

currence of event *Y*, taking into account the constraints on the time distances from the two scope boundaries. The function body is similar to that in Algorithm 4 and is extended with a counter that keeps track of the number of occurrences of a boundary event found while traversing the trace elements. Since only one segment has to be identified with this function, the main loop is interrupted as soon as such a segment is found.

3.3.4 After-until

Function `applyScopeAfterUntil` takes as input an object representing an instance of the *after-until* scope and returns a lists of trace segments. The input *scope* is in the form of "`after [m1] X [at least n1 tu] until [m2] Y [at least n2 tu]`". Likewise the function defined for applying the *between-and* scope (Algorithm 3), function `applyScopeAfterUntil` invokes the auxiliary function `applyOriginalAfterUntil` (Algorithm 6) if both *m1* and *m2* are

Algorithm 6: applyOriginalAfterUntil (procedure to complement Algorithm 4)

```
Input: strings $X,Y$ and integers $n1,n2$ ($n1 = 0$, $n2 = 0$ by default), i.e., the parameters of a
       after-until scope structured as “after $X$ [at least $n1$ tu] until $Y$ [at
       least $n2$ tu]”
Output: result : a list of trace segments, as determined by the parameters of the scope
1  $size \leftarrow self.traceElements$.size()
2  if $i_1 > 0$ && $i_1 \leq size$ then
3      $segment \leftarrow self.traceElements[i_1 .. size]$
4      if $n1 \neq 0$ then
5          $segment \leftarrow$ trace elements in $segment$ with timestamps $t'$ satisfying $t' \geq t_1$
6      $result$.append($segment$)
```

Algorithm 7: applySpecialAfterUntil (procedure to substitute lines 20–23 of Algorithm 5)

```
Input: strings $X,Y$ and integers $m1,n1,m2,n2$ ($n1 = 1 = 0, n2 = 0$ by default), i.e., the
       parameters of a after-until scope structured as “after $m1$ $X$ [at least $n1$ tu]
       until $m2$ $Y$ [at least $n2$ tu]”
Output: result : a trace segment, as determined by the parameters of the scope
1  $size \leftarrow self.traceElements$.size()
2  if $i_1 > 0$ then
3      if $i_2 = 0$ && $i_1 \leq size$ then
4          $result \leftarrow self.traceElements[i_1 .. size]$
5          if $n1 \neq 0$ then
6              $result \leftarrow$ trace elements with timestamps $t'$ satisfying $t' \geq t_1$
7      else if $i_1 \leq i_2$ then
8          $result \leftarrow self.traceElements[i_1 .. i_2]$
9          if $n1 \neq 0$ || $n2 \neq 0$ then
10             $result \leftarrow$ trace elements with timestamps $t'$ satisfying $t_1 \leq t' \leq t_2$
```

not defined in the input *scope*; otherwise, auxiliary function `applySpecialAfterUntil` (Algorithm 7) is called to retrieve the segment that is determined by the specific occurrences of the boundary events.

The *after-until* scope is defined as a hybrid of *between-and* and *after* scopes. That is, given a trace, the scope selects all the trace segments determined by its twin *between-and* scope; in addition, if the (specific) right boundary does not occur after a (specific) occurrence of the left boundary, the portion from the (specific) occurrence of the left boundary to the end of the trace is included in the output (see Figure 2.5d). In function `applyOriginalAfterUntil` (Algorithm 6), For space reasons, we omit the duplicate implementation as shown in Algorithm 4, and present the additional (pseudo)code to select the possible segment for the latter case, which is then added to Algorithm 4 between line 19 and the last line.

The algorithm uses an additional variable *size* (line 1) to store the number of trace elements of the trace instance. As shown in the loop of function `applyOriginalBetweenAnd` (Algorithm 4), we iterate through the trace instance and select all the segments delimited by the occurrences of the two boundary events, in consideration of possible constraints on the distance from each boundary. Hence if there is an odd occurrence of event *X* found in the loop (i.e., $i_1 > 0$) and the position i_1 is not the last (line 2), function `applyOriginalAfterUntil` also selects the trace segment after the occurrence of event *X* (line 3). If the constraint on the distance from the left boundary is defined (line 4), the selected segment is pruned by fulfilling the constraint (line 5). At line 6, the trace segment is add into the result.

Function `applySpecialAfterUntil` (Algorithm 7) describes the new procedure that selects the trace segment after investigating the *m1*-th occurrence of event *X* and the *m2*-th occurrence of event *Y* (lines 6–19 of Algorithm 5). Notice that for space reasons, the pseudocode shown in the algorithm is only the substitute for lines 20–23 of function `applySpecialBetweenAnd`.

In the algorithm, if the *m1*-th occurrence of event *X* is found in the trace (line 2), there are two possible procedures to be carried out: the one for selecting all the trace elements after the *m1*-th occurrence of event *X* (lines 3–6); or the other for selecting the segment between the *m1*-th occurrence of event *X* and the *m2*-th occurrence of event *Y* (lines 7–10). The former is executed when the *m2*-th occurrence of event *Y* is not found in the loop (lines 6–19 of Algorithm 5) and variable i_1 is no greater than the trace size (line 3). In the algorithm, we select the trace segment from the position i_1 to the end of the trace (line 4) and prune it (line 6) by fulfilling the constraint on the distance from the left boundary (if defined). The latter is executed when the *m2*-th occurrence of event *Y* is found and variable i_1 is no greater than i_2. The implementation is identical to lines 21–23 of function `applySpecialBetweenAnd` (Algorithm 5).

3.4 OCL Functions for Patterns

In this section we present the OCL functions that are used to check if a pattern holds on a sub-trace. We show the pseudocode of functions `checkPatternUniversality`, `checkPatternExistence`, `checkPatternAbsence`, `checkPatternPrecedence`, and `checkPatternResponse` for the five patterns. These functions take as input a sub-trace and an object representing a pattern in *TemPsy*, and return whether the pattern holds on the input sub-trace.

3.4.1 Universality

Algorithm 8: checkPatternUniversality

```
Input: a trace segment subtrace and an instance of the universality pattern pattern, in the form
       "always E"
Output: true if pattern holds on subtrace; false otherwise
1 $E \leftarrow$ event name in pattern
2 for $elem \in subtrace$ do
3     if $elem.event \neq E$ then return false
4 return true
```

Function `checkPatternUniversality` (see Algorithm 8) takes in input a trace segment (denoted by the variable *subtrace*) and an instance of the *universality* pattern (denoted by the variable *pattern*). At the first line, the function obtains the event name *E* from the input *subtrace*. A loop is used (lines 2–3) to check if every element of the trace segment is an occurrence of event *E*, and the function returns `false` if any violation is found. If there is no violation reported in the loop, the function returns `true` (line 4).

3.4.2 Existence

Algorithm 9: checkPatternExistence

```
Input: a trace segment subtrace and an instance of the existence pattern pattern, in the form
       "eventually [op n] E"
Output: true if pattern holds on subtrace; false otherwise
1 $E \leftarrow$ event name in pattern
2 $op \leftarrow$ comparison operator of the bound on the number of occurrences of event $E$
3 $n \leftarrow$ threshold of the occurrence number of event $E$
4 $count \leftarrow$ the number of occurrences of event $E$ in subtrace
5 return compare(count, op, n)
```

Function `checkPatternExistence` (see Algorithm 9) defines the algorithm for checking whether an *existence* pattern holds on a certain trace segment. The function first retrieves the parameters from input *pattern*: the event name *E*, the comparison operator *op*, and the threshold of the number of event occurrences *n* (lines 1–3). The algorithm uses variable *count* to store the number of occurrences of event *E* in the input *subtrace* (line 4). The function returns the result of the invocation of the auxiliary function `compare`, which compares the value of *count* against the value of parameter *n* using the comparison operation defined by *op* (which can be "`at least`", "`at most`", or "`exactly`"). The auxiliary function `compare`, not shown here for space reasons, takes into account also the case in which *op* is null, meaning that the function returns true if the value of *count* is greater than 0.

3.4.3 Absence

The definition of function `checkPatternAbsence` (see Algorithm 10) checks whether an *absence* pattern holds on a given trace segment. Likewise function `checkPatternExistence`, the function first retrieves parameters from the input *subtrace*: the event name *E*, the comparison operator *op*, and the threshold of the number of event occurrences *n* (lines 1–3).

The major part of the function is an *if-then-else* statement which checks the two forms of *absence* pattern respectively. In the case of the traditional *absence* pattern (i.e., "`never` *E*"), the function iterates through the input *subtrace* to check whether there is any occurrence of event *E* (lines 5–6). The function returns `false` if an occurrence of event *E* is found in the loop (line 6); otherwise it returns `true` (line 7). If the pattern contains an comparison operator, variable *count* is set to the number of occurrences of event *E* in the input *subtrace* (line 9). The function returns the comparison result of inequality between *count* and *n* (line 10).

Algorithm 10: checkPatternAbsence

```
Input: a trace segment subtrace and an instance of the absence pattern pattern, in the form
       “never [op n] E”
Output: true if pattern holds on subtrace; false otherwise
1  $E \leftarrow$ event name in pattern
2  $op \leftarrow$ comparison operator of the bound on the number of occurrences of event $E$
3  $n \leftarrow$ threshold of the occurrence number of event $E$
4  if $op$ = null then
5      for $elem \in subtrace$ do
6          if $elem.event = E$ then return false
7      return true
8  else
9      $count \leftarrow$ the number of occurrences of event $E$ in subtrace
10     return $count \neq n$
```

3.4.4 Precedence

The definition of function `checkPatternPrecedence` comes in four variants, to consider the case whether no time distance is specified between the two blocks of the patterns, and the three cases with the different comparison operators (i.e., “`at least`”, “`at most`”, and “`exactly`”). In the rest of this section, we describe the functions defined for checking the four variants of *precedence* pattern, i.e., `checkPatternPrecedenceGeneral` (Algorithm 11), `checkPatternPrecedenceAtLeast` (Algorithm 13), `checkPatternPrecedenceAtMost` (Algorithm 14), and `checkPatternPrecedenceExactly` (Algorithm 15).

Algorithm 11: checkPatternPrecedenceGeneral

```
Input: a trace segment subtrace and the two events (chains) $block_1$ and $block_2$ of an instance
       of precedence pattern of the form “$block_1$ preceding $block_2$”
Output: true if pattern holds on subtrace; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event
3  $firstOfBlock2 \leftarrow block_2$.first().event
4  $(i_1, pt_1) \leftarrow (1,0)$, $(i_2, pt_2) \leftarrow (1,0)$
5  for $elem \in subtrace$ do
6      $e \leftarrow elem.event$
7      $t \leftarrow elem.timestamp$
8      if $e = firstOfBlock1$ then $(i_1, pt_1) \leftarrow (2, t)$
9      else if $i_1 > 1$ then $(i_1, pt_1) \leftarrow$ match$(block_1, i_1, pt_1, e, t)$
10     if $i_1 = size_1 + 1$ then return true
11     if $e = firstOfBlock2$ then $(i_2, pt_2) \leftarrow (2, t)$
12     else if $i_2 > 1$ then $(i_2, pt_2) \leftarrow$ match$(block_2, i_2, pt_2, e, t)$
13     if $i_2 = size_2 + 1$ then return false
14 return true
```

Function `checkPatternPrecedenceGeneral` takes in input a trace segment and the parameters of an instance of the *precedence* pattern: $block_1$, $block_2$. Notice that $block_1$ and $block_2$ can be either an atomic event or a chain of events with optional constraints on the time distances in between.

This *precedence* pattern prescribes that each occurrence of $block_2$ must be preceded by an occurrence of $block_1$. In practice, we need to check whether there is an occurrence of $block_1$ before the *first* occurrence of $block_2$, since this implies that any other possible occurrence of $block_2$ occurring after the first one is preceded by an occurrence of $block_1$. We report a violation if the algorithm cannot find an occurrence of $block_1$ before the first occurrence of $block_2$.

The algorithm uses some auxiliary variables: $size_1$ and $size_2$ keep track of the number of events to match in each block; *firstOfBlock1* and *firstOfBlock2* contain the event of each block's first element. Moreover, the integer tuple (i_1,pt_1) (respectively (i_2,pt_2)) is used to determine whether the trace element being checked is a match of the next event in $block_1$ (respectively, $block_2$). More specifically, element i_1 (respectively, i_2) stores the position within $block_1$ (respectively, $block_2$) of the next event to be matched; element pt_1 (respectively, pt_2) stores the timestamp of the previous trace element matched at $block1[i_1-1]$ (respectively, $block2[i_2-1]$).

The function contains a loop that iterates through all the elements of the input *subtrace*, trying to match each element with $block1[i_1]$ (lines 8–10) and with $block2[i_2]$ (lines 11–13). As for matching $block_1$, if the current trace element matches the first event of $block_1$ (line 8), the variable i_1 is set to 2 and pt_1 is updated with the current timestamp. Otherwise, if the next event of *block1* to be matched is not the first, an auxiliary function `match` is called to match the event defined at i_1.

Algorithm 12: match

```
Input: an events chain block, a tuple $(i, pt)$ of which $i$ $(i > 1)$ stores the position (within block)
       of the event to be checked, $pt$ stores the timestamp of the previous trace element if it
       was a match for $block[i-1]$, and a trace element $(e, t)$ to be matched with $block[i]$
Output: $(i+1,t)$ if the trace element is a match for $block[i]$; $(1,0)$ otherwise
1 if $e = block[i].event$ then
2     $op \leftarrow block[i].timeDistance.op$
3     $t' \leftarrow pt + block[i].timeDistance.value$
4     if compare($t$, $op$, $t'$) then $(i,pt) \leftarrow (i+1,t)$
5 else $(i,pt) \leftarrow (1,0)$
6 return $(i, pt)$
```

Function `match` takes in input five parameters: an events chain *block*, two integer parameters i and pt, of which i $(i > 1)$ stores the position (within *block*) of the event to be checked and pt stores the timestamp of the previous trace element (if it was a match for $block[i-1]$), and the two parameters of a trace element (e,t) to be matched with $block[i]$. The function updates the tuple (i,pt) if the input element is a match for $block[i]$; or else it sets the tuple to $(1,0)$. More specifically, if the current element is an occurrence of the event defined at $block1[i_1]$ (with i_1 being greater than 1) (line 1), and if the constraint on the distance (if defined[3]) from the previous event at $block1[i_1-1]$ holds (line 4),

[3]The pseudocode for dealing with the case when the distance between block elements is not defined has been omitted

variable i_1 is incremented and variable pt_1 is updated with the timestamp of current trace element (line 4). Otherwise, the tuple (i_1,pt_1) is reset on line 5. Note that function `match` will be reused in the following functions.

At line 10 of function `checkPatternPrecedenceGeneral`, if the matched event is the last event of $block_1$, namely, an occurrence of *block1* has been found preceding any possible occurrence of *block2*, the algorithm then stops and returns with a positive result. Within each single iteration of the loop, the algorithm also checks whether the current trace element is part of an occurrence of $block_2$ (lines 11–13). If the occurrence of the *first* event of $block_2$ is detected (line 11), the variable i_1 is set to 2 and pt_1 is updated with the current timestamp. Otherwise the tuple (i_2,pt_2) is updated by calling function `match`. In accordance with the semantics of the pattern, a violation is reported when an occurrence of *block2* is fully matched (line 13). The algorithm returns `true` (line 14) when there is no violation found in the loop.

Algorithm 13: checkPatternPrecedenceAtLeast

```
Input: a trace segment subtrace and the parameters of an instance of precedence pattern of the
       form "$block_1$ preceding at least n tu $block_2$": two events (chains) $block_1$
       and $block_2$, and a threshold $n$ of the time distance between $block_1$ and $block_2$
Output: true if pattern holds on subtrace; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event
3  $firstOfBlock2 \leftarrow block_2$.first().event
4  $(i_1,pt_1) \leftarrow (1,0)$, $(i_2,pt_2) \leftarrow (1,0)$
5  $flag_1 \leftarrow$ true
6  for $elem \in subtrace$ do
7      $e \leftarrow elem.event$
8      $t \leftarrow elem.timestamp$
9      if $flag_1$ then
10         if $e = firstOfBlock1$ then $(i_1,pt_1) \leftarrow (2,t)$
11         else if $i_1 > 1$ then $(i_1,pt_1) \leftarrow$ match($block_1, i_1, pt_1, e, t$)
12         if $i_1 = size_1 + 1$ then $flag_1 \leftarrow$ false
13     if $e = firstOfBlock2$ then
14         if $flag_1 \parallel t < pt_1 + n$ then $(i_2,pt_2) \leftarrow (2,t)$
15         else return true
16     else if $i_2 > 1$ then $(i_2,pt_2) \leftarrow$ match($block_2, i_2, pt_2, e, t$)
17     if $i_2 = size_2 + 1$ then return false
18 return true
```

Function `checkPatternPrecedenceAtLeast` takes in input a trace segment and the parameters of an instance of the *precedence* pattern: $block_1$, $block_2$, and n, the threshold of the time distance between them. The strategy of this function is likewise the previous one, except that it needs to also check whether the time distance from the first occurrence of $block_1$ to the *first* occurrence of

for simplicity.

block$_2$ is more than n. Hence, a violation is reported either when there is no occurrence of *block*$_1$ before the first occurrence of *block*$_2$ or if the distance between the two blocks is less than n.

Apart from the auxiliary variables used in the previous function, we introduce an additional boolean variable *flag*$_1$ (line 5), which changes from `true` to `false` when the first occurrence of *block*$_1$ is fully matched, i.e., all its individual events have been matched. In lines 9–12, while *flag*$_1$ is `true`, the algorithm checks whether the current element of the input *subtrace* is part of an occurrence of *block*$_1$. The major code is the same as the corresponding part of the previous function; however, instead of stopping with a positive result, the loop continues by setting the variable *flag*$_1$ to `false` (line 12) when the matched event is the last event of *block*$_1$. As for matching *block*$_2$, if the occurrence of the *first* event of *block*$_2$ is detected (line 13), there are two cases that may lead to a violation. Either *block*$_1$ has not been fully matched yet (i.e., variable *flag*$_1$ is `true`) or it has been fully matched but the timestamp of current trace element (that matches the first element of *block*$_2$) violates the constraint on the distance between *block*$_1$ and *block*$_2$. If one of these two conditions holds, the algorithm goes on (line 14) to match[4] the rest of *block*$_2$ (line 16), since the current element might actually not be part of a whole instance of *block*$_2$. If both of these conditions are not satisfied (line 15), it means that there is no violation, i.e., the first block has been fully matched and the distance constraint between the two blocks is satisfied; hence, there is no need to match[5] the remainder of *block*$_2$ and the algorithm returns `true`. Otherwise the algorithm invokes function *match* to match the current element with $block2[i_2]$ (line 16). The function reports a violation when *block*$_2$ is fully matched (line 17); otherwise, it returns `true` in the end (line 18).

The input of function `checkPatternPrecedenceAtMost` (Algorithm 14) is the same as function `checkPatternPrecedenceAtLeast`. This function checks whether there is an occurrence of *block*$_1$ within n time unit(s) prior to the occurrence of *block*$_2$. The brute-force checking algorithm demands tracing each occurrence of *block*$_2$ and checking whether the distance from the nearest occurrence of *block*$_1$ is no more than n. In this function, our strategy is to only examine the occurrences of *block*$_2$ that may raise violation and discontinue the matches for possible occurrences of *block*$_2$ when the time distance constraint "`at most n tu`" is already fulfilled. Hence in the auxiliary variables, we introduce *criticalInstant* to enable measuring the distance between the latest occurrence of *block*$_1$ and a matched occurrence of the first event of *block*$_2$.

Each iteration of the loop consists of two portions, to check whether the current trace element is part of an occurrence of *block*$_1$ (lines 9–13) and/or an occurrence of *block*$_2$ (lines 14–16). The function uses a duplicate algorithm for function `checkPatternPrecedenceGeneral` to match each element with *block*$_1$ (lines 9–10). If the matched event is the last of *block*$_1$ (line 11), the variable *criticalInstant* is set to the sum of the timestamp of current element and n (line 12), and the tuple (i_1, pt_1) is reset (line 13). As for matching *block*$_2$, if the current trace element is a match for the first event of *block*$_2$, and only if the timestamp of the element is greater than the value of *criticalInstant* (line 14), the tuple (i_2, pt_2) is set to $(2, t)$. That is, the algorithm only tries to match the occurrences of *block*$_2$ which may be in violation of the constraint "`at most` *n* `tu`". Otherwise if i_2 is greater than 1, function `match` is called to match $block2[i_2]$. If the matched event is the last of *block*$_2$, the function reports a violation (line 16). If there is no violation detected in the loop, the algorithm returns `true`

[4]Notice that in this case a violation is reported only if *block*$_2$ is fully matched (line 17).

[5]This is derived from the formal semantics of the `preceding` operator, in which the match of the first block, at the proper time distance, is defined as the consequent of the logical implication that formalizes the semantics of the operator.

Algorithm 14: checkPatternPrecedenceAtMost

```
Input: a trace segment subtrace and the parameters of an instance of precedence pattern of the
       form "$block_1$ preceding at most n tu $block_2$": two events (chains) $block_1$
       and $block_2$, and a maximum $n$ of the time distance between $block_1$ and $block_2$
Output: true if pattern holds on subtrace; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event
3  $firstOfBlock2 \leftarrow block_2$.first().event
4  $(i_1,pt_1) \leftarrow (1,0)$, $(i_2,pt_2) \leftarrow (1,0)$
5  $criticalInstant \leftarrow 0$
6  for $elem \in subtrace$ do
7      $e \leftarrow elem.event$
8      $t \leftarrow elem.timestamp$
9      if $e = firstOfBlock1$ then $(i_1,pt_1) \leftarrow (2,t)$
10     else if $i_1 > 1$ then $(i_1,pt_1) \leftarrow$ match$(block_1, i_1, pt_1, e, t)$
11     if $i_1 = size_1 + 1$ then
12         $criticalInstant \leftarrow t + n$
13         $(i_1,pt_1) \leftarrow (1,0)$
14     if $e = firstOfBlock2$ && $t > criticalInstant$ then $(i_2,pt_2) \leftarrow (2,t)$
15     else if $i_2 > 1$ then $(i_2,pt_2) \leftarrow$ match$(block_2, i_2, pt_2, e, t)$
16     if $i_2 = size_2 + 1$ then return false
17 return true
```

(line 17) at the end of the function.

Function `checkPatternPrecedenceExactly` (Algorithm 15) checks whether each occurrence of *block*$_2$ is preceded by an occurrence of *block*$_1$, with an exact distance *n*. The key of this algorithm is to keep the timestamps of all occurrences of *block*$_1$ that may satisfy the distance constraint and fulfilling the distance constraint when the first event of *block*$_2$ is detected. To enable this evaluation, We introduce variable *criticalInstants* (line 5) to store the timestamps and initialize it to an empty list.

The major body of the function is still a loop that checks whether each trace element of the input *subtrace* is part of an occurrence of *block*$_1$ (lines 9–13) and/or an occurrence of *block*$_2$ (lines 14–18). After trying to match the current trace element with *block*$_1$ (lines 9–10), if *block*$_1$ is already fully matched (line 11), the sum of the timestamp of current element and *n* is appended to variable *criticalInstants*. When the first event of *block*$_2$ is detected (line 15), if the timestamp of current element is included in *criticalInstants*, the function removes those items that are beyond the distance *n* from *criticalInstants*; otherwise, the tuple (i_2,pt_2) is set to $(2,t)$. Function `match` is called for matching the current trace element with $block2[i_2]$ if i_2 is greater than 1. A violation is reported when an occurrence of *block*$_2$ is detected (line 18). If there is no violation reported in the loop, the function returns `true` in the end (line 19).

Algorithm 15: checkPatternPrecedenceExactly

```
Input: a trace segment subtrace and the parameters of an instance of precedence pattern of the
       form "$block_1$ preceding exactly n tu $block_2$": two events (chains) $block_1$
       and $block_2$, and the exact time distance $n$ between $block_1$ and $block_2$
Output: true if pattern holds on subtrace; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event
3  $firstOfBlock2 \leftarrow block_2$.first().event
4  $(i_1,pt_1) \leftarrow (1,0)$, $(i_2,pt_2) \leftarrow (1,0)$
5  $criticalInstants \leftarrow []$
6  for $elem \in subtrace$ do
7      $e \leftarrow elem.event$
8      $t \leftarrow elem.timestamp$
9      if $e = firstOfBlock1$ then $(i_1,pt_1) \leftarrow (2,t)$
10     else if $i_1 > 1$ then $(i_1,pt_1) \leftarrow$ match$(block_1,i_1,pt_1,e,t)$
11     if $i_1 = size_1 + 1$ then
12         criticalInstants.append($t+n$)
13         $(i_1,pt_1) \leftarrow (1,0)$
14     if $e = firstOfBlock2$ then
15         if $t \in criticalInstants$ then $criticalInstants \leftarrow \{t' \mid t' \in criticalInstants \,\&\&\, t' > t\}$
16         else $(i_2,pt_2) \leftarrow (2,t)$
17     else if $i_2 > 1$ then $(i_2,pt_2) \leftarrow$ match$(block_2,i_2,pt_2,e,t)$
18     if $i_2 = size_2 + 1$ then return false
19 return true
```

3.4.5 Response

In the previous subsection, we have described the functions that check the four variants of the *precedence* pattern on a given trace segment. Likewise, function `checkPatternResponse` invokes one of four auxiliary functions to accomplish the check of a *response* pattern, which may contain no constraint on the time distance between the two blocks, or use one of the three comparison operators (i.e., "`at least`", "`at most`", and "`exactly`") in the constraint. In the rest of this section, we present the four auxiliary functions `checkPatternResponseGeneral` (Algorithm 16), `checkPatternResponseAtLeast` (Algorithm 17), `checkPatternResponseAtMost` (Algorithm 18), and `checkPatternResponseExactly` (Algorithm 19).

Function `checkPatternResponseGeneral` takes in input a trace segment and the parameters of an instance of a *response* pattern: $block_1$, $block_2$, and checks whether all occurrences of $block_2$ are followed by an occurrence of $block_1$. Likewise function `checkpatternprecedenceplain`, the algorithm uses a set of similar auxiliary variables: $size_1$ and $size_2$ keep track of the number of events to match in each block; *firstofblock1* and *firstofblock2* contain the event of each block's first element; the integer tuple (i_1,pt_1) (respectively (i_2,pt_2)) is used to determine whether the trace element being checked is a match of the next event in $block_1$ (respectively, $block_2$). Moreover, we designate variable *result* as the current result of the check (line 4), which is initialized to `true`, and is set to `false` (respectively `true`) when an occurrence of $block_2$ (respectively $block_1$) is found.

Algorithm 16: checkPatternResponseGeneral

```
Input: a trace segment $subtrace$ and the two events (chains) $block_1$ and $block_2$ of an instance
       of $response$ pattern of the form "$block_1$ responding $block_2$"
Output: true if $pattern$ holds on $subtrace$; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$, $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,pt_1) \leftarrow (1,0)$, $(i_2,pt_2) \leftarrow (1,0)$
4  $result \leftarrow$ true
5  for $elem \in subtrace$ do
6      $e \leftarrow elem.event$
7      $t \leftarrow elem.timestamp$
8      if $e = firstOfBlock2$ then $(i_2,pt_2) \leftarrow (2,t)$
9      else if $i_2 > 1$ then $(i_2,pt_2) \leftarrow$ match$(block_2,i_2,pt_2,e,t)$
10     if $i_2 = size_2 + 1$ then
11         $(i_2,pt_2) \leftarrow (1,0)$
12         $result \leftarrow$ false
13     if $!result$ then
14         if $e = firstOfBlock1$ then $(i_1,pt_1) \leftarrow (2,t)$
15         else if $i_1 > 1$ then $(i_1,pt_1) \leftarrow$ match$(block_1,i_1,pt_1,e,t)$
16         if $i_1 = size_1 + 1$ then
17             $(i_1,pt_1) \leftarrow (1,0)$
18             $result \leftarrow$ true
19 return $result$
```

The major body of function `checkpatternresponseplain` is a loop that iterates through the input *subtrace* to match each element with $block_2$ (lines 8–12) and with $block_1$ (lines 13–18). In the first portion of each iteration, if the current trace element matches the first event of $block_2$, the variable i_2 is set to 2 and pt_2 is updated with the current timestamp (line 8); otherwise, if i_2 is greater than 1, we invoke the auxiliary function `match` to match the trace element with the event defined at $block2[i_2]$, in consideration of a possible constraint defined on the distance from the previous event at $block2[i_2 - 1]$) (line 9). If the matched event is the last event of $block_2$ (line 10), the tuple (i_2,pt_2) is reset and variable *result* is set to `false`. That is, the result of the check will remain negative unless an occurrence of $block_1$ is found.

Hence, within each single iteration of the loop, if variable *result* is `false` (line 13), the algorithm also checks whether the current trace element is part of an occurrence of $block_1$ (lines 14–18). If the occurrence of the *first* event of $block_2$ is detected, the variable i_1 is set to 2 and pt_1 is updated with the current timestamp (line 14). Otherwise if i_2 is greater than 1, function `match` is called to check whether the current trace element is a match for $block1[i_1]$. If $block_1$ is fully matched (line 16), the tuple (i_1,pt_1) is reset and variable *result* is set to `true`.

After checking the pattern on the input *subtrace* in the loop, the algorithm returns the value of variable *result* (line 19).

Given a trace segment, function `checkPatternResponseAtLeast` (Algorithm 17) checks

```
Algorithm 17: checkPatternResponseAtLeast
Input: a trace segment subtrace and the parameters of an instance of response pattern of the
       form "$block_1$ responding at least n tu $block_2$": two events (chains) $block_1$
       and $block_2$, and a threshold $n$ of the time distance between $block_1$ and $block_2$
Output: true if pattern holds on subtrace; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event, $firstOfBlock2 \leftarrow block_2$.first().event
3  $(i_1,pt_1) \leftarrow (1,0)$, $(i_2,pt_2) \leftarrow (1,0)$
4  $criticalInstant \leftarrow 0$
5  $result \leftarrow$ true
6  for $elem \in subtrace$ do
7      $e \leftarrow elem.event$
8      $t \leftarrow elem.timestamp$
9      if $e = firstOfBlock2$ then $(i_2,pt_2) \leftarrow (2,t)$
10     else if $i_2 > 1$ then $(i_2,pt_2) \leftarrow$ match$(block_2, i_2, pt_2, e, t)$
11     if $i_2 = size_2 + 1$ then
12         $(i_2,pt_2) \leftarrow (1,0)$
13         $criticalInstant \leftarrow t + n$
14         $result \leftarrow$ false
15     if $!result$ then
16         if $e = firstOfBlock1$ && $t >= criticalInstant$ then $(i_1,pt_1) \leftarrow (2,t)$
17         else if $i_1 > 1$ then $(i_1,pt_1) \leftarrow$ match$(block_1, i_1, pt_1, e, t)$
18         if $i_1 = size_1 + 1$ then
19             $(i_1,pt_1) \leftarrow (1,0)$
20             $result \leftarrow$ true
21 return $result$
```

whether there is an occurrence of *block*$_1$ following all occurrences of *block*$_2$, fulfilling the constraint "`at least n tu`" on the distance between each pair of occurrences of the two blocks. The function takes in input a trace segment *subtrace*, the parameters of an instance of a *response* pattern: *block*$_1$, *block*$_2$, and the threshold *n* of the time distance between the two blocks.

In addition to the previous algorithm, this function introduces variable *criticalInstant* (line 4) to help check the fulfillment of the distance constraint "`at least n tu`" when an occurrence of *block*$_1$ is detected, while variable *result* is `false`. The strategy of this function is likewise function `checkPatternResponseGeneral`, except for two differences. In each iteration, when *block*$_2$ is fully matched (line 11), variable *criticalInstant* is set to the sum of the timestamp of current trace element and *n* (line 13), apart from other changes. The variable *criticalInstant* is then used to assess the constraint "`at least n tu`" when the first event of *block*$_1$ is matched. More specifically, the process of matching *block*$_1$ proceeds only is the timestamp of the trace element is also confirmed no smaller than *criticalInstant* (line 16). This additional guard ensures that the current *result* of the check is set to `true` (line 20) only if there exists such an occurrence of *block*$_1$ that fulfills the constraint on the distance from the closest occurrence of *block*$_2$. The algorithm returns the value of variable *result* (line 21), if there is no violation reported in the loop.

Algorithm 18: checkPatternResponseAtMost

```
Input: a trace segment subtrace and the parameters of an instance of response pattern of the
       form "$block_1$ responding at most n tu $block_2$": two events (chains) $block_1$
       and $block_2$, and a maximum $n$ of the time distance between $block_1$ and $block_2$
Output: true if pattern holds on subtrace; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event, $firstOfBlock2 \leftarrow block_2$.first().event
3  $(i_1, pt_1) \leftarrow (1,0)$, $(i_2, pt_2) \leftarrow (1,0)$
4  $criticalInstant \leftarrow 0$
5  $result \leftarrow$ true
6  for $elem \in subtrace$ do
7      $e \leftarrow elem.event$
8      $t \leftarrow elem.timestamp$
9      if $result$ then
10         if $e = firstOfBlock2$ then $(i_2, pt_2) \leftarrow (2, t)$
11         else if $i_2 > 1$ then $(i_2, pt_2) \leftarrow$ match$(block_2, i_2, pt_2, e, t)$
12         if $i_2 = size_2 + 1$ then
13             $(i_2, pt_2) \leftarrow (1,0)$
14             $criticalInstant \leftarrow t + n$
15             $result \leftarrow$ false
16     else
17         if $e = firstOfBlock1$ then
18             if $t <= criticalInstant$ then $(i_1, pt_1) \leftarrow (2, t)$
19             else return false
20         else if $i_1 > 1$ then $(i_1, pt_1) \leftarrow$ match$(block_1, i_1, pt_1, e, t)$
21         if $i_1 = size_1 + 1$ then
22             $(i_1, pt_1) \leftarrow (1,0)$
23             $result \leftarrow$ true
24 return $result$
```

The definition of function `checkPatternResponseAtMost` (Algorithm 18) describes the procedure of checking whether there exists an occurrence of *block*$_1$ following each occurrence of *block*$_2$, while the distance between each of them is no shorter than *n*. The algorithm uses the same set of variables as function `checkPatternPrecedenceAtLeast`, but their strategies differ due to the different semantics of the two patterns.

In the loop of the algorithm, we try to find a new occurrence of *block*$_2$ while the variable *result* is `true` (lines 9–15); otherwise, we try to find a valid occurrence of *block*$_1$ (lines 16–23) in response to the marked occurrence of *block*$_2$. The reason behind this algorithm is the fact that the distances from all the occurrences of *block*$_2$ (if exist) after a matched occurrence of *block*$_2$ are as well shorter than *n* with respect to a valid occurrence of *block*$_1$. Hence, if the variable *result* is `true` (line 9), the algorithm uses the same procedure (with respect to function `checkPatternPrecedenceAtLeast`) to check if the current trace element is part of an occurrence of *block*$_2$ (lines 10–15). Otherwise, if the current trace element is a match for the first event of *block*$_1$, the algorithm proceeds if the timestamp of current element is no greater than *criticalInstant* (line 18); or else it returns `false` (line 19), to indicate that the distance between the marked occurrence of $block_2$ and the next possible occurrence of $block_1$ is in violation of the constraint on the distance between the two blocks (i.e., "`at most n tu`"). If there is no violation of the distance constraint, the variable *result* remains `false` until a valid $block_1$ is fully matched (line 23). If there is no violation reported in the loop, the function returns the value of variable *result* (line 24).

Algorithm 19: checkPatternResponseExactly

```
Input: a trace segment $subtrace$ and the parameters of an instance of $response$ pattern of the
       form "$block_1$ responding exactly n tu $block_2$": two events (chains) $block_1$
       and $block_2$, and the exact time distance $n$ between $block_1$ and $block_2$
Output: true if $pattern$ holds on $subtrace$; false otherwise
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$, $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1, pt_1) \leftarrow (1,0)$, $(i_2, pt_2) \leftarrow (1,0)$
4  $criticalInstants \leftarrow []$
5  for $elem \in subtrace$ do
6      $e \leftarrow elem.event$
7      $t \leftarrow elem.timestamp$
8      if $e = firstOfBlock2$ then $(i_2, pt_2) \leftarrow (2, t)$
9      else if $i_2 > 1$ then $(i_2, pt_2) \leftarrow$ match$(block_2, i_2, pt_2, e, t)$
10     if $i_2 = size_2 + 1$ then
11         $(i_2, pt_2) \leftarrow (1,0)$
12         $criticalInstants$.append($t + n$)
13     if $criticalInstants$.notEmpty() then
14         if $e = firstOfBlock1$ then
15             if $t = criticalInstants$.first() then $(i_1, pt_1) \leftarrow (2, t)$
16             else return false
17         else if $i_1 > 1$ then $(i_1, pt_1) \leftarrow$ match$(block_1, i_1, pt_1, e, t)$
18         if $i_1 = size_1 + 1$ then
19             $criticalInstants$.remove($1$)
20             $(i_1, pt_1) \leftarrow (1,0)$
21 return $criticalInstants$.isEmpty()
```

Function `checkPatternResponseExactly` (Algorithm 19) defines the algorithm that checks whether each occurrence of $block_2$ is followed by an occurrence of $block_1$, with an exact time distance n. We store the timestamps of all the occurrences of $block_2$ in variable *criticalInstants* and assess the fulfillment of the constraint "`exactly n tu`" whenever an occurrence of the first event of $block_1$ is detected.

In the function, we iterate through the input *subtrace* to check whether its trace element is part of an occurrence of $block_2$ (lines 8–12) and/or an occurrence of $block_1$ (lines 15–20). The code for matching a trace element with $block_2$ (lines 8–9) is identical to the corresponding part of the previous function. If the matched event is the last of $block_2$ (line 10), besides resetting the tuple (i_2, pt_2) to

$(1,0)$, the algorithm also appends the sum of the timestamp of the matched event and n to the list *criticalInstants*. Notice that in this function we use the emptiness of the list *criticalInstants* instead of a boolean variable to indicate the current result of the check. Unless the list is empty (line 13), we try to match the current trace element with $block_1$ (lines 14–17). If the trace element is an occurrence of the first event of $block_1$ (line 14), if its timestamp is equal to the first of the variable *criticalInstants*, the pointer i_1 increments and pt_1 is set to the timestamp of the trace element (line 15); otherwise a violation is reported (line 16) to indicate there is no possible occurrence of $block_1$ which can fulfill the constraint "`exactly n tu`". Otherwise if i_1 is greater than 1, function `match` is called to check whether the current trace element is a match for $block1[i_1]$ (line 17). If the matched event is the the last of $block_1$ (line 18), namely, if an occurrence of $block_1$ has been found following the first of the marked occurrences of $block_2$ with the exact time distance n, the first item of variable *criticalInstants* is removed (line 19) and the tuple (i_1,pt_1) is reset to $(1,0)$. If there is no violation reported in the loop, the function returns the emptiness of the list *criticalInstants* (line 21).

3.5 The Approach at Work

We now show how the approach works on a simple example. Consider the trace shown in Figure 3.4 and the property "Event X shall happen at least twice before the third occurrence of event Y", which can be expressed in *TemPsy* as "`before 3` Y `eventually at least 2` X", using a *before* scope combined with an *existence* pattern.

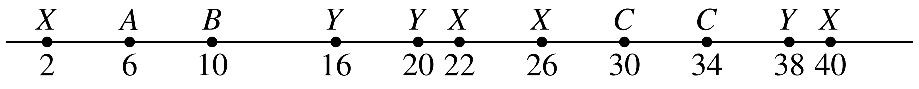


Figure 3.4: An event trace on which to evaluate the property described in Section 3.5; events are above the line, timestamps below

Checking this property on the trace using our model-driven approach is reduced to the evaluation of the OCL invariant shown in Figure 3.3; this evaluation goes as follows.

After extracting the scope and pattern from the property and assigning them to variables *scope* and *pattern* (line 3 in Figure 3.3), function `applyScopeBefore` (detailed in Algorithm 1) is invoked to select the sub-traces determined by the parameters of *scope*. In this example, parameter m is 3, the event name X is "`Y`", and parameters *op* and n are undefined because the scope has no constraint on the time distance from the scope boundary.

The statement at line 7 of Algorithm 1 will determine the timestamp of the third occurrence of event Y (38 in this case) and assign it to variable t. Since the parameter *op* is undefined, the case statement at line 17 of the algorithm will be executed, selecting the sub-trace containing events with a timestamp less than or equal to 38, i.e., the sub-trace having the event X at timestamp 2 as first event and the event Y at timestamp 38 as last event. This sub-trace is the only element contained in the list returned by Algorithm 1.

The evaluation of the OCL invariant shown in Figure 3.3 continues with the evaluation of the expression `subtraces->forAll(subtrace | checkPatternExistence (subtrace, pattern));` in this case, variable *subtraces* contains the list of sub-traces returned by function `applyScopeBefore`, as discussed above. Function `checkPatternExistence` will be invoked once (because list *subtraces* contains only one element), taking in input the sub-trace and variable *pattern*, to check the pattern over the sub-trace. In this example, for Algorithm 9, the parameter corresponding to the event name *E* is "`X`", the comparison operator *op* is "at least", and the parameter *n* is 2. The execution of the statement at line 4 in Algorithm 9 will yield 3 in the variable *count*, since there are three occurrences of event *X* in the input sub-trace. Afterwards, the value of *count* is compared to the parameter *n* using the comparison operator *op*; in this case, the algorithm will return true (since 3 > 2), indicating that the property is satisfied on this sub-trace.

Since there are no more sub-traces on which to apply function `checkPatternExistence`, the evaluation of the invariant will return *true*, indicating that the input property is not violated by the trace.

3.6 Tool Implementation

We have implemented TEMPSY-CHECK, our model-driven approach to offline trace checking of *TemPsy* properties, following the flowchart shown in Figure 3.2. The TEMPSY-CHECK tool is publicly available at `http://weidou.github.io/TemPsy-Check`.

TEMPSY-CHECK takes in input an instance of class `Trace` in the conceptual model for execution traces (Figure 3.1) and an instance of class `TemPsyExpression` corresponding to the non-terminal ⟨*TemPsyExpression*⟩ of *TemPsy*'s grammar (Figure 2.2). Both the models for execution traces and the *TemPsy* language are represented as Ecore models. More specifically, the domain-specific language *TemPsy* has been defined using the Xtext framework [Eclipse, 2015b] and the Ecore model of the DSL has been derived from its Xtext definition. We leveraged the code generation facility provided by the Eclipse Modeling Framework (EMF) to generate the Java implementation of the Ecore models.

In the implementation, we have developed file readers for loading trace instances and *TemPsy* expressions. TEMPSY-CHECK can load trace instances from log files either in CSV or XMI format; it can be extended to support other formats thanks to its design following the *Strategy Pattern*. The tool takes in input a list of *TemPsy* expressions (defined using the textual notation shown in Fig. 2.2) and converts them into an XMI-based format to load at run time. Moreover, we have developed a Java class `ConstraintFactory` to help build OCL constraints corresponding to the input *TemPsy* expressions. In the OCL constraints, we use those auxiliary OCL functions that implement the algorithms described in the previous subsections, to check the input *TemPsy* expressions on the trace instance (see also Section 3.2). The evaluation of the OCL invariants is done using the OCL checker included in Eclipse OCL [Eclipse, 2015a].

3.7 Evaluation

In this section we report on the evaluation of TEMPSY-CHECK. The evaluation focuses on the scalability of the tool, to assess the relationship between the time taken to check a property on a trace and the structural properties of the trace (e.g., length, distribution of events) and the type of property to check. We also compare the performance of TEMPSY-CHECK with a state-of-the-art alternative technology.

3.7.1 Experiment settings

We have conducted our evaluation using a benchmark consisting of a subset of the properties extracted from the requirements specification documents of the eGovernment application developed by our partner, described in Section 2.6.2. Out of the 47 properties documented in the case study, we left out of the benchmark the nine properties using the *after-until* pattern. Properties of this type can be rewritten using the *between-and* scope, possibly in conjunction with an *after* scope: for this reason, they would not have provided additional insights to our scalability analysis. The 38 properties used for the evaluation are listed in a sanitized form in Table 3.1. The actual textual description of each property has been omitted for confidentiality reasons; for each property we only detail its structure in terms of *scope* + *pattern*. The events involved in the property (e.g., “a citizen requests a certificate”) are denoted using uppercase letters.

These properties have been checked on *synthesized* traces. We use synthesized traces instead of real ones because: 1) based on our experience, real traces are often inadequate to cover a large range of trace lengths and a variety of properties; 2) we wanted to have great diversity in terms of occurrences of patterns in the traces, while being able to control this diversity; 3) real traces are valuable to assess fault detection capabilities, while in our case we focus on the scalability of the approach; 4) if we had used real traces, they could not be shared for forming a public benchmark, even when sanitized. By using synthesized traces we are able to control in a systematic way the factors (such as trace length, sub-trace(s) length and position, frequency and distance of events) that could impact the execution time for a specific type of property. At the same time, we are also able to randomly set other factors, to avoid any bias.

To synthesize these traces we implemented a trace generator program. This program allows for generating diverse (in terms of size, patterns, scopes, event positions and frequency) and realistic traces, without introducing bias. The generator takes in input a property, the desired length of the trace to generate and additional parameters depending on the type of property given in input and the factors one wants to control. To determine the position in the trace of the events occurring in the input property, the generator takes into account the temporal and timing constraints prescribed by the semantics of the scope and the pattern used in the property. Positions in the trace that are deemed not relevant for the evaluation of the property are filled with a dummy event. The details of the trace generation strategy depend on the scope and pattern used in the properties and are discussed in the next subsections. As an additional contribution of the thesis, we also make available in the TEMPSY-CHECK GitHub repository the artifacts used in the evaluation, to contribute to the building of a public repository of case studies for evaluating trace checking/run-time verification procedures.

The next three subsections report on the checking of properties using, respectively, the *globally*, *before/after*, and *between-and* scopes. For each group of properties we first describe the trace generation strategy and then present and discuss the results. The section ends with a discussion of the results and of the threats to validity. Notice that out of the three types of scope considered for the evaluation, the properties using a *globally* scope represent the most challenging in terms of scalability, since the semantics of this scope guarantees that the pattern (used in the property to check) will be evaluated through the entire length of the trace.

Moreover, to assess scalability, we also need a baseline of comparison. Such baseline should be the best available tool that can be considered an alternative to TEMPSY-CHECK. We identified such a

Table 3.1: *TemPsy* properties used for the evaluation

P1: globally always *A*
P2: globally never *B*
P3: globally eventually at least 2 *A*
P4: globally eventually at most 3 *A*
P5: globally *A* responding at most 1000 tu *B*
P6: globally *A* responding exactly 1000 tu *B*
P7: globally *A* preceding at most 6000 tu *B*
P8: globally *A* preceding at least 100 tu *B*
P9: globally *A* preceding exactly 100 tu *B*
P10: globally *A*, *B* preceding at least 1000 tu *C*, *D*
P11: globally *A* responding at least 1000 tu *B*, *C*
P12: globally *A* responding *B*
P13: before *A* eventually *B*
P14: before 3 *A* eventually at least 2 *B*
P15: before 2 *A* never *B*
P16: before *A* *B* responding at most 3000 tu *C*
P17: before *A* at least 1000 tu *B* responding at least 1000 tu *C*
P18: before *A* *B*, # at most 6000 tu *C* preceding *D*
P19: before 3 *A* *B*, # at least 1000 tu *C* preceding *D*
P20: before *A* *B* preceding *C*
P21: after *A* at most 5000 tu eventually *B*
P22: after *A* always *B*
P23: after 2 *A* exactly 5000 tu eventually *B*
P24: after *A* *B* responding at least 1000 tu *C*
P25: after *A* *B* preceding at most 3000 tu *C*, *D*
P26: after 2 *A* at most 3000 tu *B* preceding *C*, *D*
P27: after 2 *A* never *B*
P28: after *A* at most 1000 tu *B* responding at most 10 tu *C*
P29: after *A* *B* preceding at least 2000 tu *C*
P30: after *A* eventually at most 6 *B*
P31: after 2 *A* at least 5000 tu eventually *B*
P32: between *A* and *B* always *C*
P33: between *A* at least 1000 tu and *B* at least 500 tu never *C*
P34: between *A* and *B* *C* responding at least 1000 tu *D*
P35: between *A* and *B* never exactly 2 *C*
P36: between 3 *A* and *B* always *C*
P37: between *A* at least 1000 tu and 2 *B* *C* preceding at least 1000 tu *D*
P38: between 2 *A* and 2 *B* eventually at most 10 *C*

tool among the participants to the "offline monitoring" track of the first international Competition on Software for Runtime Verification [Bartocci et al., 2014] (CSRV 2014), held in September 2014 as a satellite event of the 14th International conference on Runtime Verification (RV'14). Out of the four tools (RiTHM2 [Navabpour et al., 2013], MonPoly [Basin et al., 2012], STePr, QEA [Barringer et al., 2012]) qualified for the final round of the competition, RiTHM2 and STePr were not publicly available[6] at the time of writing. Between the remaining two, we chose MONPOLY over QEA because only the former supports a real specification language (MFOTL, a metric first-order temporal logic) that is conceptually close to *TemPsy*. On the other hand, QEA does not support any input language and uses an automata-based formalism: the user has to write a Java program that builds the automaton corresponding to the property to check. To perform the comparison with MONPOLY, we manually translated the properties into MFOTL formulae; these formulae are also available in the TEMPSY-CHECK GitHub repository. We remark that our goal, in this comparison, is not to fare better than existing technology, but to verify that an MDE approach to offline trace checking is viable from a scalability standpoint.

The results reported in this section have been measured on a desktop computer with a 3 GHz Intel Dual-Core i7 CPU and 16GB of memory, running Eclipse DSL Tools v. 4.6.0M3 (Neon Milestone 3), JavaSE-1.7 (Java SE v. 1.8.0_25-b17, Java HotSpot (TM) 64-Bit Server VM v. 25.25-b02, mixed mode), Eclipse OCL v. 6.0.1, and MONPOLY v. 1.1.6. All measurements reported correspond to the average value over 100 runs of the check procedure (on the same trace, for the same property).

3.7.2 Properties using the *globally* scope

Properties defined with the *globally* scope are the most important for assessing the scalability of our approach with respect to the trace length. Indeed, the semantics of this scope requires the tool to check the property pattern through the *entire* trace, while in the case of the other scopes, property patterns are checked only on some segments of the input trace (i.e., on sub-traces). In our collection of properties there are 12 using the scope *globally*, in combination with various patterns; they correspond to properties P1–P12 listed in Table 3.1.

3.7.2.1 Research Questions

For this type of properties, given that they are the most challenging in terms of scalability, we address the following research questions:

RQ-G1) What is the relation between the execution time of the trace checking procedure and the length of a trace?

RQ-G2) What are the types of pattern most taxing on the execution time?

RQ-G3) How does TEMPSY-CHECK compare with MONPOLY in terms of execution time?

3.7.2.2 Trace generation strategy

In the case of the *globally* scope the generation of the trace is determined only by the semantics of the pattern used in the property.

For the *universality* pattern, we repeat the event occurring in it through the entire trace.

[6] The first version of RITHM is available but it only supports run-time verification of C programs. As for STEPR, no reference is available in the competition report [Bartocci et al., 2014] or online.

For the *existence* pattern, we first determine the number n of occurrences to generate, based on the bound indicated in the property. If the bound is expressed as "`at least` m" or "`at most` m" we randomly generate n with a uniform distribution on the range $[m, \text{trace length}]$, respectively $[0, m]$; if the bound is expressed as "`exactly` m", n is set to m. Afterwards, we randomly generate (with a uniform distribution on the range $[1, \text{trace length}]$) n positions in the trace where to put the occurrences of the event specified in the property.

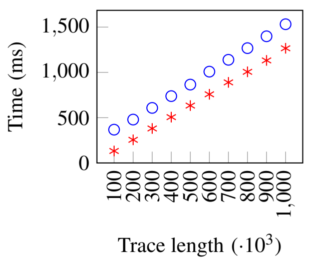
For the *absence* pattern, if the property has the form `never A`, the trace is generated without any occurrence of the event A. If the property has the form `never exactly` m `A`, we randomly generate n with a uniform distribution on the range $[0, \ldots, m-1, m+1, \ldots, \text{trace length}]$.

In the case of a property containing a *precedence* or *response* pattern, we generate a number of occurrences of events (involved in the property) equal to 10% of the length of the trace. This value has been selected based on the frequency of events observed in the application whose requirements are expressed through the properties shown in Table 3.1. The simplest case is for a property like `globally B responding at most 10 tu A`: assuming a trace length of 1M, we would generate 50K occurrences of the pattern (i.e., pairs of `A` and `B`), for a total of 100K occurrences of `A` and `B`. More complex cases have to take into account the event chains used in the property. For the distribution of the occurrences of the pattern across the trace we have to consider the distance between events. For example, for the property aforementioned, each occurrence of the response pattern would span over at most 10 time units; this is the maximum distance between an occurrence of `A` and the corresponding occurrence of `B`. The number of pattern occurrences to generate and the maximum time span of each pattern occurrence are the parameters used to randomly allot the pattern occurrences over the trace, according to a uniform distribution.

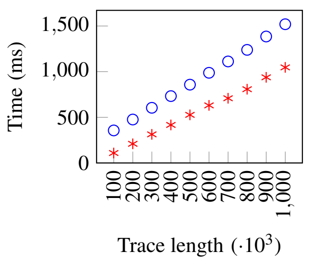
3.7.2.3 Evaluation

We run the trace checking procedure for properties P1–P12; each property was checked on ten different traces, with length (i.e., number of events) varying from 100K to 1M. The twelve plots in Figure 3.5 depict the execution time of TEMPSY-CHECK (denoted by ○) and of MONPOLY (denoted by $*$) for each of the properties P1–P12, for different trace lengths. The execution time for both tools has been measured using the `time` Unix command.

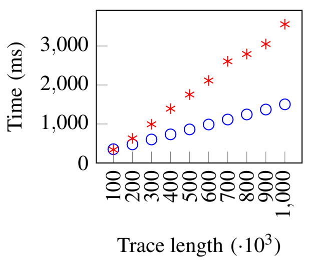
We answer RQ-G1 by observing that the time taken by TEMPSY-CHECK ranges from about one hundred milliseconds to a bit more than two seconds, and increases linearly with the length of the trace, depending on the type of property. To answer RQ-G2, the results show that the properties more taxing on the execution time are those with a *response* or *precedence* pattern (e.g., P5, P6, P7, P9, P11). Regarding RQ-G3, we observe that the time taken by MONPOLY ranges from about one hundred milliseconds to a bit less than eight seconds, and is also linear with respect to the length of the trace. MONPOLY takes longer for checking properties with a *(bounded) existence* pattern (e.g., P3, P4) and with a *precedence* pattern that contains a distance constraint of type "`at least`" (e.g., P10). We can answer RQ-G3 stating that, except for the case of properties P3, P4, and P10, the two tools perform almost similarly, with absolute differences between execution times that are quite small (less than one second). In the case of properties P3, P4, and P10, TEMPSY-CHECK performs much better than MONPOLY. A possible explanation for the slower time of MONPOLY for these properties could be the structure of the corresponding MFOTL formulae, which contain several nested temporal operators to express the "eventually at least/at most" pattern (P3, P4) and an event chain (P10).



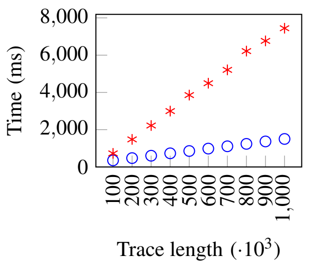
(a) P1



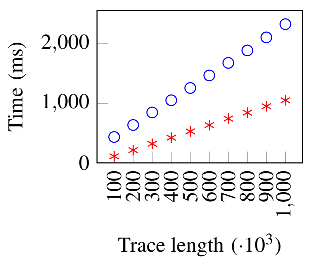
(b) P2



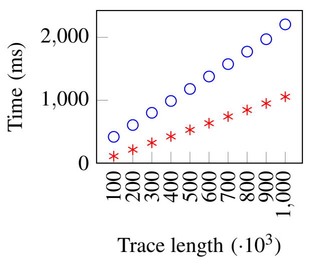
(c) P3



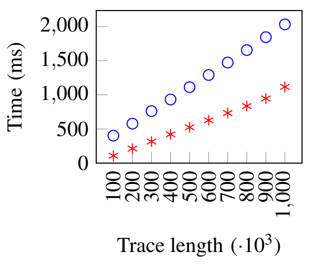
(d) P4



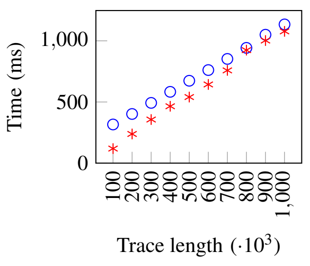
(e) P5



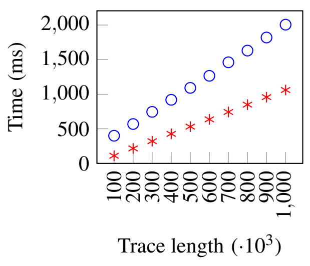
(f) P6



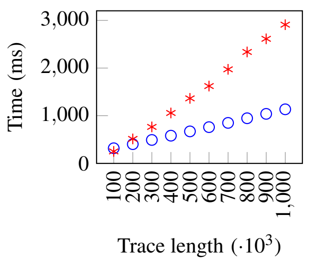
(g) P7



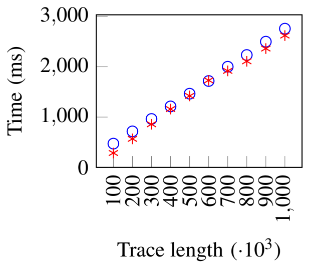
(h) P8



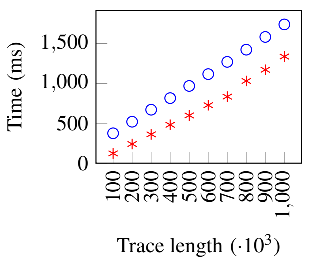
(i) P9



(j) P10



(k) P11



(l) P12

Figure 3.5: Comparison between the execution time of TEMPSY-CHECK (○) and of MONPOLY (∗) for properties with the *globally* scope

The execution times discussed above include not only the time to perform the actual check, but also the time to parse/load the trace to check[7]. As shown in Figure 3.6, the average trace loading time for TEMPSY-CHECK, measured through instrumentation, ranges from 55 ms to 550 ms, growing linearly for various trace lengths. Notice that for checking a single property on a trace with TEMPSY-CHECK, the trace loading time can take, for larger traces, from one-fourth to one-third of the total execution time. Although these values for the trace loading time can seem high, we expect the loading time not to impact on the total execution time in the case of *batch property checking*, i.e., checking multiple properties at the same time on a trace. Checking in batch mode a set of properties, rather than individual ones, is common in enterprise scenarios in which, for example, the set of properties to check is decided by the entity that has invoked a business process [Baresi and Guinea, 2005].

To further investigate this aspect, we compared the execution time of TEMPSY-CHECK and MONPOLY for batch checking ten properties (P3–P12), over ten traces, with length ranging from 1M to 10M. These traces have been obtained by concatenating the traces used for the experiment described above, and by renaming the events within each trace being concatenated, to avoid name clashes. We

[7]The trace loading time is not available in the output of MONPOLY.

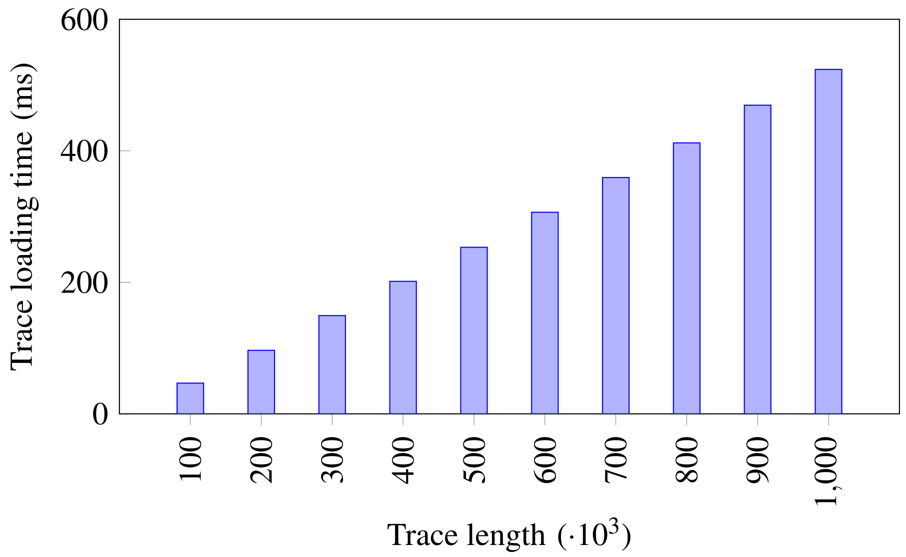


Figure 3.6: Trace loading time of TEMPSY-CHECK for traces with various lengths

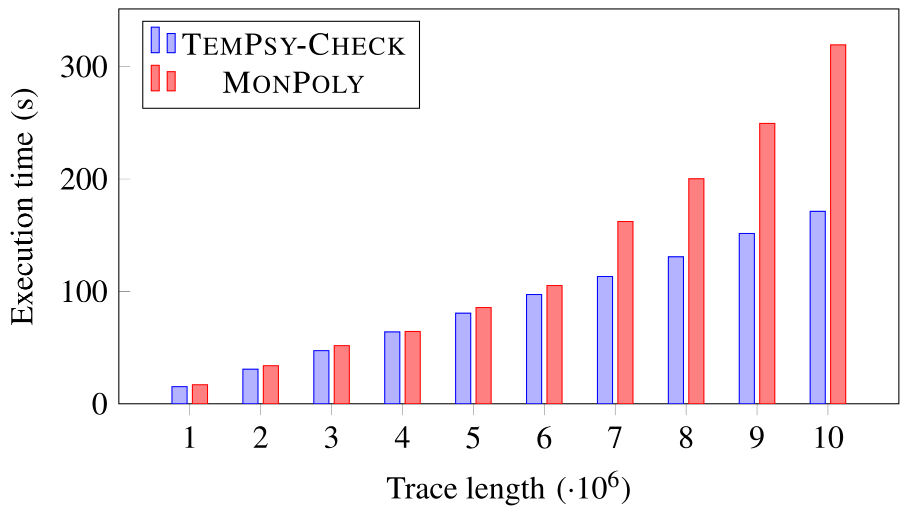


Figure 3.7: Comparison of the execution time for the batch checking of ten properties with the *globally* scope

executed TEMPSY-CHECK by providing in input the list of the ten properties to check. We executed MONPOLY by providing in input one formula corresponding to the conjunction of the ten formulae equivalent to properties P3–P12. Figure 3.7 shows the result of the comparison: the performance of the two tools are similar for traces of length up to six millions; over this threshold, MONPOLY gets slower.

3.7.3 Properties using the *before/after* scope

Properties defined using the *before/after* scopes, differently from the ones using a *globally* scope, have to be checked only on the portion of the trace delimited by the scope boundary. Hence, their scalability does not relate in a direct way with the length of the trace. Nevertheless, they can help us assess whether and how the type of property (e.g., the scope used within the property) impacts on the total execution time. We have checked eight properties with the *before* scope (properties P13–P20 in Table 3.1) and eleven properties with the *after* scope (properties P21–P31 in Table 3.1).

3.7.3.1 Research Questions

For this type of properties, to assess how the type of scope used in them impacts on the total execution time, we address the following research questions:

RQ-BEAF1) What is the relation between the time to compute the boundary of the scope and the position of the boundary?

RQ-BEAF2) What are the types of scope most expensive to compute?

Notice that we do not compare with MONPOLY since the concept of "scope" is not a first-class object in MFOTL formulae.

3.7.3.2 Trace generation strategy

As remarked above, for this type of properties the scalability of the checking procedure does not relate in a direct way with the length of the trace. Hence, for both types of scopes, we fix the length of the generated trace to 100K. To answer the research questions above, we vary the length of the sub-trace as determined by the scope boundary, i.e., we vary the position of the boundary event in the trace. In the case of properties with a *before* scope, the boundary event is placed in positions from 10K to 100K, with a 10K step increment; similarly, for properties with an *after* scope, the position of the boundary event varies from 10K to 90K, with a 10K step increment.

For properties referring to a specific occurrence of an event in their scope part, such as `before 3 B`...or `after 4 A`..., we only control the position of the actual scope boundary (e.g., the third occurrence of `B` or the fourth occurrence of `A` in the examples above). The other previous occurrences of the boundary event are generated in random positions using a uniform distribution ove the range $[0, \text{position of the boundary}]$ (for properties with a *before* scope), and over the range $[\text{position of the boundary}, \text{trace length}]$ (for properties with an *after* scope).

The generation of the patterns corresponding to the actual properties follows the steps described in Section 3.7.2.2.

3.7.3.3 Evaluation

We instrumented TEMPSY-CHECK to report the time taken to compute the boundary of a scope (i.e., to determine the sub-trace on which to check each property pattern), hereafter referred to as *scope time*, as well as the time to check the pattern on the sub-trace, hereafter referred to as *pattern time*. More specifically, scope time corresponds to the time taken to evaluate expressions of type `applyScope*S*` in Figure 3.3, while pattern time corresponds to the time taken to evaluate expressions of type `checkPattern*P*` in Figure 3.3. For each property and trace, we calculated scope time by subtracting pattern time from the total execution time; we measured pattern time by

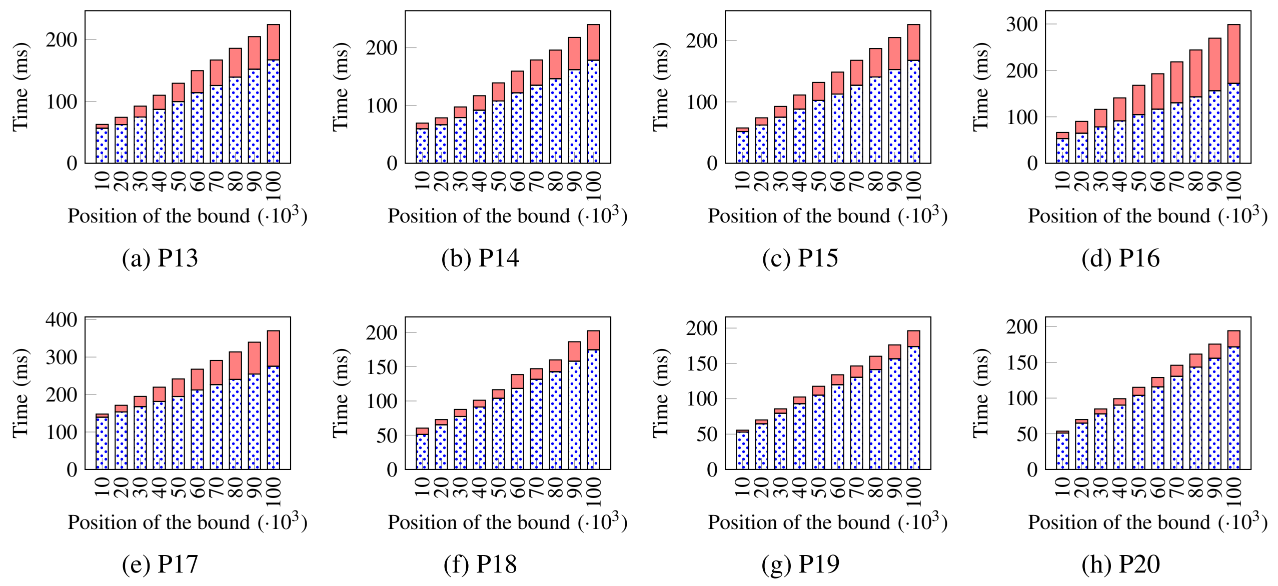


Figure 3.8: Scope time ▨ and pattern time 🟥 of TEMPSY-CHECK for checking properties with a *before* scope

first replacing the property scope with the *globally* scope and then checking the new property on the sub-trace determined by the original scope.

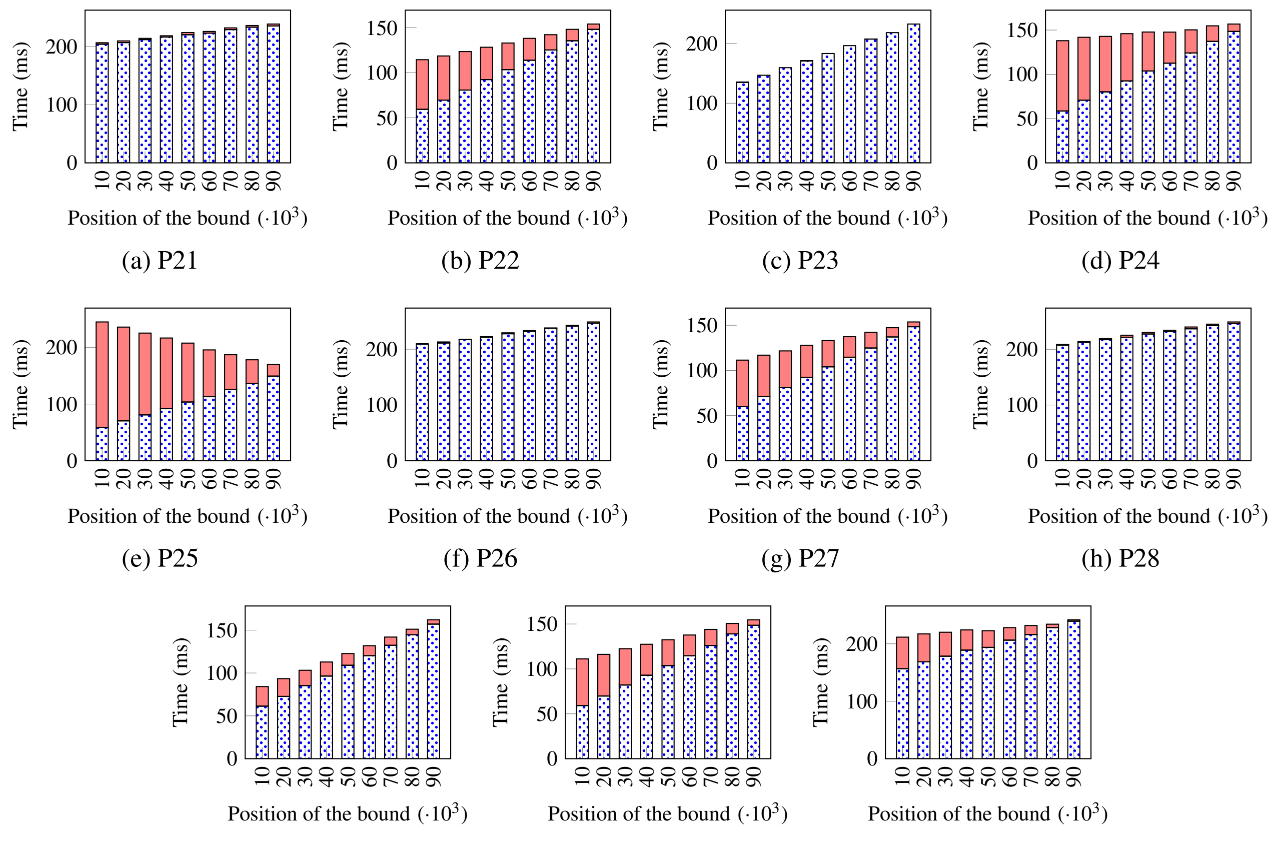
Figure 3.8 and 3.9 show the scope time (denoted by ▨) and the pattern time (denoted by 🟥) for checking, respectively, properties P13–P20 (with a *before* scope) and property P21–P31 (with an *after* scope), when varying the position of the scope boundary. Notice that while in the case of a *before* scope a higher position of the bound corresponds to a longer length of the sub-trace, in the case of an *after* scope a lower position of the bounds corresponds to a longer length.

To answer RQ-BEAF1, we observe from the plots that both in the case of the *before* scope and in the case of the *after* scope, the scope time grows linear with respect to the position of the scope boundary. This is due to the increase of the length of the sub-trace delimited by the scope boundary.

We answer RQ-BEAF2 by looking at the scope time for properties P17, P21, P23, P26, P28, P31. These properties are the most taxing in terms of scope time because the scope boundary is defined with a distance constraint. This is particularly true for the cases in which the boundary is defined using an "`at most`" constraint (see P21, P26, and P28).

3.7.4 Properties using the *between-and* scope

Properties with a *between-and* scope, similarly to the ones with a *before/after* scope, are checked on a portion of trace provided in input. Depending on the variant of this scope, the portion of the trace on which properties are checked might include one or more segments. The scopes used in properties P32–P35 can potentially select multiple segments on a trace, while the scopes in properties P36–P38 select exactly one segment on a trace, as determined by the specific event occurrence used in the scope boundaries (e.g., as in the case of `between 3 A and 2 B`).



(a) P21 (b) P22 (c) P23 (d) P24 (e) P25 (f) P26 (g) P27 (h) P28 (i) P29 (j) P30 (k) P31

Figure 3.9: Scope time ▨ and pattern time 🟥 of TEMPSY-CHECK for checking properties with an *after* scope

3.7.4.1 Research Questions

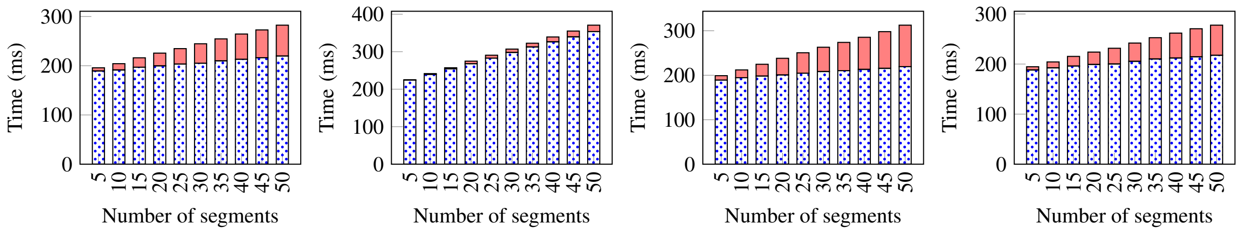
For this type of properties, given the two variants of the *between-and* scope, we address the following research questions:

RQ-BA1) For the scope variant that can select multiple segments on the trace, given a fixed length for the segments, what is the relation between the number of segments and the time to compute the scope?

RQ-BA2) For the scope variant that can select multiple segments on the trace, given a fixed number of segments, what is the relation between the length of the segment and the time to compute the scope?

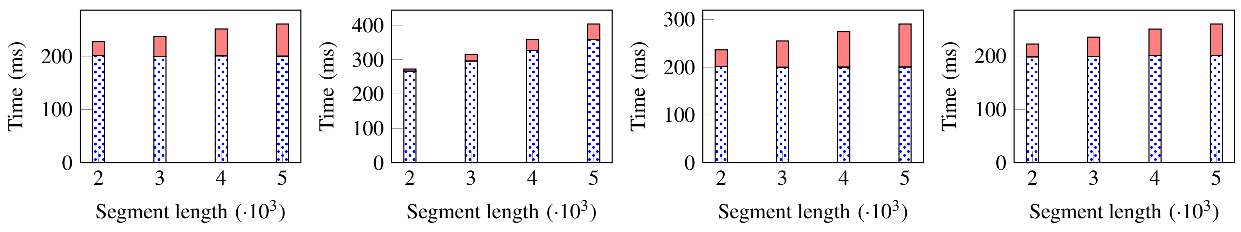
RQ-BA3) For the scope variant that can select only a single segment, given a fixed length for this segment, what is the relation between the position of the segment and the time to compute the scope?

RQ-BA4) For the scope variant that can select only a single segment, given a fixed position of this segment, what is the relation between the length of the segment and the time to compute the scope?



(a) P32 (b) P33 (c) P34 (d) P35

Figure 3.10: Scope time ▨ and pattern time 🟥 of TEMPSY-CHECK for checking properties with a *between-and* scope (multiple segments, fixed length)



(a) P32 (b) P33 (c) P34 (d) P35

Figure 3.11: Scope time ▨ and pattern time 🟥 of TEMPSY-CHECK for checking properties with a *between-and* scope (fixed number of segments, various lengths)

Notice that also in this case we do not compare with MONPOLY because the concept of "scope" is not a first-class object in MFOTL formulae.

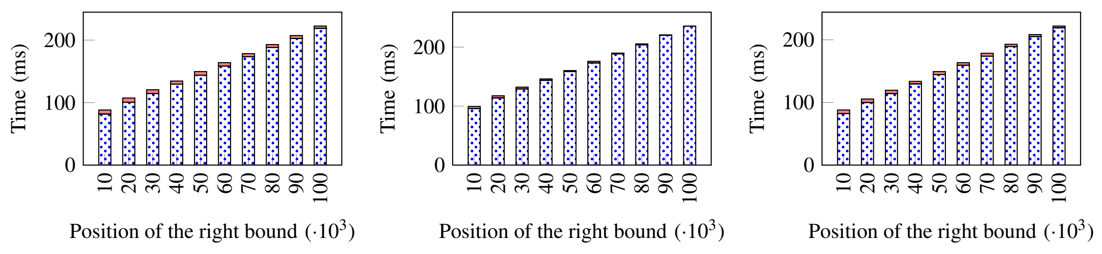
3.7.4.2 Trace generation strategy

For both types of *between-and* scope variants, we fix the length of the generated trace to 100K. To answer RQ-BA1 and RQ-BA2 we consider properties P32–P35. For these properties, we control two parameters for the trace generation: the length L of each segment selected by the scope and the number of segments N. By fixing L to 2000, we can split the 100K trace into 50 segments. The generator varies the number N of actual segments to generate from 5 to 50, with a 5-step increment. By fixing N to 20, and assuming a minimum length of 2000 for a segment (given the time constraints in P33), the generator produces traces with segments of length varying from 2000 to 5000, with 1K-step increment.

To answer RQ-BA3 and RQ-BA4 we consider properties P36–P38. For these properties we control two parameters: the length L' of the segment and the position P of one of its bounds. By fixing L' to 10K, we vary the position of the right bound from position 10K to position 100K with 10K-step increment, i.e., we vary the position of the segment in the trace. By fixing the position P to 10001, we can vary L' from 10000 to 90000, with 10K-step increments.

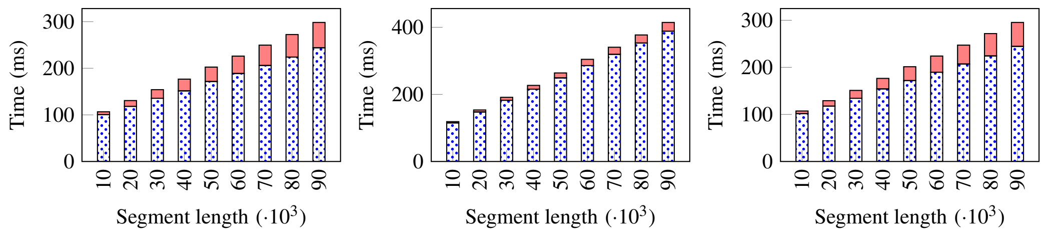
3.7.4.3 Evaluation

As done above for the case of properties with a *before/after* scope, we also distinguish between scope time and pattern time for checking properties with a *between-and* scope.



(a) P36 (b) P37 (c) P38

Figure 3.12: Scope time ▨ and pattern time 🟥 of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment of fixed length, different positions)



(a) P36 (b) P37 (c) P38

Figure 3.13: Scope time ▨ and pattern time 🟥 of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment, various lengths)

To answer RQ-BA1 we observe the plot in Figure 3.10. The scope time for properties P32–P35 when varying the number of segments (as determined by the scope) on which to check the property pattern, slightly increases with the number of segments to consider; the higher scope time for property P33 is due to the presence of a time distance constraint for the (left) scope boundary.

We answer RQ-BA2 by looking at the plot in Figure 3.11. In the case of checking properties P32–P35 when fixing the number of segments to 20 and varying the segment length from 2000 to 5000, the scope time is almost constant (about 200 ms) for all properties but P33, because of the time distance constraint for the (left) scope boundary.

The answer to RQ-BA3 can be found by looking at the plot in Figure 3.12. In the case of checking properties P36–P38 when varying the position of the segment on which the property pattern is checked and keeping the segment length constant, the scope time increases linearly with respect to the position of the segment.

We answer RQ-BA4 by observing the plot in Figure 3.13. In the case of checking properties P36–P38 when varying the length of the segment, the scope time increases linearly with respect to the length of the segment.

3.7.5 Discussion

The results presented in the previous subsections have shown the feasibility of applying our model-driven approach for offline trace checking in realistic settings.

Our TEMPSY-CHECK tool is a viable technology from a performance standpoint as it can analyze very large traces (with one million events) in about two seconds. The tool scales linearly with respect to the length of the input trace to check. Notice that "the input trace to check" can correspond also to a sub-trace of an actual, larger execution trace. This can be the case for properties referring to events occurring in time windows (see, for example, the service provisioning patterns presented in [Bianculli et al., 2012]). In these cases, one can first isolate from the original trace the window of interest and then feed the latter to our tool.

We have also compared the performance of our implementation to MONPOLY, a comparable, state-of-art tool. Despite the fact that MONPOLY is a tool that implements a dedicated algorithm [Basin et al., 2008] for trace checking of temporal logic properties, our TEMPSY-CHECK tool (which relies on a generalist OCL checker) not only achieves similar results, but in some cases it also performs better than MONPOLY.

We also remark that writing some of the properties in MFOTL was challenging (despite previous knowledge of MFOTL), much more than when using *TemPsy*. This challenge could be overcome by defining properties in *TemPsy* and then providing an automatic translation to MFOTL formulae or, dually, by building a system of property specification patterns on top of MFOTL. In both cases, one would have satisfied one of our requirements (R1, see Chapter 1) and could have then relied on MONPOLY for trace checking. While this could be in principle a viable approach, it would not fulfill another requirement (R2, see Chapter 1), which entails to rely on standard and stable MDE technology for checking temporal properties. We remark that these requirements are not specific to this project, but are more general because 1) analysts may not be able to handle the mathematical background required by temporal logic; and 2) there are many contexts where solutions have to be engineered by using standardized MDE technologies.

Overall, we can conclude that a model-driven approach to offline trace checking of realistic temporal properties is viable, even on very large traces, and compares favorably with the state of the art.

3.7.5.1 Threats to validity

The main threat to the validity of the results presented above is the intrinsic presence of errors in the toolchain we developed. We tried to compensate for this by thoroughly testing the checker with traces and properties for which the oracle was previously known. Another potential threat is the fact that we have performed trace checking on *synthesized* traces. Real execution traces might be different, in terms of events occurrences and time distances. However, this threat does not affect our research question on scalability, as we want to analyze the execution time as a function of a number of parameters (e.g., trace length), while varying randomly other aspects (e.g., position of certain events). As explained at the beginning of this section, for that purpose, synthesized traces are better than real ones as they guarantee we have the data to perform our analysis by controlling certain factors and varying others randomly. Nevertheless, real traces (with faults in the system) could be helpful to assess the cost-benefit of the proposed trace checking procedure; this is out of the scope of this thesis. Finally, as for the comparison with MONPOLY, we remark that its specification language (MFOTL) is more expressive than *TemPsy* (see also Section 2.5), hence the performance of MONPOLY could have been negatively affected by the more complex implementation needed to support a richer specification language. Moreover, the MFOTL properties that we wrote to perform the comparison described in Section 3.7.2 could be written in a different, but semantically-equivalent form that could lead to different results. For example, properties P3 and P4 contain several nested operators, which impact on the checking performance of MONPOLY. We tried to mitigate this aspect by having the MFOTL formulae written by a person with ten years of experience in formal specification (and verification) with temporal logics. Furthermore, we believe that in practice, it might be hard anyway for practitioners (with limited background in temporal logic) to find out what is the optimal way to express a property in MFOTL.

Chapter 4

Model-driven Violation Reporting for Trace Checking

In the previous chapter, we presented a scalable model-driven procedure for checking execution traces of *TemPsy* properties. Nevertheless, when a trace is in violation of a *TemPsy* property, the binary result (i.e., `false`) provided by TEMPSY-CHECK does not provide any clue about the reasons for the violation(s). To complement that procedure, in this chapter, we present TEMPSY-REPORT, a model-driven approach to systematically collecting violation information from a trace in violation of *TemPsy* properties. We also present an interactive visualization tool for navigating and analyzing the violation information collected by TEMPSY-REPORT.

In the rest of the chapter, we first characterize *TemPsy* violations in Section 4.1; in Section 4.2 we give an overview of the model-driven approach for collecting violation information from a faulty trace; we describe how the violation information is collected from the trace with OCL functions defined on the trace model in Section 4.3; we give a brief introduction of the implementation of TEMPSY-REPORT in Section 4.4. We conclude the chapter with the description of the visualization tool for understanding *TemPsy* violations and the evaluation of the scalability of the TEMPSY-REPORT tool.

4.1 Characterization of *TemPsy* Violations

In this section, we characterize the temporal violations that can occur for *TemPsy* properties. According to the definition of *TemPsy* patterns, we classify the violations into eight types, as defined with examples below. We consider an execution trace composed of a list of trace elements which are delimited by commas and enclosed in a pair of brackets; each trace element is represented as a pair consisting of the event name and a timestamp.

UNOC *UNexpected OCcurrence*. This type of violations are triggered by unexpected occurrences of the event specified in an *occurrence* pattern, i.e., *absence* or *existence*. The *absence* pattern, by definition, can be violated by any occurrence of the event specified in the pattern. It is also the case for the variant (with `exactly` as the comparison operator) of the *absence* pattern, when the number of event occurrences in a trace equals the number specified in the pattern. For the *existence* pattern out of the four variants, two can be violated by an unexpected occurrence, i.e., the variants that contain `at most` or `exactly` in the constraint on the number of occurrences. For instance, given a trace $[(a,2),(a,3),(a,5)]$ and a *TemPsy* property "`globally eventually at most 2` *a*", the third trace element is unexpected and hence violates the property.

NSOC *No-Show OCcurrence*. This type of violations takes place at the trace elements which should have been occurrences of the event specified in a *universality* or *existence* pattern. An `NSOC` violation must be reported whenever a trace element does not match the event specified in a *universality* pattern. In the case of the *existence* pattern, when the comparison operator is `exactly`/`at least` or is omitted, an `NSOC` violation has to be reported if the actual number of occurrences of the event is less than what specified in the property. For instance, given a trace $[(a,2),(b,3),(b,5)]$ and a *TemPsy* property "`globally eventually at least 2` *a*", an `NSOC` violation should be reported after the first trace element, since the number of occurrences of event *a* is less than two.

NSOR *No-Show ORder*. This type of violations is triggered when the first block of events does not occur in accordance with the order defined by an *order* pattern, i.e., *precedence* or *response*. For instance, if we check property "`globally` *a* `preceding` *b*" on a trace $[(b,2),(a,3),(c,5)]$, an `NSOR` violation should be reported since there is no occurrence of event *a* before the occurrence of event *b*.

WTO *Wrong Temporal Order*. This type of violations is specific to an *order* pattern (i.e., *precedence* or *response*) that contains a constraint on the time distance between the two blocks specified in the pattern. In such a pattern, the constraint on the distance between the two blocks is specified in the form of "`[at least | at most | exactly]` *n* `tu`"; we call *n* the *critical distance*. Given the timestamp *t* of an occurrence of the second block of a *precedence* pattern (respectively, a *response* pattern), we call instant $t-n$ (respectively, $t+n$) its *critical instant*. In general, an *order* pattern is satisfied when all the the following conditions hold for all occurrences of the second block: C1) given an occurrence of the second block, there exists one occurrence of the first block that complies with the order defined in the pattern; C2) given an occurrence of the second block, there exists one corresponding occurrence of the first block that satisfies the constraint on the distance between the two blocks (if defined); C3) given an occurrence of the second block, there exists one corresponding occurrence of the first block that satisfies all the distance constraints within the first block (if defined). A `WTO` violation is triggered when C2 fails to hold for an occurrence of the second block. If the failure of either condition C2 or C3 for an occurrence of the second block correlates with multiple occurrences of the first block, we only consider the closest one(s) beside the *critical instant*. For instance, given a trace $[(a,2),(b,6),(a,7),(b,10)(c,15)]$ and a property "`globally` *a*`, #at least 3 tu` *b* `preceding at most 2 tu` *c*", although there are two occurrences of the first block (i.e., the events chain defined by "*a*`, #at least 3 tu` *b*") precedes the second block (i.e., event *c*), the distances from them to the occurrence of event *c* are more than 3, which violates the constraint "`at most 2 tu`"; thus we report a `WTO` violation with the offending occurrence of event *c* and the second occurrence of the events chain *a*, *b*, which is closer to the critical instant $15-2$. For space reasons, we will not use multiple occurrences of the first block in the following examples unless necessary.

WTC *Wrong Temporal Chain*. This type of violations is specific to an *order* pattern that contains at least one constraint on the time distance between two consecutive events specified in the first block. A

`WTC` violation is triggered when condition C3 (as defined above) fails to hold for an occurrence of the second block. For instance, given a trace $[(a,2),(b,3),(c,5)]$ and a property "`globally` *a*`, #at least 3 tu` *b* `preceding` *c*", the chain of events *a* and *b* does occur before event *c*, but the constraint "`at least 3 tu`" is violated by the time distance between the first two trace elements, and therefore a `WTC` violation is triggered.

`WTOC` *Wrong Temporal Order and Chain.* This type of violations is specific to an *order* pattern that contains not only a constraint on the distance between the two blocks but at least one constraint on the distance between two consecutive events within the first block. A `WTOC` violation is triggered when both conditions C2 and C3 (as defined above) fail to hold for an occurrence of the second block. For instance, given a trace $[(a,2),(b,3),(c,9)]$ and a property "`globally` *a*`, #at least 3 tu` *b* `preceding at most 2 tu` *c*", the constraint "`at least 3 tu`" within the first block and the constraint "`at most 2 tu`" between the two blocks are both violated. In this case, a `WTOC` violation should be reported instead of `WTO` or `WTC`.

`LVRI` *Left Valid and Right Invalid*[1]. This type of violations is specific to an *order* pattern that meets the same criteria defined for `WTOC`. A `LVRI` is triggered when C2 fails to hold with respect to an occurrence the second block, and C3 holds for the closest occurrence of the first block on the left of the *critical instant* but fails to hold for the closest occurrence of the first block on the right of the *critical instant*. For instance, given a trace $[(a,2),(b,5),(a,7),(b,8),(c,10)]$ and a property "`globally` *a*`, #at least 2 tu` *b* `preceding at most 4 tu` *c*", though the time distance between the first occurrence of the pair of event *a* and *b* is valid, the constraint "`at most 4 tu`" between the two blocks fails to hold with respect to the occurrence of event *c*; in contrast, the second occurrence of the first block violates the distance constraint "`at least 2 tu`" specified in the block, though it is within 4 time units from the occurrence of the event *c*; and hence, an `LVRI` violation should be reported.

`LIRV` *Left Invalid and Right Valid.* This violation type is dual to `LVRI`, which is triggered when C2 fails to hold for an occurrence of the second block, and C3 fails to hold for the closest occurrence of the first block on the left of the *critical instant* but holds for the closest occurrence of the first block on the right of the *critical instant*. For instance, given a trace $[(a,2),(b,3),(a,5),(b,8),(c,10)]$ and a property "`globally` *a*`, #at least 2 tu` *b* `preceding exactly 4 tu` *c*", both the occurrences of the first block violate the constraint "`exactly 4 tu`" and only the right one conforms to the constraint defined within the first block. Hence, an `LIRV` violation should be reported.

Notice that when condition C2 fails to hold for an occurrence of the second block and C3 fails to hold for the two closest occurrences of the first block on both sides of the *critical instant*, we group such violations into `WTOC`.

4.2 Overview of the Approach

Our procedure for collecting violation information from a trace in violation of *TemPsy* properties is sketched in Figure 4.1, which follows a similar workflow as TEMPSY-CHECK (Figure 3.2). In a

[1]An occurrence of an events chain that satisfies all the time distance constraints in the chain is referred to as a *valid* occurrence, otherwise we call it an *invalid* occurrence if any of the time distance constraints is violated.

regular scenario, our procedure is indeed invoked after the trace checking procedure reports a negative outcome. Rather than instantiating the models for the trace and *TemPsy* language from raw data, TEMPSY-REPORT takes in input the same trace instance and *TemPsy* expressions. The key step of the approach is to evaluate an OCL query on the trace instance. The evaluation of this query, which can be done using standard OCL tools, is semantically equivalent to collecting violation information from the trace instance with respect to the *TemPsy* properties provided in input.

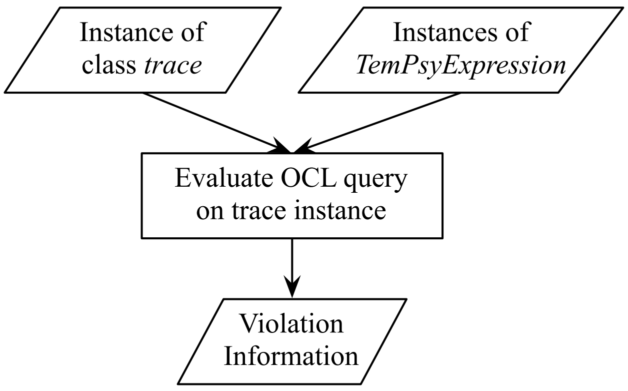

Instance of class *trace*
Instances of *TemPsyExpression*
Evaluate OCL query on trace instance
Violation Information

Figure 4.1: Overview of the approach to violation information collection

```
1 let property:TemPsy::TemPsyExpression = self.properties->at(P1),
2     subtraces:Sequence(Tuple(begin:Integer, end:Integer)) = P2(property
        .scope)
3 in  subtraces->iterate(
4       subtrace:Tuple(begin:Integer, end:Integer);
5       result:Sequence(Tuple(begin:Integer, end:Integer, violations:
           OclAny)) = Sequence{} |
6       let newViolations:OclAny = P3(subtrace.begin, subtrace.end,
           property.pattern) in
7         if newViolations->notempty() then
8           result->append(
9             Tuple{begin:Integer = subtrace.begin,
10                  end:Integer = subtrace.end,
11                  violations:OclAny = newViolations})
12        else result endif)
```

Figure 4.2: The template for OCL queries on a trace for collecting violation information with respect to *TemPsy* expressions

As shown in Figure 4.2, we have defined a template for various OCL queries on the `Trace` class, to collect the violation information from a trace instance with respect to the associated *TemPsy* expressions, which conceptually corresponds to applying the semantics of the pattern used in the property on a set of sub-traces, as defined by the scope used in the property. The template contains three placeholders that are underlined in the "`let...in`" clauses, i.e., P1, P2, and P3. The placeholder P1 at the first line represents a positive integer, which is used by the attribute `self.properties` (through the function `at`) to access the the *TemPsy* property to be investigated. At line 2, we invoke an auxiliary function of the form `applyScope*S*` that is represented by the placeholder P2. This function takes the scope used in the property (accessed through the expression `property.scope`) as input and returns a list of sub-traces, each of which is denoted by the positions of the two boundaries, as defined by the scope semantics. Depending on the type of the scope, there are five auxiliary functions to substitute for the placeholder P2, namely, `applyScopeGlobally`, `applyScopeBefore`, `applyScopeAfter`, `applyScopeBetweenAnd`, and `applyScopeAfterUntil`. Since the definition of these OCL functions are identical[2] to the ones presented in Chapter 3, we will not repeat the explanation in this section. In lines 3–12, we navigate each sub-trace to collect the violation information, which is stored in a list of triples at line 5. Each triple consists of two integers (i.e., the variable *begin* and *end*) that indicate the boundaries of each trace segment, and the information violation of type `OclAny`. To collect the violation information within each sub-trace, at line 6, we invoke another auxiliary function of the form `reportPattern*P*` that is represented by the placeholder P3. This function takes as input the two boundaries of a sub-trace and the pattern used in the property (accessed through the expression `property.pattern`) and returns the information about the violations. With respect to the pattern type, there are five functions to substitute for the placeholder P3, i.e., `reportPatternUniversality`, `reportPatternExistence`, `reportPatternAbsence`, `reportPatternPrecedence`, and `reportPatternResponse`. Notice that the super type `OclAny` does not imply that the five functions return the same type of violation information. As explained in the next subsection, in fact, the data type varies according to the violation types referring to a specific pattern. In each iteration, unless there is violation found in the sub-trace, in lines 8–11, a new triple that contains the violation information is appended to the result.

In Section 4.3, we explain the `reportPattern*P*` functions; for readability and space reasons, all the code snippets are presented as pseudocode and only the *globally* scope is used in the examples of *TemPsy* properties.

4.3 OCL Functions for collecting violation information

In this section, we present the pseudocode of the `reportPattern*P*` functions that are used to collect violation information for the five *TemPsy* patterns. These OCL functions take as input a pair of integers representing the boundaries of a sub-trace and an instance of a *TemPsy* pattern, and return the violation information in the sub-trace. Note that though we use the super type `OclAny` to receive the output of these functions in the template of OCL queries (Figure 4.2), the specific result type varies according to the semantics of the given *TemPsy* pattern.

4.3.1 Universality

By definition, a *universality* pattern can lead only to `NSOC` violations. As shown in Algorithm 20, we collect all the locations at which the event specified in the pattern does not occur. In reference to the super type `OclAny` used in the template (Figure 4.2) for receiving the result of the functions `reportPattern*P*`, the specific type of the function `reportPatternUniversality` is `Sequence(Integer)`.

```
Algorithm 20: reportPatternUniversality
    Input: begin, end: the boundaries of a sub-trace; pattern: an instance of the universality
           pattern of the form "always E"
    Output: result: a list of locations at which violations occur
1   $E \leftarrow$ event name in pattern
2   $result \leftarrow []$
3   for $i \leftarrow begin$ to $end$ do
4       if $self.traceElements[i] \neq E$ then result.append(i)
5   return result
```

Table 4.1: Violation types of the *existence* pattern

Pattern representative	Condition	NSOC	UNOC
`eventually` E	$\lvert\{E\}\rvert = 0$	+	
`eventually at least` n E	$\lvert\{E\}\rvert < n$	+	
`eventually at most` n E	$\lvert\{E\}\rvert > n$		+
`eventually exactly` n E	$\lvert\{E\}\rvert < n$	+	
	$\lvert\{E\}\rvert > n$		+

Legend. $\lvert\{E\}\rvert$: the number of occurrences of event E in a given trace segment.

4.3.2 Existence

As shown in Table 4.1, according to the number of occurrences of the event specified in an *existence* pattern, the violations can be classified as either `NSOC` or `UNOC` type. The procedure for collecting violation information for the *existence* pattern is shown in Algorithm 21. The procedure takes as input the boundaries of a sub-trace and an instance of the *existence* pattern, and returns a pair comprised of a list of locations at which violations occur and the type of the violations. In lines 5–7, all the occurrence locations of event E are retrieved from the sub-trace. Depending on the threshold n of the bound on the number of occurrences of event E, we determine the violation type and the offending occurrences (lines 9–15). If the comparison operator is present and the number of occurrences is less than n, we collect `NSOC` violations with all the occurrence locations of event E (line 11) when the comparison operator is either "`at least`" or "`exactly`" (line 10). Otherwise if the number of the occurrences exceeds n and the comparison operator is either "`at most`" or "`exactly`" (line 12), we collect `UNOC` violations with the additional locations after the n-th occurrence of event E (line 13). Or if the pattern has no constraint on the number of occurrences and there is no occurrence of event E, the procedure returns an `NSOC` violation with an empty list.

4.3.3 Absence

The *absence* pattern about an event E can be specified in two forms, i.e., "`never` E" or "`never exactly` n E", in which n is a natural number. When a sub-trace violates one of these two forms of *absence* pattern, it is because all the occurrences of event E are not allowed, and thus the violation

[2]The algorithms of the OCL functions are semantically identical to the ones described in Chapter 3, though the result type of the functions are different.

Algorithm 21: reportPatternExistence

```
Input: begin, end: the boundaries of a sub-trace; pattern: an instance of the existence pattern
       of the form "eventually [op n] E"
Output: result: a pair which contains a violation type and a list of locations upon which
        violations occur
E ← event name in pattern
op ← comparison operator of the bound on the number of occurrences of event E
n ← threshold of the occurrence number of event E
result ← null, locations ← []
for i ← begin to end do
    if self.traceElements[i] = E then
        locations.append(i)
count ← locations.size()
if op ≠ null then
    if count < n && op ≠ "at most" then
        result ← (NSOC, locations)
    else if count > n && op ≠ "at least" then
        result ← (UNOC, locations[n + 1, count])
else if count = 0 then result ← (NSOC, [])
return result
```

Algorithm 22: reportPatternAbsence

```
Input: begin, end: the boundaries of a sub-trace; pattern: an instance of the absence pattern of
       the form "never [exactly n] E"
Output: result: a list of locations at which violations occur
E ← event name in pattern
result ← []
for i ← begin to end do
    if self.traceElements[i] = E then result.append(i)
return result
```

is classified as the UNOC type. Hence, in Algorithm 22, the procedure collects all the occurrence locations of event *E* (lines 3–4).

4.3.4 Precedence

In Section 4.1, we classify all the possible violations of the *precedence* and *response* patterns into six types, i.e., NSOR, WTO, WTC, WTOC, LIRV, and LVRI. Function reportPatternPrecedence defines the algorithm for collecting violation information of a *precedence* pattern from a faulty trace segment. It is realized by four auxiliary functions reportPatternPrecedenceGeneral (Algorithm 23), reportPatternPrecedenceAtLeast (Algorithm 25), reportPatternPrecedenceAtMost (Algorithm 26), and reportPatternPrecedenceExactly (Algorithm 27), depending on the definition of the constraint on the distance between the two blocks specified in the *precedence* pattern.

Algorithm 23: reportPatternPrecedenceGeneral

```
Input: $begin$, $end$: the boundaries of a sub-trace; $block_1$, $block_2$: the two blocks of an instance
       of the precedence pattern of the form "$block_1$ preceding $block_2$"
Output: $result$: a list of 3-tuples, each of which consists of a violation type, a list of the
        locations of offending occurrences of $block_2$, and a list of the locations of
        corresponding occurrences of $block_1$
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$  $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $flag \leftarrow$ true, $l_w \leftarrow 0$
5  $result \leftarrow []$
6  for $i \leftarrow begin$ to $end$ do
7      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
8      if $e = firstOfBlock1$ then $(i_1,t_1,flag) \leftarrow (2,t,$true$)$
9      else if $i_1 > 1$ then $(i_1,t_1,flag) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag$, $e$, $t$)
10     if $l_w > 0$ && $!flag$ then $(i_1,t_1) \leftarrow (1,0)$
11     else if $i_1 = size_1 + 1$ then
12         if $flag$ then break
13         else
14             $l_w \leftarrow i$
15             $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
16     if $e = firstOfBlock2$ then $(i_2,t_2) \leftarrow (2,t)$
17     else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match($block_1$, $i_1$, $t_1$, $e$, $t$)
18     if $i_2 = size_2 + 1$ then
19         if $l_w = 0$ then $v \leftarrow$ (NSOR, $[i]$, $[]$)
20         else $v \leftarrow$ (WTC, $[i]$, $[l_w]$)
21         $result$.append($v$)
22         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
23 return $result$
```

Function `reportPatternPrecedenceGeneral` (Algorithm 23) defines the algorithm for the variant of *precedence* pattern which contains no constraint on the distance between the two blocks. This *precedence* pattern indicates that each occurrence of $block_2$ must be preceded by an occurrence of $block_1$). Notice that each block can be either an atomic event or a chain of events with optional constraints on the time distances between two consecutive events. This function takes in input the two boundaries of a sub-trace and the two blocks of an instance of the *precedence* pattern (denoted by $block_1$ and $block_2$). Given an instance of this pattern, there are two possible types of violations (i.e., `NSOR` and `WTC`) in a faulty trace segment; and `WTC` is only possible if $block_1$ contains a distance constraint. The function returns a list of violation information, each of which consists of a violation type, a list of the locations of offending occurrences of $block_2$, as well as a list of the positions of the corresponding occurrences of $block_1$.

In the function, we store the size of $block_1$ (respectively, $block_2$) in variable $size_1$ (respectively, $size_2$) (line 1), and the first event defined in $block_1$ (respectively, $block_2$) in variable *firstOfBlock1* (respectively, *firstOfBlock2*) (line 2). We define a tuple (i_1,t_1) (respectively, (i_2,t_2)) to determine whether the trace element being matched is part of an occurrence of $block_1$ (respectively, $block_2$) (line 3). More specifically, element i_1 (respectively, i_2) stores the position within $block_1$ (respectively, $block_2$) of the next event to be matched; element t_1 (respectively, t_2) stores the timestamp of the previous trace element matched at $block_1[i_1 - 1]$ (respectively, $block_1[i_2 - 1]$). In addition, we use variable *flag* to identify whether the ongoing match of $block_1$ is consistent with the distance constraints (if defined) within the block; and if an occurrence of $block_1$ is found invalid of the constraints, the variable l_w is updated with the location (line 4). The variable *result* is used to accumulate the violation information that has been found (line 5).

The major part of the function is a loop that navigates the input sub-trace, trying to match each trace element between *begin* and *end* with $block_1[i_1]$ (lines 8–15) and with $block_2[i_2]$ (lines 16–22). According to the semantics of this *precedence* pattern, the function terminates only if a valid occurrence of $block_1$ is matched. In each iteration, we first use two local variables (e and t) to store the event name and timestamp of the current trace element (line 7). If the trace element is a match for the first event of $block_1$, the function sets the position i_1 to 2, assigns the timestamp of the trace element to t_1, and resets the *flag* to `true` (line 8). Otherwise if variable i_1 is greater than 1, the function invokes another auxiliary function `matchSecondaryBlock` (Algorithm 24) to check whether the current trace element is part of $block_1$ (line 9).

Algorithm 24: matchSecondaryBlock

Input: an events chain *block*, a 3-tuple (*i*, *pt*, *flag*) of which *i* ($i > 1$) stores the position (within *block*) of the event to be checked, *pt* stores the timestamp of the previous trace element if it was a match for the event defined at $block[i-1]$, and *flag* indicates whether the (optional) constraints on the distances within *block* hold on the matched elements, a trace element (e, t) to be matched with $block[i]$

Output: $(i+1, t, flag$ && `true`$)$ if the trace element is a valid match for $block[i]$; $(i+1, t, flag$ && `false`$)$ if the trace element matches the event defined at $block[i]$ but violates the constraint on the distance between $block[i-1]$ and $block[i]$; $(1, 0, flag)$ otherwise

```
1 if $e = block[i].event$ then
2     $op \leftarrow block[i].timeDistance.op$
3     $t' \leftarrow pt + block[i].timeDistance.value$
4     $flag \leftarrow flag$ && compare ($t$, $op$, $t'$)
5     return $(i+1, t, flag)$
6 else return $(1, 0, flag)$
```

Function `matchSecondaryBlock` takes in input six parameters: an events chain *block*; a 3-tuple comprised of i ($i > 1$), *pt*, and *flag*, of which i stores the position (within *block*) of the event to be checked, *pt* stores the timestamp of the previous trace element (if it was a match for $block[i-1]$), and *flag* indicates whether the (optional) constraints on the distances within *block* hold on the matched elements; and the two parameters of a trace element (e,t) to be matched with $block[i]$. The function updates the 3-tuple $(i, pt, flag)$ if the input element is a match for $block[i]$ (lines 1–5); or else it sets the tuple to $(1, 0, flag)$ (line 6). More specifically, if the trace element is a valid match for $block[i]$ (line 5), the position i is incremented, the variable pt is set to the timestamp of the trace element, and $flag$ is set to the logical conjunction of itself and the satisfaction of the constraint on the distance defined between $block[i-1]$ and $block[i]$ (line 4); or else the function returns $(1,0,flag)$.

At line 9 of Algorithm 23, function `reportPatternPrecedenceGeneral` updates the variables i_1, t_1, $flag_1$ with the result of function `matchSecondaryBlock`, and continues to check whether the matched event is part of an invalid occurrence of $block_1$. If there is already an invalid occurrence of $block_1$ (i.e., $l_w > 0$) and the matched event violates the constraint on the distance between $block_1[i_1-1]$ and $block_1[i_1]$, the algorithm resets the tuple (i_1,t_1) for another round of match for a valid occurrence of $block_1$ (line 10). If it is not the case and $block_1$ is fully matched, if variable $flag_1$ is `true`, the function ceases the collection of violation information, namely it becomes unnecessary to continue since the pattern will always be satisfied even if $block_2$ occurs after this *valid* occurrence of $block_1$ (line 12); otherwise, the variable l_w is updated with the position i of this *invalid* occurrence of $block_1$ (line 14) and the variables i_1,t_1,i_2,t_2 are reset (line 15).

In each iteration, if the function has not yet found a valid occurrence of $block_1$, it also checks whether the current trace element is part of an occurrence of $block_2$ (lines 16–22). If an occurrence of the first event of $block_2$ is detected, the variable i_2 is set to 2 and variable t_2 is set to the timestamp of current trace element (line 16). Otherwise, if variable i_2 is already greater than 1, the algorithm calls function `match` (see Algorithm 12 introduced in Section 3.4.4) to match the current trace element with $block_2[i_2]$ (line 17).

The algorithm reports a violation when $block_2$ is fully matched (lines 18–22). More specifically, if there is no invalid occurrence of $block_1$ found yet (i.e., l_w is still 0), an `NSOR` violation is reported, accompanied by a list that contains the location of current trace element (indicating this offending occurrence of $block_2$) and an empty list indicating there is no corresponding occurrences of $block_1$ (line 19); otherwise a `wtc` violation is reported with $[i]$ and a list that contains the location $[l_w]$ of the corresponding invalid occurrence of $block_1$ (line 20). The recognized violation is then added to the *result* (line 21) and the tuples (i_1,t_1) and (i_2,t_2) are reset (line 22). The function returns variable *result* after investigating the entire trace segment (line 23).

Function `reportPatternPrecedenceAtLeast` defines the algorithm for collecting violations from a faulty trace segment which violates a *precedence* pattern that contains a constraint using the comparison operator "`at least`" on the distance between its two blocks. The function takes in input the two boundaries of a sub-trace and the parameters of an instance of the *precedence* pattern: $block_1$, $block_2$, and the threshold n of the time distance between them, and returns a list of violations with the same type as of function `reportPatternPrecedenceGeneral`.

This variant of the *precedence* pattern prescribes that each occurrence of $block_2$ must be preceded by an occurrence of $block_1$, with at least a distance equal to n. Given a faulty trace segment, we recognize an offending occurrence of $block_2$ when any of the following conditions holds: 1) there is no *valid* occurrence of $block_1$ before the occurrence of $block_2$; 2) the distance between the farthest *valid* occurrence of $block_1$ and the occurrence of $block_2$ is less than n; According to the characterization of violations, there are five possible violation types: `NSOR`, `WTO`, `WTC`, `WTOC`, and `LIRV`.

For instance, given a *precedence* pattern "`a, #at most 10 tu b preceding at least 5 tu c`" and a trace segment $[(a,8),(c,10),(a,15),(b,30),(c,32),(c,40)]$, the function navigates

Algorithm 25: reportPatternPrecedenceAtLeast

```
Input: begin, end: the boundaries of a sub-trace; block_1, block_2, n: the parameters of an
       instance of the precedence pattern of the form "block_1 preceding at least n tu block_2"
Output: result: a list of 3-tuples, each of which consists of a violation type, a list of the
        location of offending occurrences of block_2, a list of the locations of corresponding
        occurrences of block_1
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$   $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $(l_c,t_c) \leftarrow (0,0)$, $(l_w,t_w) \leftarrow (0,0)$
5  $flag_1 \leftarrow$ true, $flag_2 \leftarrow$ false
6  $result \leftarrow []$
7  for $i \leftarrow begin$ to $end$ do
8      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
9      if $l_c = 0$ then
10         if $e = firstOfBlock1$ then $(i_1,t_1,flag_1) \leftarrow (2,t,$true$)$
11         else if $i_1 > 1$ then
12             $(i_1,t_1,flag_1) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag_1$, $e$, $t$)
13         if $l_w > 0$ && $!flag_1$ then $(i_1,t_1) \leftarrow (1,0)$
14         else if $i_1 = size_1 + 1$ then
15             if $flag_1$ then $(l_c,t_c) \leftarrow (i,t)$
16             else $(l_w,t_w) \leftarrow (i,t)$
17             $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
18     else if $t - t_c \geq n$ then break
19     if $e = firstOfBlock2$ then
20         $(i_2,t_2) \leftarrow (2,t)$
21         $flag_2 \leftarrow (t_w > 0)$ && $(t - t_w < n)$
22     else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match($block_1$, $i_1$, $t_1$, $e$, $t$)
23     if $i_2 = size_2 + 1$ then
24         if $l_c = l_w = 0$ then $v \leftarrow$ (NSOR, $[i]$, $[]$)
25         else if $l_w = 0$ then $v \leftarrow$ (WTO, $[i]$, $[l_c]$)
26         else if $flag_2$ then $v \leftarrow$ (WTOC, $[i]$, $[l_w]$)
27         else if $l_c = 0$ then $v \leftarrow$ (WTC, $[i]$, $[l_w]$)
28         else $v \leftarrow$ (LIRV, $[i]$, $[l_c, l_w]$)
29         $result$.append($v$)
30         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
31 return $result$
```

the trace segment and collects three violations. First, an NSOR violation is collected at the first occurrence of the second block (i.e., event c), since no occurrence of the first block (i.e., the events chain defined by "a, #at most 10 tu b") is detected before that occurrence. At the second occurrence of event c, a WTOC violation is collected since the occurrence of the first block violates the distance constraint "`at most 10 tu`" as well as the threshold of the distance between the two blocks (i.e., "`at least 5 tu`"). The latter constraint is then satisfied at the third occurrence of event c, and hence a `WTC` violation is collected.

In the function, besides the variables $size_1$, $size_2$, *firstOfBlock1*, *firstOfBlock2*, (i_1,t_1), (i_2,t_2), which have the same usage as the ones in Algorithm 23, we use a tuple (l_c,t_c) (respectively, (l_w,t_w)) to record the latest valid (respectively, invalid) occurrence of $block_1$. More specifically, element l_c (respectively, l_w) keeps track of the location of the latest valid (respectively, invalid) occurrence of $block_1$; element t_c (respectively, t_w) stores the timestamp of the latest valid (respectively, invalid) occurrence of $block_1$. In addition, we use variable $flag_1$ to identify whether the ongoing match of $block_1$ is consistent with the distance constraints (if defined) within the block; and use variable $flag_2$ to identify whether the closest occurrence of $block_1$ breaks not only the distance constraints within $block_1$ but the constraint on the distance between the two blocks with respect to an occurrence of $block_2$ (line 5).

The function uses a loop to navigate the input sub-trace, trying to match each trace element with $block_1[i_1]$ (lines 10–17) and with $block_2[i_2]$ (lines 19–30). In accordance with the semantics of this *precedence* pattern, the function ceases searching for $block_1$ once a valid occurrence of $block_1$ has been found. Hence if variable l_c is still 0 (line 9), if the trace element is a match for the first event of $block_1$, the function sets the position i_1 to 2, assigns the timestamp of the trace element to t_1, and resets the *flag* to `true` (line 10); otherwise if variable i_1 is already greater than 1, the function invokes the auxiliary function `matchSecondaryBlock` to check whether the current trace element is a match for $block_1[i_1]$ (line 12). The function resets the variables i_1 and t_1 if variable l_w is already set to a positive position and the matched event violates the constraint on the distance from $block_1[i_1-1]$ and $block_1[i_1]$ (line 13). Otherwise if the matched event is the last event of $block_1$, the function assigns the position and timestamp of current trace element to (l_c,t_c) if variable is true (line 15), or otherwise to (l_w,t_w) (line 16) and resets the variables i_1,t_1,i_2,t_2 (line 17). If a valid occurrence of $block_1$ has been found (i.e., $l_c > 0$) and the distance between that occurrence to current element is more than or equal to n (line 18), namely the pattern is not able to be violated anymore, the function ceases the loop.

In the loop, the function continues to check whether the current trace element matches a valid occurrence of $block_2$ (lines 19–30). If the first event of $block_2$ is found (line 19), the function moves on to matching the second event of $block_2$ (line 20) and sets variable $flag_2$ to the `true` if variable t_w is positive and the distance between the invalid occurrence of $block_1$ and the current trace element is less than n (line 21). Otherwise if the pointer i_2 is greater than 1, we invoke function `match` (Algorithm 12) to match the current trace element with $block_2[i_2]$ (line 22). The function collects a violation if the matched event is the last of $block_2$ (line 23). More specifically, an `NSOR` violation is reported with a list containing the location of current trace element and an empty list if there is no occurrence of $block_1$ before this occurrence of $block_2$ (line 24); otherwise, a `WTO` violation is reported with $[i]$ and a list containing the location of the corresponding valid occurrence of $block_1$ (i.e., $[l_c]$), if there is no invalid occurrence of $block_1$ (line 25); otherwise, a `WTOC` violation is reported with $[i]$ and the location of the invalid occurrence of $block_1$ (i.e., $[l_w]$) if the variable $flag_2$ is `true`, meaning thats the constraint "`at least` n `tu`" is violated by the first (invalid) occurrence of $block_1$ (line 26); otherwise, a `WTC` violation is reported with $[i]$ and $[l_w]$, if there is no valid occurrence of $block_1$ (line 27); or else, an `LIRV` violation is reported with $[i]$ and a list containing both the invalid and valid occurrences of $block_1$ $[l_w,l_c]$ (line 28). The recognized violation is then appended to the *result* (line 29) and the tuples (i_1,t_1) and (i_2,t_2) are reset (line 30). The function returns the violations stored in *result* after the loop (line 31).

Algorithm 26: reportPatternPrecedenceAtMost

```
Input: $begin$, $end$: the boundaries of a sub-trace; $block_1$, $block_2$, $n$: the parameters of an
       instance of the precedence pattern of the form "$block_1$ preceding at most $n$
       tu $block_2$"
Output: $result$: a list of 3-tuples, each of which consists of a violation type, a list of the
        locations of offending occurrences of $block_2$, and a list of the locations of
        corresponding occurrences of $block_1$
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$  $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $(l_c,t_c) \leftarrow (0,0)$, $(l_w,t_w) \leftarrow (0,0)$
5  $flag_1 \leftarrow$ true, $flag_2 \leftarrow$ false
6  $result \leftarrow []$
7  for $i \leftarrow begin$ to $end$ do
8      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
9      if $e = firstOfBlock1$ then $(i_1,t_1,flag_1) \leftarrow (2,t,$true$)$
10     else if $i_1 > 1$ then $(i_1,t_1,flag_1) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag_1$, $e$, $t$)
11     if $i_1 = size_1 + 1$ then
12         if $flag_1$ then $(l_c,t_c) \leftarrow (i,t)$
13         else $(l_w,t_w) \leftarrow (i,t)$
14         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
15     if $e = firstOfBlock2$ && $(t_c = 0 \mid\mid t - t_c > n)$ then
16         $(i_2,t_2) \leftarrow (2,t)$
17         $flag_2 \leftarrow (t_w > 0)$ && $(t - t_w > n)$
18     else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match($block_1$, $i_1$, $t_1$, $e$, $t$)
19     if $i_2 = size_2 + 1$ then
20         if $l_c = l_w = 0$ then $v \leftarrow$ (NSOR, $[i]$, $[]$)
21         else if $l_w < l_c$ then $v \leftarrow$ (WTO, $[i]$, $[l_c]$)
22         else if $flag_2$ then $v \leftarrow$ (WTOC, $[i]$, $[l_w]$)
23         else if $l_c = 0$ then $v \leftarrow$ (WTC, $[i]$, $[l_w]$)
24         else $v \leftarrow$ (LVRI, $[i]$, $[l_c, l_w]$)
25         $result$.append($v$)
26         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
27 return $result$
```

Function `reportPatternPrecedenceAtMost` defines the algorithm for collecting violation information from a faulty trace segment which violates a *precedence* pattern that contains a constraint using the comparison operator "`at most`" on the distance between the two blocks. The function takes as input the two boundaries of a sub-trace and the parameters of an instance of the *precedence* pattern: *block*$_1$, *block*$_2$, and the maximum *n* of the time distance between the two blocks. The function returns a list of violation information, each of which consists of a violation type, a list that contains the locations of occurrences of $block_2$ where violations occur, and the locations of corresponding occurrences of $block_1$.

This *precedence* pattern specifies that each occurrence of $block_2$ must be preceded, within n time units, by an occurrence of $block_1$. Hence, an instance of this pattern can be violated when one of the following conditions holds: 1) there is no occurrence of $block_1$ before an occurrence of $block_2$; 2) the occurrences of $block_1$ are all beyond n time units prior to an occurrence of $block_2$. According to the characterization of violations, given a faulty trace segment, there are five possible violation types: `NSOR`, `WTO`, `WTC`, `WTOC`, and `LVRI`.

The main part of the function is the loop that navigates the input sub-trace, trying to match each trace element with $block_1[i_1]$ (lines 9–14) and with $block_2[i_2]$ (lines 15–26). According to the semantics of this *precedence* pattern, we try to find a new occurrence of $block_1$, that conforms to the constraint "`at most` *n* `tu`" . The function uses the same code to update the variables i_1, t_1, and $flag_1$ for matching $block_1[i_1]$ lines 9–10. When the matched event is the last event of $block_1$, the function assigns the current location and timestamp to the variables (l_c,t_c) if $flag_1$ is `true`, or otherwise to (l_w,t_w) (lines 11–14) and resets variables i_1,t_1,i_2,t_2 (line 14).

In the loop, the algorithm also checks whether the current trace element is part of an occurrence of $block_2$. If an occurrence of the first event of $block_2$ is detected and there is no valid occurrence of $block_1$ or the distance from the previous valid occurrence $block_1$ violates the constraint "`at most` *n* `tu`" (i.e., $t - t_c > n$) (line 15), then the function moves on to match the second event of $block_2$ (line 16). In addition, the variable $flag_2$ is also updated with the conjunction "$t_w > 0$ && $t - t_w > n$", which indicates whether the time distance from the previous invalid occurrence of $block_1$ (if present) is in violation of the constraint on the distance between the two blocks. Otherwise, if variable i_2 is greater than 1, the algorithm calls function `match` to match the current trace element with $block_2[i_2]$ (line 18).

The algorithm reports a violation when $block_2$ is fully matched (lines 19–26). There are five possible types of violation to be reported, depending on the value of variables l_c, l_w: an `NSOR` violation is reported with a list that contains the location of current trace element and an empty list of occurrences of $block_1$, if there is no occurrence of $block_1$ before the occurrence of $block_2$ (line 20); otherwise, a `WTO` violation is reported with $[i]$ and a list of the location of the valid occurrence of $block_1$ (i.e., $[l_c]$), when the valid occurrence of $block_1$ is closer to the occurrence of $block_2$ than the invalid one (line 21); otherwise, a `WTOC` violation is reported with $[i]$ and a list of the location of the invalid occurrence of $block_1$ if variable $flag_2$ is `true`, meaning that the constraint "`at most` *n* `tu`" is violated by the closest invalid occurrence of $block_1$ (line 22); otherwise, a `WTC` violation is reported with $[i]$ and $[l_w]$, if the distance between the invalid occurrence $block_1$ and the current occurrence of $block_2$ is less than or equal to n (line 23); otherwise, an `LVRI` violation is reported with $[i]$ and $[l_c, l_w]$ (line 24), according to the definition of `LVRI`. The recognized violation is then added to the *result* (line 25) and the tuples (i_1,t_1) and (i_2,t_2) are reset (line 26). After the loop, the function returns variable *result* (line 27).

Function `reportPatternPrecedenceExactly` defines the algorithm for collecting violation information from a faulty trace segment that violates a *precedence* pattern that contains a constraint using the comparison operator "`exactly`" on the distance between the two blocks. The *precedence* pattern prescribes that each occurrence of $block_2$ must be preceded by an occurrence of $block_1$ with an exact distance n. Hence, an instance of this pattern can be violated if there is either no

Algorithm 27: reportPatternPrecedenceExactly

```
Input: $begin$, $end$: the boundaries of a sub-trace; $block_1$, $block_2$, $n$: the parameters of an
       instance of the precedence pattern "$block_1$ preceding exactly $n$ tu $block_2$"
Output: $result$: a list of 3-tuples, each of which consists of a violation type, a list of an
        occurrence location of $block_2$, and a list of corresponding occurrence locations of
        $block_1$
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$  $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $flag \leftarrow$ true, $ct \leftarrow 0$, $candidates \leftarrow []$, $result \leftarrow []$
5  for $i \leftarrow begin$ to $end$ do
6      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
7      if $e = firstOfBlock1$ then $(i_1,t_1,flag) \leftarrow (2,t,$true$)$
8      else if $i_1 > 1$ then $(i_1,t_1,flag) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag$, $e$, $t$)
9      if $i_1 = size_1 + 1$ then $candidates$.append($i_1$, $t_1$, $flag$), $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
10     if $e = firstOfBlock2$ then
11         if $\exists l', (l', t-n,$true$) \in candidates$ then $(i_2,t_2) \leftarrow (1,0)$
12         else $(i_2,t_2,ct) \leftarrow (2,t,t)$
13     else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match($block_1$, $i_1$, $t_1$, $e$, $t$)
14     if $i_2 = size_2 + 1$ then
15         if $candidates$.isEmpty() then $result$.append((NSOR, $[i]$, $[]$))
16         else
17             $L \leftarrow \{l' \mid (l',t',flag') \in candidates \;\&\&\; ct - t' = n\}$
18             if $L$.notEmpty() then $result$.append((WTC, $[i]$, $L$))
19             else
20                 $(l'_1, flag'_1) \leftarrow$ last($\{(l',flag') \mid (l',t',flag') \in candidates \;\&\&\; ct - t' > n\}$)
21                 $(l'_2, flag'_2) \leftarrow$ first($\{(l',flag') \mid (l',t',flag') \in candidates \;\&\&\; ct - t' < n\}$)
22                 if $flag'_1$ then
23                     if $flag'_2$ then $result$.append((WTO, $[i]$, $[l'_1, l'_2]$))
24                     else if $l'_2 \,!= $ null then $result$.append((LVRI, $[i]$, $[l'_1, l'_2]$))
25                     else $result$.append((WTO, $[i]$, $[l'_1]$))
26                 else if $l'_1 \,!= $ null then
27                     if $flag'_2$ then $result$.append((LIRV, $[i]$, $[l'_1, l'_2]$))
28                     else if $l'_2 \,!= $ null then $result$.append((WTOC, $[i]$, $[l'_1, l'_2]$))
29                     else $result$.append((WTOC, $[i]$, $[l'_1]$))
30                 else if $flag'_2$ then $result$.append((WTO, $[i]$, $[l'_2]$))
31                 else $result$.append((WTOC, $[i]$, $[l'_2]$))
32         $candidates \leftarrow \{(l',t',flag') \mid (l',t',flag') \in candidates \;\&\&\; ct - t' < n\}$
33         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
34 return $result$
```

occurrence of $block_1$ before an occurrence of $block_2$ or all the valid occurrences of $block_1$ cannot meet the constraint about the distance from an occurrence of $block_2$. According to the characterization of violations, given a faulty trace segment, there are five possible violation types: `NSOR`, `WTO`, `WTC`, `WTOC`, `LVRI`, and `LIRV`.

This function uses a set of auxiliary variables similar to those in the previous functions. In addition, we use variable *ct* to store the timestamp of a trace element that is a match with the first event of $block_2$, and use variable *candidates* to store all the occurrences of $block_1$ regardless of the compliance of the distance constraints defined within the block.

The main part of the function is the loop that navigates the input sub-trace, trying to match each trace element with $block_1[i_1]$ (lines 7–9) and with $block_2[i_2]$ (lines 10–33). According to the semantics of this *precedence* pattern, we have to store all the possible occurrences of $block_1$ which may lead to violations. In the loop, if $block_1$ is fully matched, besides resetting variables i_1, t_1, i_2, t_2, the function adds the value of i_1, t_1, *flag* to the list *candidates* (line 9). If an occurrence of the first event of $block_2$ is detected (line 10), if there exists a valid occurrence of $block_1$ satisfying the distance constraint "`exactly` *n* `tu`", the function resets the variables i_2, t_2 (line 11), since there will be no violation even if there is a match for $block_2$ beginning at the current trace element; otherwise, the function moves on to match the second event of $block_2$ (line 12). If variable i_2 is already greater than 1, the algorithm calls function `match` to match the current trace element with $block_2[i_2]$ (line 13).

The algorithm reports a violation when $block_2$ is fully matched (lines 14–33). There are six possible types of violations. If there is no occurrence of $block_1$ (i.e., *candidates* is empty), an `NSOR` violation is reported with a list that contains the location of the current trace element and an empty list of occurrences of $block_1$ (line 15). Otherwise, we check whether there exists an occurrence of $block_1$, and the distance between it and the head of this matched occurrence of $block_2$ is exactly *n* (line 17). If it does exist, then it must be *invalid* with respect to the distance constraints within $block_1$, and hence we report a `WTC` violation with the occurrence locations of the two blocks. If the violation is neither of the aforementioned types, we use temporary variables l'_1 and $flag'_1$ (respectively, l'_2 and $flag'_2$) to store the last (respectively, the first) element of the portion of *candidates*, in which all the occurrences of $block_1$ are more than *n* (respectively, less than *n*) time units prior to this matched occurrence of $block_2$; and we report a violation depending on their value. More specifically, a `WTO` violation is reported if both variables $flag'_1$ and $flag'_2$ are `true` (lines 23, 25, 30), that is, if the occurrences of $block_1$ close to the expected instant conform to the distance constraints defined within the block. If $flag'_1$ is `true` but $flag'_2$ is `false`, an `LVRI` violation is reported (line 24); otherwise, if it is the opposite, an `LIRV` violation is reported (line 27). Otherwise, a `WTOC` violation is reported if both $flag'_1$ and $flag'_2$ is `false` (lines 28, 29, 31). After collecting the violation, the function prunes the list *candidates* by removing all the occurrences of $block_1$ that are impossible to satisfy the pattern (line 32) and resets the tuples (i_1, t_1) and (i_2, t_2) (line 33). The function returns variable *result* after the loop finishes (line 34).

4.3.5 Response

Function `reportPatternResponse` is the dual of function `reportPatternPrecedence`. It invokes one of four auxiliary functions to collect the information violation with respect to a *response* pattern, which may contain no constraint on the time distance between the two blocks, or use one of the three comparison operators (i.e., "`at least`", "`at most`", and "`exactly`") in the constraint. In the rest of this section, we explain the four functions `reportPatternResponseGeneral`

(Algorithm 28), `reportPatternResponseAtLeast` (Algorithm 29), `reportPatternResponseAtMost` (Algorithm 30), and `reportPatternResponseExactly` (Algorithm 31).

Algorithm 28: reportPatternResponseGeneral

```
Input: begin, end: the boundaries of a sub-trace; block_1, block_2: the two blocks of an instance
       of the response pattern of the form "block_1 responding block_2"
Output: result: a list of 3-tuples, each of which consists of a violation type, a list of the
        locations of offending occurrences of block_2, and a list of the locations of
        corresponding occurrences of block_1
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().event, $firstOfBlock2 \leftarrow block_2$.first().event
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $flag \leftarrow$ true, $l_w \leftarrow 0$, $candidates \leftarrow []$, $result \leftarrow []$
5  for $i \leftarrow begin$ to $end$ do
6      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
7      if $e = firstOfBlock2$ then $(i_2,t_2) \leftarrow (2,t)$
8      else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match$(block_2, i_2, t_2, e, t)$
9      if $i_2 = size_2 + 1$ then
10         $candidates$.append($(i_2,t_2)$)
11         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
12     if $candidates$.notEmpty() && $e = firstOfBlock1$ then $(i_1,t_1,flag) \leftarrow (2,t,$true$)$
13     else if $i_1 > 1$ then $(i_1,t_1,flag) \leftarrow$ matchSecondaryBlock($block_1, i_1, t_1, flag, e, t$)
14     if $i_1 = size_1 + 1$ then
15         if $flag$ then $candidates \leftarrow []$
16         else $l_w \leftarrow i$
17         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
18 if $candidates$.notEmpty() then
19     $L_1 \leftarrow \{l' \mid (l',t') \in candidates \;\&\&\; l' < l_w\}$
20     if $L_1$.notEmpty() then $result$.append((WTC, $L_1$, $[l_w]$))
21     $L_2 \leftarrow \{l' \mid (l',t') \in candidates \;\&\&\; l' > l_w\}$
22     if $L_2$.notEmpty() then $result$.append((NSOR, $L_2$, []))
23 return $result$
```

We define function `reportPatternResponseGeneral` (Algorithm 28) to collect violations from a faulty trace segment that is in violation of a *response* pattern which contains no constraint on the distance between the two blocks. The function has the same input and output type as its dual function `reportPatternPrecedenceGeneral`. The *response* pattern prescribes that each occurrence of $block_2$ must be followed by an occurrence of $block_1$), so there are two possible types of violations (i.e., `NSOR` and `WTC`); `WTC` is only possible if $block_1$ contains a distance constraint. In addition to the variables used in its dual function, the function introduces the variable *candidates* to store the occurrences of $block_2$ that may be violated.

First, the function uses a loop to navigate the input sub-trace from *begin* to *end*, trying to match each trace element with $block_2[i_2]$ (lines 7–11) and with $block_1[i_1]$ (lines 12–17). More specifically, in each iteration, the function checks whether the current trace element is part of an occurrence of $block_2$ (lines 7–8). If an occurrence of $block_2$ is fully matched, the function adds the value of variables i_2 t_2 to *candidates* (line 10) and resets variables i_1,t_1,i_2,t_2 (line 11). On the other hand, if the list *candidates* is not empty, the function checks whether the current trace element is part of $block_1$ (lines 12–13). Until an occurrence of $block_1$ is fully matched (line 14), if it conforms to the distance constraints within the block (i.e., *flag* is `true`), that is, all the occurrences of $block_2$ stored in *candidates* are followed by a *valid* occurrence of $block_1$, the function empties *candidates* (line 15); otherwise the function updates variable l_w with the current position (line 16). Besides, variables i_1,t_1,i_2,t_2 are reset (line 17).

After the loop, the function checks if the list *candidates* still contains occurrences of $block_2$ (line 18). As discussed before, there are two possible types of violations, `WTC` and `NSOR`. For the occurrences of $block_2$ that happen prior to the position l_w (line 19), the function reports a group of `WTC` violations with $[l_w]$ that indicates the corresponding *invalid* occurrence of $block_1$ (line 20). For the occurrences of $block_2$ that happen after the position l_w, since they are not followed by any occurrence of $block_1$, the function reports a group of `NSOR` violations (line 22).

Function `reportPatternResponseAtLeast` defines the violation information collection algorithm for the variant of *response* pattern that uses "`at least`" as the comparison operator in the distance constraint between the two blocks. The function has the same input and output types as function `reportPatternResponseGeneral`. This type of *response* pattern stipulates that an occurrence of $block_2$ must be followed by an occurrence of $block_1$ with a minimum distance *n*; it can lead to five types of violation: `NSOR`, `WTO`, `WTC`, `WTOC`, and `LVRI`.

In addition to the variables used in function `reportPatternResponseGeneral`, this function uses variables l_c and t_c (respectively, l_w and t_w) to store the latest occurrence of $block_1$ which is *valid* (respectively, *invalid*) with respect to the the distance constraints within the block and introduces variable *ct* to save the timestamp of the first event of a potential match for $block_1$. In each iteration for searching for occurrences of $block_2$ and with $block_1$, there is a slight difference from the corresponding portion described in Algorithm 28 when an occurrence of $block_1$ is fully matched (lines 14–19). More specifically, if the matched occurrence of $block_1$ has no violation of the distance constraints defined within the block, the function removes the occurrences of $block_2$ that have been satisfied from the list `candidates` (line 16) and updates variable l_c with the current position (i.e., *i*) and variable t_c with the timestamp of the head of the matched occurrence (i.e., *ct*) (line 17). Otherwise, the function assigns *i*, *ct* to variables l_w, t_w (line 17).

After the loop, the function collects violations if the list *candidates* contains unsatisfied occurrences of $block_2$ (lines 20–28). The function extracts all the offending positions from *candidates* and stores in variable *L* (line 21); it also extracts the positions that are smaller than l_w and stores them in variable L_I, and finally it also extracts the positions that are smaller than l_c and stores them in variable L_V (line 21). In addition, the function extracts the positions of those occurrences of $block_2$ that meet the distance constraint "`at least` *n* `tu`" with respect to the last *invalid* occurrence of $block_1$, and stores them in variable L_I'. If there exist occurrences of $block_2$ which are neither followed by a *valid* nor by an *invalid* occurrence of $block_1$, the function reports a set of `NSOR` violations accompanied by a list of offending occurrences of $block_2$ stored in $L - L_V - L_I$ (line 24). If the last *valid* occurrence of $block_1$ is found after the *invalid* one and variable L_V is not empty, the function reports a set of `WTO`

Algorithm 29: reportPatternResponseAtLeast

```
Input: $begin$, $end$: the boundaries of a sub-trace; $block_1$, $block_2$, $n$: the parameters of an
        instance of the response pattern of the form "$block_1$ responding at least $n$
        tu $block_2$"
Output: $result$: a list of 3-tuples, each of which consists of a violation type, a list of the
        locations of offending occurrences of $block_2$, and a list of the locations of
        corresponding occurrences of $block_1$
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$, $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$, $(l_c,t_c) \leftarrow (0,0)$, $(l_w,t_w) \leftarrow (0,0)$
4  $flag \leftarrow$ true, $ct \leftarrow 0$, $candidates \leftarrow []$, $result \leftarrow []$
5  for $i \leftarrow begin$ to $end$ do
6      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
7      if $e = firstOfBlock2$ then $(i_2,t_2) \leftarrow (2,t)$
8      else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match$(block_2,i_2,t_2,e,t)$
9      if $i_2 = size_2 + 1$ then
10         $candidates$.append($(i_2,t_2)$)
11         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
12     if $candidates$.notEmpty() && $e = firstOfBlock1$ then $(i_1,t_1,ct) \leftarrow (2,t,t)$
13     else if $i_1 > 1$ then $(i_1,t_1,flag) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag$, $e$, $t$)
14     if $i_1 = size_1 + 1$ then
15         if $flag$ then
16             $candidates \leftarrow \{(l',t') \mid (l',t') \in candidates \ \&\& \ ct - t' < n\}$
17             $(l_c,t_c) \leftarrow (i,ct)$
18         else $(l_w,t_w) \leftarrow (i,ct)$
19         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
20 if $candidates$.notEmpty() then
21     $L \leftarrow \{l' \mid (l',t') \in candidates\}$
22     $L_I \leftarrow \{l' \mid (l',t') \in candidates \ \&\& \ l' < l_w\}$, $L_V \leftarrow \{l' \mid (l',t') \in candidates \ \&\& \ l' < l_c\}$
23     $L'_I \leftarrow \{l' \mid (l',t') \in candidates \ \&\& \ t_w - t' \geq n\}$
24     if $L - L_V - L_I \neq \emptyset$ then $result$.append((NSOR, $L - L_V - L_I$, []))
25     if $l_w < l_c$ && $L_V \neq \emptyset$ then $result$.append((WTO, $L_V$, $[l_c]$))
26     else if $l_c < l_w$ then
27         if $L_I - L'_I \neq \emptyset$ then $result$.append((WTOC, $L_I - L'_I$, $[l_w]$))
28         if $L_V \cap L'_I \neq \emptyset$ then $result$.append((LVRI, $L'_I \cap L_V$, $[l_c, l_w]$))
29 return $result$
```

violations accompanied by L_V and $[l_c]$ (line 25). Otherwise if the l_w is greater than l_c, if there exist occurrences of $block_2$ before the position l_w and the distance constraint "at least n tu" is violated by the *invalid* occurrence of $block_1$ at l_w (i.e., $L_I - L'_I \neq \emptyset$), the function reports a set WTOC violations; if there exist occurrences of $block_2$ followed both by a *valid* occurrence of $block_1$ which violates the distance constraint and by an *invalid* occurrence of $block_1$ which meets the distance constraint (i.e., $L_V \cap L'_I \neq \emptyset$), the function reports a set LVRI violations.

Algorithm 30: reportPatternResponseAtMost

```
Input: $begin$, $end$: the boundaries of a sub-trace; $block_1$, $block_2$, $n$: the parameters of an
       instance of the response pattern of the form "$block_1$ responding at most $n$ tu
       $block_2$"
Output: $result$: a list of 3-tuples, each of which consists of a violation type, a list of the
        location of offending occurrences of $block_2$, and a list of the locations of
        corresponding occurrences of $block_1$
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first().$event$, $firstOfBlock2 \leftarrow block_2$.first().$event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $flag \leftarrow$ true, $ct \leftarrow 0$, $l_w \leftarrow 0$, $candidates \leftarrow []$, $result \leftarrow []$
5  for $i \leftarrow begin$ to $end$ do
6      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
7      if $e = firstOfBlock2$ then $(i_2,t_2) \leftarrow (2,t)$
8      else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match$(block_2, i_2, t_2, e, t)$
9      if $i_2 = size_2 + 1$ then
10         $candidates$.append($(i_2,t_2)$)
11         $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
12     if $candidates$.notEmpty() && $e = firstOfBlock1$ then $(i_1,t_1,ct) \leftarrow (2,t,t)$
13     else if $i_1 > 1$ then $(i_1,t_1,flag) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag$, $e$, $t$)
14     if $i_1 = size_1 + 1$ then
15         $L_1 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' < l_w$ && $ct - t' > n\}$
16         $L_2 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' > l_w$ && $ct - t' > n\}$
17         if $flag$ then
18             if $L_1$.notEmpty() then $result$.append((LIRV, $L_1$, $[l_w, i]$))
19             if $L_2$.notEmpty() then $result$.append((WTO, $L_2$, $[i]$))
20             $candidates \leftarrow []$
21         else
22             if $L_1$.notEmpty() then $result$.append((WTC, $L_1$, $[l_w]$))
23             if $L_2$.notEmpty() then $result$.append((WTOC, $L_2$, $[i]$))
24             $candidates \leftarrow \{(l',t') \mid (l',t') \in candidates$ && $ct - t' \leq n\}$
25             $l_w \leftarrow i$
26             $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
27 if $candidates$.notEmpty() then
28     $L_1 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' < l_w\}$
29     if $L_1$.notEmpty() then $result$.append((WTC, $L_1$, $[l_w]$))
30     $L_2 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' > l_w\}$
31     if $L_2$.notEmpty() then $result$.append((NSOR, $L_2$, []))
32 return $result$
```

Function reportPatternResponseAtMost defines the algorithm for collecting violations from a faulty trace segment which is in violation of a *response* pattern using "at most" in the distance constraint between the two blocks. The pattern can be violated either when an occurrence of *block*$_2$ is not followed by an occurrence of *block*$_1$ or the distances between an occurrence of *block*$_2$ and all valid occurrences of *block*$_1$ are more than *n*. So there are five possible violation types for this pattern: `NSOR`, `WTO`, `WTC`, `WTOC`, and `LIRV`. The function has the same input and output types as the the previous homologous functions.

The function contains a loop that inspects trace elements from the position *begin* to *end* of the trace instance and tries to find occurrences of the pair of *block*$_2$ (lines 7–11) and *block*$_1$ (lines 12–26). The pseudocode (lines 7–13) for matching the two blocks is the same as the previous functions until *block*$_1$ is fully matched (line 14).

In lines 15–26, the algorithm examines the *response* pattern and update *result* if any violation emerges. More specifically, from the list *candidates*, the algorithm extracts the occurrence locations of *block*$_2$ that fails fulfilling the maximum limit on the distance to the matched occurrence of *block*$_1$, and assigns the ones before (respectively, after) the previous (invalid) occurrence of *block*$_1$ (i.e., l_w) to variable L_1 and (respectively, variable L_2) (lines 15–16). According to the definition of the pattern, violations should be reported if one of the variables is not empty. If the current matched occurrence of *block*$_1$ complies with the distance constraints (if defined) within the block, (line 17), if variable L_1 is not empty, a group of `LIRV` violations, the locations of offending occurrences of *block*$_2$ stored in variable L_1, and a list with the two locations of the corresponding occurrence of *block*$_1$, is appended to *result* (line 18); if variable L_2 is not empty, a group of violations that contains the violation type `WTO`, variable L_2, and a list with the current location, is collected (line 19). The function then empties variable *candidates* (line 20). Otherwise if the current matched occurrence of *block*$_1$ is in violation of the distance constraints defined within the block, if variable L_1 is not empty, a group of violations consisting of the violation type `WTC`, variable L_1, and a list containing l_w, is collected (line 22); if variable L_2 is not empty, a group of violations consisting of the violation type `WTOC`, variable L_2, and a list with the current location, is collected (line 23). The variable *candidates* is updated by retaining the occurrences of *block*$_2$ which comply with the constraint "`at most n tu`" with respect to the current matched occurrence of *block*$_1$ (line 24). Moreover, the variable l_w is updated with the current location that indicates a new invalid occurrence of *block*$_1$ (line 25).

The violation information is accumulated in variable *result* through the iteration. After the loop (line 27) the algorithm also has to check if the list *candidates* is not empty. More specifically, the algorithm collects `WTC` violations when there exist occurrences of *block*$_2$ before l_w (line 29), and collects `NSOR` violations when there exist occurrences of *block*$_2$ after which no occurrence of *block*$_1$ is found (line 31).

Function `reportPatternResponseExactly` defines the algorithm for collecting violation information for the variant of *response* pattern that uses "`exactly`" in the distance constraint between the two blocks. The function has the same input and output types as the other homologous functions. There are six possible violation types for this type of *response* pattern: `NSOR`, `WTO`, `WTC`, `WTOC`, `LVRI`, and `LIRV`.

Instead of using variable l_w (in function `reportPatternResponseAtMost`) to save the location of the latest occurrence of *block*$_1$ that is invalid with respect to the distance constraints defined in the block, in this function we introduce a pair of variables $(l_l, flag_l)$ (line 4) to store the location and validity of the latest occurrence of *block*$_1$.

Algorithm 31: reportPatternResponseExactly

```
Input: $begin$, $end$: the boundaries of a sub-trace; $block_1$, $block_2$, $n$: the parameters of an
       instance of the $response$ pattern "$block_1$ responding exactly $n$ tu $block_2$"
Output: $result$: a list of 3-tuples, each of which consists of a violation type, a list of the
        locations of offending occurrences of $block_2$, and a list of the locations of
        corresponding occurrences of $block_1$
1  $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2  $firstOfBlock1 \leftarrow block_1$.first()$.event$, $firstOfBlock2 \leftarrow block_2$.first()$.event$
3  $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
4  $flag \leftarrow$ true, $ct \leftarrow 0$, $(l_l, flag_l) \leftarrow (0,$ true$)$, $candidates \leftarrow []$, $result \leftarrow []$
5  for $i \leftarrow begin$ to $end$ do
6      $elem \leftarrow self.traceElements[i]$, $(e,t) \leftarrow (elem.event, elem.timestamp)$
7      if $e = firstOfBlock2$ then $(i_2,t_2) \leftarrow (2,t)$
8      else if $i_2 > 1$ then $(i_2,t_2) \leftarrow$ match$(block_2, i_2, t_2, e, t)$
9      if $i_2 = size_2 + 1$ then $candidates$.append($(i_2,t_2)$), $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
10     if $candidates$.notEmpty() && $e = firstOfBlock1$ then $(i_1,t_1,ct) \leftarrow (2,t,t)$
11     else if $i_1 > 1$ then $(i_1,t_1,flag) \leftarrow$ matchSecondaryBlock($block_1$, $i_1$, $t_1$, $flag$, $e$, $t$)
12     if $i_1 = size_1 + 1$ then
13         $L \leftarrow \{l' \mid (l',t') \in candidates$ && $ct - t' = n\}$
14         if !$flag$ && $L$.notEmpty() then $result$.append((WTC, $L$, $[i]$))
15         $L_1 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' < l_l$ && $ct - t' > n\}$
16         $L_2 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' > l_l$ && $ct - t' > n\}$
17         if $L_1$.notEmpty() then
18             if $flag$ then
19                 if $flag_l$ then $result$.append((WTO, $L_1$, $[l_l, i]$))
20                 else $result$.append((LIRV, $L_1$, $[l_l, i]$))
21             else
22                 if $flag_l$ then $result$.append((LVRI, $L_1$, $[l_l, i]$))
23                 else $result$.append((WTOC, $L_1$, $[l_l, i]$))
24         if $L_2$.notEmpty() then
25             if $flag$ then $result$.append((WTO, $L_1$, $[i]$))
26             else $result$.append((WTOC, $L_1$, $[i]$))
27         $candidates \leftarrow \{(l',t') \mid (l',t') \in candidates$ && $ct - t' < n\}$
28         $(l_l, flag_l) \leftarrow (i, flag)$, $(i_1,t_1) \leftarrow (1,0)$, $(i_2,t_2) \leftarrow (1,0)$
29 if $candidates$.notEmpty() then
30     $L_1 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' < l_l\}$, $L_2 \leftarrow \{l' \mid (l',t') \in candidates$ && $l' > l_l\}$
31     if $L_1$.notEmpty() then
32         if $flag_l$ then $result$.append((WTO, $L_1$, $[l_l]$))
33         else $result$.append((WTOC, $L_1$, $[l_l]$))
34     if $L_2$.notEmpty() then $result$.append((NSOR, $L_2$, $[]$))
35 return $result$
```

In the loop (lines 5–28) that searches for occurrences of $block_2$ and $block_1$, the function differs from the previous functions when an occurrence of $block_1$ is fully matched (lines 12–28). More specifically, we first retrieve the occurrence of $block_2$ that meets the distance constraint with respect to this match with $block_1$ (line 13). If the result is not empty and the match for $block_1$ is valid with respect to the distance constraints within the block, we collect a `WTO` violation with the offending occurrence of $block_2$ and the current location. In addition, we extract the locations of the occurrences of $block_2$ that happen before the previous occurrence of $block_1$ (i.e., l_l) and farther than n to the current occurrence of $block_1$, and store them in variable L_1 (line 15); we also store the locations of the occurrences of $block_2$ after l_l and also farther than n to the current match for $block_1$ in L_2 (line 16). If variable L_1 is not empty (line 17), we collect a set of violations with L_1 and a list comprised of both the locations of the previous and current occurrences of $block_1$: if the current match of $block_1$ is valid (i.e., *flag* is true), depending on the validity of the previous match for $block_1$, we collect either a set of `WTO` (line 19) or `LIRV` (line 20) violations; otherwise if the current match is invalid, we collect either a set of `LVRI` (line 22) or `WTOC` (line 23) violations depending on the validity of the previous occurrence of $block_1$. If variable L_2 is not empty (line 24), we collect a set of violations with L_2 and a list containing the current location i: if the current match for $block_1$ is valid, we collect a set of `WTO` violations (line 25) or else collect a set of `WTOC` violations (line 26). In addition to violation information collection, we prune variable *candidates* by removing the matches for $block_2$ that have been examined (line 27) and also update the variables $l_l, flag_1, i_1, t_1, i_2, t_2$.

After the analysis of the entire input trace, if variable *candidates* still contains some matches for $block_2$ that have not been examined (line 29), we use the position of the last occurrence of $block_1$ (l_l) to split the locations stored in *candidates* into two parts (i.e., L_1 and L_2) (line 30) and collect violation information: if variable L_1 is not empty, if the last occurrence of $block_1$ is valid, we collect a set of `WTO` violations (line 32) or else a set of `WTOC` violations (line 33); if variable L_2 is not empty, we report a set of `NSOR` violations (line 34).

4.4 Tool Implementation

We have implemented the model-driven procedure for collecting violation information from a trace in the tool TEMPSY-REPORT. The TEMPSY-REPORT tool extends the implementation of TEMPSY-CHECK (Section 3.6) and is also based on Eclipse OCL [Eclipse, 2015a]; it is publicly available at `http://weidou.github.io/TemPsy-Report`.

TEMPSY-REPORT takes in input the same trace instance and *TemPsy* expressions (checked by TEMPSY-CHECK) and hence shares the definitions of the conceptual model for execution traces and the *TemPsy* language with TEMPSY-CHECK. The tool extends the Java class `ConstraintFactory` to build OCL queries following the template shown in Figure 4.2. More specifically, given a trace instance and a set of *TemPsy* expressions that TEMPSY-CHECK has determined as violating the trace, TEMPSY-REPORT creates OCL queries on the trace instance with the help of `ConstraintFactory`, depending on the types of the scope and pattern used by each *TemPsy* expression. We have implemented all the OCL functions (defined on class `Trace`) for *TemPsy* scopes and patterns as described in the previous subsection, to collect information on the violations. The evaluation of the OCL queries is done using the `evaluate` function provided by Eclipse OCL.

TEMPSY-REPORT can both output structured text containing the violation information, and also insert the latter in a MongoDB[3] (using the *Mongo Java Driver*[4]). In the next subsection, we will present how the violation information stored in MongoDB is used by our visualization tool.

4.5 Visualizing Violation Information

4.5.1 Requirements

The violation information collected by TEMPSY-REPORT can be displayed as textual output, as done by the main state-of-the-art trace checking tools [Donzé, 2010, Basin et al., 2012, Reger et al., 2015, Luo et al., 2014]. However, complex structured textual output is cumbersome to inspect, especially when a violation can be the result of different causes (i.e., event occurrences) and thus navigating both the violations list and the events trace is required.

Based on the discussions with our partner, we define the following high-level requirements for a tool for the visualization of violation information, as an alternative to plaintext output: R1) user-friendly navigation of a trace; R2) easy access to violations in a trace; R3) useful information for understanding violations. In the remainder of this section, we first define the functionality of the target tool, as prescribed by these requirements; then, we illustrate the implementation of the main features.

4.5.2 Functionality

We defined the following seven features for visualizing the violation information in a trace:

F1 to show the details of a trace element when mousing over its data point;

F2 to zoom in/out of the trace with a self-adaptive granularity of data points;

F3 to slide the displayed time window in both directions along the x-axis;

F4 to mark out the locations of violations as well as the corresponding data points and intervals;

F5 to display the reasons and other details of violations in a navigable fashion (separately from the display of the trace);

F6 to inspect each violation (in the trace display) by clicking on either its corresponding data point or the link embedded in its description;

F7 to highlight the violated portion of a *TemPsy* property upon inspecting a violation.

As shown in Table 4.2, the seven features strongly contribute to fulfilling the three requirements. The first feature, as the basis for inspecting traces, shows the event name and timestamp of a trace element, and hence assists in navigating the traces. The second feature contributes to both R1 and R3, since it provides a handy facility, which allows users not only to examine interesting trace elements

[3]`https://www.mongodb.com`
[4]`https://mongodb.github.io/mongo-java-driver`

Table 4.2: Mapping between the requirements and the functionality of the target visualization tool

Requirement	Functionality
R1	F1 F2 F3
R2	F4 F5 F6
R3	F2 F4 F5 F7

in close-up but also to visualize the distribution of violations, by means of feature F4. Furthermore, the third feature helps to refer to adjacent trace elements when navigating a portion of a trace.

While the aforementioned three features are common functionality for generic data visualization, the remaining features are specific to the context of visualizing violations in traces. The fourth feature helps identify violations in the display of a trace, and the fifth explicitly lists the details of all violations. Both features not only contribute to a better understanding of violations but also enable effortless access to violations across a trace, as described in the feature F6. Together, they contribute to fulfilling the second requirement. In addition to F2, F4, and F5, the last feature highlights the portion of a property that is violated and thus contribute to fulfilling R3 (related to understanding violations).

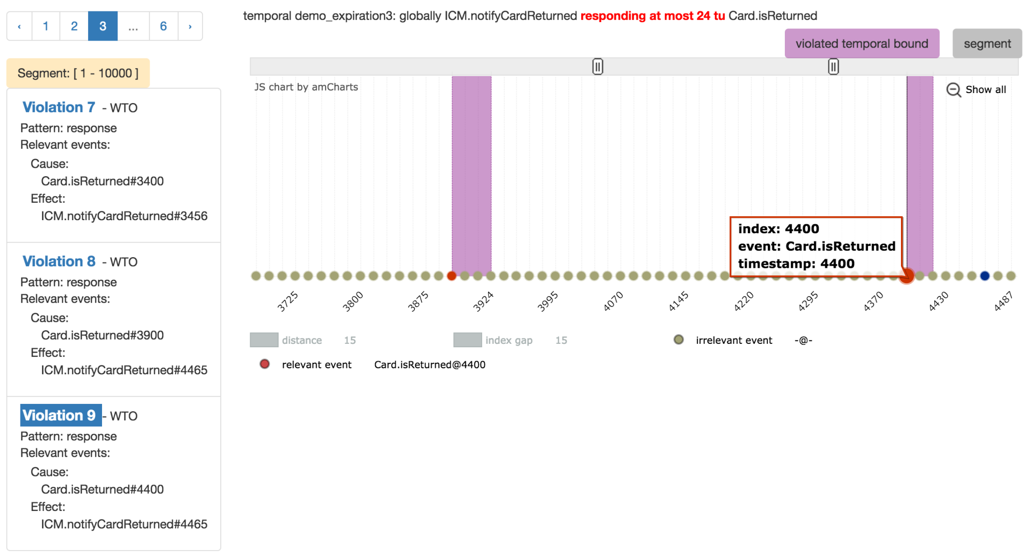
4.5.3 Implementation

We have developed an interactive visualization tool, implementing the features above; it is publicly available at `http://weidou.github.io/TemPsy-Violation-Visualization`. The tool is based on technologies such as amCharts[5], Meter.js[6], AngularJS[7], MongoDB[3], and ElasticSearch[8]. The tool leverages the violation information collected by TEMPSY-REPORT and saved into MongoDB. It also stores a copy of the trace in ElasticSearch, to allow for fast data retrieval from the trace.

The trace is displayed in chronological order using a chart component. A data point in the chart corresponds to a trace element; the timestamp of each trace element is the x-axis value of the corresponding data point, while the y-axis is set to a constant.

Figure 4.3 shows a screenshot of the tool, which highlights the ninth violation on an execution trace. The *TemPsy* property “`globally ICM.notifyCardReturned responding at most 24 tu Card.isReturned`” consists of a *globally* scope and a bounded *response* pattern. On the right of the screenshot, the execution trace is visualized chronologically, within which the occurrences of irrelevant events are in olive and the data points referring to violations are highlighted with red and blue colors. The red points are the occurrences of event `Card.isReturned` which violate the response pattern; the blue points are the closest (in time) occurrences of event `ICM.notifyCardReturned` which occur after an occurrence of event `Card.isReturned` but beyond the distance of 24 time units. As a general rule, for the *TemPsy* patterns which contain only one event, i.e., *universality*, *existence*, and *absence*, we use the red color to highlight the data points referring to violations; for the *order* patterns, i.e., *precedence* and *response*, we use the red color to

[5]`https://www.amcharts.com`
[6]`https://www.meteor.com`
[7]`https://angularjs.org`
[8]`https://www.elastic.co`



Segment: [1 - 10000]

Violation 7 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#3400
Effect:
ICM.notifyCardReturned#3456

Violation 8 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#3900
Effect:
ICM.notifyCardReturned#4465

Violation 9 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#4400
Effect:
ICM.notifyCardReturned#4465

index: 4400
event: Card.isReturned
timestamp: 4400

Figure 4.3: Screenshot of our visualization tool for understanding *TemPsy* violations

highlight the offending occurrences of the event(s) specified in the second block, and use the blue color to highlight the closest occurrences of the event(s) specified in the first block, which correspond to the red data points. In addition, as shown in the screenshot, we mark the time window of 24 time units in purple at each red point, to help identify the bound of the time distance between the two blocks of an *order* pattern. This is how the tool implements feature F4. The screenshot also showcases the general features, i.e., F1, F2, and F3. As shown within the red-framed callout, the details of the occurrence of the event `Card.isReturned` at the timestamp 4400 appears when hovering over its data point (F1); as for F2 and F3, the current sub-trace can be reached with three actions: zooming in for inspecting the ninth violation at timestamp 4400, zooming out to show the right region of the violation, and sliding to the left using the horizontal sliding bar on top of the chart. Above the chart, the text "`responding at most 24 tu`" is highlighted in red in the property, to indicate that the ninth violation is about the constraint on the time distance between the two blocks; this implement feature F7. On the left of the screenshot, the violation information is navigable through a list (feature F5). Moreover, the user can zoom in for inspecting each violation by clicking on the link within each violation information block, or by clicking on the data points in red/blue (feature F6).

The visualization tool can display on-the-fly the distribution of the violations when displaying the entire trace, by automatically adapting the granularity of data points. Furthermore, as shown at the top of the sliding bar, the button labeled with *segment* (or *segments*) enables toggling the highlight(s) of the sub-trace(s) delimited by the scope used in the given property. A the bottom of the chart, we also provide toggles for inspecting the trace elements; for example, the one labeled with *irrelevant event*, toggles the visualization of the events that are not specified in the pattern.

4.6 Evaluation

In this section, we evaluate the scalability of TEMPSY-REPORT, by investigating the relationship among the execution time of TEMPSY-REPORT, the number of violations, the violation type, the structural properties of a trace (e.g., length, distribution of violations and events), and the type of the pattern used by a *TemPsy* property. For each type of *TemPsy* patterns, we address two research questions:

RQ1) Given a fixed number of violations, what is the relation between the execution time of TEMPSY-REPORT and the length of a trace?

RQ2) Given a fixed length of a trace, what is the relation between the execution time of TEMPSY-REPORT and the number of violations?

4.6.1 Experiment settings

From the requirements specification documents of an eGovernment application developed by our partner, we have extracted 47 temporal properties. Since the scalability of the violation information collection procedure mainly correlates with the *pattern* used in a *TemPsy* property, we select the ones that use the *globally* scope (12 out of the 47) as the benchmark. The 12 properties are listed in a sanitized form in Table 4.3. For confidentiality reasons, we only keep the structure of each property, in terms of *scope* + *pattern*; the events involved in the property (e.g., "a citizen requests a certificate") are denoted by uppercase letters.

Table 4.3: *TemPsy* properties used for the evaluation

P1: globally always A
P2: globally never B
P3: globally eventually at least 2 A
P4: globally eventually at most 3 A
P5: globally A responding at most 1000 tu B
P6: globally A responding exactly 1000 tu B
P7: globally A preceding at most 6000 tu B
P8: globally A preceding at least 100 tu B
P9: globally A preceding exactly 100 tu B
P10: globally A, B preceding at least 1000 tu C, D
P11: globally A responding at least 1000 tu B, C
P12: globally A responding B

We use *synthesized* traces to assess the scalability of the procedure for collecting violation information. There are several reasons to do this: 1) based on our experience, real traces are often inadequate to cover a large range of trace lengths and a variety of properties; 2) we wanted to have great diversity in terms of occurrences of violations in the traces, while being able to control this diversity; 3) if we had used real traces, they could not be shared for forming a public benchmark, even when sanitized. By using synthesized traces we are able to control in a systematic way the factors (such as trace length, frequency and types of violations) that could impact the execution time for a specific type of pattern. At the same time, we are also able to randomly set other factors, e.g., distance between events, to avoid any bias. To synthesize these traces we implemented a trace generator program. The program takes in input a *TemPsy* pattern and generates faulty traces for the evaluation. The details of the trace generation strategy depend on the input pattern and are discussed in the next subsections. As an additional contribution of the thesis, we also make available in the TEMPSY-REPORT GitHub repository the artifacts used in the evaluation, to contribute to the building of a public repository of case studies for evaluating the approaches to violation reporting for trace checking.

To address research question RQ1, in the trace generator program we fix the number of violations[9] to 1K, and generate 10 traces with various lengths from 100K to 1M, with a 100K step increment. For the research question RQ2, we fix the length of the trace to 1M, and vary the number of violations[9] from 1K to 10K, with a 1K step increment. The key to avoiding bias is to distribute the violations (or events) evenly and to generate their positions randomly within the slots determined by the trace length and number of violations. The positions related to a violation are generated randomly by taking into account the temporal and timing constraints prescribed by the semantics of the input pattern; the other positions are filled with an irrelevant event.

In the following subsections, we first describe the specific trace generation strategies for each *TemPsy* pattern and then report on the evaluation results by applying our approach to the 12 properties.

The results reported in this section have been measured on a desktop computer with a 3 GHz Intel Dual-Core i7 CPU and 16GB of memory, running Eclipse DSL Tools v. 4.6.0M3 (Neon Milestone 3), JavaSE-1.7 (Java SE v. 1.8.0_25-b17, Java HotSpot (TM) 64-Bit Server VM v. 25.25-b02, mixed mode), Eclipse OCL v. 6.0.1. All measurements reported correspond to the average value over 100 runs of the procedure (on the same trace, for the same property).

4.6.2 Trace Generation Strategies

Properties using the *universality* pattern. For the *universality* pattern like property P1 in Table 4.3, there is only one possible violation type, i.e., `NSOC`. Hence after randomly generating the violation positions, the trace generator fills them with a dummy event (e.g., event "*B*"), and fills the others with event *A*.

Properties using the *existence* pattern. Out of the 12 properties, there are two properties using the *existence* pattern, i.e., P3 and P4. The strategies for generating faulty traces for properties defined using the *existence* pattern depend on the comparison operator used in the pattern.

Property P3 can only contain `NSOC` violations, when the number of occurrences of event *A* is less than 2; and hence, we can only address RQ1 since the number of offending occurrences of event *A* can hardly be varied ("`at least 2`"). In the trace generator, we fix the number of occurrences of event *A* to 1, which is the smallest value between $2-1$ and the reserved number 1K, generate the position randomly in a given trace, and fill the other positions with a dummy event.

For property P4, we address both the research questions. As discussed in Section 4.3.2, there is only one type of violations for this pattern, i.e., `UNOC`. Following the general trace generation strategies discusses above settings, given the number of violations *n*, the trace generator picks the greatest value between $3+1$ and *n* as the number of violations and fills the generated positions with event *A* and the others with a dummy event.

Properties using the *absence* pattern. In the 12 properties, property P2 is the one using the *ab-*

[9]Alternatively, the number of offending occurrences of a specific event for the cases of *universality*, *absence*, and *existence*.

sence pattern. It can be violated by any occurrences of event *B*, yielding `UNOC` violations. Hence, after generating the violation positions, the trace generator fills the them with event *B* and fills the other positions with a dummy event.

Properties using the *precedence* pattern. The trace generation strategy for the properties using the *precedence* pattern differs from the previous ones, since there may be more than one possible type of violations, depending on the structure of the pattern. So for each possible violation type, we generate a set of traces that contains only the specific type of violations, to avoid bias.

Out of the 12 properties, there are four properties that use the *precedence* patterns, i.e., P7–P10. For all the properties, given the number of violations *n*, the trace generator first divides the trace into *n* portions with the same length, and then fills a specific violation at a random and valid position.

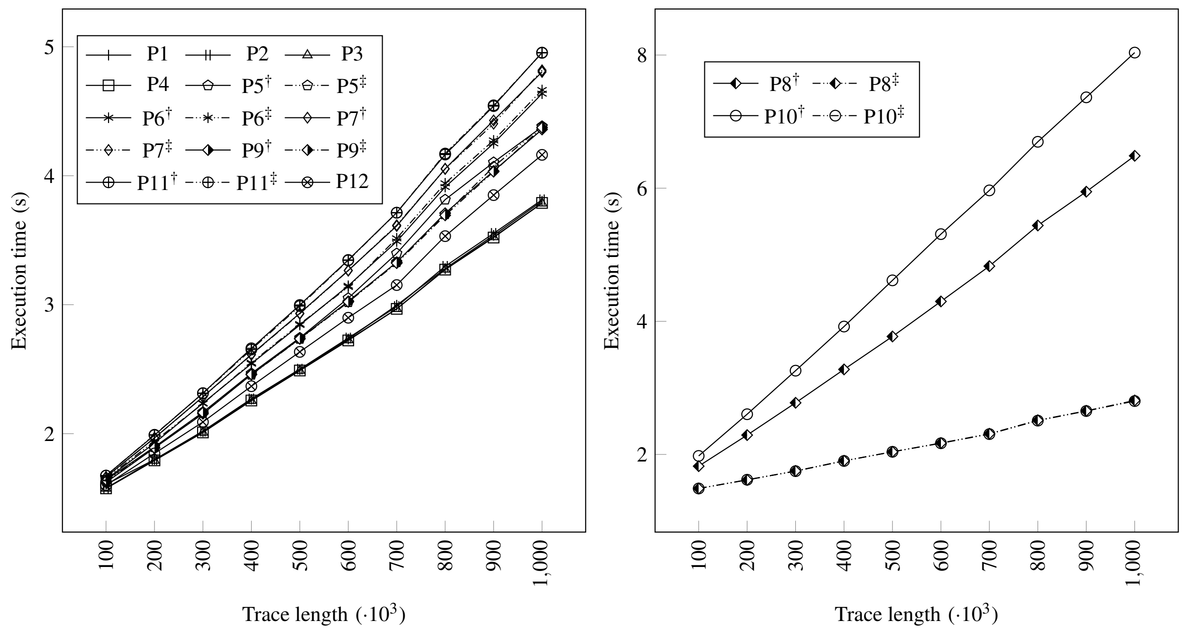
The *precedence* patterns used in the four properties can yield either `NSOR` or `WTO` violations. Hence, given the number of violations *n*, we generate traces that contain *n* occurrences of the second block (i.e., *n* `NSOR` violations), which for properties P7–P9 is event *B* and for property P10 is the events chain "C,D" . We also generate traces that contain *n* pairs of the two blocks defined in the patterns, which are in violation of the constraint on the time distance between the two blocks, i.e., *n* `WTO` violations. The strategy for choosing the value of the distance between the two blocks depends on the type of the comparison operator used in the distance constraint. Property P7 contains constraint "`at most 6000 tu`" between event *A* and *B*, so the s distance is randomly generated from $(6000, 6000*1.1]$. For property P8 which uses "`at least 100 tu`", the trace generator picks a random value from $[1,100)$. The trace generation strategy for property P10 is similar. As for property P9, which uses "`exactly 100 tu`", the distance is randomly chosen from the two ranges $[1,100)$ and $(100,100*1.1]$. According to the definition of the *precedence* pattern, an event chain that contains no distance constraint such as A,B is arranged consecutively at a randomly generated position and the distance between them is fixed to 1.

Properties using the *response* pattern. There are four properties using the *response* patterns, i.e., P5, P6, P11, and P12. For property P12, `NSOR` is the only possible type of violations. Given the number of violations *n*, the trace generator distributes event *B* randomly in all the *n* slots. For the other three properties, there are two possible violation types: `NSOR` and `WTO`. The trace generation strategies for them are similar to the ones used in the corresponding *precedence* patterns.

4.6.3 Evaluation Results

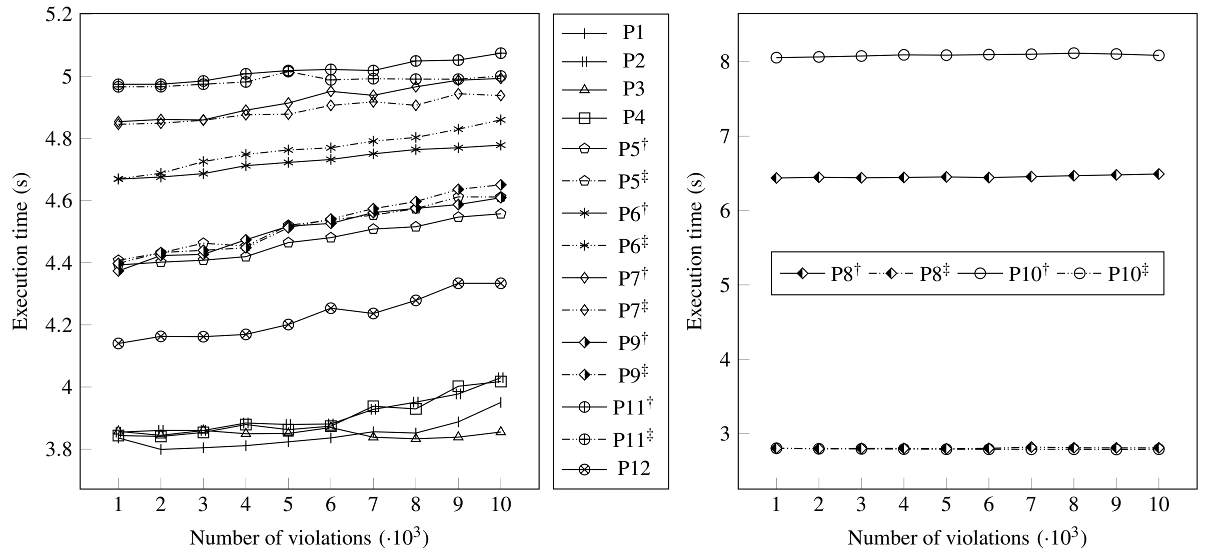
To answer the two research questions, the TEMPSY-REPORT tool has been run with properties P1–P12 on two sets of synthesized traces. As discussed in the previous subsection, since there are two possible types of violations (i.e., `NSOR` and `WTO`) for properties P5–P11, for each of the properties, we have generated two traces, each containing one of the two violation types.

In the experiments for answering RQ1, we generated traces by fixing the number of violations to 1K and varying the length from 100K to 1M. The 19 plots in Figure 4.4 depict the relation between the execution time (y-axis) of TEMPSY-REPORT and the trace length (x-axis). The execution of TEMPSY-REPORT on a trace containing `NSOR` violations is denoted by adding a superscript † to the property's name and a superscript ‡ is used to indicate the execution on a trace with `WTO` violations. Notice that we split the plots into two parts (i.e., Figures 4.4a and 4.4b), since the distance constraint



(a) P1–P7, P9, P11–P12 (b) P8, P10

Figure 4.4: Execution time of TEMPSY-REPORT for collecting violation information from faulty traces (fixed number of violations, various lengths)



(a) P1–P7, P9, P11–P12 (b) P8, P10

Figure 4.5: Execution time of TEMPSY-REPORT for collecting violation information from faulty traces (various numbers of violations, fixed length)

"`at least` *n* `tu`" used by properties P8 and P10 can be naturally satisfied after several violations, regardless of the strategy used for generating traces with `WTO` violations.

As shown in Figure 4.4, we answer RQ1 by observing that the TEMPSY-REPORT tool scales linearly as the trace length increases; the execution time ranges from about 1.5 s to 8.2 s, depending on the pattern used in the property and the violation type in the trace.

In the experiments for answering RQ2, we generated traces by fixing the length to 1M and varying the number of violations from 1K to 10K. The 19 plots in Figure 4.5 depict the relation between the execution time (y-axis) of TEMPSY-REPORT and the number of violations (x-axis). We separate the plots for P8 and P10 (i.e., Figure 4.4b) from the others (i.e., Figure 4.4a) for the same reason stated above (limited numbers of violations in the generated traces with `WTO` violations). Notice that the x-axis of Figure 4.5b is the same as the one in Figure 4.5a, since it does not affect the results.

As shown in Figure 4.5, we answer RQ2 by observing that the number of violations makes no tangible impact on the performance of the TEMPSY-REPORT tool, which stays approximately constant as the number of violations increases; the execution time ranges from about 3.8 s to 8.2 s depending on the pattern used in the property and the violation type in the trace.

Figures 4.4 and 4.5 show that TEMPSY-REPORT needs less time to collect violation information for the properties using the *universality*, *existence*, and *absence* patterns than for the properties using the *precedence* and *response* patterns. Figures 4.4b and 4.5b show that the tool spends less time in analyzing the generated traces with `WTO` violations than the ones with `NSOR` violations, for the *precedence* patterns (e.g., P8, P10) using "`at least`" (as the comparison operator) in the distance constraint between the two blocks. The plots for P8 and P10 also indicate that it requires more time to cope with the generated traces containing `NSOR` violations for the *precedence* patterns using "`at least`" in the middle distance constraint than with the traces generated for the other patterns, by given the same number of violations. This is because the worst-case execution time of function `reportPatternPrecedenceAtLeast` (Algorithm 25), in particular when the violation type is `NSOR`, is a bit more expensive than the other homologous functions. More specifically, our function has one more *if-else* statement (lines 9–17) for checking whether the first block has already been found before a match of the second block and also it takes time to confirm if the current match of the first block is invalid with respect to the distance constraints defined within the block (line 13). Moreover, the results show that our approach spends more time in collecting violation information for an *order* pattern in which the blocks contains more events (e.g., P10, P11).

4.6.4 Discussion

The evaluation results presented in the previous subsection have shown the feasibility of applying our model-driven approach to violation information collection for trace checking of temporal properties in realistic settings.

Our TEMPSY-REPORT tool is a viable technology from a performance viewpoint, as it can analyze very large traces (with one million events) in less than ten seconds. The tool scales linearly with respect to the length of the input trace and keeps approximately constant performance with respect to the number of violations. Note that "the input trace" can also correspond to a sub-trace isolated from a larger execution trace (see also Section 3.7.5). We also remark that our TEMPSY-REPORT tool fulfills requirement R2 (see Chapter 1) which calls for using standard MDE technologies.

At the time of writing, we can not find a tool in the literature that is comparable with our TEMPSY-REPORT tool, in terms of collecting *comprehensible* information (e.g., violation positions and reasons) about temporal violations detected by trace checking. A detailed comparison between the state-of-the-art tools for violation reporting for trace checking of temporal properties and TEMPSY-REPORT will be drawn in the next chapter.

Overall, our model-driven approach to violation information collection for trace checking of temporal properties is viable and is able to collect more comprehensible information about temporal violations than the state of the art.

4.6.4.1 Threats to validity

The main threat to the validity of the evaluation results is the intrinsic presence of errors in TEMPSY-REPORT. We tried to compensate for this by thoroughly testing the tool with traces and properties for which the oracle was previously known.

Another potential threat is from the *synthesized* traces that were generated by our trace generator program. As explained at the beginning of this section, synthesized traces do not affect our research question on scalability and are better than real ones as they allows us to control certain factors and varying others randomly.

Chapter 5

Related Work

The work presented in this thesis is related to MDE approaches for specifying temporal properties, to approaches for trace checking/run-time verification, and to approach for reporting violation information. We review these areas in the next three subsections.

5.1 MDE Approaches for Specifying Temporal Properties

There have been several proposals in the MDE community to define high-level specification languages for expressing temporal properties; all these proposals are realized as temporal extensions of OCL. In the rest of this section we summarize them and discuss their differences and limitations with respect to *TemPsy*.

5.1.1 Pattern-based temporal extensions of OCL

The approaches that are most similar to *TemPsy* are those that extend OCL with support for Dwyer et al.'s property specification patterns.

Table 5.1: Comparison between pattern-based temporal extensions of OCL and *TemPsy*

Language	Features				Tool
	NOOP	TDOP	SOS	TDS	support
[Flake and Müller, 2003]	-	+	-	-	-
[Küster-Filipe and Anderson, 2006]	-	-	-	*	-
[Robinson, 2008]	+	*	-	*	n/a
[Kanso and Taha, 2014]	+	*	-	*	+
TemPsy	+	+	+	+	+

Legend. NOOP: Number of Occurrences in *occurrence* Patterns; TDOP: Time Distance in *order* Patterns; SOS: Specific Occurrence in Scopes; TDS: Time Distance in Scopes; +: full support; -: no support; *: partial support; n/a: tool mentioned in the paper but not available.

Flake and Mueller [Flake and Müller, 2003] define a state-oriented OCL extension for expressing Dwyer et al.'s patterns over UML Statecharts configurations. The extension is based on the introduction of a special temporal operation, which can be applied to objects that have an associated Statechart. The evaluation of this operation at a certain time point yields the set of state configuration sequences in the time interval defined by the parameters of the operation. The extension, in addition to allowing for expressing the original definition of patterns in [Dwyer et al., 1999], adds also the support for specifying time distances in order patterns.

Küster-Filipe and Anderson [Küster-Filipe and Anderson, 2006] propose a liveness template for OCL to define future-oriented time-bounded constraints that are expressed with a time-bounded *after* scope and an *existence* pattern. This template is defined in terms of the real-time temporal logic of knowledge, interpreted over timed automata, to allow for formal reasoning. The expressiveness of this extension is very limited since it supports only one scope/pattern combination.

Robinson [Robinson, 2008] presents a temporal extension of OCL called OCL_{TM}, developed in the context of a framework for monitoring of requirements expressed using a goal model. OCL_{TM} includes all the operators corresponding to standard LTL modalities, and supports Dwyer et al.'s patterns and time distances in patterns. In this regard, it is very close to the expressiveness of *TemPsy*, though it supports neither the reference to a specific occurrence of an event in scopes nor two types of constraints (as *TemPsy* does with the keywords '`at least`' and '`exactly`') on time distances in scopes and *order* patterns.

Kanso and Taha [Kanso and Taha, 2014] introduce Temporal OCL, a pattern-based temporal extension of OCL. Although the support for temporal patterns is very similar between the two languages, Temporal OCL does not allow references to specific event occurrences in scope boundaries and does not fully support constraints on the time distance from a scope boundary (it only supports state-change events).

Table 5.1 provides a comparison of these four approaches with *TemPsy*, in terms of the following language features, derived from the analysis of the requirements specifications of our case study (see Section 2.2.1): 1) the possibility of referring to the number of occurrences of an event in *occurrence* patterns (NOOP); 2) the possibility of defining a time distance between events in *order* patterns (TDOP); 3) the possibility of referring to a specific occurrence of an event in scopes (SOS); 4) the possibility of defining a constraint on the time distance from scope boundaries (TDS). The table also indicates whether the proposed language extension includes tool support.

As you can see, *TemPsy* is the only pattern-based language that provides support for all the specific features needed for the specification of requirements in the context of our case study.

5.1.2 Other temporal extensions of OCL

Temporal extensions of OCL that are not pattern-based are mainly realized by extending the language with temporal operators borrowed from standard temporal logic, such as "always", "until", "eventually", "next". A preliminary work in this direction appeared in [Conrad and Turowski, 2001]. OCL/RT [Cengarle and Knapp, 2002] extends OCL with the notion of timestamped events (based on the original UML abstract meta-class `Event`) and two temporal operators, "always" and "sometimes". Events are associated with instances of classifiers and, by means of a special satisfaction operator, it is possible to evaluate an expression at the time instant when a certain event occurred. The OCL/RT extension allows for expressing real-time deadline and timeout constraints but requires to reason explicitly at the lowest-level of abstraction, in terms of time instants. Lavazza et al. [Lavazza et al., 2003] define the Object Temporal Logic (OTL), which allows users to write temporal constraints on Real-time UML (UML-RT) models. In particular, it supports the concepts of *Time*, *Duration*, and *Interval* to specify the time distance between events. Nevertheless, the language is modeled after the TRIO temporal logic [Morzenti et al., 1992], and the properties are written using a low level of abstraction. Ziemann and Gogolla [Ziemann and Gogolla, 2003] propose TOCL, an extension of OCL with LTL operators, to specify constraints on the temporal evolution of the system states. Being based on LTL, TOCL does not support real-time constraints. Bill et al. [Bill et al., 2014] define cOCL, an extension of OCL with CTL temporal operators to express properties over the lifetime of an instance model. These properties are then verified with an explicit state space model checking framework. Being based on CTL, cOCL does not support real-time constraints. The work on Flake and Mueller [Flake and Mueller, 2004] goes in a similar direction, proposing an extension of OCL that allows for the specification of past- and future-oriented time-bounded constraints. They do not support event-based specifications; moreover, the proposed mapping into Clocked LTL does not allow to rely on standard OCL tools. Soden and Eichler [Soden and Eichler, 2009] propose Linear Temporal OCL (LT-OCL) for languages defined over MOF meta-models in conjunction with operational semantics. LT-OCL contains the standard LTL operators. The interpretation of LT-OCL formulae is defined in the context of a MOF meta-model and its dynamic behavior specified by action semantics using the M3Actions framework.

Since all these temporal extensions of OCL are based on some temporal logic and include temporal logic operators, they intrinsically inherit the limitations of other specification approaches based on temporal logic: 1) they require strong theoretical and mathematical background, which are rarely found among practitioners; 2) they provided limited tool support, often based on prototypes that do not scale for industrial applications.

A different type of support for temporal constraints is proposed by Cabot et al. [Cabot et al., 2003]. They extend UML to use UML/OCL as a temporal conceptual modeling language, introducing the concepts of *durability* and *frequency* for the definition of temporal features of UML classifiers and associations. They define temporal operations in OCL through which it is possible to refer to any past state of the system. These operations are mapped into standard OCL by relying on the mapping of the temporally-extended conceptual schema into a conventional UML one, which explicitly instantiates the concepts of time interval and instant. However, the temporal operations are geared to express temporal integrity constraints on the model, rather than temporal properties correlating events of the system.

5.2 Trace Checking and Run-time Verification

Model-driven technologies have been used in various work on (run-time) trace and/or assertion checking. The model-driven approach for assertion checking proposed in [Zhang et al., 2005] relies on the principles of aspect-oriented programming and uses a technique called two-level aspect weaving. First, cross-cutting assertions defined using ECL, an extension of OCL, are weaved into a model defined within GME (Generic Modeling Environment [Davis, 2003]) and then the code for checking the contracts specified in the models is generated using model-driven program transformations [Gray

Table 5.2: The state of the art of violation reporting for offline trace checking of temporal properties

Tool	Bool	Positions	Reasons	Visualization
BREACH [Donzé, 2010]	+	+		*
MONPOLY [Basin et al., 2012]	+	+		
QEA [Reger et al., 2015]	+	*		
SOLOIST+ZOT [Bersani et al., 2014]	+			
RV-MONITOR [Luo et al., 2014]	+		*	

Legend. Bool: the boolean checking result; Positions: all offending positions in the faulty trace; Reasons: the reasons for all violations; Visualization: a comprehensive visualization facility for inspecting violations; *: partial support.

et al., 2004]. ECL does not support the expression of temporal constraints. An approach conceptually similar to ours is proposed in [Engels et al., 2006], in which pre- and post-conditions are expressed with visual contracts defined using graph transformations and then transformed into a code-level representation as JML (Java Modeling Language) assertions. The pre- and post-conditions that can be expressed in this framework are functional and do not support temporal expressions. Reference [Simmonds et al., 2009] proposes a model-driven approach for monitoring Web services in which temporal properties, expressed using property specification patterns [Dwyer et al., 1999], are defined with a subset of UML 2.0 Sequence Diagrams and checked at run time by translating sequence diagrams into non-deterministic finite automata. However, the properties used in this thesis, differently from those that can be expressed with *TemPsy*, do not support expressing timing requirements. Our model-driven approach for trace checking can be easily applied in scenarios where other trace models are used, as long as OCL invariants can be expressed on them; examples of these models are those proposed in [Briand et al., 2006] (designed for the reverse engineering of UML sequence diagrams from traces) and [Hamou-Lhadj and Lethbridge, 2012] (tailored for the exchange of traces corresponding to large program call trees).

This thesis is also related to the more general area of run-time verification [Leucker and Schallhart, 2009]. The majority of the approaches proposed in this area (e.g., [Finkbeiner et al., 2005, Basin et al., 2008, Basin et al., 2013, Barre et al., 2013, Bianculli et al., 2014, Bersani et al., 2014]) focuses on the verification of temporal properties expressed using some temporal logic. These approaches define the trace checking/run-time verification problem in terms of a *word problem*, i.e., the problem of whether a given word is included in some languages, and rely on formal verification tools like model checkers or SAT/SMT solvers. In our approach, we use a domain-specific specification language (*TemPsy*) and rely on standard MDE technologies.

5.3 Violation Reporting for Trace Checking

To review the state of the art for violation reporting for offline trace checking of temporal properties, we analyzed the output of the tools 1) presented as a demo and/or 2) contestants in the "offline monitoring" track of the International Competition on Runtime Verification (CRV), both included in the program of the 2014 and 2015 editions of the International Conference on Runtime Verification.

Table 5.2 summarizes whether the output of each trace checking tool includes the following four features: 1. the boolean result of the checking (i.e., false or any equivalent conclusion for a faulty trace); 2. all offending positions in the faulty trace; 3. the reasons for all violations; 4. a comprehensive visualization facility for inspecting violations. Note that four of the tools mentioned in the trace checking competitions - STEPR, AGMON [Kane et al., 2014], LOGFIRE [Havelund, 2015], OPTYSIM [Diaz et al., 2011] - are not publicly available; another tool RITHM-v2.0 [Navabpour et al., 2013], is available but does not work when executed by following the instructions specified in the *README* file on its GitHub page [Yogi Joshi, 2016].

One can see that besides the boolean output obtained when checking a faulty trace, state-of-the-art tools rarely report useful violation information. Although two of them are able to indicate the offending positions of all violations, none can systematically indicate the reason behind each violation, or integrates a comprehensive visualization facility for inspecting those violations. As a Matlab/C++ tool, BREACH [Donzé, 2010] is not developed for offline trace checking, though it can check digitized traces against Signal Temporal Logic (STL) specifications. The output of BREACH shows the boolean checking results on the plot of a trace, which is a very basic support for visualizing violations. Furthermore, BREACH does not indicate the reason for each violation and lacks features for inspecting violations (e.g., jumping to a specific violation). MONPOLY [Basin et al., 2012] is an offline trace checking tool that prints out the offending log lines when violations are found in a trace. However, MONPOLY only provides textual output and it does not explain the reason for each violation. QEA [Reger et al., 2015] is a trace checking tool that stops checking at the first violation; it produces the first offending line as part of output. SOLOIST+ZOT [Bersani et al., 2014] uses a bounded satisfiability checker and returns only a boolean result. RV-MONITOR [Luo et al., 2014] is a tool designed for runtime verification, but it is also able to check traces using a front-end log reader called RV-LOG [He Xiao, 2016]. RV-MONITOR does not allow keeping track of the offending positions in a faulty trace. It provides the possibility of manually writing a violation handler to print some relative information about a violation. However, since it does not produce any additional information to understand the reasons and the types of violations, the output of the handler is not very informative. Moreover, RV-MONITOR does not allow visually inspecting violations.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The correct enactment of business processes in the context of eGovernment is of vital importance to supply reliable digital solutions to citizens and enterprises, as well as to foster an effective cooperation of the various public administrations in a state. To ensure the correct enactment of a business process, offline trace checking provides a means to check the compliance of the business process with respect to its requirements, by analyzing the trace of events produced by the system at run time. In addition, a violation reporting procedure complements offline trace checking by collecting violation information from a faulty trace and presenting the data to end users, and hence allowing for understanding the violations.

In this thesis, we have presented a practical and scalable solution for the offline checking of the temporal requirements of business processes, which can be used in contexts where model-driven engineering is already a practice, where temporal specifications should be written in a domain-specific language not requiring a strong mathematical background, and where relying on standards and industry-strength tools for property checking is a fundamental prerequisite. We have applied this solution, as a case study, to the particular context of eGovernment, in collaboration with our public service partner CTIE.

This thesis has made the following contributions:

i) the *TemPsy* language, a pattern-based domain-specific language for the specification of temporal properties, developed in the context of business process models;

ii) a model-driven trace checking procedure, which relies on an optimized mapping of temporal requirements written in *TemPsy* into OCL constraints on a conceptual model of execution traces;

iii) a model-driven approach to violation information collection, which relies on the evaluation of OCL queries on an instance of the trace model;

iv) three publicly-available tools: 1) TEMPSY-CHECK and 2) TEMPSY-REPORT, implementing, respectively, the trace checking and violation information collection procedures; 3) an interactive visualization tool for navigating and analyzing the violation information collected by TEMPSY-REPORT;

v) an evaluation of the scalability of TEMPSY-CHECK and TEMPSY-REPORT, when applied to the verification of real properties derived from a case study of our public service partner.

The experimental results of the evaluation of the scalability of TEMPSY-CHECK show the feasibility of applying our model-driven offline trace checking procedure in realistic settings. The TEMPSY-CHECK tool proves favorable with respect to the state of the art. It scales linearly with respect to the length of the input trace to check and is able to analyze traces with one million events in about two seconds.

As a complement to the offline trace checking procedure, the TEMPSY-REPORT tool provides end users with the possibility to navigate and understand temporal violations; such a functionality is not common or very limited among state-of-the-art trace checking tools. The evaluation of TEMPSY-REPORT shows that TEMPSY-REPORT is able to collect violation information from large traces (with one million events) in less than ten seconds. The TEMPSY-REPORT tool scales linearly with respect to the length of the trace and keeps approximately constant performance as the number of violations increases.

6.2 Future Work

The work presented in this thesis is part of a broader project in collaboration with CTIE, on model-driven run-time verification of business processes [Dou et al., 2014c]. The next step is to embed our approaches for trace checking and violation reporting within the business process execution platform of our partner, to realize an efficient run-time verification platform for temporal properties of business process-based applications.

In the future, we also plan to extend the work along the following directions: i) conducting a usability study of the *TemPsy* language, to assess the usability with respect to other specification methods (e.g., temporal logic); ii) applying the procedures for trace checking and violation reporting to other MDE contexts different from eGovernment business process modeling, with the possibility of extending *TemPsy* with additional constructs, as required by the new application domains; iii) carrying out a usability study of our visualization tool for violation information.

Bibliography

[Alexander et al., 1977] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A pattern language. Towns, buildings, construction*. Oxford University Press.

[Autili et al., 2015] Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., and Tang, A. (2015). Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. *IEEE Trans. Softw. Eng.*, 41(7):620–638.

[Baresi et al., 2007] Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., and Spoletini, P. (2007). Validation of web service compositions. *IET Softw.*, 1(6):219–232.

[Baresi and Guinea, 2005] Baresi, L. and Guinea, S. (2005). Towards dynamic monitoring of WS-BPEL processes. In *Proc. ICSOC 2005*, volume 3826 of *LNCS*, pages 269–282. Springer.

[Baresi et al., 2009] Baresi, L., Guinea, S., Pistore, M., and Trainotti, M. (2009). Dynamo + astro: An integrated approach for BPEL monitoring. In *Proc. ICWS '09*, pages 230–237. IEEE.

[Barre et al., 2013] Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.-A., and Hallé, S. (2013). MapReduce for parallel trace validation of LTL properties. In *Proc. RV 2012*, volume 7687 of *LNCS*, pages 184–198. Springer.

[Barringer et al., 2012] Barringer, H., Falcone, Y., Havelund, K., Reger, G., and Rydeheard, D. (2012). Quantified event automata: Towards expressive and efficient runtime monitors. In *Proc. FM 2012*, volume 7436 of *LNCS*, pages 68–84. Springer.

[Bartocci et al., 2014] Bartocci, E., Bonakdarpour, B., and Falcone, Y. (2014). First international competition on software for runtime verification. In *Proc. RV 2014*, volume 8734 of *LNCS*, pages 1–9. Springer.

[Basin et al., 2012] Basin, D., Harvan, M., Klaedtke, F., and Zălinescu, E. (2012). MONPOLY: Monitoring usage-control policies. In *Proc. RV 2011*, volume 7186 of *LNCS*, pages 360–364.

[Basin et al., 2013] Basin, D., Klaedtke, F., Marinovic, S., and Zălinescu, E. (2013). Monitoring of temporal first-order properties with aggregations. In *Proc. RV 2013*, volume 8174 of *LNCS*, pages 40–58. Springer.

[Basin et al., 2008] Basin, D., Klaedtke, F., Müller, S., and Pfitzmann, B. (2008). Runtime monitoring of metric first-order temporal properties. In *Proc. FSTTCS '08*, pages 49–60. IBFI Schloss Dagstuhl.

[Bersani et al., 2016] Bersani, M., Bianculli, D., Ghezzi, C., Krstić, S., and San Pietro, P. (2016). Efficient large-scale trace checking using MapReduce. In *Proc. ICSE 2016*. ACM. to be published.

[Bersani et al., 2014] Bersani, M. M., Bianculli, D., Ghezzi, C., Krstić, S., and San Pietro, P. (2014). SMT-based checking of SOLOIST over sparse traces. In *Proc. FASE 2014*, volume 8411 of *LNCS*, pages 276–290. Springer.

[Bianculli et al., 2014] Bianculli, D., Ghezzi, C., and Krstić, S. (2014). Trace checking of metric temporal logic with aggregating modalities using MapReduce. In *Proc. SEFM 2014*, volume 8702 of *LNCS*, pages 144–158. Springer.

[Bianculli et al., 2012] Bianculli, D., Ghezzi, C., Pautasso, C., and Senti, P. (2012). Specification patterns from research to industry: a case study in service-based applications. In *Proc. ICSE 2012*, pages 968–976. IEEE.

[Bianculli et al., 2013] Bianculli, D., Ghezzi, C., and San Pietro, P. (2013). The tale of SOLOIST: a specification language for service compositions interactions. In *Proc. FACS'12*, volume 7684 of *LNCS*, pages 55–72. Springer.

[Bianculli et al., 2007] Bianculli, D., Ghezzi, C., and Spoletini, P. (2007). A model checking approach to verify BPEL4WS workflows. In *Proc. SOCA '07*, pages 13–20. IEEE.

[Bill et al., 2014] Bill, R., Gabmeyer, S., Kaufmann, P., and Seidl, M. (2014). Model checking of CTL-extended OCL specifications. In *Proc. SLE 2014*, volume 8706 of *LNCS*, pages 221–240. Springer.

[Brambilla et al., 2010] Brambilla, M., Butti, S., and Fraternali, P. (2010). WebRatio BPM: A tool for designing and deploying business processes on the web. In *Proc. ICWE 2010*, volume 6189 of *LNCS*, pages 415–429. Springer.

[Briand et al., 2006] Briand, L. C., Labiche, Y., and Leduc, J. (2006). Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Softw. Eng.*, 32(9):642–663.

[Cabot et al., 2003] Cabot, J., Olivé, A., and Teniente, E. (2003). Representing temporal information in UML. In *Proc. UML 2003*, volume 2863 of *LNCS*, pages 44–59. Springer.

[Cengarle and Knapp, 2002] Cengarle, M. and Knapp, A. (2002). Towards OCL/RT. In *Proc. FME 2002*, volume 2391 of *LNCS*, pages 390–409. Springer.

[Conrad and Turowski, 2001] Conrad, S. and Turowski, K. (2001). Temporal OCL: Meeting specification demands for business components. In *Unified Modeling Language*, pages 151–165. IGI Global.

[Davis, 2003] Davis, J. (2003). GME: The generic modeling environment. In *Companion of the Proc. of OOPSLA '03*, pages 82–83. ACM.

[Diaz et al., 2011] Diaz, A., Merino, P., and Salmeron, A. (2011). Obtaining models for realistic mobile network simulations using real traces. *IEEE Communications Letters*, 15(7):782–784.

[Donzé, 2010] Donzé, A. (2010). Breach, a toolbox for verification and parameter synthesis of hybrid systems. In Touili, T., Cook, B., and Jackson, P., editors, *Proc. CAV 2010*, pages 167–170. Springer.

[Dou et al., 2014a] Dou, W., Bianculli, D., and Briand, L. (2014a). A model-based approach to offline trace checking of temporal properties with OCL. Technical Report TR-SnT-2014-5, SnT Centre - University of Luxembourg.

[Dou et al., 2014b] Dou, W., Bianculli, D., and Briand, L. (2014b). OCLR: a more expressive, pattern-based temporal extension of OCL. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 51–66. Springer.

[Dou et al., 2014c] Dou, W., Bianculli, D., and Briand, L. (2014c). Revisiting model-driven engineering for run-time verification of business processes. In *Proc. SAM 2014*, volume 8769 of *LNCS*, pages 190–197. Springer.

[Dwyer et al., 1999] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proc. ICSE 1999*, pages 411–420. IEEE.

[Eclipse, 2015a] Eclipse (2015a). Eclipse OCL tools. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[Eclipse, 2015b] Eclipse (2015b). Xtext–Language Engineering Made Easy! `http://www.eclipse.org/Xtext/`.

[Engels et al., 2006] Engels, G., Lohmann, M., Sauer, S., and Heckel, R. (2006). Model-driven monitoring: An application of graph transformation for design by contract. In *Proc. ICGT 2006*, volume 4178 of *LNCS*, pages 336–350. Springer.

[Felder and Morzenti, 1994] Felder, M. and Morzenti, A. (1994). Validating real-time systems by history-checking TRIO specifications. *ACM Trans. Softw. Eng. Methodol.*, 3(4):308–339.

[Finkbeiner et al., 2005] Finkbeiner, B., Sankaranarayanan, S., and Sipma, H. (2005). Collecting statistics over runtime executions. *Form. Method Syst. Des.*, 27:253–274.

[Flake and Mueller, 2004] Flake, S. and Mueller, W. (2004). Past- and future-oriented time-bounded temporal properties with OCL. In *Proc. SEFM 2004*, pages 154–163. IEEE.

[Flake and Müller, 2003] Flake, S. and Müller, W. (2003). Expressing property specification patterns with OCL. In *Proc. SERP '03*, pages 595–603. CSREA Press.

[Fu et al., 2004] Fu, X., Bultan, T., and Su, J. (2004). Analysis of interacting BPEL web services. In *Proc. WWW '04*, pages 621–630. ACM.

[Furia et al., 2012] Furia, C. A., Mandrioli, D., Morzenti, A., and Rossi, M. (2012). *Modeling Time in Computing*. Springer.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Gray et al., 2004] Gray, J., Zhang, J., Lin, Y., Roychoudhury, S., Wu, H., Sudarsan, R., Gokhale, A., Neema, S., Shi, F., and Bapty, T. (2004). Model-driven program transformation of a large avionics framework. In *Proc. GPCE 2004*, volume 3286 of *LNCS*, pages 361–378. Springer.

[Gruhn and Laue, 2006] Gruhn, V. and Laue, R. (2006). Patterns for timed property specifications. *Electron. Notes Theor. Comput. Sci.*, 153(2):117–133.

[Grunske, 2008] Grunske, L. (2008). Specification patterns for probabilistic quality properties. In *Proc. ICSE 2008*, pages 31–40. ACM.

[Hallé et al., 2009] Hallé, S., Villemaire, R., and Cherkaoui, O. (2009). Specifying and validating data-aware temporal web service properties. *IEEE Trans. Softw. Eng.*, 35(5):669–683.

[Hamou-Lhadj and Lethbridge, 2012] Hamou-Lhadj, A. and Lethbridge, T. C. (2012). A metamodel for the compact but lossless exchange of execution traces. *Softw. Syst. Model.*, 11(1):77–98.

[Havelund, 2015] Havelund, K. (2015). Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170.

[He Xiao, 2016] He Xiao (2016). RV-Log. `https://github.com/runtimeverification/RV-Log/tree/crv15`.

[Kallel et al., 2009] Kallel, S., Charfi, A., Dinkelaker, T., Mezini, M., and Jmaiel, M. (2009). Specifying and monitoring temporal properties in web services compositions. In *Proc. ECOWS '09*, pages 148–157. IEEE Computer Society.

[Kane et al., 2014] Kane, A., Fuhrman, T., and Koopman, P. (2014). Monitor based oracles for cyber-physical system testing: practical experience report. In *Proc. DSN 2014*, pages 148–155. IEEE.

[Kanso and Taha, 2014] Kanso, B. and Taha, S. (2014). Specification of temporal properties with OCL. *Sci. Comput. Program.*, 96, Part 4:527–551.

[Konrad and Cheng, 2005] Konrad, S. and Cheng, B. H. C. (2005). Real-time specification patterns. In *Proc. ICSE '05*, pages 372–381. ACM.

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

[Küster-Filipe and Anderson, 2006] Küster-Filipe, J. and Anderson, S. (2006). On a time enriched OCL liveness template. *STTT*, 8(2):156–166.

[Lavazza et al., 2003] Lavazza, L., Morasca, S., and Morzenti, A. (2003). A dual language approach extension to UML for the development of time-critical component-based systems. *Electron. Notes Theor. Comput. Sci.*, 82(6):121–132.

[Leucker and Schallhart, 2009] Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303.

[Li et al., 2005] Li, Z., Han, J., and Jin, Y. (2005). Pattern-based specification and validation of web services interaction properties. In *Proc. ICSOC 2005*, volume 3826 of *LNCS*, pages 73–86. Springer.

[Lumpe et al., 2011] Lumpe, M., Meedeniya, I., and Grunske, L. (2011). PSPWizard: machine-assisted definition of temporal logical properties with specification patterns. In *Proc. ESEC/FSE '11*, pages 468–471. ACM.

[Luo et al., 2014] Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P. O., Şerbănuţă, T. F., and Roşu, G. (2014). RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In Bonakdarpour, B. and Smolka, S. A., editors, *Proc. RV 2014*, pages 285–300. Springer.

[Morzenti et al., 1992] Morzenti, A., Mandrioli, D., and Ghezzi, C. (1992). A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14:521–573.

[Mrad et al., 2013] Mrad, A., Ahmed, S., Hallé, S., and Beaudet, E. (2013). BabelTrace: A collection of transducers for trace validation. In *Proc. RV 2012*, volume 7687 of *LNCS*, pages 126–130. Springer.

[Navabpour et al., 2013] Navabpour, S., Joshi, Y., Wu, W., Berkovich, S., Medhat, R., Bonakdarpour, B., and Fischmeister, S. (2013). RiTHM: A tool for enabling time-triggered runtime verification for C programs. In *Proc. ESEC/FSE 2013*, pages 603–606. ACM.

[OMG, 2011a] OMG (2011a). BPMN Specification. `http://www.bpmn.org`.

[OMG, 2011b] OMG (2011b). Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.1. `http://www.omg.org/spec/MARTE/1.1/PDF`.

[OMG, 2012] OMG (2012). ISO/IEC 19507 (OCL v2.3.1). `http://www.omg.org/spec/OCL/ISO/19507/PDF`.

[Post et al., 2012] Post, A., Menzel, I., Hoenicke, J., and Podelski, A. (2012). Automotive behavioral requirements expressed in a specification pattern system: A case study at BOSCH. *Requir. Eng.*, 17(1):19–33.

[Raimondi et al., 2008] Raimondi, F., Skene, J., and Emmerich, W. (2008). Efficient online monitoring of web-service SLAs. In *Proc. SIGSOFT '08/FSE-16*, pages 170–180. ACM.

[Reger et al., 2015] Reger, G., Cruz, H. C., and Rydeheard, D. (2015). MarQ: Monitoring at runtime with QEA. In Baier, C. and Tinelli, C., editors, *Proc. TACAS 2015*, pages 596–610. Springer.

[Robinson, 2008] Robinson, W. N. (2008). Extended OCL for goal monitoring. *ECEASST*, 9.

[Simmonds et al., 2009] Simmonds, J., Gan, Y., Chechik, M., Nejati, S., O'Farrell, B., Litani, E., and Waterhouse, J. (2009). Runtime monitoring of web service conversations. *IEEE Trans. Serv. Comput.*, 2(3):223–244.

[Soden and Eichler, 2009] Soden, M. and Eichler, H. (2009). Temporal extensions of OCL revisited. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 190–205. Springer.

[Software AG, 2014] Software AG (2014). ARIS. `http://www.softwareag.com/corporate/products/aris/default.asp`.

[Yogi Joshi, 2016] Yogi Joshi (2016). RiTHM-v2.0. `https://github.com/yogirjoshi/maven-repo`.

[Zhang et al., 2005] Zhang, J., Gray, J., and Lin, Y. (2005). A model-driven approach to enforce crosscutting assertion checking. In *Proc. MACS '05*, pages 1–5. ACM.

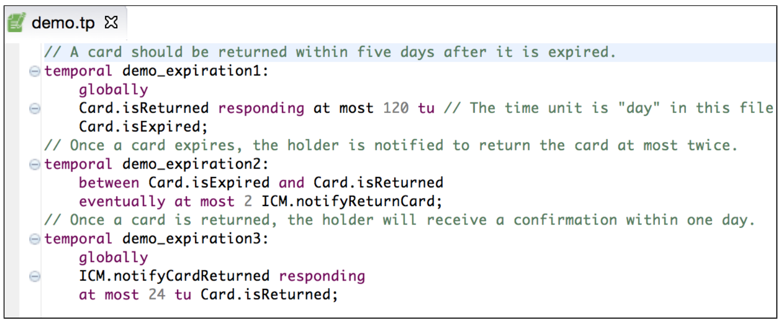
[Ziemann and Gogolla, 2003] Ziemann, P. and Gogolla, M. (2003). OCL extended with temporal logic. In *Proc. PSI 2003*, volume 2890 of *LNCS*, pages 351–357. Springer.

Appendix A

Tool Support

As mentioned in Chapter 3 and Chapter 4, we implemented TEMPSY-CHECK, TEMPSY-REPORT, and TEMPSY-VIOLATIONVISUALIZATION to facilitate offline trace checking of temporal properties. A basic scenario of checking temporal properties of a system on a certain execution trace includes three steps: 1) specifying the temporal properties as *TemPsy* expressions; 2) checking the *TemPsy* expressions on the trace; 3) invoking the violation reporting procedure to inspect violation information if any *TemPsy* expression is violated.

In the rest of this section, we illustrate how our *TemPsy* language, trace checking procedure, and violation reporting procedure can be combined to facilitate offline trace checking of these temporal properties. In the example, we use three temporal properties of the business process presented in Section 2.6 and an execution trace that consists of 10000 events.



demo.tp

```
// A card should be returned within five days after it is expired.
temporal demo_expiration1:
    globally
    Card.isReturned responding at most 120 tu // The time unit is "day" in this file
    Card.isExpired;
// Once a card expires, the holder is notified to return the card at most twice.
temporal demo_expiration2:
    between Card.isExpired and Card.isReturned
    eventually at most 2 ICM.notifyReturnCard;
// Once a card is returned, the holder will receive a confirmation within one day.
temporal demo_expiration3:
    globally
    ICM.notifyCardReturned responding
    at most 24 tu Card.isReturned;
```

Figure A.1: *TemPsy* editor

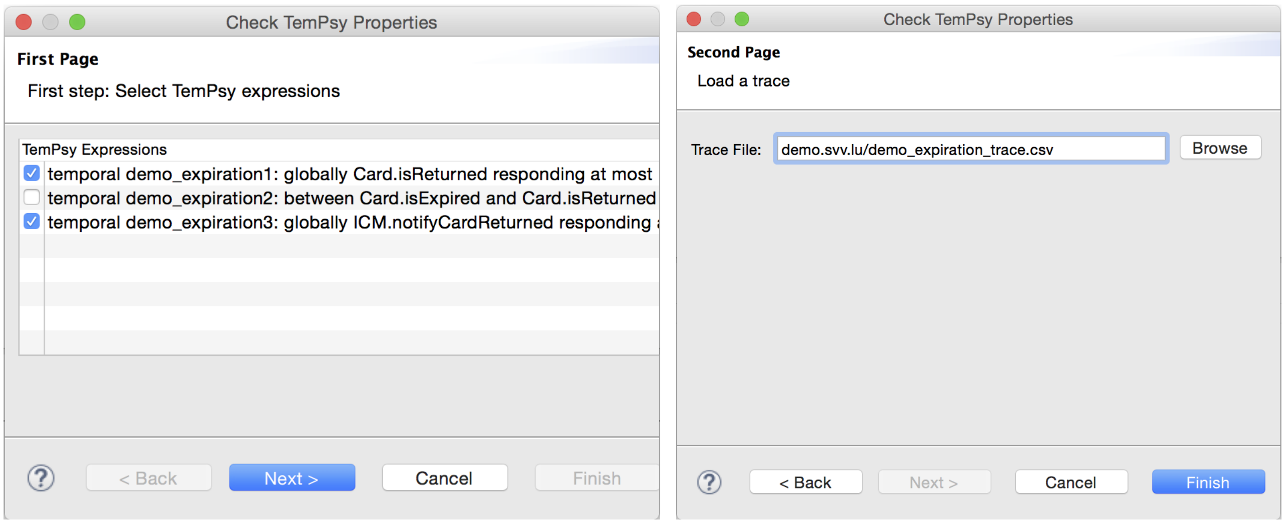
Specification of temporal properties. The three temporal properties are:

1) A card should be returned within five days after it expired.
2) If a card expires, the card holder is notified to return the card at most twice.
3) Once a card is returned, the card holder will receive a confirmation within one day.

They can be specified with the following *TemPsy* expressions:

1) `globally Card.isReturned responding at most 120 tu Card.isExpired`
2) `globally between Card.isExpired and Card.isReturned eventually at most 2 ICM.notifyReturned`
3) `globally ICM.notifyCardReturned responding at most 24 tu Card.isReturned`

The specification of temporal properties is supported by a *TemPsy* editor generated by Xtext. The Xtext definition of the *TemPsy* language is publicly available in the repository[1] of TEMPSY-CHECK.



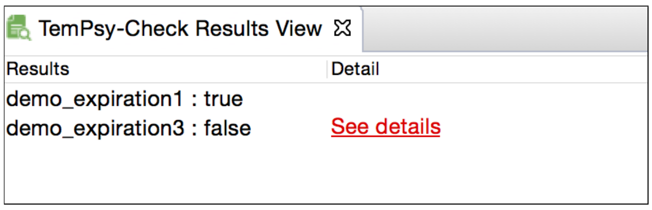
✓ temporal demo_expiration1: globally Card.isReturned responding at most
☐ temporal demo_expiration2: between Card.isExpired and Card.isReturned
✓ temporal demo_expiration3: globally ICM.notifyCardReturned responding a

Trace File: demo.svv.lu/demo_expiration_trace.csv

(a) Selecting *TemPsy* expressions (b) Loading an execution trace

Figure A.2: Checking *TemPsy* expressions on an execution trace

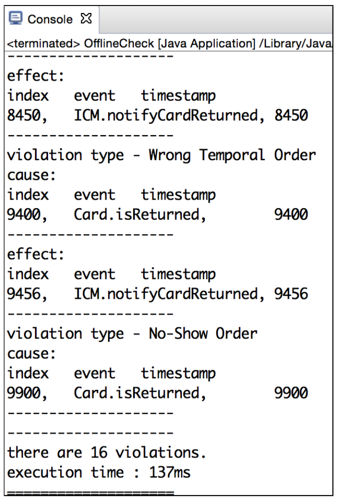
Trace checking procedure. There are two steps in the trace checking procedure. Given the three *TemPsy* expressions and the execution trace, the user first selects the expressions of interest (as shown in Figure A.2a) and then loads the trace to check (as shown in Figure A.2b). The checking results of TEMPSY-CHECK are shown in Figure A.3. If a *TemPsy* expression is violated by the trace, we provide a hyperlink with the text "See details" to invoke our violation reporting procedure.



Results	Detail
demo_expiration1 : true	
demo_expiration3 : false	See details

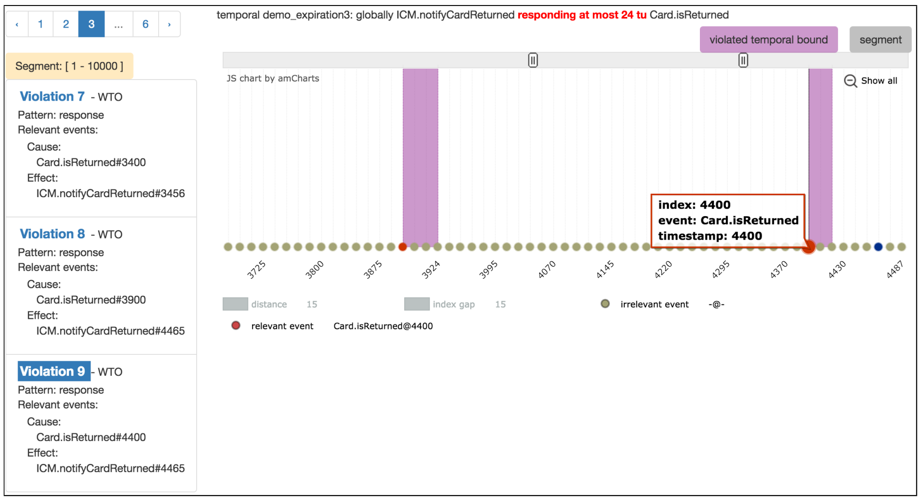
Figure A.3: Checking results of TEMPSY-CHECK

[1]`http://weidou.github.io/TemPsy-Check`



```
Console ☒
<terminated> OfflineCheck [Java Application] /Library/Java
--------------------
effect:
index   event   timestamp
8450,   ICM.notifyCardReturned, 8450
--------------------
violation type - Wrong Temporal Order
cause:
index   event   timestamp
9400,   Card.isReturned,        9400
--------------------
effect:
index   event   timestamp
9456,   ICM.notifyCardReturned, 9456
--------------------
violation type - No-Show Order
cause:
index   event   timestamp
9900,   Card.isReturned,        9900
--------------------
--------------------
there are 16 violations.
execution time : 137ms
====================
```

(a) Excerpt of the textual output



temporal demo_expiration3: globally ICM.notifyCardReturned **responding at most 24 tu** Card.isReturned

Segment: [1 - 10000]

Violation 7 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#3400
Effect:
ICM.notifyCardReturned#3456

Violation 8 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#3900
Effect:
ICM.notifyCardReturned#4465

Violation 9 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#4400
Effect:
ICM.notifyCardReturned#4465

index: 4400
event: Card.isReturned
timestamp: 4400

(b) Excerpt of the graphical output

Figure A.4: Presentation of violation information

Violation reporting procedure. As presented in Chapter 4, our violation reporting procedure consists of two steps: 1) collecting information about the violations and 2) presenting this information (textually and graphically) to the user. For instance, as shown in the checking results (Figure A.3), the third *TemPsy* expression is violated. Our TEMPSY-REPORT tool collects the violation information from the input trace and displays the information textually (as shown in Figure A.4a). The tool stores this information into MongoDB and calls our TEMPSY-VIOLATIONVISUALIZATION tool for visualization. As shown in Figure A.4b, the violation information can be navigated using our visualization tool.