Dissertation Defense Committee

Prof. Dr.-Ing. Lionel Briand, Dissertation Supervisor  
*University of Luxembourg, Luxembourg*

Dr. Radu State, Chairman  
*University of Luxembourg, Luxembourg*

Dr. Mehrdad Sabetzadeh, Vice Chairman  
*University of Luxembourg, Luxembourg*

Prof. Dr.-Ing. Cesare Pautasso  
*Università della Svizzera italiana, Switzerland*

Prof Dr.-Ing. Carlo Alberto Furia  
*Chalmers University of Technology, Sweden*

Dr. Domenico Bianculli, Expert in an advisory capacity (Co-supervisor)  
*University of Luxembourg, Luxembourg*
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.
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

## Contents

List of Figures ix

List of Tables xi

Acronyms xiii

1 Introduction 1

1.1 Motivation ........................................ 1

1.2 Research Contribution .......................... 3

1.3 Dissemination ................................. 6

1.4 Organization of the Dissertation ............... 6

2 The TemPsy Language 9

2.1 Background: Property Specification Patterns .......... 9

2.2 Definition of TemPsy .............................. 11

2.2.1 Eliciting the requirements of the language ....... 11

2.2.2 Design ........................................ 11

2.2.3 Syntax ........................................ 12

2.2.4 TemPsy at Work .............................. 14

2.3 Informal Semantics ............................... 15

2.3.1 Scopes ........................................ 15

2.3.2 Patterns ..................................... 16

2.4 Formal Semantics ................................ 18

2.4.1 Events and Trace .............................. 18

2.4.2 Temporal expressions ......................... 19

2.5 Expressivity ...................................... 23

2.6 Applying TemPsy ................................. 23

2.6.1 Business process models ....................... 24

2.6.2 Requirement specifications ..................... 25

2.6.3 Adoption of TemPsy by our partner .............. 27

3 Model-driven Offline Trace Checking of Temporal Properties 31

3.1 Conceptual Model for Execution Traces .............. 31

3.2 Overview of the Approach .......................... 32

3.3 OCL Functions for Scopes .......................... 33
3.3.1 Before ................................................. 34
3.3.2 After .................................................. 35
3.3.3 Between-and .............................. 36
3.3.4 After-until ........................................... 39
3.4 OCL Functions for Patterns ............. 41
  3.4.1 Universality ........................................ 41
  3.4.2 Existence ............................................ 42
  3.4.3 Absence ............................................... 42
  3.4.4 Precedence .......................................... 43
  3.4.5 Response ............................................. 48
3.5 The Approach at Work .................. 53
3.6 Tool Implementation ..................... 54
3.7 Evaluation ............................................. 54
  3.7.1 Experiment settings ......................... 55
  3.7.2 Properties using the globally scope ... 57
    3.7.2.1 Research Questions ....................... 57
    3.7.2.2 Trace generation strategy ............... 57
    3.7.2.3 Evaluation ...................................... 58
  3.7.3 Properties using the before/after scope 60
    3.7.3.1 Research Questions ....................... 61
    3.7.3.2 Trace generation strategy ............... 61
    3.7.3.3 Evaluation ...................................... 61
  3.7.4 Properties using the between-and scope 62
    3.7.4.1 Research Questions ....................... 63
    3.7.4.2 Trace generation strategy ............... 64
    3.7.4.3 Evaluation ...................................... 64
  3.7.5 Discussion .......................................... 65
    3.7.5.1 Threats to validity ....................... 66
4  Model-driven Violation Reporting for Trace Checking 69
  4.1 Characterization of TemPsy Violations ........................ 69
  4.2 Overview of the Approach ................ 71
  4.3 OCL Functions for collecting violation information .................. 73
    4.3.1 Universality ...................................... 73
    4.3.2 Existence .......................................... 74
    4.3.3 Absence ............................................... 74
    4.3.4 Precedence .......................................... 75
    4.3.5 Response ............................................. 84
  4.4 Tool Implementation ..................... 91
  4.5 Visualizing Violation Information ............. 92
    4.5.1 Requirements ..................................... 92
    4.5.2 Functionality ..................................... 92
    4.5.3 Implementation .................................... 93
  4.6 Evaluation ............................................. 94
    4.6.1 Experiment settings ............................. 95
    4.6.2 Trace Generation Strategies ................. 96
4.6.3 Evaluation Results ..................................................... 97
4.6.4 Discussion ................................................................. 99
4.6.4.1 Threats to validity .................................................. 100

5 Related Work ................................................................. 101
  5.1 MDE Approaches for Specifying Temporal Properties .......... 101
     5.1.1 Pattern-based temporal extensions of OCL ................. 101
     5.1.2 Other temporal extensions of OCL ....................... 102
  5.2 Trace Checking and Run-time Verification ....................... 103
  5.3 Violation Reporting for Trace Checking ....................... 104

6 Conclusions and Future Work ........................................ 107
  6.1 Conclusions ............................................................. 107
  6.2 Future Work ............................................................ 108

Bibliography ................................................................. 109

A Tool Support ............................................................... 115
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Scopes in the catalogue of property specification patterns in [Dwyer et al., 1999]</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Syntax of <em>TemPsy</em></td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>An event trace on which to evaluate the properties described in Section 2.2.4; events are above the line, timestamps below</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>A sample trace for the description of scopes</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Examples of <em>TemPsy</em> scopes</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Example trace for illustrating the precedence and response patterns</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>Conceptual model of the ICM business process</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>Activity diagrams of three fragments of the ICM business process</td>
<td>24</td>
</tr>
<tr>
<td>2.9</td>
<td>Example of the graphical notation for <em>TemPsy</em></td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Conceptual model for execution traces</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of the approach to offline trace checking of <em>TemPsy</em> properties</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>OCL invariant for checking <em>TemPsy</em> properties on a trace</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>An event trace on which to evaluate the property described in Section 3.5; events are above the line, timestamps below</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison between the execution time of TEMPSY-CHECK and of MONPOLY for properties with the <em>globally</em> scope</td>
<td>59</td>
</tr>
<tr>
<td>3.6</td>
<td>Trace loading time of TEMPSY-CHECK for traces with various lengths</td>
<td>60</td>
</tr>
<tr>
<td>3.7</td>
<td>Comparison of the execution time for the batch checking of ten properties with the <em>globally</em> scope</td>
<td>60</td>
</tr>
<tr>
<td>3.8</td>
<td>Scope time and pattern time of TEMPSY-CHECK for checking properties with a <em>before</em> scope</td>
<td>62</td>
</tr>
<tr>
<td>3.9</td>
<td>Scope time and pattern time of TEMPSY-CHECK for checking properties with an <em>after</em> scope</td>
<td>63</td>
</tr>
<tr>
<td>3.10</td>
<td>Scope time and pattern time of TEMPSY-CHECK for checking properties with a <em>between-and</em> scope (multiple segments, fixed length)</td>
<td>64</td>
</tr>
<tr>
<td>3.11</td>
<td>Scope time and pattern time of TEMPSY-CHECK for checking properties with a <em>between-and</em> scope (fixed number of segments, various lengths)</td>
<td>64</td>
</tr>
<tr>
<td>3.12</td>
<td>Scope time and pattern time of TEMPSY-CHECK for checking properties with a <em>between-and</em> scope (single segment of fixed length, different positions)</td>
<td>65</td>
</tr>
<tr>
<td>3.13</td>
<td>Scope time and pattern time of TEMPSY-CHECK for checking properties with a <em>between-and</em> scope (single segment, various lengths)</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of the approach to violation information collection</td>
<td>72</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.2</td>
<td>The template for OCL queries on a trace for collecting violation information with respect to TemPsy expressions</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>Screenshot of our visualization tool for understanding TemPsy violations</td>
<td>94</td>
</tr>
<tr>
<td>4.4</td>
<td>Execution time of TEMPSY-REPORT for collecting violation information from faulty traces (fixed number of violations, various lengths)</td>
<td>98</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution time of TEMPSY-REPORT for collecting violation information from faulty traces (various numbers of violations, fixed length)</td>
<td>98</td>
</tr>
<tr>
<td>A.1</td>
<td>TemPsy editor</td>
<td>115</td>
</tr>
<tr>
<td>A.2</td>
<td>Checking TemPsy expressions on an execution trace</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>(a) Selecting TemPsy expressions</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>(b) Loading an execution trace</td>
<td>116</td>
</tr>
<tr>
<td>A.3</td>
<td>Checking results of TEMPSY-CHECK</td>
<td>116</td>
</tr>
<tr>
<td>A.4</td>
<td>Presentation of violation information</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>(a) Excerpt of the textual output</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>(b) Excerpt of the graphical output</td>
<td>117</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Distribution of requirements from the ICM business process in terms of the combination of scopes and patterns ................................................................. 28
3.1 TemPsy properties used for the evaluation ..................................................... 56
4.1 Violation types of the existence pattern ......................................................... 74
4.2 Mapping between the requirements and the functionality of the target visualization tool ................................................................. 93
4.3 TemPsy properties used for the evaluation ..................................................... 95
5.1 Comparison between pattern-based temporal extensions of OCL and TemPsy ...... 101
5.2 The state of the art of violation reporting for offline trace checking of temporal properties ................................................................. 104
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.
Chapter 1. Introduction

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.
//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


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


  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


  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.
Chapter 2. The TemPsy Language

**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.
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…” . 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…”.

- 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
2.2. Definition of TemPsy

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

Figure 2.2: Syntax of TemPsy

boundaries (expressed with \langle \text{Boundary1} \rangle or \langle \text{Boundary2} \rangle) that denote a specific occurrence of an event as a boundary, possibly with a time distance; notice that \langle \text{Boundary2} \rangle represents a syntactic restriction of \langle \text{Boundary1} \rangle. Chains of events, used in precedence and response patterns, are defined as \langle \text{EventChainExp} \rangle, which denotes a comma-separated list of events, possibly with a time distance (\langle \text{TimeDistanceExp} \rangle) 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.
Chapter 2. The TemPsy Language

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 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.
2.3. Informal Semantics

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⟩; ‘$m$’, ‘$m_1$’, ‘$m_2$’, ‘$n$’, ‘$n_1$’, and ‘$n_2$’ 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 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 [$m_1$] $X$ [at least $n_1$ tu] and [$m_2$] $Y$ [at least $n_2$ 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 $m_1=1$ and $m_2=1$, 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, 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.

---

4The scopes that contain constraints on time distance from the boundary events (with “at least” and “exactly”) are closed on the bounds.
### 2.3. Informal Semantics

**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;
Chapter 2. The TemPsy Language

• “eventually exactly mA” 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 (e0, . . . , en−1), where e0 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 $\tau : \mathbb{N} \to \mathbb{N}$, 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 $\lambda$, 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 $\lambda$,

- $\lambda(i)$ 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 $\lambda$ 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 $\lambda$.

### 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 $\bot$ 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 $\langle\text{Scope}\rangle$ 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 $\phi_s : \Lambda \to 2^\Lambda$ as follows:

**globally:** $\phi_{[\text{globally}]}(\lambda) = \{ \lambda \}$

**before:**

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

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

**after:**
Chapter 2. The TemPsy Language

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

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

between-and:

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

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:
2.4. Formal Semantics

- $\phi_{\text{after } e_1 \text{ until } e_2}(\lambda) = \phi_{\text{between } e_1 \text{ and } e_2}(\lambda) \cup \{\lambda(\eta + 1 : n - 1) | \eta = \min(\{\gamma | \gamma \leq n - 2, \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n - 1, \lambda(k) \neq e_2\})\}$

- $\phi_{\text{after } e_1 \text{ until } e_2 \text{ at least } d \text{ tu}}(\lambda) = \phi_{\text{between } e_1 \text{ and } e_2 \text{ at least } d \text{ tu}}(\lambda) \cup \{\lambda(\eta + 1 : n - 1) | \eta = \min(\{\gamma | \gamma \leq n - 2, \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n - 1, \lambda(k) \neq e_2\})\}$

- $\phi_{\text{after } e_1 \text{ at least } b \text{ tu until } e_2}(\lambda) = \phi_{\text{between } e_1 \text{ at least } b \text{ tu and } e_2 \text{ at least } d \text{ tu}}(\lambda) \cup \{\lambda(\eta' : n - 1) | \eta' = \min(\{\gamma | \text{td}(\eta, \gamma) \geq b\}), \eta = \min(\{\gamma | \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n - 1, \lambda(k) \neq e_2\})\}$

- $\phi_{\text{after } e_1 \text{ at least } b \text{ tu until } e_2 \text{ at least } d \text{ tu}}(\lambda) = \phi_{\text{between } e_1 \text{ at least } b \text{ tu and } e_2 \text{ at least } d \text{ tu}}(\lambda) \cup \{\lambda(\eta' : n - 1) | \eta' = \min(\{\gamma | \text{td}(\eta, \gamma) \geq b\}), \lambda(\eta) = e_1, \#(\lambda, 0, \eta, e_1) = x, \#(\lambda, \eta + 1, n - 1, e_2) < y\}$

- $\phi_{\text{after } e_1 \text{ at least } b \text{ tu until } e_2 \text{ at least } d \text{ tu}}(\lambda) = \phi_{\text{between } e_1 \text{ at least } b \text{ tu and } e_2 \text{ at least } d \text{ tu}}(\lambda) \cup \{\lambda(\eta' : n - 1) | \eta' = \min(\{\gamma | \text{td}(\eta, \gamma) \geq b\}), \lambda(\eta) = e_1, \#(\lambda, 0, \eta, e_1) = x, \#(\lambda, \eta + 1, n - 1, e_2) < y\}$

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

**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 $e_0, t_1, e_1, \ldots, t_m, e_m \ldots$. 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 $t_i = \# \bowtie \delta_i \text{ tu}$ with $\delta_i \in \mathbb{N}^+$ and $\bowtie \in \{\text{at least, at most, exactly}\}$; when $t_i$ is undefined we use the notation $t_i = \perp$. Function $\text{len}(EC)$ returns the length $m$ of an $m$-length EventChain $EC$.

**Event and EventChain matching function.** Let $\lambda$ be an $n$-length trace, $EC$ be an $m$-length EventChain ($1 \leq m \leq n$). The matching function $\text{match}$ 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 $\text{match}(\lambda, EC, i) = \text{true}$, with $i, 0 \leq i \leq n - 1$, if $\lambda(i) = e$. For an event chain $EC = e_0, t_1, e_1, \ldots, t_m, e_m \ldots$, we have $\text{match}(\lambda, EC, i) = \text{true}$, 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,
Chapter 2. The TemPsy Language

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

\[
\begin{align*}
\text{td}(t_{j-1}, t_j) &\geq \delta_j \text{ if } \triangleright= \text{ at least;} \\
\text{td}(t_{j-1}, t_j) &\leq \delta_j \text{ if } \triangleright= \text{ at most;} \quad \text{and} \\
\text{td}(t_{j-1}, t_j) &\neq \delta_j \text{ if } \triangleright= \text{ exactly.}
\end{align*}
\]

For an events chain \( EC = e_0, t_1, e_1, \ldots, t_{m-1}, e_{m-1} \) we also define two auxiliary functions \( \text{first}(\lambda, EC, i) \) and \( \text{last}(\lambda, EC, i) \), 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 \( \lambda \).

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

**Pattern.** Let \( p \) be a pattern defined by the non-terminal \( \langle \text{Pattern} \rangle \) 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 \( \psi_p : \Lambda \rightarrow \{\text{true,false}\} \) as follows:

**universality:** \( \psi_{[\text{always} \ e]}(\lambda) \iff \forall i, 0 \leq i \leq n - 1, \lambda(i) = e \)

**absence:**
- \( \psi_{[\text{never} \ e]}(\lambda) \iff \forall i, 0 \leq i \leq n - 1, \lambda(i) \neq e \)
- \( \psi_{[\text{never exactly} \ m \ e]}(\lambda) \iff \#(\lambda, 0, n - 1, e) \neq m \)

**existence:**
- \( \psi_{[\text{eventually} \ e]}(\lambda) \iff \exists i, 0 \leq i \leq n - 1, \lambda(i) = e \)
- \( \psi_{[\text{eventually} \triangleright= \ m \ e]}(\lambda) \iff \#(\lambda, 0, n - 1, e) \triangleright= m \)

where \( \triangle = \begin{cases} 
\geq, \text{ if } \triangleright= \text{ at least;} \\
\leq, \text{ if } \triangleright= \text{ at most;} \\
\neq, \text{ if } \triangleright= \text{ exactly.}
\end{cases} \)

**precedence:**
- \( \psi_{[EC_1 \text{ preceding} \ EC_2]}(\lambda) \iff \forall i, 0 \leq i \leq n - 1, \text{match}(\lambda, EC_2, i) \Rightarrow \\
\exists j, 0 \leq j \leq i - len(EC_1), \text{match}(\lambda, EC_1, j) \)
- \( \psi_{[EC_1 \text{ preceding} \triangleright= \, b \, \text{ tu} \, EC_2]}(\lambda) \iff \forall i, 0 \leq i \leq n - 1, \text{match}(\lambda, EC_2, i) \Rightarrow \\
\exists j, 0 \leq j \leq i - len(EC_1), \text{match}(\lambda, EC_1, j) \text{ and } (\text{first}(\lambda, EC_2, i) - \text{last}(\lambda, EC_1, j)) \triangleright= b \\
\geq, \text{ if } \triangleright= \text{ at least;} \\
\leq, \text{ if } \triangleright= \text{ at most;} \\
\neq, \text{ if } \triangleright= \text{ exactly.}
\)

**response:**
- \( \psi_{[EC_1 \text{ responding} \ EC_2]}(\lambda) \iff \forall i, 0 \leq i \leq n - 1, \text{match}(\lambda, EC_2, i) \Rightarrow \exists j, i + len(EC_2) \leq j \leq n - 1, \text{match}(\lambda, EC_1, j) \)
- \( \psi_{[EC_1 \text{ responding} \triangleright= \, b \, \text{ tu} \, EC_2]}(\lambda) \iff \forall i, 0 \leq i \leq n - 1, \text{match}(\lambda, EC_2, i) \Rightarrow \exists j, i + len(EC_2) \leq j \leq n - 1, \text{match}(\lambda, EC_1, j) \) and

22
2.5 Expressivity

\[(\text{first}(\lambda, EC_1, i) - \text{last}(\lambda, EC_2, j)) \triangle b\]

where \(\triangle = \begin{cases} 
\geq, & \text{if } \trianglerighteq \text{ at least;} \\
\leq, & \text{if } \trianglerighteq \text{ at most;} \\
=, & \text{if } \trianglerighteq \text{ exactly.}
\end{cases}\)

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

### 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 \(\langle\text{scope, pattern}\rangle\) 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.
Chapter 2. The TemPsy Language

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 Cards 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
Chapter 2. The TemPsy Language

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:

```plaintext
1 temporal L1:
2   after Card.isLost
3   at most 24*3600 tu
4   ICM.recallTempCard
5   responding at most 10*24*3600 tu
6   ICM.issueTempCard
7 temporal L2:
8   after Card.isLost
9   ICM.produceCard
10  preceding at least 2*24*3600 tu
11  ICM.issueCard
12 temporal L3:
13   after Card.isLost
14   until ICM.issueCard
15   ICM.recallCard,
16   ICM.recallTempCard,
17   responding at most 3*24*3600 tu
18   ICM.produceCard,
19   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:
2.6. Applying TemPsy

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:

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; 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.

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

<table>
<thead>
<tr>
<th>scope+pattern</th>
<th># of requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>globally+universality</td>
<td>1</td>
</tr>
<tr>
<td>globally+absence</td>
<td>1</td>
</tr>
<tr>
<td>globally+existence</td>
<td>2</td>
</tr>
<tr>
<td>globally+precedence</td>
<td>4</td>
</tr>
<tr>
<td>globally+response</td>
<td>4</td>
</tr>
<tr>
<td>before+absence</td>
<td>1</td>
</tr>
<tr>
<td>before+existence</td>
<td>2</td>
</tr>
<tr>
<td>before+precedence</td>
<td>3</td>
</tr>
<tr>
<td>before+response</td>
<td>2</td>
</tr>
<tr>
<td>after+universality</td>
<td>1</td>
</tr>
<tr>
<td>after+absence</td>
<td>1</td>
</tr>
<tr>
<td>after+existence</td>
<td>4</td>
</tr>
<tr>
<td>after+precedence</td>
<td>3</td>
</tr>
<tr>
<td>after+response</td>
<td>2</td>
</tr>
<tr>
<td>between-and+universality</td>
<td>2</td>
</tr>
<tr>
<td>between-and+absence</td>
<td>2</td>
</tr>
<tr>
<td>between-and+existence</td>
<td>1</td>
</tr>
<tr>
<td>between-and+precedence</td>
<td>1</td>
</tr>
<tr>
<td>between-and+response</td>
<td>1</td>
</tr>
<tr>
<td>after-until+universality</td>
<td>1</td>
</tr>
<tr>
<td>after-until+absence</td>
<td>1</td>
</tr>
<tr>
<td>after-until+existence</td>
<td>4</td>
</tr>
<tr>
<td>after-until+precedence</td>
<td>1</td>
</tr>
<tr>
<td>after-until+response</td>
<td>2</td>
</tr>
</tbody>
</table>
2.6. Applying TemPsy

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 $\rho$ over a trace $\lambda$, to the problem of evaluating an OCL constraint (semantically equivalent to $\rho$) on an instance of a conceptual model for execution traces (equivalent to $\lambda$).

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
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

Figure 3.1: Conceptual model for execution traces

traceElements

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.
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.
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

```plaintext
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::UNIVERSITY 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.
3.3. OCL Functions for Scopes

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 ← event name of the scope boundary
2. m ← index of the specific occurrence of event X
3. op ← comparison operator of the constraint on time distance
4. n ← time distance from the m-th occurrence of X
5. result ← [], segment ← []
6. if m = null then m ← 1
7. t ← timestamp of the m-th occurrence of event X
8. if t ≠ null then
   9. switch op do
      10. case “at least” do
          11. segment ← trace elements with timestamp t' satisfying t' ≤ t − n
      12. case “at most” do
          13. segment ← trace elements with timestamp t' satisfying t − n ≤ t' < t
      14. case “exactly” do
          15. segment ← trace elements with timestamp equal to t − n
      16. otherwise do
          17. segment ← 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 $m = \text{null}$ then $m \leftarrow 1$
7 $t \leftarrow$ timestamp of the $m$-th occurrence of event $X$
8 if $t \neq \text{null}$ 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
3.3. OCL Functions for Scopes

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

1. X ← event name of the left boundary
2. Y ← event name of the right boundary
3. m1 ← index of the specific occurrence of event X
4. m2 ← index of the specific occurrence of event Y
5. n1 ← lower bound of the time distance from the m1-th occurrence of event X
6. n2 ← lower bound of the time distance from the m2-th occurrence of event Y
7. result ← []
8. if m1 = null && m2 = null then
   9.      result ← applyOriginalBetweenAnd(X, n1, Y, n2)
10. else
11.    if m1 = null then m1 ← 1
12.    if m2 = null then m2 ← 1
13.    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 (i1, t1) to keep track of the starting point of a trace segment. More precisely, element i1 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 t1 points to the instant that is n1 time units after the occurrence of the left bound of the segment. The tuple (i2, t2) is defined in a similar way, to keep track of the end point of a trace segment (characterized by an occurrence of event Y).
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

Algorithm 4: applyOriginalBetweenAnd

**Input:** strings \(X, Y\) and integers \(n_1, n_2\) (\(n_1 = 0, n_2 = 0\) by default), i.e., the parameters of a between-and scope structured as “between \(X\) [at least \(n_1\) tu] and \(Y\) [at least \(n_2\) tu]”

**Output:** \(\text{result}\) : a list of trace segments, as determined by the parameters of the scope

1. \(\text{result} \leftarrow \emptyset\)
2. \(\text{index} \leftarrow 0\)
3. \((i_1, t_1) \leftarrow (0, 0)\)
4. \((i_2, t_2) \leftarrow (0, 0)\)
5. for \(\text{elem} \in \text{self}.\text{traceElements}\) do
6. \(\text{index} \leftarrow \text{index} + 1\)
7. \(e \leftarrow \text{elem}.\text{event}\)
8. \(t \leftarrow \text{elem}.\text{timestamp}\)
9. if \(i_1 = 0\) then
10. if \(e = X\) then
11. \((i_1, t_1) \leftarrow (\text{index} + 1, t + n_1)\)
12. else if \(e = Y \& \& \text{index} > i_1\) then
13. \((i_2, t_2) \leftarrow (\text{index} - 1, t - n_2)\)
14. \(\text{segment} \leftarrow \text{self}.\text{traceElements}[i_1..i_2]\)
15. if \(n_1 \neq 0 \| n_2 \neq 0\) then
16. \(\text{result} \leftarrow \text{trace elements in segment with timestamps } t' \text{ satisfying } t_1 \leq t' \leq t_2\)
17. \(\text{result}.\text{append}(\text{segment})\)
18. \((i_1, t_1) \leftarrow (0, 0)\)
19. \((i_2, t_2) \leftarrow (0, 0)\)
20. return \(\text{result}\)

At each iteration of the loop (lines 5–19), for each element of the trace, the function first increments the variable \(\text{index}\) 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 \(n_1\) 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 \(n_2\) 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 \(\text{result}\) 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 \(m_1\) and \(m_2\), 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 \(m_1\)-th occurrence of event \(X\) and the \(m_2\)-th oc-
3.3. OCL Functions for Scopes

Algorithm 5: applySpecialBetweenAnd

Input: strings \( X, Y \) and integers \( m_1, n_1, m_2, n_2 \) (\( n_1 = 0, n_2 = 0 \) by default), i.e., the parameters of a \( \text{between-and} \) scope structured as “\( \text{between } m_1 \ X \ \text{[at least } n_1 \ \text{tu}] \) and \( m_2 \ Y \ \text{[at least } n_2 \ \text{tu}] \)”

Output: result : a trace segment, as determined by the parameters of the scope

1. \( \text{result} \leftarrow [] \)
2. \( \text{index} \leftarrow 0 \)
3. \( (i_1, t_1) \leftarrow (0, 0) \)
4. \( (i_2, t_2) \leftarrow (0, 0) \)
5. \( \text{count} \leftarrow 0 \)
6. for elem \( \in \text{self.traceElements} \) do
7. \( \text{index} \leftarrow \text{index} + 1 \)
8. \( e \leftarrow \text{elem.event} \)
9. \( t \leftarrow \text{elem.timestamp} \)
10. if \( i_1 = 0 \land e = X \) then
11. \( \text{count} \leftarrow \text{count} + 1 \)
12. \( \text{if count} = m_1 \) then
13. \( (i_1, t_1) \leftarrow (\text{index} + 1, t + n_1) \)
14. \( \text{count} \leftarrow 0 \)
15. \( \) else if \( i_2 = 0 \land e = Y \) then
16. \( \text{count} \leftarrow \text{count} + 1 \)
17. \( \text{if count} = m_2 \) then
18. \( (i_2, t_2) \leftarrow (\text{index} - 1, t - n_2) \)
19. \( \) break
20. \( \) if \( i_1 > 0 \land i_1 \leq i_2 \) then
21. \( \text{result} \leftarrow \text{self.traceElements}[i_1 \ldots i_2] \)
22. \( \) if \( n_1 \neq 0 \lor n_2 \neq 0 \) then
23. \( \) \( \text{result} \leftarrow \text{trace elements with timestamps } t' \text{ satisfying } t_1 \leq t' \leq t_2 \)
24. \( \) 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 [\( m_1 \) \( X \) [at least \( n_1 \) tu] until [\( m_2 \) \( Y \) [at least \( n_2 \) tu]]].” Likewise the function defined for applying the between-and scope (Algorithm 3), function applyScopeAfterUntil invokes the auxiliary function applyOriginalAfterUntil (Algorithm 6) if both \( m_1 \) and \( m_2 \) are
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

Algorithm 6: applyOriginalAfterUntil (procedure to complement Algorithm 4)

Input: strings \(X, Y\) and integers \(n_1, n_2\) (\(n_1 = 0, n_2 = 0\) by default), i.e., the parameters of a after-until scope structured as “after \(X\) [at least \(n_1\) tu] until \(Y\) [at least \(n_2\) tu]”

Output: result: a list of trace segments, as determined by the parameters of the scope

1. \(size \leftarrow \text{self.traceElements.size}()\)
2. if \(i_1 > 0 \land i_1 \leq size\) then
3.   \(\text{segment} \leftarrow \text{self.traceElements}[i_1..size]\)
4.   if \(n_1 \neq 0\) then
5.     \(\text{segment} \leftarrow \text{trace elements in segment with timestamps } t'\text{ satisfying } t' \geq t_1\)
6.   result.append(segment)

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

Input: strings \(X, Y\) and integers \(m_1, n_1, m_2, n_2\) (\(n_1 = 1, n_2 = 0\) by default), i.e., the parameters of a after-until scope structured as “after \(m_1\) \(X\) [at least \(n_1\) tu] until \(m_2\) \(Y\) [at least \(n_2\) tu]”

Output: result: a trace segment, as determined by the parameters of the scope

1. \(size \leftarrow \text{self.traceElements.size}()\)
2. if \(i_1 > 0\) then
3.   if \(i_2 = 0 \land i_1 \leq size\) then
4.     \(\text{result} \leftarrow \text{self.traceElements}[i_1..size]\)
5.     if \(n_1 \neq 0\) then
6.       \(\text{result} \leftarrow \text{trace elements with timestamps } t'\text{ satisfying } t' \geq t_1\)
7.   else if \(i_1 \leq i_2\) then
8.     \(\text{result} \leftarrow \text{self.traceElements}[i_1..i_2]\)
9.     if \(n_1 \neq 0 \lor n_2 \neq 0\) then
10.    \(\text{result} \leftarrow \text{trace elements with timestamps } t'\text{ satisfying } t_1 \leq t' \leq t_2\)

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 \text{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 \text{applySpecialAfterUntil} (Algorithm 7) describes the new procedure that selects the trace segment after investigating the $m_1$-th occurrence of event $X$ and the $m_2$-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 \text{applySpecialBetweenAnd}.

In the algorithm, if the $m_1$-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 $m_1$-th occurrence of event $X$ (lines 3–6); or the other for selecting the segment between the $m_1$-th occurrence of event $X$ and the $m_2$-th occurrence of event $Y$ (lines 7–10). The former is executed when the $m_2$-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 $m_2$-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 \text{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 \text{checkPatternUniversality}, \text{checkPatternExistence}, \text{checkPatternAbsence}, \text{checkPatternPrecedence}, and \text{checkPatternResponse} for the five patterns. These functions take as input a sub-trace and an object representing a pattern in \text{TemPsy}, and return whether the pattern holds on the input sub-trace.

#### 3.4.1 Universality

Algorithm 8: \text{checkPatternUniversality}

\begin{verbatim}
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 $E$ ← event name in pattern
2 for elem ∈ subtrace do
3   if elem.event ≠ $E$ then return false
4 return true
\end{verbatim}

Function \text{checkPatternUniversality} (see Algorithm 8) takes in input a trace segment (denoted by the variable \text{subtrace}) and an instance of the \text{universality} pattern (denoted by the variable \text{pattern}).
3.4.2 Existence

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).
3.4. OCL Functions for Patterns

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 ← event name in pattern
2. op ← comparison operator of the bound on the number of occurrences of event E
3. n ← threshold of the occurrence number of event E

if op = null then
  for elem ∈ subtrace do
    if elem.event = E then return false
  return true
else
  count ← the number of occurrences of event E in subtrace
  return count ≠ 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) block1 and block2 of an instance of precedence pattern of the form “block1 preceding block2”

Output: true if pattern holds on subtrace; false otherwise

1. size1, size2 ← the sizes of block1 and block2
2. firstOfBlock1 ← block1.first().event
3. firstOfBlock2 ← block2.first().event
4. \((i_1, pt_1) \leftarrow (1, 0), (i_2, pt_2) \leftarrow (1, 0)\)

for elem ∈ subtrace do
  e ← elem.event
  t ← elem.timestamp
  if e = firstOfBlock1 then \((i_1, pt_1) \leftarrow (2, t)\)
  else if \(i_1 > 1\) then \((i_1, pt_1) \leftarrow \text{match}(block_1, i_1, pt_1, e, t)\)
  if \(i_1 = \text{size}_1 + 1\) then return true
  if e = firstOfBlock2 then \((i_2, pt_2) \leftarrow (2, t)\)
  else if \(i_2 > 1\) then \((i_2, pt_2) \leftarrow \text{match}(block_2, i_2, pt_2, e, t)\)
  if \(i_2 = \text{size}_2 + 1\) then return false

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. $op ← block[i].timeDistance.op$
   3. $t' ← pt + block[i].timeDistance.value$
   4. if $compare(t, op, t')$ then $(i, pt) ← (i + 1, t)$
5. else $(i, pt) ← (1, 0)$
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 `block1`, 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 `block2` (lines 11–13). If the occurrence of the first event of `block2` 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 “`block1` preceding at least $n$ tu `block2`”; two events (chains) `block1` and `block2`, and a threshold $n$ of the time distance between `block1` and `block2`

**Output:** true if pattern holds on `subtrace`; false otherwise

1. $size_1, size_2 \leftarrow$ the sizes of `block1` and `block2`
2. `firstOfBlock1` $\leftarrow$ `block1.first().event`
3. `firstOfBlock2` $\leftarrow$ `block2.first().event`
4. $(i_1, pt_1) \leftarrow (1, 0)$, $(i_2, pt_2) \leftarrow (1, 0)$
5. `flag1` $\leftarrow$ `true`
6. **for elem $\in$ subtrace do**
7.   $e \leftarrow$ `elem.event`
8.   $t \leftarrow$ `elem.timestamp`
9.   **if `flag1` 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(block1, i_1, pt_1, e, t)`
12.  **if** $i_1 =$ `size1` + 1 **then** `flag1` $\leftarrow$ `false`
13.  **if** $e =$ `firstOfBlock2` **then**
14.    **if** `flag1` $\parallel t < pt_1 + n$ **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(block2, i_2, pt_2, e, t)`
17.  **if** $i_2 =$ `size2` + 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: `block1`, `block2`, 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 `block1` 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 block_2[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, p_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, p_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 block_2[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.

^4Notice that in this case a violation is reported only if block_2 is fully matched (line 17).

^5This 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 “block₁ preceding at most n to block₂”: two events (chains) block₁ and block₂, and a maximum n of the time distance between block₁ and block₂

Output: true if pattern holds on subtrace; false otherwise

1. size₁, size₂ ← the sizes of block₁ and block₂
2. firstOfBlock1 ← block₁.first().event
3. firstOfBlock2 ← block₂.first().event
4. (i₁, pt₁) ← (1, 0), (i₂, pt₂) ← (1, 0)
5. criticalInstant ← 0
6. for elem ∈ subtrace do
7.   e ← elem.event
8.   t ← elem.timestamp
9.   if e = firstOfBlock1 then (i₁, pt₁) ← (2, t)
10.  else if i₁ > 1 then (i₁, pt₁) ← match(block₁, i₁, pt₁, e, t)
11.  if i₁ = size₁ + 1 then
12.     criticalInstant ← t + n
13.     (i₁, pt₁) ← (1, 0)
14.  if e = firstOfBlock2 && t > criticalInstant then (i₂, pt₂) ← (2, t)
15.  else if i₂ > 1 then (i₂, pt₂) ← match(block₂, i₂, pt₂, e, t)
16.  if i₂ = size₂ + 1 then return false
17. return true

(line 17) at the end of the function.

Function checkPatternPrecedenceExactly (Algorithm 15) checks whether each occurrence of block₂ is preceded by an occurrence of block₁, with an exact distance n. The key of this algorithm is to keep the timestamps of all occurrences of block₁ that may satisfy the distance constraint and fulfilling the distance constraint when the first event of block₂ 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 block₁ (lines 9–13) and/or an occurrence of block₂ (lines 14–18). After trying to match the current trace element with block₁ (lines 9–10), if block₁ 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 block₂ 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 (i₂, pt₂) is set to (2, t). Function match is called for matching the current trace element with block₂[i₂] if i₂ is greater than 1. A violation is reported when an occurrence of block₂ 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 "block₁ preceding exactly n to block₂": two events (chains) block₁ and block₂, and the exact time distance n between block₁ and block₂

Output: true if pattern holds on subtrace; false otherwise

1. size₁, size₂ ← the sizes of block₁ and block₂
2. firstOfBlock1 ← block₁.first().event
3. firstOfBlock2 ← block₂.first().event
4. (i₁, pt₁) ← (1, 0), (i₂, pt₂) ← (1, 0)
5. criticalInstants ← []
6. for elem ∈ subtrace do
7.   e ← elem.event
8.   t ← elem.timestamp
9.   if e = firstOfBlock1 then (i₁, pt₁) ← (2, t)
10.  else if i₁ > 1 then (i₁, pt₁) ← match(block₁, i₁, pt₁, e, t)
11.  if i₁ = size₁ + 1 then
12.      criticalInstants.append(t + n)
13.      (i₁, pt₁) ← (1, 0)
14.  if e = firstOfBlock2 then
15.      if t ∈ criticalInstants then criticalInstants ← {t′ | t′ ∈ criticalInstants && t′ > t}
16.      else (i₂, pt₂) ← (2, t)
17.  else if i₂ > 1 then (i₂, pt₂) ← match(block₂, i₂, pt₂, e, t)
18.  if i₂ = size₂ + 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: block₁, block₂, and checks whether all occurrences of block₂ are followed by an occurrence of block₁. Likewise function checkPatternPrecedenceplain, the algorithm uses a set of similar auxiliary variables: size₁ and size₂ 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 (i₁, pt₁) (respectively (i₂, pt₂)) is used to determine whether the trace element being checked is a match of the next event in block₁ (respectively, block₂). 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 block₂ (respectively block₁) is found.
Algorithm 16: checkPatternResponseGeneral

Input: a trace segment subtrace and the two events (chains) block1 and block2 of an instance of response pattern of the form “block1 responding block2”

Output: true if pattern holds on subtrace; false otherwise

1. size1, size2 ← the sizes of block1 and block2
2. firstOfBlock1 ← block1.first().event, firstOfBlock2 ← block2.first().event
3. (i1, pt1) ← (1, 0), (i2, pt2) ← (1, 0)
4. result ← true
5. for elem ∈ subtrace do
   6.     e ← elem.event
   7.     t ← elem.timestamp
   8.     if e = firstOfBlock2 then (i1, pt1) ← (2, t)
   9.     else if i2 > 1 then (i2, pt2) ← match(block2, i2, pt2, e, t)
  10.    if i2 = size2 + 1 then
  11.       (i2, pt2) ← (1, 0)
  12.       result ← false
  13.    if !result then
  14.       if e = firstOfBlock1 then (i1, pt1) ← (2, t)
  15.       else if i1 > 1 then (i1, pt1) ← match(block1, i1, pt1, e, t)
  16.       if i1 = size1 + 1 then
  17.         (i1, pt1) ← (1, 0)
  18.       result ← true
  19. return result

The major body of function checkPatternResponsePlain is a loop that iterates through the input subtrace to match each element with block2 (lines 8–12) and with block1 (lines 13–18). In the first portion of each iteration, if the current trace element matches the first event of block2, the variable i2 is set to 2 and pt2 is updated with the current timestamp (line 8); otherwise, if i2 is greater than 1, we invoke the auxiliary function match to match the trace element with the event defined at block2[i2], in consideration of a possible constraint defined on the distance from the previous event at block2[i2 – 1]) (line 9). If the matched event is the last event of block2 (line 10), the tuple (i2, pt2) is reset and variable result is set to false. That is, the result of the check will remain negative unless an occurrence of block1 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 block1 (lines 14–18). If the occurrence of the first event of block2 is detected, the variable i1 is set to 2 and pt1 is updated with the current timestamp (line 14). Otherwise if i2 is greater than 1, function match is called to check whether the current trace element is a match for block1[i1]. If block1 is fully matched (line 16), the tuple (i1, pt1) 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
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

Algorithm 17: checkPatternResponseAtLeast

**Input:** a trace segment subtrace and the parameters of an instance of response pattern of the form “block₁ responding at least n tu block₂”: two events (chains) block₁ and block₂, and a threshold n of the time distance between block₁ and block₂

**Output:** true if pattern holds on subtrace; false otherwise

```plaintext
1. size₁, size₂ ← the sizes of block₁ and block₂
2. firstOfBlock₁ ← block₁.first().event, firstOfBlock₂ ← block₂.first().event
3. (i₁, pt₁) ← (1, 0), (i₂, pt₂) ← (1, 0)
4. criticalInstant ← 0
5. result ← true
6. for elem ∈ subtrace do
7.     e ← elem.event
8.     t ← elem.timestamp
9.     if e = firstOfBlock₂ then (i₂, pt₂) ← (2, t)
10.    else if i₂ > 1 then (i₂, pt₂) ← match(block₂, i₂, pt₂, e, t)
11.    if i₂ = size₂ + 1 then
12.         (i₂, pt₂) ← (1, 0)
13.         criticalInstant ← t + n
14.         result ← false
15.    if !result then
16.        if e = firstOfBlock₁ && t >= criticalInstant then (i₁, pt₁) ← (2, t)
17.        else if i₁ > 1 then (i₁, pt₁) ← match(block₁, i₁, pt₁, e, t)
18.        if i₁ = size₁ + 1 then
19.            (i₁, pt₁) ← (1, 0)
20.            result ← true
21. return result
```

whether there is an occurrence of block₁ following all occurrences of block₂, 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: block₁, block₂, 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 block₁ is detected, while variable result is false. The strategy of this function is likewise function checkPatternResponseGeneral, except for two differences. In each iteration, when block₂ 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 block₁ is matched. More specifically, the process of matching block₁ 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 block₁ that fulfills the constraint on the distance from the closest occurrence of block₂. The algorithm returns the value of variable result (line 21), if there is no violation reported in the loop.

50
### 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

```plaintext
1. `size_1`, `size_2` ← the sizes of `block_1` and `block_2`
2. `firstOfBlock1` ← `block_1`.first().event, `firstOfBlock2` ← `block_2`.first().event
3. `(i_1, pt_1) ← (1, 0), (i_2, pt_2) ← (1, 0)
4. `criticalInstant` ← 0
5. `result` ← `true`
6. for `elem` \(\in\) `subtrace` do
7.     `e` ← `elem`.event
8.     `t` ← `elem`.timestamp
9.     if `result` then
10.        if `e` = `firstOfBlock2` then `(i_2, pt_2) ← (2, t)`
11.           else if `i_2` > 1 then `(i_2, pt_2) ←` match(`block_2`, `i_2`, `pt_2`, `e`, `t`)
12.              if `i_2` = `size_2` + 1 then
13.                 `(i_2, pt_2) ← (1, 0)
14.                 `criticalInstant` ← `t` + `n`
15.                 `result` ← `false`
16.     else
17.         if `e` = `firstOfBlock1` then
18.              if `t` <= `criticalInstant` then `(i_1, pt_1) ← (2, t)`
19.                 else return `false`
20.              else if `i_1` > 1 then `(i_1, pt_1) ←` match(`block_1`, `i_1`, `pt_1`, `e`, `t`)
21.                 if `i_1` = `size_1` + 1 then
22.                     `(i_1, pt_1) ← (1, 0)
23.                     `result` ← `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 \texttt{criticalInstant} (line 18); or else it returns \texttt{false} (line 19), to indicate that the distance between the marked occurrence of \texttt{block2} and the next possible occurrence of \texttt{block1} 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 \texttt{result} remains \texttt{false} until a valid \texttt{block1} is fully matched (line 23). If there is no violation reported in the loop, the function returns the value of variable \texttt{result} (line 24).

\begin{algorithm}
\caption{checkPatternResponseExactly}
\begin{algorithmic}
\Input{a trace segment \texttt{subtrace} and the parameters of an instance of \texttt{response} pattern of the form “\texttt{block1 responding exactly n tu block2}”: two events (chains) \texttt{block1} and \texttt{block2}, and the exact time distance \textit{n} between \texttt{block1} and \texttt{block2}}
\Output{\texttt{true} if pattern holds on \texttt{subtrace}; \texttt{false} otherwise}
\State \texttt{size1}, \texttt{size2} ← the sizes of \texttt{block1} and \texttt{block2}
\State \texttt{firstOfBlock1} ← \texttt{block1.first().event}, \texttt{firstOfBlock2} ← \texttt{block2.first().event}
\State \texttt{(i1,pt1)} ← (1,0), \texttt{(i2,pt2)} ← (1,0)
\State \texttt{criticalInstants} ← []
\For{\texttt{elem} ∈ \texttt{subtrace}}
\State \texttt{e} ← \texttt{elem.event}
\State \texttt{t} ← \texttt{elem.timestamp}
\If{\texttt{e} = \texttt{firstOfBlock2}} \texttt{(i2,pt2)} ← (2,\textit{t})
\ElseIf{\texttt{i2} > 1} \texttt{(i2,pt2)} ← \texttt{match(block2,i2,pt2,e,t)}
\EndIf
\If{\texttt{i2} = \texttt{size2} + 1} \texttt{criticalInstants.append(\textit{t} + n)}
\EndIf
\EndFor
\If{\texttt{criticalInstants notEmpty}()} \texttt{then}
\If{\texttt{e} = \texttt{firstOfBlock1}}
\If{\texttt{t} = \texttt{criticalInstants.first}()} \texttt{(i1,pt1)} ← (2,\textit{t})
\Else \textbf{return} \texttt{false}
\EndIf
\ElseIf{\texttt{i1} > 1} \texttt{(i1,pt1)} ← \texttt{match(block1,i1,pt1,e,t)}
\EndIf
\EndIf
\If{\texttt{i1} = \texttt{size1} + 1} \texttt{criticalInstants.remove(1)}
\EndIf
\EndIf
\State \texttt{(i1,pt1)} ← (1,0)
\EndFor
\State \textbf{return} \texttt{criticalInstants.isEmpty}()
\end{algorithmic}
\end{algorithm}

Function \texttt{checkPatternResponseExactly} (Algorithm 19) defines the algorithm that checks whether each occurrence of \texttt{block2} is followed by an occurrence of \texttt{block1}, with an exact time distance \textit{n}. We store the timestamps of all the occurrences of \texttt{block2} in variable \texttt{criticalInstants} and assess the fulfillment of the constraint “\texttt{exactly n tu}” whenever an occurrence of the first event of \texttt{block1} is detected.

In the function, we iterate through the input \texttt{subtrace} to check whether its trace element is part of an occurrence of \texttt{block2} (lines 8–12) and/or an occurrence of \texttt{block1} (lines 15–20). The code for matching a trace element with \texttt{block2} (lines 8–9) is identical to the corresponding part of the previous function. If the matched event is the last of \texttt{block2} (line 10), besides resetting the tuple \texttt{(i2,pt2)} to
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](image)

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 \langle TemPsyExpression \rangle 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. Evaluation

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

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

...
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 \textit{globally} scope

Properties defined with the \textit{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 \textit{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 \textit{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 \textit{globally} scope the generation of the trace is determined only by the semantics of the pattern used in the property.

For the \textit{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.
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

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 $\text{TEMPSY-CHECK}$ (denoted by $\circ$) and of $\text{MONPOLY}$ (denoted by $\ast$) 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 $\text{TEMPSY-CHECK}$ 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 $\text{MONPOLY}$ 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. $\text{MONPOLY}$ 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, $\text{TEMPSY-CHECK}$ performs much better than $\text{MONPOLY}$. A possible explanation for the slower time of $\text{MONPOLY}$ 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).
3.7. Evaluation

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.
Chapter 3. Model-driven Offline Trace Checking of Temporal Properties

![Figure 3.6: Trace loading time of TEMPsy-CHECK for traces with various lengths](image)

![Figure 3.7: Comparison of the execution time for the batch checking of ten properties with the globally scope](image)

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
3.7. Evaluation

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 over the range [0, position of the boundary] (for properties with a before scope), and over the range [position of the boundary, 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
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).
3.7. Evaluation

![Graphs showing time and position of the bound for different segments.](image)

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?
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.
3.7. Evaluation

In Figure 3.12, we observe the scope time and pattern time of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment of fixed length, different positions):

![Graphs showing scope and pattern time for different properties](a) P36  (b) P37  (c) P38

In Figure 3.13, we observe the scope time and pattern time of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment, various lengths):

![Graphs showing scope and pattern time for different properties](a) P36  (b) P37  (c) P38

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.

---

Chapter 4. Model-driven Violation Reporting for Trace Checking
4.2 Overview of the Approach

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.

![Diagram of the approach to violation information collection](image)

**Figure 4.1:** Overview of the approach to violation information collection

```plaintext
let property:TemPsy::TemPsyExpression = self.properties->at(P1),
subtraces:Sequence(Tuple(begin:Integer, end:Integer)) = P2(property.scope)
in subtraces->iterate(
subtrace:Tuple(begin:Integer, end:Integer);
result:Sequence(Tuple(begin:Integer, end:Integer, violations:OclAny)) = Sequence{} |
let newViolations:OclAny = P3(subtrace.begin, subtrace.end, property.pattern) in
if newViolations->notempty() then
result->append(
Tuple{begin:Integer = subtrace.begin,
end:Integer = subtrace.end,
violations:OclAny = newViolations})
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
4.3. OCL Functions for collecting violation information

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.

### 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.

#### 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 ← []
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

<table>
<thead>
<tr>
<th>Pattern representative</th>
<th>Condition</th>
<th>NSOC</th>
<th>UNOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventually ( E )</td>
<td>(</td>
<td>{E}</td>
<td>= 0 )</td>
</tr>
<tr>
<td>eventually at least ( n E )</td>
<td>(</td>
<td>{E}</td>
<td>&lt; n )</td>
</tr>
<tr>
<td>eventually at most ( n E )</td>
<td>(</td>
<td>{E}</td>
<td>&gt; n )</td>
</tr>
<tr>
<td>eventually exactly ( n E )</td>
<td>(</td>
<td>{E}</td>
<td>&lt; n )</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>{E}</td>
<td>&gt; n )</td>
</tr>
</tbody>
</table>

Legend. \(|\{E\}|\): 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

---

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.
4.3. OCL Functions for collecting violation information

Algorithm 21: reportPatternExistence

**Input:** begin, end: the boundaries of a sub-trace; pattern: an instance of the \emph{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

1. \(E\) ← event name in pattern
2. \(op\) ← comparison operator of the bound on the number of occurrences of event \(E\)
3. \(n\) ← threshold of the occurrence number of event \(E\)
4. \(result\) ← null, locations ← []
5. for \(i \leftarrow begin \) to \(end \) do
6. if self.traceElements\([i]\) = \(E\) then locations.append\((i)\)
7. \(count\) ← locations.size()
8. if \(op \neq \text{null} \) then
9. if \(count < n \&\& op \neq \text{“at most”} \) then
10. \(result\) ← (NSOC, locations)
11. else if \(count > n \&\& op \neq \text{“at least”} \) then
12. \(result\) ← (UNOC, locations\([n + 1, count]\))
13. else if \(count = 0\) then \(result\) ← (NSOC, [])
14. return \(result\)

Algorithm 22: reportPatternAbsence

**Input:** begin, end: the boundaries of a sub-trace; pattern: an instance of the \emph{absence} pattern of the form “never [exactly \(n\) ] \(E\)”

**Output:** result: a list of locations at which violations occur

1. \(E\) ← event name in pattern
2. \(result\) ← []
3. for \(i \leftarrow begin\) to \(end\) do
4. if self.traceElements\([i]\) = \(E\) then result.append\((i)\)
5. 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 \emph{precedence} and \emph{response} patterns into six types, i.e., NSOR, WTO, WTC, WTC, WTOC, LIRV, and LVRI. Function reportPatternPrecedence defines the algorithm for collecting violation information of a \emph{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; block₁, block₂: the two blocks of an instance of the precedence pattern of the form “block₁ preceding block₂” |
| Output: result: a list of 3-tuples, each of which consists of a violation type, a list of the locations of offending occurrences of block₂, and a list of the locations of corresponding occurrences of block₁ |

1. size₁, size₂ ← the sizes of block₁ and block₂
2. firstOfBlock₁ ← block₁.first().event firstOfBlock₂ ← block₂.first().event
3. (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
4. flag ← true, lₜ ← 0
5. result ← []
6. for i ← begin to end do
7.    elem ← self.traceElements[i], (e, t) ← (elem.event, elem.timestamp)
8.    if e = firstOfBlock₁ then (i₁, t₁, flag) ← (2, t, true)
9.    else if i₁ > 1 then (i₁, t₁, flag) ← matchSecondaryBlock(block₁, i₁, t₁, flag, e, t)
10.   if lₜ > 0 & & !flag then (i₁, t₁) ← (1, 0)
11.   else if i₁ = size₁ + 1 then
12.       if flag then break
13.       else
14.          lₜ ← i
15.          (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
16.   if e = firstOfBlock₂ then (i₂, t₂) ← (2, t)
17.   else if i₂ > 1 then (i₂, t₂) ← match(block₁, i₁, t₁, e, t)
18.   if i₂ = size₂ + 1 then
19.       if lₜ = 0 then v ← (NSOR, [i], [])
20.       else v ← (WTC, [i], [lₜ])
21.       result.append(v)
22.       (i₁, t₁) ← (1, 0), (i₂, t₂) ← (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 block₂ must be preceded by an occurrence of block₁. 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 block₁ and block₂). 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 block₁ 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 block₂, as well as a list of the positions of the corresponding occurrences of block₁.
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$[i_1 − 1]$ (respectively, block$[i_2 − 1]$). 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$[i_1]$(lines 8–15) and with block$[i_2]$(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. \hspace{1em} $op \leftarrow block[i].timeDistance.op$
3. \hspace{1em} $t' \leftarrow pt + block[i].timeDistance.value$
4. \hspace{1em} $flag \leftarrow flag \&\& compare(t, op, t')$
5. \hspace{1em} 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 \texttt{reportPatternPrecedenceGeneral} updates the variables \( i_1, t_1, flag \) with the result of function \texttt{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 \emph{valid} occurrence of \( block_1 \) (line 12); otherwise, the variable \( l_w \) is updated with the position \( i \) of this \emph{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 \texttt{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 \texttt{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 \texttt{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 \texttt{result} (line 21) and the tuples \((i_1, t_1)\) and \((i_2, t_2)\) are reset (line 22). The function returns variable \( \texttt{result} \) after investigating the entire trace segment (line 23).

Function \texttt{reportPatternPrecedenceAtLeast} defines the algorithm for collecting violations from a faulty trace segment which violates a \emph{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 \emph{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 \texttt{reportPatternPrecedenceGeneral}.

This variant of the \emph{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 \emph{valid} occurrence of \( block_1 \) before the occurrence of \( block_2 \); 2) the distance between the farthest \emph{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: \texttt{NSOR}, \texttt{WTO}, \texttt{WTC}, \texttt{WTOC}, and \texttt{LIRV}.

For instance, given a \emph{precedence} pattern “\texttt{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; block1, block2, n: the parameters of an instance of the precedence pattern of the form “block1 preceding at least n tu block2”

Output: result: a list of 3-tuples, each of which consists of a violation type, a list of the location of offending occurrences of block2, a list of the locations of corresponding occurrences of block1

1 size1, size2 ← the sizes of block1 and block2
2 firstOfBlock1 ← block1.first().event firstOfBlock2 ← block2.first().event
3 (i1, t1) ← (1, 0), (i2, t2) ← (1, 0)
4 (lc, tc) ← (0, 0), (lw, tw) ← (0, 0)
5 flag1 ← true, flag2 ← false
6 result ← []
7 for i ← begin to end do
8     elem ← self.traceElements[i], (e, t) ← (elem.event, elem.timestamp)
9     if lc = 0 then
10        if e = firstOfBlock1 then (i1, t1, flag1) ← (2, t, true)
11           else if i1 > 1 then
12              (i1, t1, flag1) ← matchSecondaryBlock(block1, i1, t1, flag1, e, t)
13           if lw > 0 && !flag1 then (i1, t1) ← (1, 0)
14           else if i1 = size1 + 1 then
15              if flag1 then (lc, tc) ← (i, t)
16              else (lw, tw) ← (i, t)
17              (i1, t1) ← (1, 0), (i2, t2) ← (1, 0)
18        else if i - tc ≥ n then break
19           if e = firstOfBlock2 then
20              (i2, t2) ← (2, t)
21              flag2 ← (lw > 0) && (t - tw < n)
22           else if i2 > 1 then (i2, t2) ← match(block1, i1, t1, e, t)
23           if i2 = size2 + 1 then
24              if lc = lw = 0 then v ← (NSOR, [i], [])
25              else if lw = 0 then v ← (WTOC, [i], [lc])
26              else if flag2 then v ← (WTOC, [i], [lw])
27              else if lc = 0 then v ← (LIRC, [i], [lc, lw])
28              else v ← (LIRC, [i], [lc, lw])
29              result.append(v)
30              (i1, t1) ← (1, 0), (i2, t2) ← (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 that 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{align*}
\text{begin, end:} & \text{ the boundaries of a sub-trace; block}_1, \text{block}_2, n: \text{ the parameters of an instance of the precedence pattern of the form } \text{“block}_1 \text{ preceding at most } n \text{ block}_2}\end{align*}

**Output:** \(result\): a list of 3-tuples, each of which consists of a violation type, a list of the locations of offending occurrences of \(block_1\), and a list of the locations of corresponding occurrences of \(block_2\)

1. \(size_1, size_2 \leftarrow \text{the sizes of block}_1 \text{ and block}_2\)
2. \(firstOfBlock1 \leftarrow \text{block}_1.\text{first()}.\text{event}\)
3. \(firstOfBlock2 \leftarrow \text{block}_2.\text{first()}.\text{event}\)
4. \((i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0)\)
5. \((l_c, t_c) \leftarrow (0, 0), (l_w, t_w) \leftarrow (0, 0)\)
6. \(\text{flag}_1 \leftarrow \text{true}, \text{flag}_2 \leftarrow \text{false}\)
7. \(result \leftarrow []\)
8. \(\text{for } i \leftarrow \text{begin } \text{to end } \text{do}\)
9. \(\text{elem} \leftarrow \text{self.traceElements}[i], (e, t) \leftarrow (\text{elem.event}, \text{elem.timestamp})\)
10. \(\text{if } e = \text{firstOfBlock}_1 \text{ then } (i_1, t_1, \text{flag}_1) \leftarrow (2, t, \text{true})\)
11. \(\text{else if } i_1 > 1 \text{ then } (i_1, t_1, \text{flag}_1) \leftarrow \text{matchSecondaryBlock} (\text{block}_1, i_1, t_1, \text{flag}_1, e, t)\)
12. \(\text{if } i_1 = \text{size}_1 + 1 \text{ then}\)
13. \(\text{if } \text{flag}_1 \text{ then } (l_1, t_c) \leftarrow (i, t)\)
14. \(\text{else } (l_w, t_c) \leftarrow (i, t)\)
15. \(\text{if } e = \text{firstOfBlock}_2 \text{ and } (t_c = 0 \text{ or } t - t_c > n) \text{ then}\)
16. \( (l_2, t_2) \leftarrow (2, t)\)
17. \(\text{flag}_2 \leftarrow \text{true}\)
18. \(\text{else if } i_2 > 1 \text{ then } (i_2, t_2) \leftarrow \text{match} (\text{block}_1, i_1, t_1, e, t)\)
19. \(\text{if } i_2 = \text{size}_2 + 1 \text{ then}\)
20. \(\text{if } l_c = l_w = 0 \text{ then } v \leftarrow (\text{NSOR}, [i], [])\)
21. \(\text{else if } l_w < l_c \text{ then } v \leftarrow (\text{WTO}, [i], [l_c])\)
22. \(\text{else if } \text{flag}_2 \text{ then } v \leftarrow (\text{WTOC}, [i], [l_w])\)
23. \(\text{else if } l_c = 0 \text{ then } v \leftarrow (\text{WTC}, [i], [l_w])\)
24. \(\text{else } v \leftarrow (\text{LVRI}, [i], [l_c, l_w])\)
25. \(\text{result.append}(v)\)
26. \((i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0)\)
27. \(\text{return } result\)

Function \text{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: \(\text{block}_1, \text{block}_2, \text{and the maximum } n\) 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 block₂ where violations occur, and the locations of corresponding occurrences of block₁.

This precedence pattern specifies that each occurrence of block₂ must be preceded, within n time units, by an occurrence of block₁. Hence, an instance of this pattern can be violated when one of the following conditions holds: 1) there is no occurrence of block₁ before an occurrence of block₂; 2) the occurrences of block₁ are all beyond n time units prior to an occurrence of block₂. 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 block₁[i₁] (lines 9–14) and with block₂[i₂] (lines 15–26). According to the semantics of this precedence pattern, we try to find a new occurrence of block₁, that conforms to the constraint “at most n tu”. The function uses the same code to update the variables i₁, t₁, and flag₁ for matching block₁[i₁] lines 9–10. When the matched event is the last event of block₁, the function assigns the current location and timestamp to the variables (l_c, t_c) if flag₁ is true, or otherwise to (l_w, t_w) (lines 11–14) and resets variables i₁, t₁, i₂, t₂ (line 14).

In the loop, the algorithm also checks whether the current trace element is part of an occurrence of block₂. If an occurrence of the first event of block₂ is detected and there is no valid occurrence of block₁ or the distance from the previous valid occurrence block₁ violates the constraint “at most n tu” (i.e., t − l_c > n) (line 15), then the function moves on to match the second event of block₂ (line 16). In addition, the variable flag₂ 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 block₁ (if present) is in violation of the constraint on the distance between the two blocks. Otherwise, if variable i₂ is greater than 1, the algorithm calls function match to match the current trace element with block₂[i₂] (line 18).

The algorithm reports a violation when block₂ 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 block₁, if there is no occurrence of block₁ before the occurrence of block₂ (line 20); otherwise, a WTO violation is reported with [i] and a list of the location of the valid occurrence of block₁ (i.e., [l_c]), when the valid occurrence of block₁ is closer to the occurrence of block₂ 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 block₁ if variable flag₂ is true, meaning that the constraint “at most n tu” is violated by the closest invalid occurrence of block₁ (line 22); otherwise, a WTC violation is reported with [i] and [l_w], if the distance between the invalid occurrence block₁ and the current occurrence of block₂ 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 (i₁, t₁) and (i₂, t₂) 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 block₂ must be preceded by an occurrence of block₁ with an exact distance n. Hence, an instance of this pattern can be violated if there is either no
4.3. OCL Functions for collecting violation information

**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 to 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\(_1\)

\(\text{size}_1, \text{size}_2 \leftarrow \text{the sizes of block}_1 \text{ and block}_2\)

\(\text{firstOfBlock}_1 \leftarrow \text{block}_1.\text{first()}.\text{event} \quad \text{firstOfBlock}_2 \leftarrow \text{block}_2.\text{first()}.\text{event}\)

\((i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0)\)

flag \leftarrow true, ct \leftarrow 0, candidates \leftarrow [], result \leftarrow []

**for** \(i \leftarrow \text{begin} \text{ to end do}\)

\(\text{elem} \leftarrow \text{self}.\text{traceElements}[i], (e, t) \leftarrow (\text{elem}.\text{event}, \text{elem}.\text{timestamp})\)

**if** \(e \leftarrow \text{firstOfBlock}_1\) **then** \((i_1, t_1, \text{flag}) \leftarrow (2, t, \text{true})\)

**else if** \(i_1 > 1\) **then** \((i_1, t_1, \text{flag}) \leftarrow \text{matchSecondaryBlock} (\text{block}_1, i_1, t_1, \text{flag}, e, t)\)

**if** \(i_1 = \text{size}_1 + 1\) **then** candidates.append\((i_1, t_1, \text{flag})\), \((i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0)\)

**if** \(e \leftarrow \text{firstOfBlock}_2\) **then**

**if** \(\exists \ell', (\ell', t - n, \text{true}) \in \text{candidates}\) **then** \((i_2, t_2) \leftarrow (1, 0)\)

**else** \((i_2, t_2, ct) \leftarrow (2, t, t)\)

**else if** \(i_2 > 1\) **then** \((i_2, t_2) \leftarrow \text{match} (\text{block}_1, i_1, t_1, e, t)\)

**if** \(i_2 = \text{size}_2 + 1\) **then**

**if** candidates.isEmpty() **then** result.append\(((\text{NSOR}, [i], []))\)

**else**

\(L \leftarrow \emptyset, (i', t', \text{flag'}) \in \text{candidates} \&\& ct - t' = n\)

**if** \(L, \text{notEmpty}()\) **then** result.append\(((\text{WTO}, [i], L))\)

**else**

\((l_1', \text{flag}_{l_1}') \leftarrow \text{last}\(((l', \text{flag'}) | (l', t', \text{flag'}) \in \text{candidates} \&\& ct - t' > n)\)

\((l_2', \text{flag}_{l_2}') \leftarrow \text{first}\(((l', \text{flag'}) | (l', t', \text{flag'}) \in \text{candidates} \&\& ct - t' < n)\)

**if** \(\text{flag}_{l_1}'\) **then**

**if** \(\text{flag}_{l_2}'\) **then** result.append\(((\text{WTO}, [i], [l_1', l_2']))\)

**else if** \(l_1' \neq \text{null}\) **then** result.append\(((\text{LIRV}, [i], [l_1', l_2']))\)

**else result.append\(((\text{WTOC}, [i], [l_1']))\)\)

**else if** \(l_1' \neq \text{null}\) **then**

**if** \(\text{flag}_{l_2}'\) **then** result.append\(((\text{LIRV}, [i], [l_1', l_2']))\)

**else if** \(l_2' \neq \text{null}\) **then** result.append\(((\text{WTOC}, [i], [l_2']))\)

**else result.append\(((\text{WTOC}, [i], [l_2']))\)\)

**else result.append\(((\text{WTOC}, [i], [l_2']))\)\)

**else result.append\(((\text{WTOC}, [i], [l_2']))\)\)

candidates \leftarrow \emptyset, (i', t', \text{flag'}) \in \text{candidates} \&\& ct - t' < n\)

\((i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0)\)

**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
4.3. OCL Functions for collecting violation information

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

**Algorithm 28: reportPatternResponseGeneral**

**Input:** begin, end: the boundaries of a sub-trace; block₁, block₂: the two blocks of an instance of the response pattern of the form “block₁ responding block₂”

**Output:** result: a list of 3-tuples, each of which consists of a violation type, a list of the locations of offending occurrences of block₂, and a list of the locations of corresponding occurrences of block₁

```plaintext
1 size₁, size₂ ← the sizes of block₁ and block₂
2 firstOfBlock₁ ← block₁.first().event, firstOfBlock₂ ← block₂.first().event
3 (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
4 flag ← true, lₜ ← 0, candidates ← [], result ← []
5 for i ← begin to end do
6     elem ← self.traceElements[i], (e, t) ← (elem.event, elem.timestamp)
7     if e = firstOfBlock₂ then (i₂, t₂) ← (2, t)
8     else if i₂ > 1 then (i₂, t₂) ← match(block₂, i₂, t₂, e, t)
9     if i₂ = size₂ + 1 then
10        candidates.append((i₂, t₂))
11        (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
12        if candidates.notEmpty() && e = firstOfBlock₁ then (i₁, t₁, flag) ← (2, t, true)
13        else if i₁ > 1 then (i₁, t₁, flag) ← matchSecondaryBlock(block₁, i₁, t₁, flag, e, t)
14        if i₁ = size₁ + 1 then
15           if flag then candidates ← []
16           else lₜ ← i
17           (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
18        if candidates.notEmpty() then
19           L₁ ← {l’ | (l’, t’) ∈ candidates && l’ < lₜ}
20           if L₁.notEmpty() then result.append((WTC, L₁, [lₜ]))
21           L₂ ← {l’ | (l’, t’) ∈ candidates && l’ > lₜ}
22           if L₂.notEmpty() then result.append((NSOR, L₂, []))
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 block₂ must be followed by an occurrence of block₁, so there are two possible types of violations (i.e., NSOR and WTC); WTC is only possible if block₁ 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 block₂ 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 block₂[i₂] (lines 7–11) and with block₁[i₁] (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 $L'_t$, 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 $n + t_u$” with respect to the last invalid occurrence of block$_1$, and stores them in variable $L'_f$. 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 $L - L_v - L_f$ (line 24). If the last invalid occurrence of block$_1$ is found after the invalid one and variable $L_v$ is not empty, the function reports a set of WTO
violations accompanied by $LV$ and $|lc|$ (line 25). Otherwise if the $lw$ is greater than $lc$, if there exist occurrences of $block_2$ before the position $lw$ and the distance constraint “at least $n$ tu” is violated by the invalid occurrence of $block_1$ at $lw$ (i.e., $L_1 - L'_I \neq \emptyset$), the function reports a set $\text{WTOC}$ 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., $LV \cap L'_I \neq \emptyset$), the function reports a set $\text{LVRI}$ violations.

4.3. OCL Functions for collecting violation information

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. $firstOfBlock_1 \leftarrow block_1.first().event, firstOfBlock_2 \leftarrow block_2.first().event$
3. $(i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0), (lc, tc) \leftarrow (0, 0), (lw, tw) \leftarrow (0, 0)$
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 $e = firstOfBlock_2$ 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 $candidates.isEmpty()$ & $e = firstOfBlock_1$ 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.         $(lc, tc) \leftarrow (i, ct)$
18.        else $(lw, tw) \leftarrow (i, ct)$
19.       $(i_1, t_1) \leftarrow (1, 0), (i_2, t_2) \leftarrow (1, 0)$
20.     if $candidates.isEmpty()$ then
21.       $L \leftarrow \{l' \mid (l', t') \in candidates\}$
22.       $L_1 \leftarrow \{l' \mid (l', t') \in candidates \& l' < lw\}, LV \leftarrow \{l' \mid (l', t') \in candidates \& l' < lc\}$
23.       $L'_{L} \leftarrow \{l' \mid (l', t') \in candidates \& lw - t' \geq n\}$
24.       if $L - LV - L_1 \neq \emptyset$ then $result.append((\text{NSOR}, L - LV - L_1, []))$
25.       if $lw < lc \& LV \neq \emptyset$ then $result.append((\text{WTOC}, LV, [lw]))$
26.     else if $lw < lc$ then
27.       if $L_1 - L'_I \neq \emptyset$ then $result.append((\text{WTOC}, L_1 - L'_I, [lw]))$
28.       if $LV \cap L'_I \neq \emptyset$ then $result.append((LVRI, LV \cap L'_I, [lc, lw]))$
29. return result
Algorithm 30: reportPatternResponseAtMost

**Input**: begin, end: the boundaries of a sub-trace; block₁, block₂, n: the parameters of an instance of the response pattern of the form “block₁ responding at most n tu block₂”

**Output**: result: a list of 3-tuples, each of which consists of a violation type, a list of the location of offending occurrences of block₂, and a list of the locations of corresponding occurrences of block₁

1. size₁, size₂ ← the sizes of block₁ and block₂
2. firstOfBlock₁ ← block₁.first().event, firstOfBlock₂ ← block₂.first().event
3. (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
4. flag ← true, ct ← 0, lᵳ ← 0, candidates ← [], result ← []
5. for i ← begin to end do
   6.   elem ← self.traceElements[i], (e, t) ← (elem.event, elem.timestamp)
   7.   if e = firstOfBlock₂ then (i₂, t₂) ← (2, t)
   8.   else if i₂ > 1 then (i₂, t₂) ← match(block₂, i₂, t₂, e, t)
   9.   if i₂ = size₂ + 1 then
      10.      candidates.append((i₂, t₂))
      11.      (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
   12.   if candidates.isEmpty() & e = firstOfBlock₁ then (i₁, t₁, ct) ← (2, t, t)
   13.   else if i₁ > 1 then (i₁, t₁, flag) ← matchSecondaryBlock(block₁, i₁, t₁, flag, e, t)
   14.   if i₁ = size₁ + 1 then
      15.      L₁ ← \{l′ | (l′, t′) ∈ candidates & l′ < lᵳ & ct − t′ > n\}
      16.      L₂ ← \{l′ | (l′, t′) ∈ candidates & l′ > lᵳ & ct − t′ > n\}
     17.     if flag then
      18.        if L₁.isEmpty() then result.append((LIRV, L₁, [lᵳ, i]))
      19.        if L₂.isEmpty() then result.append((WTO, L₂, [i]))
      20.        candidates ← []
      21.     else
      22.        if L₁.isEmpty() then result.append((WTC, L₁, [lᵳ]))
      23.        if L₂.isEmpty() then result.append((WTOC, L₂, [i]))
      24.        candidates ← \{(l′, t′) | (l′, t′) ∈ candidates & ct − t′ ≤ n\}
      25.        lᵳ ← i
      26.        (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0)
   27.   if candidates.isEmpty() then
      28.      L₁ ← \{l′ | (l′, t′) ∈ candidates & l′ < lᵳ\}
      29.      if L₁.isEmpty() then result.append((WTC, L₁, [lᵳ]))
      30.      L₂ ← \{l′ | (l′, t′) ∈ candidates & l′ > lᵳ\}
      31.      if L₂.isEmpty() then result.append((NSOR, L₂, []))
   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\textsubscript{2} is not followed by an occurrence of block\textsubscript{1} or the distances between an occurrence of block\textsubscript{2} and all valid occurrences of block\textsubscript{1} are more than \(n\). So there are five possible violation types for this pattern: \textit{NSOR}, \textit{WTO}, \textit{WTC}, \textit{WT\textsubscript{OC}}, and \textit{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 \textit{begin} to \textit{end} of the trace instance and tries to find occurrences of the pair of block\textsubscript{2} (lines 7–11) and block\textsubscript{1} (lines 12–26). The pseudocode (lines 7–13) for matching the two blocks is the same as the previous functions until block\textsubscript{1} is fully matched (line 14).

In lines 15–26, the algorithm examines the \textit{response} pattern and update \textit{result} if any violation emerges. More specifically, from the list candidates, the algorithm extracts the occurrence locations of block\textsubscript{2} that fails fulfilling the maximum limit on the distance to the matched occurrence of block\textsubscript{1}, and assigns the ones before (respectively, after) the previous (invalid) occurrence of block\textsubscript{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\textsubscript{1} complies with the distance constraints (if defined) within the block, (line 17), if variable \(L\_1\) is not empty, a group of \textit{LIRV} violations, the locations of offending occurrences of block\textsubscript{2} stored in variable \(L\_1\), and a list with the two locations of the corresponding occurrence of block\textsubscript{1}, is appended to \textit{result} (line 18); if variable \(L\_2\) is not empty, a group of violations that contains the violation type \textit{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\textsubscript{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 \textit{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 \textit{WT\textsubscript{OC}}, 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\textsubscript{2} which comply with the constraint “at most \(n\) tu” with respect to the current matched occurrence of block\textsubscript{1} (line 24). Moreover, the variable \(l\_w\) is updated with the current location that indicates a new invalid occurrence of block\textsubscript{1} (line 25).

The violation information is accumulated in variable \textit{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 \textit{WTC} violations when there exist occurrences of block\textsubscript{2} before \(l\_w\) (line 29), and collects \textit{NSOR} violations when there exist occurrences of block\textsubscript{2} after which no occurrence of block\textsubscript{1} is found (line 31).

Function \texttt{reportPatternResponseExactly} defines the algorithm for collecting violation information for the variant of \textit{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 \textit{response} pattern: \textit{NSOR}, \textit{WTO}, \textit{WTC}, \textit{WT\textsubscript{OC}}, \textit{LVRI}, and \textit{LIRV}.

Instead of using variable \(l\_w\) (in function \texttt{reportPatternResponseAtMost}) to save the location of the latest occurrence of block\textsubscript{1} that is invalid with respect to the distance constraints defined in the block, in this function we introduce a pair of variables \((l_i, flag_i)\) (line 4) to store the location and validity of the latest occurrence of block\textsubscript{1}.
Algorithm 31: reportPatternResponseExactly

**Input:** begin, end: the boundaries of a sub-trace; block₁, block₂, n: the parameters of an instance of the response pattern “block₁ responding exactly n tu block₂”

**Output:** result: a list of 3-tuples, each of which consists of a violation type, a list of the locations of offending occurrences of block₂, and a list of the locations of corresponding occurrences of block₁

1. \( size₁, size₂ \) ← the sizes of block₁ and block₂
2. \( firstOfBlock₁ ← block₁.first().event, firstOfBlock₂ ← block₂.first().event \)
3. \( (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0) \)
4. \( flag ← \text{true}, ct ← 0, (l₁, flag₁) ← (0, \text{true}), candidates ← [], result ← [] \)
5. for \( i ← \text{begin to end do} \)
6. \( \text{elem} ← \text{self.traceElements}[i], (e, t) ← (\text{elem.event, elem.timestamp}) \)
7. else if \( i₂ > 1 \) then \( (i₂, t₂) ← \text{match(block₂, i₂, t₂, e, t)} \)
8. if \( i₂ = size₂ + 1 \) then \( \text{candidates.append}((i₂, t₂)) \), \( (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0) \)
9. if \( \text{candidates.isEmpty()} \) \&\& \( e = \text{firstOfBlock₁} \) then \( (i₁, t₁, ct) ← (2, t, t) \)
10. if \( i₁ > 1 \) then \( (i₁, t₁, flag) ← \text{matchSecondaryBlock} (block₁, i₁, t₁, flag, e, t) \)
11. if \( i₁ = size₁ + 1 \) then
12. \( L ← \{ l' | (l', t') ∈ \text{candidates} \&\& ct - t' = n \} \)
13. if \( ! \text{flag} \) \&\& \( \text{L.isEmpty()} \) then \( \text{result.append}((\text{WTC, } L, [i]) \)
14. \( L₁ ← \{ l' | (l', t') ∈ \text{candidates} \&\& l' < l₁ \&\& ct - t' > n \} \)
15. \( L₂ ← \{ l' | (l', t') ∈ \text{candidates} \&\& l' > l₁ \&\& ct - t' > n \} \)
16. if \( L₁ \not\text{isEmpty()} \) then
17. \( \text{if flag then} \)
18. \( \text{if flag₁ then result.append}((\text{WTO, } L₁, [l₁, i]) \)
19. \( \text{else result.append}((\text{LIRV, } L₁, [l₁, i]) \)
20. \( \text{else} \)
21. \( \text{if flag₁ then result.append}((\text{LVRI, } L₁, [l₁, i]) \)
22. \( \text{else result.append}((\text{WTOC, } L₁, [l₁, i]) \)
23. if \( L₂ \not\text{isEmpty()} \) then
24. \( \text{if flag then result.append}((\text{WTO, } L₂, [i]) \)
25. \( \text{else result.append}((\text{WTOC, } L₂, [i]) \)
26. \( \text{candidates} ← \{ (l', t') | (l', t') ∈ \text{candidates} \&\& ct - t' < n \} \)
27. \( (l₁, flag₁) ← (i, flag), (i₁, t₁) ← (1, 0), (i₂, t₂) ← (1, 0) ) \)
28. if \( \text{candidates.isEmpty()} \) then
29. \( L₁ ← \{ l' | (l', t') ∈ \text{candidates} \&\& l' < l₁ \}, L₂ ← \{ l' | (l', t') ∈ \text{candidates} \&\& l' > l₁ \} \)
30. if \( L₁ \not\text{isEmpty()} \) then
31. \( \text{if flag then result.append}((\text{WTO, } L₁, [l₁]) \)
32. \( \text{else result.append}((\text{WTOC, } L₁, [l₁]) \)
33. if \( L₂ \not\text{isEmpty()} \) then \( \text{result.append}((\text{NSOR, } L₂, []) \)
34. return result
In the loop (lines 5–28) that searches for occurrences of block2 and block1, the function differs from the previous functions when an occurrence of block1 is fully matched (lines 12–28). More specifically, we first retrieve the occurrence of block2 that meets the distance constraint with respect to this match with block1 (line 13). If the result is not empty and the match for block1 is valid with respect to the distance constraints within the block, we collect a WTO violation with the offending occurrence of block2 and the current location. In addition, we extract the locations of the occurrences of block2 that happen before the previous occurrence of block1 (i.e., l1) and farther than n to the current occurrence of block1, and store them in variable L1 (line 15); we also store the locations of the occurrences of block2 after l1 and also farther than n to the current match for block1 in L2 (line 16). If variable L1 is not empty (line 17), we collect a set of violations with L1 and a list comprised of both the locations of the previous and current occurrences of block1: if the current match of block1 is valid (i.e., flag is true), depending on the validity of the previous match for block1, 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 block1. If variable L2 is not empty (line 24), we collect a set of violations with L2 and a list containing the current location i: if the current match for block1 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 block2 that have been examined (line 27) and also update the variables l1, flag1, i1, t1, l2, t2.

After the analysis of the entire input trace, if variable candidates still contains some matches for block2 that have not been examined (line 29), we use the position of the last occurrence of block1 (l1) to split the locations stored in candidates into two parts (i.e., L1 and L2) (line 30) and collect violation information: if variable L1 is not empty, if the last occurrence of block1 is valid, we collect a set of WTO violations (line 32) or else a set of WTOC violations (line 33); if variable L2 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 TemPsi 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
Chapter 4. Model-driven Violation Reporting for Trace Checking

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

92
### 4.5. Visualizing Violation Information

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

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>F1 F2 F3</td>
</tr>
<tr>
<td>R2</td>
<td>F4 F5 F6</td>
</tr>
<tr>
<td>R3</td>
<td>F2 F4 F5 F7</td>
</tr>
</tbody>
</table>

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](http://weidou.github.io/TemPsy-Violation-Visualization). The tool is based on technologies such as amCharts⁵, Meter.js⁶, AngularJS⁷, MongoDB³, and ElasticSearch⁸. 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

---

⁵https://www.amcharts.com
⁶https://www.meteor.com
⁷https://angularjs.org
⁸https://www.elastic.co
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 \textit{TemPsy} property. For each type of \textit{TemPsy} patterns, we address two research questions:

RQ1) Given a fixed number of violations, what is the relation between the execution time of \texttt{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 \texttt{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 \textit{pattern} used in a \textit{TemPsy} property, we select the ones that use the \textit{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 \textit{scope} + \textit{pattern}; the events involved in the property (e.g., “a citizen requests a certificate”) are denoted by uppercase letters.

<table>
<thead>
<tr>
<th>Table 4.3: \textit{TemPsy} properties used for the evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: globally always \textit{A}</td>
</tr>
<tr>
<td>P2: globally never \textit{B}</td>
</tr>
<tr>
<td>P3: globally eventually at least 2 \textit{A}</td>
</tr>
<tr>
<td>P4: globally eventually at most 3 \textit{A}</td>
</tr>
<tr>
<td>P5: globally \textit{A} responding at most 1000 tu \textit{B}</td>
</tr>
<tr>
<td>P6: globally \textit{A} responding exactly 1000 tu \textit{B}</td>
</tr>
<tr>
<td>P7: globally \textit{A} preceding at most 6000 tu \textit{B}</td>
</tr>
<tr>
<td>P8: globally \textit{A} preceding at least 100 tu \textit{B}</td>
</tr>
<tr>
<td>P9: globally \textit{A} preceding exactly 100 tu \textit{B}</td>
</tr>
<tr>
<td>P10: globally \textit{A}, \textit{B} preceding at least 1000 tu \textit{C}, \textit{D}</td>
</tr>
<tr>
<td>P11: globally \textit{A} responding at least 1000 tu \textit{B}, \textit{C}</td>
</tr>
<tr>
<td>P12: globally \textit{A} responding \textit{B}</td>
</tr>
</tbody>
</table>

We use \textit{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 \textit{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 \texttt{TEMPsy-Report} GitHub
Chapter 4. Model-driven Violation Reporting for Trace Checking

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.
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 $(6000, 6000 \times 1.1)$. 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 $(100, 100 \times 1.1]$. 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
Chapter 4. Model-driven Violation Reporting for Trace Checking

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

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 rea-
sons) 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

<table>
<thead>
<tr>
<th>Language</th>
<th>Features</th>
<th>Tool support</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Flake and Müller, 2003]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[Küster-Filipe and Anderson, 2006]</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>[Robinson, 2008]</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>[Kanso and Taha, 2014]</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>TemPsy</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

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.
Chapter 5. Related Work

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
Chapter 5. Related Work

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Bool</th>
<th>Positions</th>
<th>Reasons</th>
<th>Visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREACH [Donzé, 2010]</td>
<td>+</td>
<td>+</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>MONPOLY [Basin et al., 2012]</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QEA [Reger et al., 2015]</td>
<td>+</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOLOIST+ZOT [Bersani et al., 2014]</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RV-MONITOR [Luo et al., 2014]</td>
<td>+</td>
<td></td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

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
Chapter 6. Conclusions and Future Work

- A 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


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.

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:
Appendix A. Tool Support

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 TEMSY-CHECK.

![Figure A.2: Checking TemPsy expressions on an execution trace](image)

**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 TEMSY-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.

![Figure A.3: Checking results of TEMSY-CHECK](image)

\(^1\)http://weidou.github.io/TemPsy-Check
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.

Figure A.4: Presentation of violation information