POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

TRACE CHECKING OF QUANTITATIVE PROPERTIES

Doctoral Dissertation of:
Srđan Krstić

Supervisor:
Prof. Carlo Ghezzi

Co-supervisor:
Dr. Domenico Bianculli
Prof. Pierluigi San Pietro

Tutor:
Prof. Donatella Sciuto

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2015 – Cycle XXVIII

Acknowledgements

It is really hard to look back on the past three years and find the appropriate words that can sum them up in a meaningful way. Three years ago, I was not able to fully appreciate most of the concepts in this thesis. Considering only these two data points, one can plot two different graphs representing my ability to conduct research and my disposition towards it, respectively. The former monotonically increases, while the latter resembles a fairly dangerous roller-coaster, but unlike a typical roller-coaster, it ends at its highest point. Fortunately for me, I was not alone during this three-year-long ride, there were many people that have indebted me greatly and I would like to dedicate these pages to them.

First of all, this work would not have been possible without the technical and moral support of my thesis advisor, *Carlo Ghezzi*. He supervised me back when I pursued my masters degree, introduced me to the field of verification and opened many doors for me in the research community. His enthusiastic supervision has been a continuous source of encouragement and I consider myself fortunate that I had access to his valuable guidance. His experience and accomplishments [37] will remain an inspiration to me for years to come.

I would like to thank my formal and informal co-advisors *Domenico Bianculli* and *Pierluigi San Pietro* for being invaluable source of guidance and inspiration. Domenico devoted a lot of time to track my progress, engage me in fruitful discussions and groom my writing and presentation skills. I am fairly certain that I have empirical evidence of his unlimited patience. Pierluigi helped me immensely to carry out the formal aspects of my research and often our lengthy discussions led to exciting new ideas, like lazy semantics for MTL.

I would also like to thank my co-authors *Marcello Bersani, Alessio Gambi* and *Giovanni Paolo Gibilisco* for their help in developing many ideas presented in this thesis. I also thank *Dimitra Giannakopoulou* for hosting me at NASA Ames and introducing me to the exciting research done at NASA, which I look forward to contribute to myself in the future.

I was grateful to share many moments that go beyond work with my colleagues and friends from the DEEPSE research group: *Michele Ciavotta, Diego Perez, Christos Tsigkanos, Marco Scavuzzo, Mohammad Mehdi Pourhashem, Claudio Menghi, Alessandro Rizzi, Damian Andrew Tamburri, Clément Quinton, Manuel Mazzara, Mehrnoosh Askarpour, Luca Florio, Narges Shahmandi, Amir Molzam Sharifloo, Adnan Shahzada, Marco Miglierina, Federica Panella, Riccardo Desantis, Luigi Manco, Santo Lombardo, Va-*

lerio Panzica La Manna, Mikhail Afanasov, Mohammad Ghafari, Alessandra Viale, Danilo Ardagna, Marco Funaro, Matteo Rossi, Liliana Pasquale, Giordano Tamburelli, Mario Sangiorgio, Elisabetta Di Nitto, Luciano Baresi and *Matteo Pradella.*

I must thank people most responsible for the inclines in my roller-coaster graph — my friends: *Milan Andrejević, Dusan Stefanović, Aleksandar Blagotić, Milan Marković, Nenad Mladenović, Ivan Stojanović, Milena Cvetkov, Aleksandar Aleksić, Stefan Sladić, Dušan Vučković, Ivor Didović, Nikola Pavlović, Mira Vasić, Ana Veljković, Iva Stoyanova, Mina Todorović, Damián Soriano, Mercedes Sarua, Dana Dorneanu, Denis Ćutić, Guru Basavaraja, Karthik Duraisami, Majid Shokrolahi, Amir Ashouri, Nikola Pejčić, Nemanja Stolić* and many others. Even though I do not get to see everybody as much as I would like, the impact of our friendship remains and still contributes to my interests and personality.

Finally, my father *Bratislav* and mother *Suzana* have provided me with more help, encouragement and forbearance than anyone can hope for and resisted asking too many questions about how the PhD was going. I am grateful to my girlfriend *Ivana* for her love, for sharing ups and downs, and for reminding me when necessary that computer science is not the most important thing in life. Without the unconditional love and support from my closest ones, I would not have been able to write this thesis. Therefore, I dedicate this thesis to them.

Abstract

SOFTWARE engineering has dramatically changed over the past decade and many of the changes have challenged our most basic assumptions about the nature of the software products that we develop. The most important realization is that modern software has a very complex interaction with the environment in which it executes and it is often not safe to assume that the behavior of the environment is stable. Designing software that anticipates changes in the environment makes the software itself exhibit dynamic behavior that can only be observed at run time. This asks for verification techniques that complement design-time approaches and puts forward *trace checking* as a viable complementary choice for verifying modern software. Trace checking is an automatic procedure for evaluating a formal specification over a trace of recorded events produced by a system after execution. The output of the procedure states whether the system behaves according to its specification.

The goal of this thesis is to develop general and efficient trace checking procedures that support a broad class of *quantitative properties*. Quantitative properties can be seen as *constraints* on quantifiable values observed in an execution of a system. Quantitative properties typically express non-functional requirements, like constraints on resource utilization (e.g., number of computation resources, power consumption, costs), constraints on the runtime characteristics of the environment (e.g., arrival rates, response time), or constraints on the runtime behavior of the system (e.g., timing constraints, QoS, availability, fault tolerance).

The first part of the thesis discusses two algorithms that implement the

satisfiability procedure for SOLOIST — a specification language based on metric temporal logic (MTL) used to express quantitative properties. We show how a satisfiability procedure can be used to perform trace checking and apply the proposed approach to an extensive case study in the domain of cloud-based elastic systems. The second part of the thesis focuses on the problem of distributed trace checking and provides algorithms that rely on existing distributed computation frameworks (like MapReduce and Spark) to efficiently check SOLOIST specifications over very large traces. The thesis also contributes to the state of the art in MTL trace checking by proposing a novel decomposition technique for MTL formulae. This decomposition provides a scalable way of trace checking formulae with large time intervals. Due to known restrictions of the standard point-based MTL semantics we facilitate the decomposition by proposing an alternative semantics for MTL, called *lazy* semantics. The new semantics is more powerful than point-based semantics and possesses certain properties that allow us to decompose any MTL formula into an equivalent MTL formula with smaller time intervals.

Sommario

La disciplina dell'ingegneria del software è cambiata radicalmente nell'ultimo decennio e molti dei cambiamenti hanno stravolto le nostre assunzioni fondamentali sulla natura del software che sviluppiamo. La consapevolezza più importante è che il software moderno ha una complessa interazione con l'ambiente in cui viene eseguito e spesso non si può supporre che il comportamento dell'ambiente sia stabile. La progettazione di software in grado di anticipare i cambiamenti dell'ambiente determina che il software stesso esibisca un comportamento dinamico, osservabile solo in fase di esecuzione. Questo tipo di software richiede tecniche di verifica che siano di complemento agli approcci usati durante lo sviluppo e propone la tecnica di *verifica di tracce* (trace checking) come una soluzione praticabile e complementare per la verifica del software moderno. La verifica di tracce è una procedura automatica per la valutazione di una specifica formale su una traccia di eventi prodotti dal sistema e salvati dopo l'esecuzione del sistema stesso. L'output di questa procedura indica se il sistema è stato conforme alla sua specifica.

L'obiettivo di questa tesi è quello di sviluppare procedure di trace checking che siano generali ed efficienti e che supportino un'ampia classe di *proprietà quantitative*. Quest'ultime sono dei *vincoli* su valori quantificabili osservati durante l'esecuzione di un sistema. Questo tipo di proprietà esprime tipicamente requisiti non funzionali, come i vincoli sull'utilizzazione delle risorse (e.g., numero di risorse, consumo di energia, costi), vincoli sulle caratteristiche di esecuzione dell'ambiente (e.g., frequenza delle richieste, tempo di risposta), o vincoli sul comportamento del sistema durante

la sua esecuzione (per esempio, vincoli temporali, qualità del servizio, affidabilità, tolleranza ai guasti). La prima parte della tesi descrive due algoritmi che implementano la procedura di soddisfacibilità per SOLOIST - un linguaggio di specifica basato sulla logica temporale con metrica (MTL) ed utilizzato per esprimere proprietà quantitative. La tesi mostra come si possa usare una procedura di soddisfacibilità per eseguire la verifica di tracce e descrive l'applicazione di questa procedura ad un caso di studio nel settore dei sistemi elastici basati su infrastrutture cloud. La seconda parte della tesi si concentra sul problema della verifica distribuita di tracce e descrive algoritmi basati su modelli di calcolo distribuito (come MapReduce e Spark) per la verifica, in modo efficiente, di specifiche SOLOIST su tracce di esecuzione molto grandi. La tesi considera anche il dominio della verifica su tracce di proprietà espresse in MTL e propone una nuova tecnica per la decomposizione di formule MTL. Questa tecnica permette di effettuare la verifica di tracce, in modo scalabile, anche in presenza di formule con intervalli temporali molto ampi. La tecnica di decomposizione proposta supera le limitazioni della semantica standard di MTL attraverso la definizione di una nuova semantica per MTL, chiamata *"lazy"*. Questa nuova semantica è più espressiva della semantica standard e possiede alcune proprietà che permettono di decomporre qualsiasi formula MTL in una formula MTL equivalente con intervalli di tempo più piccoli.

Contents

Part I

Prologue

CHAPTER 1

Introduction

1.1 Quantitative properties of Open-world Software

Traditional software systems are products of early approaches in software engineering that typically employ waterfall software process [109] in the development phase and handle all the changes after the deployment via software change requests [24]. These early approaches were designed with certain fundamental assumptions about the nature of the software under development:

- software requirements are considered to be *known* and *stable* before the development;
- software modules are *statically* bound to each other;
- there is a *single* organization responsible for the complete lifecycle of software;
- software is deployed on a *well-known* and *centralized* infrastructure;
- software runs in a static environment, that *does not* change.

The last assumption is the most important one and it is usually phrased as "the external world in which the software executes, is *closed* [9]". This means that requirements leading to a specification of the software system's interaction with the external world can capture all phenomena of interest.

However, modern software systems invalidate all the assumptions above. In almost all the practical cases requirements cannot be fully gathered upfront and frozen before the development [95] since the stakeholders often do not know beforehand what they expect from a system and the requirements get refined as the stakeholders interact with the initial prototypes of the system. Modern systems are assembled out of components that provide a specific functionality. The most prominent examples of this approach are service-based applications (SBA) [62], often created as compositions of many existing services using service orchestration languages [7]. Bindings among the services are delayed until the execution, thus the specific functionality of SBA is only known at run-time. Software increasingly relies on functionalities coming from *third party* organizations. For example, many Android applications rely on the functionality provided by Google's location service [2]. Therefore the maintenance of such software cannot be performed within a single organization. Finally, with the advent of cloud computing, many applications run on a cloud-based infrastructure that provides virtualized and distributed computing resources shared among many users and typically not under direct control by the owners of the applications.

Therefore, we say that the modern software systems are embedded in an *open world* [9], characterized by continuous change in the environment in which they are situated and in the requirements they have to meet. The dynamic behavior of such systems makes traditional design-time verification approaches unfeasible, because they cannot analyze all the behaviors that can emerge at run time. For this reason, techniques like *run-time verification*[1] [90] and *trace checking*[2] [13] have become promising alternatives. While run-time verification checks the behavior of a system *during* its execution, trace checking is a *post-mortem* technique. In other words, to perform trace checking one must first collect and store relevant execution data (called *execution traces* or *logs*) produced by the system and then check them *offline* against the system specifications. This activity is often done to inspect server logs, crash reports, and test traces, in order to analyze problems encountered at run time. More precisely, trace checking is an automatic procedure for evaluating a formal specification over a trace of

[1] Also called run-time monitoring [50] or policy monitoring [19]

[2] Also called *trace validation* [97] or *history checking* [64].

recorded events produced by a system after execution. The output of the procedure is called *verdict* and states whether the system's behavior conforms to its specification.

Although runtime verification detects a violation of a property immediately during the execution, it also introduces unnecessary computational overhead and may affect the system it checks. The difficulty of the problem of checking if properties of a system hold during its execution directly depends on the complexity of the properties being checked and, in turn, on the expressiveness of the specification language being used. Research in this domain [67,74] classifies runtime verification algorithms as algorithms that must analyze the events in a trace in the order in which they occurred and keep in memory only some finite amount of information about the trace seen so far. Trace checking algorithms, on the other hand, can analyze a trace in an arbitrary order and typically have linear space complexity, i.e., in the worst case they need to consider the entire trace, seen so far, during the evaluation. Trace checking is therefore an *offline* procedure that does not introduce any overhead in the execution of the system, other than collecting logs. More importantly, trace checking can support more expressive specification languages than runtime verification [63].

In this thesis the main focus will be on the specific class of properties we call *quantitative* properties. In the last decade, the research efforts in the area of software verification have mainly focused on verifying qualitative properties of systems (e.g., safety or liveness). However, many important software characteristics can be quantitative, such as the ones related to non-functional requirements like response-time, throughput, availability or some domain specific functional properties. For example, the following properties express bounds on quantitative values derived from a system execution:

P1: "A service always needs to respond to any request within 30 milliseconds." (response time)

P2: "The average response time of a service must not exceed 30 milliseconds, within any 10 minute time window." (average response time)

P3: "A client is allowed to submit a maximum of 3 service requests each hour." (throughput)

P4: "Never allocate more than 3 machines within 2 minute time window." (resource thrashing)

Informally, we can define quantitative properties as any *constraints* on quantifiable values from a system execution [86]. Typically, these quanti-

ties are related to timing, like in P1. The constraints may be defined only over a subset of the time domain, like in P2, we refer to quantities calculated for each 10 minute time window. Properties can refer to an array of quantities, for example timing relations between pairs events (P2) or multiplicities of some event (P3 and P4). In the properties that refer to multiple quantities, we can use aggregations. For example, in P2 we use average, while in P3 and P4 we use maximum.

At the moment there is no consolidated research into verification of quantitative properties of open-world software, hence this thesis addresses this issue by proposing different trace checking techniques.

1.2 Problem Statement and Research Goals

In [9] the authors state what are the research challenges for open-world software. The authors identify the need for more consolidated approaches for *specification, verification, monitoring, trust, implementation,* and *self-management* of open-world software. Some of the issues have been addressed in recent years, namely in [32] the author deals with specification, verification and reputation management in the context of service-based applications. In [65] the author addresses the issues of verification and self-management, while in [113] the author proposes specification and verification techniques for the quality of service requirements of open-world software. In [94] the author deals with the verification of open-world software in the context of incomplete models.

This thesis addresses the issue of verification of open-world software in the context of quantitative properties with the goal of providing practical and scalable approaches based on trace checking. This can be precisely stated with the following overall research goal:

> *"To study quantitative properties of systems occurring in practice and provide a practical and scalable approach to verification, driven by the selected specification language suitable to express such properties."*

As stated above this research is driven by quantitative properties encountered in practice [29,35]. In order to formally specify them, we have chosen a specification language, called SOLOIST [36], as a baseline language. Although SOLOIST is designed to express properties of service interactions, it is based on metric temporal logic (MTL) [83] and it is able to specify both functional and non-functional properties of systems. Moreover, since the original version of the language from [36] is undecidable due to the

first-order quantification, this thesis focuses on a propositional fragment of SOLOIST.

Given the choice of a specification language, the overall research goal can be decomposed into two research sub-goals:

Research goal 1 - Decision procedure for SOLOIST

To demonstrate that significant verification activities are possible, the chosen specification language must be decidable. A language is decidable if there exists an algorithm that always terminates and given any formula of the language it can find a satisfying assignment if and only if the formula is satisfiable. To provide a practical and scalable approach to verification, one needs to develop not only a proof of the decidability of the language, but also a decision procedure which is efficient and easily implementable in existing verification engines (such as SAT or SMT solvers). For the propositional fragment of SOLOIST decidability has been shown in [36] via reduction to LTL. However the reduction did not provide an efficient and practical decision procedure. Showing the existence of a practically relevant decision procedure for SOLOIST is expressed as the following research goal:

> *"To study the SOLOIST specification language and develop a decision procedure that provides a general framework to perform different verification use cases using SOLOIST, including trace checking, and which is amenable to rapid development of prototype tools."*

Research goal 2 - Scalable trace checking of SOLOIST

The problem of checking a logged event trace against a temporal logic specification arises in many practical cases. Unfortunately, known algorithms for an expressive logic like MTL do not scale with respect to two crucial dimensions: the length of the trace and the size of the time interval of the formula to be checked. These issues are formulated as the second research goal:

> *"To develop a scalable, efficient and practical trace checking algorithm for SOLOIST."*

1.3 Contributions

In this section we outline the contributions of the thesis, mapping them to the research goals stated above.

Contribution 1 - Decision procedure for SOLOIST

We have implemented two efficient decision procedures for SOLOIST that make use of state-of-the-art SMT solver. The implementation is a translation that reduces the problem of SOLOIST satisfiability to satisfiability of a particular logic supported by the SMT solver theories. The major difference between the two implemented procedures is the logic targeted by the translation and how it encodes the satisfying assignment of SOLOIST formulae. The two procedures are described in Chapters 3 and 4, respectively. An efficient decision procedure provides a general framework for building a SOLOIST verification suite that supports many verification use cases. An instance of such a use case is trace checking; in Chapter 5, we exploit the implemented decision procedures for SOLOIST to perform trace checking and show how the two decision procedures can be used complementarily.

Contribution 2 - Scalable trace checking of SOLOIST

The problem of the algorithms for trace checking logics based on MTL is that they do not scale with respect to two crucial dimensions: the length of the trace and the size of the time interval of the formula to be checked. We address the former issue in Chapter 8 by proposing a distributed and parallel trace checking algorithm that can take advantage of modern cloud computing and programming frameworks like MapReduce and Spark. We address the latter issue in Chapter 9 by proposing an alternative semantics for MTL, called *lazy* semantics. Lazy semantics possesses certain properties that allow us to decompose any MTL formula into an equivalent MTL formula with all time intervals of its temporal operators limited by some constant. This decomposition plays a major role in the context of (distributed) trace checking of formulae with large time intervals.

Contribution 3 - Specifying quantitative properties

This contribution does not map to neither of the research goals, but it is rather a side effect of the previous contributions. In the process of specifying quantitative properties we have encountered many complex cases where SOLOIST is not expressive enough. Therefore, in Chapter 6 we extend

SOLOIST with arithmetical constrains (SOLOIST$^{\mathcal{A}}$) that allow us to express complex quantitative properties of cloud-based elastic systems, like *elasticity* or *resource thrashing*. Another contribution towards specifying quantitative properties is *lazy* semantics. Besides allowing us to optimize our distributed trace checking algorithm, we believe that lazy semantics makes the process of specifying system properties using MTL more intuitive.

1.4 Dissemination

The research I have conducted during the PhD program has lead to several publications. This section lists them in order in which they are presented in the thesis. There are also several unpublished reports listed in this section that are related to the thesis.

Conference and workshop papers

- Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Offline trace checking of quantitative properties of service-based applications. *In Proceedings of the 7h International Conference on Service Oriented Computing and Application (SOCA 2014)*, IEEE, 2014.

  This paper is the basis of Chapter 3 where we introduce SOLOIST decision procedure based on CLTLB($\mathcal{D}$). In Chapter 5 we report on the evaluation of the trace checking algorithm tailored for dense traces that is based on the decision procedure.

- Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. *In Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014)*, volume 8411 of LNCS, pages 276-290. Springer, 2014

  This paper is the basis of Chapter 4 where we introduce SOLOIST decision procedure based on QF-EUFIDL. In Chapter 5 we also report on the evaluation of the trace checking algorithm and some of its applications.

- Marcello Maria Bersani, Domenico Bianculli, Schahram Dustdar, Alessio Gambi, Carlo Ghezzi, and Srđan Krstić. Towards the formalization of properties of cloud-based elastic systems. *In Proceedings of the 6th*

International Workshop on Principles of Engineering Service-oriented Systems (PESOS 2014), co-located with ICSE 2014, ACM, 2014

This paper reports on applying our trace checking algorithms to check properties of Cloud-based Elastic systems. This is detailed in Chapter 6 where we introduce property specification patterns that we identified for Cloud-based Elastic systems and we report on the effort to check the properties of *Elastic Doodle* service.

- Domenico Bianculli, Carlo Ghezzi, and Srđan Krstić. Trace checking of Metric Temporal Logic with Aggregating Modalities using MapReduce. *In Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM 2014)*, Springer, 2014.

  This paper is the basis of Chapter 8 where we introduce a scalable distributed trace checking algorithm for SOLOIST based on the MapReduce paradigm.

- Srđan Krstić. Quantitative Properties of Software Systems: Specification, Verification, and Synthesis. *In Proceedings of the 36th International Conference on Software Engineering, (ICSE 2014), Doctoral Symposium*, ACM, 2014

  This paper motivated the need for verification of quantitative properties of open-world software and describes my long-term research agenda.

- Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using MapReduce. *In Proceedings of the 38th International Conference on Software Engineering (ICSE 2016),* ACM, 2016.

  This paper is the basis of Chapter 9 where we introduce optimizations for the trace checking algorithm introduced in Chapter 8. It proposes an alternative semantics for MTL that enjoys certain properties useful to perform decomposition of MTL formulae. The decomposition can be tunes and it is very useful for a distributed algorithm.

Unpublished reports

- Giovanni Paolo Gibilisco and Srđan Krstić. InstaCluster: Building A Big Data Cluster in Minutes. *Technical Report, arXiv:1508.04973*

  This paper is the basis of the evaluation setup described in Chapter 10 where we present a tool that we developed to provision BigData enabled clusters on the Amazon public cloud infrastructure in order to use it for testing our algorithm.

1.5 Structure of the Thesis

This thesis is structured in four parts. Part I overviews the problem statement and contributions of the thesis in Chapter 1, while in Chapter 2 it provides some background information on notations, formal logic and programming models used in the thesis. Part II has five chapters: Chapter 3 introduces the decision procedure for SOLOIST based on CLTLB($\mathcal{D}$), while Chapter 4 introduces the decision procedure for SOLOIST based on QF-EUFIDL. Chapter 5 shows how the two decision procedures can be used complementary for trace checking of SOLOIST specifications and presents the evaluation of their performance. Chapter 6 shows our comprehensive case study in specification and verification of properties of cloud-based elastic systems using SOLOIST and its decision procedures. Chapter 7 surveys related work in the area of verification of quantitative properties and describes other languages that can be used to specify quantitative properties. Part III has three chapters: Chapter 8 presents our distributed and parallel trace checking algorithm based on MapReduce and Spark. Chapter 9 describes a memory optimization for the algorithm that relies on new alternative semantics for MTL, called *lazy* semantics. Chapter 10 presents a comprehensive evaluation of the algorithm. Chapter 11 surveys related work in the area of distributed trace checking and describes other attempts at providing alternative semantics for MTL. Finally, Chapter 12 in Part IV provides some concluding remarks, discusses limitations of the proposed approaches and how these limitations can be addressed as part of future work.

CHAPTER 2

Preliminaries

This chapter provides some preliminary information on notations, formal logic and programming models used in the remaining of the thesis. Section 2.1 introduces two standard temporal logics: *metric temporal logic* (MTL) and *constrained linear temporal logic* (CLTLB($\mathcal{D}$)), that are able to express real-time properties of systems. In Section 2.2 we introduce SOLOIST temporal logic designed to capture properties of service composition interactions. The novelty of SOLOIST is its ability to formalize aggregated behaviors of systems. Section 2.3 introduces the concept of *bounded satisfiability checking* (BSC) and a tool, called ZOT that supports BSC and implements the decision procedure for CLTLB($\mathcal{D}$). We use CLTLB($\mathcal{D}$) and ZOT to introduce the first decision procedure for SOLOIST in Chapter 3. BSC is the core principle used to perform trace checking by exploiting a decision procedure of a language. Section 2.4 introduces the satisfiability modulo theories (SMT) solvers and two particular theories: *theory of integer difference logic* (IDL) and *theory of equality and uninterpreted functions* (EUF). These concepts are used to introduce the second decision procedure for SOLOIST in Chapter 4. Finally, in Section 2.5 we introduce MapReduce and Spark, the distributed programming models used to implement distributed and parallel trace checking algorithms.

2.1 Temporal Logic for Specifying System Properties

Problem specifications are essential for designing, validating, documenting, communicating, reengineering, and reusing solutions. Formality helps in obtaining higher-quality specifications within such processes; it also provides the basis for their automated support. Temporal logic is a convenient formalism for formally specifying properties of systems. Informally, a model of a temporal logic formula is an infinite sequence of events that satisfy it. Each temporal logic formula defines a set of such sequences. Such a set of sequences of events in turn represents the admissible system behavior. A given system satisfies a temporal logic formula if all of its computations (i.e., sequences of events) belong to the set defined by the formula. The most well known temporal logic is linear temporal logic (LTL) and its equally expressive variant propositional linear temporal logic with both past and future modalities (PLTLB) [91]. LTL can express the order among the events in a sequence, but cannot enforce the timing relations between them. This lead to the development of real-time temporal logics as extensions of LTL, most notably metric temporal logic (MTL).

2.1.1 Metric Temporal Logic (MTL)

Let I be any non-empty interval over $\mathbb{N}$ and let Π be a finite set of atomic propositions. The syntax of MTL is defined by the following grammar, where $p \in \Pi$ and U_I is the metric "*Until*" operator:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi \mathsf{U}_I \phi \mid \phi \mathsf{S}_I \phi$$

Informally, MTL extends the well known "*Until*" temporal operator of the classical LTL with an interval that indicates the time distance within which its right-hand side subformula must hold. For example, property *"It is always true when a student accesses a homework assignment, he/she can provide or modify the answer within a week before a professor revokes the access."* is expressed as:

$$\mathsf{G}(\mathsf{access} \rightarrow (\mathsf{can\_write} \vee \mathsf{can\_modify}) \mathsf{U}_{(0,168]} \mathsf{revoke}) \tag{2.1}$$

where temporal operator $\mathsf{U}_{(0,168]}$ states that its right operand, the revoke predicate, must occur within a week (or 168 hours) from the moment of access (expressed by the access predicate). It also states that the left operand must be continuously true until that happens. Operator G (called "*Globally*") simply states that property always holds. The operator "*Globally*" and the other boolean and temporal operators can be derived using the

usual conventions: "*Eventually*" is defined as $\mathsf{F}_I\phi \equiv \top\mathsf{U}_I\phi$; "*Globally*" is defined as $\mathsf{G}_I\phi \equiv \neg\mathsf{F}_I\neg\phi$; "*Eventually in the Past*" or just "*Past*" is defined as $\mathsf{P}_I\phi \equiv \top\mathsf{S}_I\phi$; "*Globally in the Past*" or "*Historically*" is defined as $\mathsf{H}_I\phi \equiv \neg\mathsf{P}_I\neg\phi$, where $\top$ means "true". We adopt the convention that an interval of the form $[i, i]$ is written as "$= i$". The interval $[0, +\infty)$ in temporal operators is omitted for conciseness. We introduce the following shorthand notation: $\mathsf{F}^K(\phi) \equiv \underbrace{\mathsf{F}\mathsf{F}\ldots\mathsf{F}}_{K \text{ times}}(\phi)$, with $\mathsf{F}^0(\phi) = \phi$. Traditional semantics for MTL is called point-based semantics and we will denote it as MTL_P semantics.

MTL_P semantics. We focus on the finite-word semantics of MTL. A *timed sequence* $\tau = \tau_0\tau_1\ldots\tau_{|\tau|-1}$ is a sequence of values $\tau_i \in \mathbb{R}$ with $\tau_i > 0$, such that, $\tau_i < \tau_{i+1}$ for each $0 \leq i < |\tau|$, i.e., the sequence is *strictly monotonic*. A *word* σ over the alphabet 2^Π is a sequence $\sigma_0\sigma_1\ldots\sigma_{|\sigma|-1}$ such that $\sigma_i \in 2^\Pi$ for all $0 \leq i < |\sigma|$, where $|\sigma|$ denotes the length of the word. A *timed word* [5] $\omega = \omega_0\omega_1\ldots\omega_{|\omega|-1}$ is a word over $2^\Pi \times \mathbb{R}$, i.e., a sequence of pairs $\omega_i = (\sigma_i, \tau_i)$ where $\sigma_0\ldots\sigma_{|\omega|-1}$ is a word over 2^Π and $\tau_0\ldots\tau_{|\omega|-1}$ is a timed sequence. Note that in this definition i refers to a particular *position* in the timed word ω, while τ_i refers to the *time instant* at the position i. We abuse the notation and represent a timed word equivalently as a pair containing a word and a timed sequence of the same length, i.e., $\omega = (\sigma, \tau)$. A *timed language* over 2^Π is a set of timed words over 2^Π. MTL_P semantics on timed words is given below in Formula (2.2), where the point-based satisfaction relation $\models_P$ is defined with respect to a timed word (σ, τ), a position $i \in \mathbb{N}$, and MTL formulae ϕ and ψ.

$$\begin{aligned}
(\sigma,\tau,i) &\models_P p \text{ iff } p \in \sigma_i \text{ for } p \in \Pi \\
(\sigma,\tau,i) &\models_P \neg\phi \text{ iff } (\sigma,\tau,i) \not\models_P \phi \\
(\sigma,\tau,i) &\models_P \phi \vee \psi \text{ iff } (\sigma,\tau,i) \models_P \phi \text{ or } (\sigma,\tau,i) \models_P \psi \\
(\sigma,\tau,i) &\models_P \phi\mathsf{U}_I\psi \text{ iff } \exists j.(i \leq j < |\sigma| \text{ and } \tau_j - \tau_i \in I \text{ and} \\
(\sigma,\tau,j) &\models_P \psi \text{ and } \forall k.(i < k < j \text{ then } (\sigma,\tau,k) \models_P \phi)
\end{aligned} \tag{2.2}$$

Note that, due to the strictly monotonic definition of the timed sequence τ, the metric "*Next*" and "*Yesterday*" operators can be defined as $\mathsf{X}_I\phi \equiv \bot\mathsf{U}_{I-\{0\}}\phi$ and $\mathsf{Y}_I\phi \equiv \bot\mathsf{S}_{I-\{0\}}\phi$ respectively, where $\bot$ means "false". $L_p(\phi)$ is a timed language defined by a formula ϕ when interpreted according to the MTL_P semantics, i.e., $L_p(\phi) = \{(\sigma,\tau) \mid (\sigma,\tau,0) \models_P \phi\}$

2.1.2 Constrained Linear Temporal Logic (CLTLB($\mathcal{D}$))

CLTLB($\mathcal{D}$) [26] is an extension of PLTLB (Propositional Linear Temporal Logic with both past and future modalities) [91] augmented with atomic formulae built over a constraint system $\mathcal{D}$. In practice, CLTLB($\mathcal{D}$) defines a set of *variables* C and *arithmetical constraints* over a constraint system $\mathcal{D}$; in our case, $\mathcal{D}$ is the structure $(\mathbb{Z}, =, +)$. For this particular combination, decidability of CLTLB($\mathcal{D}$) has been proven in [54]. Variables (also called *counters*) receive a separate evaluation at each time instant. In addition to the standard PLTLB temporal operators "*Since*" and "*Until*", CLTLB($\mathcal{D}$) introduces the new construct of *arithmetic temporal term*, defined as

$$\alpha := c \mid x \mid \alpha + c \mid \mathsf{Y}(\alpha) \mid \mathsf{X}(\alpha)$$

where $c \in \mathbb{Z}$ is a constant, $x \in C$ is a counter and Y and X are temporal operators applied to counters. These temporal operators for counters return the value of the counter in the previous and in the next time instant, respectively. Note that we use a syntactically sugared version of PLTLB using metric temporal operators over time intervals, such as U_I. Since time is discrete, they are just a convenient shorthand [104]. The syntax of CLTLB($\mathcal{D}$) is the following:

$$\phi ::= p \mid \alpha = \alpha \mid \alpha < \alpha \neg\phi \mid \phi \wedge \phi \mid \phi \mathsf{U}_I \phi \mid \phi \mathsf{S}_I \phi$$

where p is an atomic proposition, S_I, U_I are the usual "*Since*", and "*Until*" modalities of PLTLB. Additional temporal modalities (like G, "Globally", and W, "Weak Until") can be defined using the usual conventions. An example of a CLTLB($\mathcal{D}$) formula is $\mathsf{G}(\phi \rightarrow \mathsf{X}(y) = y + 1)$, which states that whenever ϕ is true, the value of counter y in the next time instant must be incremented of 1 with respect to the value at the current time instant.

CLTLB($\mathcal{D}$) formulae admit finite, ultimately periodic two-part models (π,δ). Function $\pi : \mathbb{N} \rightarrow \mathcal{P}(\Pi)$ associates a subset of the propositions with each time instant, while function $\delta : \mathbb{N} \times C \rightarrow \mathbb{Z}$ defines the value of counters at each time position. Hereafter, this two-part model will be graphically represented as in Fig. 2.1: the topmost row (above the timeline)

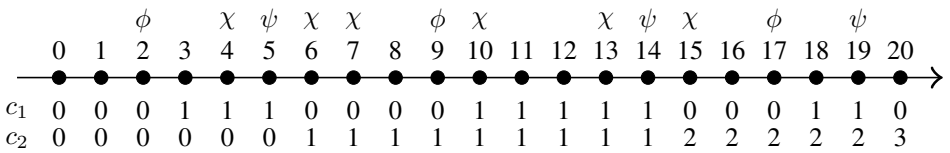


Figure 2.1: *CLTLB($\mathcal{D}$) two-part model example*

represents function π (e.g., $\pi(5) = \{\psi\}$ or $\pi(11) = \{\}$); the rows of integers below the timeline represent function δ, i.e., the values of each counter defined in the model. In the example in the figure there are two counters, as shown on the left: c_1 and c_2; the δ function is defined so that we have, for example in correspondence with the sixth time instant (position #5), $\delta(5, c_1) = 1$ and $\delta(5, c_2) = 0$.

2.2 SOLOIST

SOLOIST acronym stands for *SpecificatiOn Language fOr servIce compoSitions inTeractions*, introduced in [36]. SOLOIST was designed with the goal of supporting the common specification patterns found for service provisioning; It is a propositional metric temporal logic with additional aggregating modalities. These modalities have been defined based on an extensive field study [35] of the requirements specifications in the context of service-based applications, and they are tailored to express the most common requirements occurring in practice. The study — performed in collaboration with an industrial partner — analyzed more than 900 requirements specifications, extracted both from research papers and industrial data, and led to the identification of a new class of specification patterns, specific to the domain of service provisioning (in addition to the well-known ones like those defined in [60, 82]). The service provisioning patterns refer to: S1) average response time; S2) counting the number of events; S3) average number of events; S4) maximum number of events; S5) absolute time; S6) unbounded elapsed time; S7) data-awareness.

The syntax of SOLOIST is defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathsf{U}_I \phi \mid \phi \mathsf{S}_I \phi \mid \mathfrak{C}^K_{\bowtie n}(\phi) \mid \mathfrak{U}^{K,h}_{\bowtie n}(\phi)$$

$$\mid \mathfrak{M}^{K,h}_{\bowtie n}(\phi) \mid \mathfrak{D}^K_{\bowtie n}(\phi, \phi)$$

where $p \in \Pi$, with Π being a finite set of atoms; I is a nonempty interval over $\mathbb{N}$; n, K, h range over $\mathbb{N}$; $\bowtie \in \{<, \leq, \geq, >, =\}$. The arguments ϕ of modalities $\mathfrak{C}, \mathfrak{U}, \mathfrak{M}, \mathfrak{D}$ are restricted to atoms in Π. Moreover, the two arguments in the $\mathfrak{D}$ modality are required to be different atoms.

The U_I and S_I modalities have the usual meaning in temporal logic ("*Until*" and its past counterpart "*Since*"), but they are bound to interval I; additional temporal modalities (like G, "*Globally*") can be defined from them using the usual conventions. The remaining modalities are called *aggregate* modalities and are used to express the specification patterns S1–S4 mentioned above. The $\mathfrak{C}^K_{\bowtie n}(\phi)$ modality states a bound (represented by

$$
\begin{array}{lll}
((\sigma,\tau),i) \models p & \text{iff} & p \in \sigma_i \\
((\sigma,\tau),i) \models \neg\phi & \text{iff} & ((\sigma,\tau),i) \not\models \phi \\
((\sigma,\tau),i) \models \phi \wedge \psi & \text{iff} & ((\sigma,\tau),i) \models \phi \wedge ((\sigma,\tau),i) \models \psi \\
((\sigma,\tau),i) \models \phi \mathsf{S}_I \psi & \text{iff} & \text{for some } j < i, \tau_i - \tau_j \in I, ((\sigma,\tau),j) \models \psi \\
 & & \text{and for all } k, j < k < i, ((\sigma,\tau),k) \models \phi \\
((\sigma,\tau),i) \models \phi \mathsf{U}_I \psi & \text{iff} & \text{for some } j > i, \tau_j - \tau_i \in I, ((\sigma,\tau),j) \models \psi \\
 & & \text{and for all } k, i < k < j, ((\sigma,\tau),k) \models \phi \\
((\sigma,\tau),i) \models \mathfrak{C}^K_{\bowtie n}(\phi) & \text{iff} & c(\tau_i - K, \tau_i, \phi) \bowtie n \text{ and } \tau_i \geq K \\
((\sigma,\tau),i) \models \mathfrak{U}^{K,h}_{\bowtie n}(\phi) & \text{iff} & \frac{c(\tau_i - \lfloor \frac{K}{h} \rfloor h, \tau_i, \phi)}{\lfloor \frac{K}{h} \rfloor} \bowtie n \text{ and } \tau_i \geq K \\
((\sigma,\tau),i) \models \mathfrak{M}^{K,h}_{\bowtie n}(\phi) & \text{iff} & \max\left\{ \bigcup_{m=0}^{\lfloor \frac{K}{h} \rfloor} \{c(lb(m), rb(m), \phi)\} \right\} \bowtie n \text{ and } \tau_i \geq K \\
((\sigma,\tau),i) \models \mathfrak{D}^K_{\bowtie n}(\phi,\psi) & \text{iff} & \frac{\sum_{(s,t)\in d(\phi,\psi,\tau_i,K)} (\tau_t - \tau_s)}{|d(\phi,\psi,\tau_i,K)|} \bowtie n \text{ and } \tau_i \geq K \text{ and } d(\phi,\psi,\tau_i,K) \neq \emptyset
\end{array}
$$

where $c(\tau_a, \tau_b, \phi) = |\{s \mid \tau_a < \tau_s \leq \tau_b \text{ and } ((\sigma,\tau),s) \models \phi\}|$, $lb(m) = \max\{\tau_i - K, \tau_i - (m+1)h\}$, $rb(m) = \tau_i - mh$, and $d(\phi,\psi,\tau_i,K) = \{(s,t) \mid \tau_i - K < \tau_s \leq \tau_i \text{ and } ((\sigma,\tau),s) \models \phi,$ $t = \min\{u \mid \tau_s < \tau_u \leq \tau_i, ((\sigma,\tau),u) \models \psi\}\}$

Figure 2.2: *Formal semantics of SOLOIST*

$\bowtie n$) on the number of occurrences of an event ϕ in the previous K time instants: it expresses pattern S2. The $\mathfrak{U}^{K,h}_{\bowtie n}(\phi)$ (respectively, $\mathfrak{M}^{K,h}_{\bowtie n}(\phi)$) modality expresses a bound on the average (respectively, maximum) number of occurrences of an event ϕ, aggregated over the set of right-aligned adjacent non-overlapping subintervals within a time window K; it corresponds to pattern S3 (respectively, S4), as in "the average/maximum number of events per hour in the last ten hours". A subtle difference in the semantics of the $\mathfrak{U}$ and $\mathfrak{M}$ modalities is that $\mathfrak{M}$ considers events in the (possibly empty) tail interval, i.e., the leftmost observation subinterval whose length is less than h, while the $\mathfrak{U}$ modality ignores them. The $\mathfrak{D}^K_{\bowtie n}(\phi,\psi)$ modality expresses a bound on the average time elapsed between a pair of specific adjacent events ϕ and ψ occurring in the previous K time instants; it can be used to express pattern S1. A more in-depth discussion on the SOLOIST aggregate modalities and their comparison to similar existing modalities can be found in [36].

The formal semantics of SOLOIST is defined on timed words (σ,τ) over $2^\Pi \times \mathbb{N}$. Figure 2.2 defines the satisfiability relation $((\sigma,\tau),i) \models \phi$ for every timed word (σ,τ), every position $i \geq 0$ and for every SOLOIST formula ϕ.

We remark that the version of SOLOIST presented here is a restriction of the original one in [36]. To simplify the presentation in the next sections, we dropped first-order quantification on finite domains (which was introduced to support data-awareness, i.e., pattern S7) and limited the argument of the

$\mathfrak{D}$ modality to only one pair of events. These restrictions are only syntactic sugar and we refer to [36] for the details of the transformations that provide support for them.

Additional notation

This section briefly introduces some concepts used throughout the thesis. The concepts apply to SOLOIST as well as its fragment — MTL. Let ϕ and ψ be SOLOIST (or MTL) formulae. The set of all proper subformulae of ϕ is denoted with $\mathsf{sub}(\phi)$; notice that for atoms $p \in \Pi$, $\mathsf{sub}(p) = \emptyset$. The *length* of a formula ϕ, denoted $|\phi|$, is defined as the number of its non-proper subformulae, i.e., $|\phi| = |\mathsf{sub}(\phi)|+1$. The *size* of (the encoding of) a formula ϕ, denoted as $||\phi||$ is equal to $|\phi| \cdot \mathsf{log}(\mu)$, where μ is the largest constant occurring the formula. The set $\mathsf{sub}_a(\phi) = \{p \mid p \in \mathsf{sub}(\phi),\ \mathsf{sub}(p) = \emptyset\}$ is the set of atoms of formula ϕ. The set $\mathsf{sub}_d(\phi) = \{\alpha \mid \alpha \in \mathsf{sub}(\phi), \forall \beta \in \mathsf{sub}(\phi), \alpha \notin \mathsf{sub}(\beta)\}$ is called the set of all *direct subformulae* of ϕ; ϕ is called the *superformula* of all formulae in $\mathsf{sub}_d(\phi)$. The set $\mathsf{sup}_\psi(\phi) = \{\alpha \mid \alpha \in \mathsf{sub}(\psi), \phi \in \mathsf{sub}_d(\alpha)\}$ is the set of all subformulae of ψ that have formula ϕ as *direct subformula*. The *height* $h(\phi)$ of ϕ is defined recursively as:

$$h(\phi) = \begin{cases} \mathsf{max}\{h(\psi) \mid \psi \in \mathsf{sub}_d(\phi)\} + 1 & \text{if } \phi \notin \Pi \\ 1 & \text{otherwise.} \end{cases}$$

For example, given the formula $\gamma = \mathsf{F}_{[2,4]}(a \wedge b)\mathsf{U}_{(30,100)}\neg c$, we have: $\mathsf{sub}(\gamma) = \{a, b, c, a \wedge b, \neg c, \mathsf{F}_{[2,4]}(a \wedge b)\}$ is the set of all proper subformulae of γ; $\mathsf{sub}_a(\gamma) = \{a, b, c\}$ is the set of atoms in γ; $\mathsf{sub}_d(\gamma) = \{\mathsf{F}_{[2,4]}(a \wedge b), \neg c\}$ is the set of direct subformulae of γ; $\mathsf{sup}_\gamma(a) = \mathsf{sup}_\gamma(b) = \{a \wedge b\}$ shows that the sets of superformulae of a and b in γ coincide; and the height of γ is 4, since $h(a) = h(b) = h(c) = 1$, $h(\neg c) = h(a \wedge b) = 2$, $h(\mathsf{F}_{[2,4]}(a \wedge b)) = 3$ and therefore $h(\gamma) = \mathsf{max}\{h(\mathsf{F}_{[2,4]}(a \wedge b)), h(\neg c)\}+1 = 4$.

2.2.1 Formalizing BPEL process interactions

We consider service compositions defined in terms of the BPEL [10] orchestration language. Very briefly, BPEL is a high-level XML-based language for the definition and execution of business processes, defined as workflows that compose external partner services. The definition of a workflow contains a set of variables; the business logic is expressed as a composition of activities. The main types of activities are primitives for communicating with other services (*receive, invoke, reply, pick*) and for executing

assignments (*assign*) to variables, as well as control-flow structures (*sequence, while, switch and parallel flows*).

Below we list some examples of properties expressed in natural language, which can be used to specify the interactions of a BPEL process. We assume that the process has invoke activities named *invA* and *invB*, three receive activities named *recvP*, *recvQ*, and *recvR* and a reply activity term that takes no parameters. The detailed workflow structure of the process as well as the other variables are of no interest for the purpose of this section and are omitted for clarity. All properties are under the scope of an implicit universal temporal quantification as in "In every process run, . . .".

P1: "The execution of activity *recvP* should alternate with the execution of activity *recvQ*, though other activities different from *recvQ* (respectively, *recvP*) can be executed in between."

P2: "The response time of activity *invB* should not exceed 4 time units."

P3: "If activity *invB* has been invoked 4 times in the past 16 units, than activity *recvR* will be executed within 32 time units."

P4: "When activity term is executed, the average response time of all the invocations of activity *invB* completed in the past 720 time units should be less than 3 time units."

P5: "When activity term is executed, the average number of invocations, in an interval of 60 time units, of activity *invB* during the past 720 time units should be less than 4".

P6: "When activity term is executed, the maximum number of invocations, in an interval of 60 time units, of activity *invB* during the past 720 time units should be less than 5".

To specify the properties above in SOLOIST we first need to map the activities of the BPEL process into its atomic propositions. Let $\mathcal{A}$ be the set of activities defined in a BPEL process; $\mathcal{A} = \mathcal{A}_{start-inv} \cup \mathcal{A}_{end-inv} \cup \mathcal{A}_{recv} \cup \mathcal{A}_{pick} \cup \mathcal{A}_{reply} \cup \mathcal{A}_{hdlr} \cup \mathcal{A}_{other}$ where:

- $\mathcal{A}_{start-inv}$ ($\mathcal{A}_{end-inv}$) is the set of start (end) events of all invoke activities[1]
- $\mathcal{A}_{recv}$ is the set of all receive activities;
- $\mathcal{A}_{pick}$ is the set of all pick activities;

[1] A synchronous invoke is characterized both by a start event and by an end event; an asynchronous invoke is characterized only by a start event.

- $\mathcal{A}_{reply}$ is the set of all reply activities;
- $\mathcal{A}_{hdlr}$ is the set of events associated with all kinds of handlers;
- $\mathcal{A}_{other}$ is the set of activities that are not an invoke, a receive, a pick, a reply, or related to a handler (e.g., an assign, a control structure activity).

A set of atomic propositions of SOLOIST $\Pi = \mathcal{A}$ corresponds to the set of all process activities.

Now we can use SOLOIST to formalize the properties above:

P1: $\mathsf{G}((\mathsf{recvP} \rightarrow \neg \mathsf{recvP}\mathsf{U}_{(0,\infty)}\mathsf{recvQ}) \wedge (\mathsf{recvQ} \rightarrow \neg \mathsf{recvQ}\mathsf{U}_{(0,\infty)}\mathsf{recvP}))$

P2: $\mathsf{G}(\mathsf{invB}_{\mathsf{start}} \rightarrow \mathsf{F}_{[0,4]}\mathsf{invB}_{\mathsf{end}})$

P3: $\mathsf{G}(\mathfrak{C}^{16}_{=14}\mathsf{invB}_{\mathsf{start}} \rightarrow \mathsf{F}_{0,32}\mathsf{recvR})$

P4: $\mathsf{G}(\mathsf{term}_{\mathsf{end}} \rightarrow \mathfrak{D}^{720}_{<3}(\mathsf{invB}_{\mathsf{start}}, \mathsf{invB}_{\mathsf{end}}))$

P5: $\mathsf{G}(\mathsf{term}_{\mathsf{end}} \rightarrow \mathfrak{U}^{720,60}_{\leq 4}(\mathsf{invB}_{\mathsf{start}}))$

P6: $\mathsf{G}(\mathsf{term}_{\mathsf{end}} \rightarrow \mathfrak{M}^{720,60}_{\leq 5}(\mathsf{invB}_{\mathsf{start}}))$

2.2.2 Translating SOLOIST into LTL

In this section we sketch the translation of SOLOIST into a linear temporal logic, to show that the two languages are equivalent. The translation summarized here encodes SOLOIST into PLTLB. This is done in two steps for temporal operators (like U_I and S_I); they are first translated into a variant of linear temporal logic called MPLTLB (Metric Linear Temporal Logic with Past) [104]. MPLTLB is a syntactically-sugared version of classical PLTLB [80], defined over a mono-infinite discrete model of time represented by words. The second step is their translation from MPLTLB to

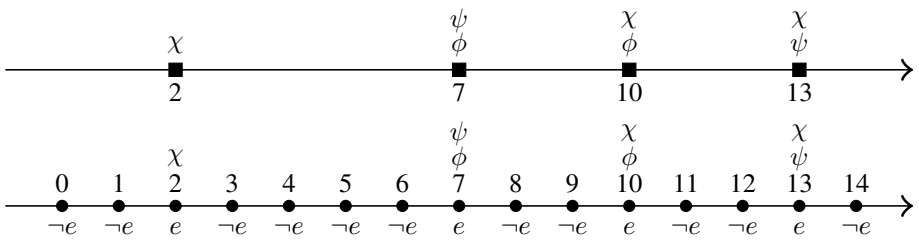


Figure 2.3: *Conversion from timed word to word*

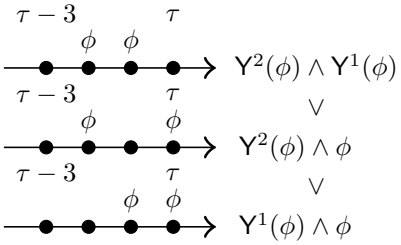


Figure 2.4: *Basic translation of the formula* $\mathfrak{C}^3_{>1}(\phi)$

PLTLB which is a standard translation [68]. The rest of the modalities are translated directly to PLTLB.

Without loss of expressiveness, we consider only SOLOIST formulae in positive normal form, i.e., where negation may only occur on atoms (see, for example, [104]). Then, we need to bridge the gap between the semantics based on timed words, where the temporal information is denoted by a natural time-stamp, and the one used for MPLTLB, where the temporal information is implicitly defined by the integer position in an word. The two temporal models can be transformed into each other. Here we are interested in pinpointing in an MPLTLB word the positions that correspond to time-stamps in a SOLOIST timed word where events occurred. To do so, we add to the set Π a special propositional symbol e, which is true in each position corresponding to a "valid" time-stamp in the timed word. An example of this conversion is shown in Fig. 2.3, where a timed word is depicted in the timeline at the top and the equivalent word corresponds to the timeline at the bottom. Hereafter, when displaying words, we will omit the symbol e from positions in the timeline, since its presence can be implied by the

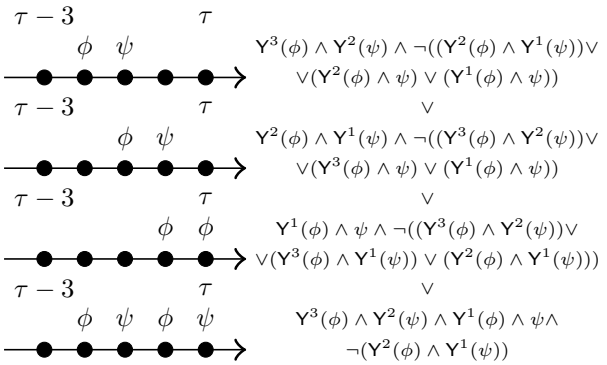


Figure 2.5: *Basic translation of the formula* $\mathfrak{D}^4_{\leq 1}\{(\phi, \psi)\}$

presence of other propositional symbols.

The translation ρ that maps SOLOIST formulae to MPLTLB is straightforward for the "standard" propositional and temporal part (U_I and S_I operators) of SOLOIST. For example, the SOLOIST formula $\phi U_I \psi$ is mapped into the MPLTLB formula $\rho(\phi U_I \psi) = (\neg e \vee \rho(\phi)) U_I (e \wedge \rho(\psi))$.

Note that, as shown in [68], MPLTLB is simply an exponentially succinct encoding of PLTLB. Below we show how metric operators from MPLTLB can be translated into PLTLB. We denote the translation with ς where $0 < l \leq u < \infty, 1 < d < \infty, 0 \leq i < \infty, 0 \leq f \leq \infty$:

$$\varsigma(\mathsf{X}_{=i}(\phi)) \equiv \underbrace{\mathsf{X}\mathsf{X}\cdots\mathsf{X}}_{i \text{ times}}(e \wedge \phi) \qquad (\mathcal{T}_X)$$

$$\varsigma(\mathsf{G}_I(\phi)) \equiv \bigwedge_{i \in I} \varsigma(X_{=i}(\phi)) \qquad (\mathcal{T}_G)$$

$$\varsigma(\phi \mathsf{U}_{\emptyset} \psi) \equiv \bot \qquad (\mathcal{T}_{U1})$$

$$\varsigma(\phi \mathsf{U}_{=i} \psi) \equiv \varsigma(\mathsf{G}_{(0,i)}(\phi)) \wedge \varsigma(\mathsf{X}_{=i}(\psi)) \qquad (\mathcal{T}_{U2})$$

$$\varsigma(\phi \mathsf{U}_{(0,d)} \psi) \equiv \mathsf{X}(\varsigma(\psi) \vee (\varsigma(\phi) \wedge \varsigma(\phi \mathsf{U}_{(0,d-1)} \psi))) \qquad (\mathcal{T}_{U3})$$

$$\varsigma(\phi \mathsf{U}_{(l,u)} \psi) \equiv \varsigma(\mathsf{G}_{(0,l]}(\phi)) \wedge \varsigma(\mathsf{X}_{=l}(\phi \mathsf{U}_{(0,u-l)} \psi)) \qquad (\mathcal{T}_{U4})$$

$$\varsigma(\phi \mathsf{U}_{(i,\infty)} \psi) \equiv \varsigma(\mathsf{G}_{(0,i]}(\phi)) \wedge \varsigma(\mathsf{X}_{=i}(\phi \mathsf{U} \psi)) \qquad (\mathcal{T}_{U5})$$

In the rest of this section we focus on the translation of the new modalities introduced by SOLOIST, i.e., $\mathfrak{C}^K_{\bowtie n}, \mathfrak{U}^{K,h}_{\bowtie n}, \mathfrak{M}^{K,h}_{\bowtie n}, \mathfrak{D}^K_{\bowtie n}$, and refer the reader to [36] for the details.

The translation of the $\mathfrak{C}$ modality considers a formula of the form $\mathfrak{C}^K_{>n}(\phi)$ as the base case. This formula is translated as a disjunction of formulae denoting all possible cases where ϕ holds $n+1$ times within the time window K. Consider, for example, the formula $\mathfrak{C}^3_{>1}(\phi)$. Within a time window of length 3, there are three possible combinations, in terms of positions on the timeline, for representing 2 (i.e., the bound n in the formula incremented by 1) occurrences of the event ϕ. These combinations are shown in Fig. 2.4 and can be characterized by the following formulae (where Y denotes the *yesterday* past MPLTLB modality): $\mathsf{Y}^2(\phi) \wedge \mathsf{Y}^1(\phi)$, $\mathsf{Y}^2(\phi) \wedge \phi$, and $\mathsf{Y}^1(\phi) \wedge \phi$; these formulae are combined in a disjunction in the final translation. The translation for other values of the $\bowtie$ operator is defined by reducing the formula to an equivalent instance of the base case, e.g., $\mathfrak{C}^K_{\leq n}(\phi)$ is equivalent to $\neg \mathfrak{C}^K_{>n}(\phi)$; other cases can be defined similarly.

The translation of the $\mathfrak{U}$ and $\mathfrak{M}$ modalities is defined in terms of the $\mathfrak{C}$ modality. A formula like $\mathfrak{U}^{K,h}_{\bowtie n}(\phi)$ is equivalent to $\mathfrak{C}^{\lfloor \frac{K}{h} \rfloor \cdot h}_{\bowtie n \cdot \lfloor \frac{K}{h} \rfloor}(\phi)$. For the $\mathfrak{M}$ modality, a formula like $\mathfrak{M}^{K,h}_{<n}(\phi)$ is equivalent to $\left(\bigwedge_{m=0}^{\lfloor \frac{K}{h} \rfloor - 1} \mathsf{Y}^{m \cdot h}(\rho(\mathfrak{C}^h_{<n}\phi))\right) \wedge$

$\left(\mathsf{Y}^{\lfloor \frac{K}{h} \rfloor \cdot h}(\rho(\mathfrak{C}_{<n}^{(K \bmod h)}\phi))\right)$.

The translation of a $\mathfrak{D}_{\bowtie n}^{K}(\phi,\psi)$ is defined as a disjunction of formulae denoting all possible occurrences of instances of the pair of events (ϕ,ψ), satisfying the bound related to the average distance. Within a time window K, the maximum number of possible instances of events pairs is $\lfloor \frac{K}{2} \rfloor$. For each of these possible numbers of instances, we define a disjunction of formulae characterizing all their possible combinations in terms of position on the timeline, such that bound on the average distance is satisfied. We also need to explicitly state that events pairs do not occur in all other time instants. Consider, for example, the formula $\mathfrak{D}_{\leq 1}^{4}(\phi,\psi)$. One possible combination of occurrences of a pair of events (ϕ,ψ) is characterized by the formula $\mathsf{Y}^3(\phi) \wedge \mathsf{Y}^2(\psi)$; the absence of other occurrences in the other time instants is denoted by the formula $\neg((\mathsf{Y}^2(\phi) \wedge \mathsf{Y}^1(\psi)) \vee (\mathsf{Y}^2(\phi) \wedge \psi) \vee (\mathsf{Y}^1(\phi) \wedge \psi))$. A similar template can be followed to characterize other combinations of events on the time line. All the possible combinations are depicted on the left side of Fig. 2.5; the disjunction of all the formulae on the right side is the complete translation of the formula.

For the $\mathfrak{D}_{\bowtie n}^{K}$ modality in general, an equivalent LTL formula is defined as follows:

$$\bigvee_{0<h\leq\lfloor \frac{K}{2} \rfloor} \left(\bigvee_{\substack{0\leq i_1<j_1<\ldots i_h<j_h<K \\ \text{and} \\ (\sum_{m=1}^{h} \frac{j_m - i_m}{h}) \bowtie n}} \begin{array}{c} \mathsf{Y}^{i_1}(e \wedge \phi) \wedge \mathsf{Y}^{j_1}(e \wedge \psi) \wedge \\ \ldots \\ \wedge \mathsf{Y}^{i_h}(e \wedge \phi) \wedge \mathsf{Y}^{j_h}(e \wedge \psi) \wedge \\ \\ \neg \left(\bigvee_{\substack{0\leq s<t<K \\ s\notin\{i_1,\ldots,i_h\} \\ t\notin\{j_1,\ldots,j_h\}}} \left(\mathsf{Y}^{s}(e \wedge \phi) \wedge \mathsf{Y}^{t}(e \wedge \psi) \right) \right) \end{array} \right)$$

This translation has been implemented in [85] as a proof of concept.

2.3 Bounded Satisfiability Checking and ZOT

Bounded satisfiability checking [105] (BSC) is a verification technique that complements bounded model checking [38] (BMC): instead of a customary operational model (e.g., a state-transition system) used in BMC, BSC supports the analysis of a *descriptive model*, denoted by a set of temporal logic formulae. With BSC, verification tasks become suitable instances of the satisfiability problem for quite large formulae (written in a certain logic), which comprehend the model of the system to analyze as well as the requirement(s) to verify. BSC has been successfully applied in the context of metric temporal logics and implemented in ZOT [105], a verification toolset based on SAT- and SMT-solvers

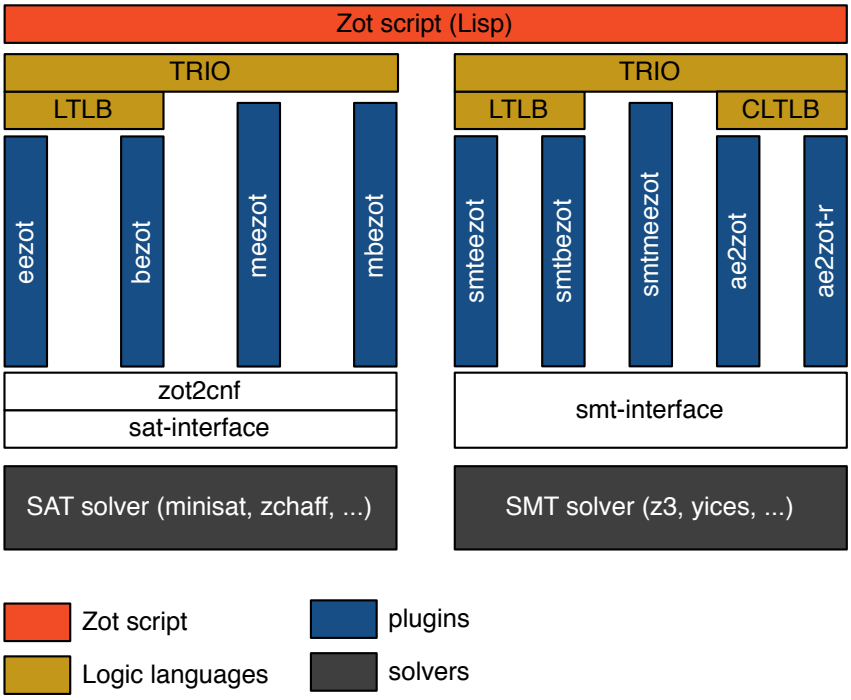


Figure 2.6: *Overview of the ZOT architecture*

ZOT is an agile and easily extendible Bounded Model/Satisfiability Checker written in Common Lisp. The tool supports BMC of different logic languages through a multi-layered plugin-based architecture: at its core layer ZOT supports formulae written in LTL (with past operators). Leveraging the core functionality, the second layer of ZOT supports the propositional, discrete-time metric temporal logic. On top of the second layer ZOT is able to support a wide variety of temporal logic languages by means of dedicated plug-ins. Figure 2.6 shows the architecture of ZOT. In this thesis we will make use of the plugin *ae2zot* (shown in Figure 2.6 as the second plugin on the right). Plugin *ae2zot* translates CLTLB($\mathcal{D}$) into the input language of an SMT solver of choice. The underlying theory used in the target language is either QF_LIA (Quantifier-Free Linear Integer Arithmetic) or QF_LRA (Quantifier-Free Linear Real Arithmetic) depending on the arithmetical constraints used in CLTLB($\mathcal{D}$).

The following steps describe a typical ZOT workflow for BSC: (i) a user writes the specification to be checked as a set of temporal logic formulae, selects the plugin and the time bound to be used; (ii) depending on the

selected plugin, ZOT encodes the received specification in a specific target logic; (iii) ZOT invokes a solver that is capable of handling the target logic; (iv) the result obtained by the solver is parsed back and presented to the user.

Zot supports both SAT solvers (e.g., MiniSat [61]) for Propositional Logic, and SMT solvers (e.g., Z3 [52]) for the decidable fragments of first-order logic, like IDL and EUF.

2.4 QF-EUFIDL and Satisfiability Modulo Theories

QF-EUFIDL

Acronym QF-EUFIDL stands for quantifier free integer difference logic formula with uninterpreted function and predicate symbols. Such a logic combines decision procedures from two theories, namely theory of equality and uninterpreted functions (EUF) and theory of integer difference logic (IDL). This combination is shown to be decidable, and the satisfiability problem is NP-complete, according to Nelson-Oppen Theorem [99].

Signature of QF-EUFIDL is $\Sigma = (C, F, \Pi, V, a)$ where C is a set of constants; F is a set of function symbols; Π is a set of atomic propositions; set V is a set of variables; and $a : F \to \mathbb{N}$ associates arity with each function symbol. Well-formed QF-EUFIDL formulae conform to the following grammar:

$$\phi ::= p \mid t = t \mid t <_d t \mid \neg\phi \mid \phi \vee \phi$$

$$t ::= c \mid v \mid f(t, \ldots, t)$$

where $p \in \Pi$ is an atomic proposition, $c \in C$ is a constant, $v \in V$ is a variable and $f \in F$ is a function symbol.

Structure of QF-EUFIDL is $(\mathbb{Z}, f^{\mathbb{Z}}, =, (<_d)_{d\in\mathbb{Z}}, v_v, v_p)$ where $f^{\mathbb{Z}} : \mathbb{Z}^{a(f)} \to \mathbb{Z}$ is the interpretation of the function symbol f with $a(f)$ over $\mathbb{Z}$. EUF theory introduces the equality relation $=$ with a standard interpretation over $\mathbb{Z}$; IDL theory introduces a family of relations $(<_d)_{d\in\mathbb{Z}}$ defined as $x <_d y \leftrightarrow x < y + d$ with $<$ having the standard interpretation over $\mathbb{Z}$. $v_v : V \to \mathbb{Z}$ is the valuation function that assigns a value to variables from V and $v_p : \Pi \to \{\top, \bot\}$ assigns truth values to the atomic propositions.

An example is $f(x) = y \wedge x <_2 y \wedge (\neg p \vee q)$, where x and y are variables, p and q are atomic propositions and f is an uninterpreted function symbol with arity 1. Note that, QF-EUFIDL can express formulae $x < y$, $x \leq y$, $x \geq y$, $x > y$ and $x = y + d$ using the $<_d$ relation as: $x <_0 y$,

$x <_0 y \vee x = y$, $\neg(x <_0 y)$, $\neg(x <_0 y \vee x = y)$ and $y <_{1-d} x \wedge x <_{d+1} y$, respectively.

Satisfiability modulo theories

Most of the work presented in this thesis makes use of solvers for Satisfiability Modulo Theories (SMT) problems, also called the SMT solvers. More specifically, SMT solvers produce a model (i.e., satisfying assignment) for the QF-EUFIDL formula.

In general, an instance of SMT problem is a generalization of a well known SAT problem over boolean formulae [112], where atoms can be formulae of an underlying theory. Theories which are typically supported by the SMT solvers are *the theory of equality and uninterpreted functions* (EUF), *the theory of quantifier-free linear arithmetic over* $\{\mathbb{Z}, \mathbb{Q}\}$ (LIA, LRA), *the difference logic theory over* $\{\mathbb{Z}, \mathbb{Q}\}$ (IDL, RDL) and *the theory of arrays and bit-vectors*. The Satisfiability problem amounts to checking whether there exists a model, i.e. an assignment to variables, functions and predicates, such that the interpretation of formula is true. When the atoms of the formulae are interpreted with respect to some theory, we are solving a satisfiability problem *modulo* that theory.

SMT solvers can also provide modes for formulae interpreted over multiple theories with *disjoint signatures*. We say that two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ have disjoint signatures when $\Sigma_1 \cap \Sigma_2 = \{=\}$ where $=$ is the symbol for the binary equality relation. In other words, signatures $\mathcal{T}_1$ and $\mathcal{T}_2$ do not have any constant, function or predicate symbol in common, except for the equality predicate. The main idea is to combine separate solvers for the theories into one by using them to solve their respective theories and exchanging the entailed equalities them. Nelson-Oppen combination method [99] identifies sufficient conditions for combining two theories over disjoint signatures.

In Chapter 4 we will use the EUF combined with IDL to encode the satisfiability problem for SOLOIST. Algorithms which solve satisfiability of formulae in EUF are based on the congruence closure of graphs [100]. Terms of a formula are encoded by nodes of a directed graph G. The equality and dependency relation between terms are encoded with edges between nodes of G. For instance, in $f(a, g(b)) = a$ the term $f(a, g(b))$ depends on a and $g(b)$, hence there are dependency edges between them and $f(a, g(b))$ and a are equal, hence there is an equality edge between them. Satisfiability reduces to checking whether the congruence closure is compatible with the structure of the formula. Let G be a graph of n vertices and m edges; the

congruence closure for G can be computed in time $\mathcal{O}(m \log m)$ and space $\mathcal{O}(nm)$, in the worst case. Details about congruence closure algorithms and implementations can be found in [56, 100, 101]. Negative cycle detection problem is exploited in [87] to solve satisfiability of conjunctions of IDL formulae of the form $ax + by \leq d$ where $a, b \in \{-1, 0, +1\}$. Constraints are represented by a direct graph G. Nodes of G represent variables and DL constraints define edges between nodes. A conjunction of constraints is satisfiable if the graph G does not contain any negative cycles. The worst case complexity is $\mathcal{O}(nm)$ where n is the number of variables involved in conjunction of m constraints. Algorithms for negative cycles detections are deeply studied in literature, for instance in [46]. The combination of EUF and IDL theories enables the satisfiability procedure for QF-EUFIDL formulae which is NP-complete [106].

2.5 Distributed Programming Models

2.5.1 MapReduce

MapReduce [53] is a programming model, initially developed by Google, for processing and analyzing large data sets using a parallel, scalable and distributed infrastructure. It allows the user to define two functions, *map* and *reduce*, that are inspired by the homonymous functions that are typically found in functional programming languages. The MapReduce modes divides the processing into two phases: the *map phase* and the *reduce phase*. Each phase has *key-value pairs* as input and output, the types of which may be chosen by the programmer. The *map* function is applied in the map phase on every *input* key-value pair. Its input is a key-value pair associated with the input data and its output is a set of *intermediate* key-value pairs; its signature is `map(k:`$\mathtt{K}_1$`,v:`$\mathtt{V}_1$`):list[(k:`$\mathtt{K}_2$`, v:`$\mathtt{V}_2$`)]`. The *reduce* function is applied in the reduce phase to all the values from the intermediate key-value pairs that have the same key to derive the output data appropriately; its signature is `reduce(k:`$\mathtt{K}_2$`, list(v:`$\mathtt{V}_2$`)):list[v:`$\mathtt{V}_2$`]`. In the definitions above, $\mathtt{K}_1$ and $\mathtt{K}_2$ are types for keys and $\mathtt{V}_1$ and $\mathtt{V}_2$ are types for values.

We illustrate how the MapReduce model works with an example that counts the number of occurrences of each word in a large collection of documents. Consider the *map* and *reduce* functions in the following pseudocode:

```
map(String key, String value)
 //key: document name
 //value: document contents
 for each word w in value:
     Emit(w,"1")
```

```
reduce(String key, Iterator values)
 //key: a word
 //values: a list of partial counts
 int result = 0
 for each v in values:
     result += ParseInt(v)
 Emit(result)
```

The *map* function takes in input a pair made of the document name and the document contents; for each word in a document, it emits a pair composed of the word itself and its associated count of occurrences (which is just 1 initially). The list of all emitted pairs is partitioned into groups and sorted according to their key for the reduction phase; in the example, pairs are grouped and sorted according to the key (i.e. the word they contain). The *reduce* function takes a word and the list (in the form of an iterator) of the aggregated partial counts: it sums all the counts emitted for each particular word (i.e., each unique key).

Besides the programming model, MapReduce also defines a framework that implements the programming model and provides, in transparent way to the user, parallelization, fault tolerance, locality optimization, and load balancing. The main assumption of the MapReduce framework is that the input data is very large that it must be split and saved on multiple interconnected machines (or nodes) in a distributed fashion. The framework supports computations over the input data by transmitting the code of the computation between the nodes rather than moving data itself. It is responsible for scheduling and executing the *map* and *reduce* function code by starting remote processes (also called *mappers* and *reducers*, respectively) on a cluster of available nodes storing the data. It also manages the necessary communication and data transfer (usually leveraging a distributed file system). A very popular open-source distribution of the MapReduce framework is Apache Hadoop [8, 118]. The execution of a Hadoop MapReduce operation (called *job*) proceeds as follows. First, the framework splits the input into blocks[2] of a certain size, then, using the so called *InputReader*, it parses the blocks in parallel and generates input key-value pairs. It then assigns each parsed input block to a mapper. A *mapper* executes the *map* function on the input key-value pairs and generates a set of intermediate key-value pairs. Notice that each run of the *map* function is stateless, i.e., the transformation of a single key-value pair does not depend on any other key-value pair. The next phase is called *shuffle and sort*: the framework takes the intermediate key-value pairs generated by each mapper, divides them into partitions (each to be processed by a reducer) possibly transmit-

[2] Also called input splits or chunks.

ting (also called *shuffling*) some pairs between the nodes in order to have complete partitions on the same node. The division of intermediate data into partitions is done by a *partitioning function*, which depends on the (user-specified) number of reducers and the keys of the intermediate pairs. If there are less reducers than the different intermediate keys, the partitioning function will assign pairs with two or more different keys to a single reducer. After partitioning (and possibly shuffling), the framework sorts the pairs based on their keys within each partition. Sorting is done based on the *comparison operator* for the keys, which can either be user-defined, or the default one[3] that establishes an arbitrary, but consistent ordering by comparing the hashes of the keys. Sorting saves time for the reducer, helping it easily distinguish when a new key is reached during the execution. The framework then invokes a reducer on each partition. Each reducer executes the *reduce* function, which produces the output data. This output is appended to a final output file for this reduce partition. The output of the MapReduce job will then be available in several files, one for each used reducer.

2.5.2 Spark

Spark [120, 121] is another computational framework, which comes with its own programming model for processing large data sets. It is designed to support a class of iterative MapReduce applications, called *applications with working sets*. These applications reuse a *working set of data* across multiple operations and cover a broad range of very common use cases in BigData analytics like:

- Iterative algorithms, like machine learning or graph algorithms
- Interactive data mining, where large volumes of data are queried repeatedly
- Streaming applications that maintain aggregate state over time

Traditional MapReduce framework is suboptimal in these cases because it is designed to perform simple acyclic operations on data. To support iterative applications, one must run a sequence of distinct MapReduce jobs, each reading data from persistent storage and writing it back after applying the map and reduce operations. Interacting with persistent storage incurs significant time costs, especially when performed repeatedly.

[3] Also called HashPartitioner

The Spark framework offers an abstraction called *resilient distributed dataset* (or RDD) to represent the working set of data that can be distributed over multiple nodes of a cluster. More precisely, an RDD is a *collection of user-defined objects* that encapsulates the information about where the objects are physically stored on the nodes of the cluster. Each RDD is split into multiple partitions, such that a partition refers to the objects that are stored on the same cluster node.

The Spark applications are written as a sequence of operations on RDDs. RDDs support two types of operations: *transformations* and *actions*. Transformations create new RDDs from the old ones by modifying their content. For example, standard operations on collections, like *map()* and *filter()* are transformations. Actions are operations that return a particular result from an RDD or write an RDD to the persistent storage. Counting the objects in the RDD is an example of an action. Transformations and actions are also different in the way the Spark framework applies them on the data. Spark keeps an acyclic directed graph of different RDDs created by transformations without applying the transformations directly on the data. Hence, every RDD has its *lineage* – a sequence of transformations applied to the RDD that represents the initial stored data. Once an action is invoked on an RDD, the Spark framework first reads the data from the storage, applies the transformations from the RDD's lineage in the chronological order and then applies the invoked action.

By default, the Spark framework recomputes the RDDs each time an action is invoked. However, it also allows the programmers to perform in memory *caching* of the RDDs obtained after applying a number of transformations. This can dramatically save time if several actions are performed on the RDDs with common lineage.

The underlying assumption of the Spark framework is that computation is more time efficient than reading or writing data to persistent storage, therefore it does not persist any intermediate computations on RDDs. If the computation fails Spark uses the RDD lineage to recompute the data only on the cluster nodes where the failure occurred.

When executing on a cluster, Spark uses a master/slave runtime architecture with one central coordinator, called the *driver*, and many distributed workers, called *executors*. When launched, Spark driver relies on an external service called *cluster manager* to obtain resources of the cluster in order to deploy the executors. Besides that, the driver executes the user code: creates RDDs; performs transformations and actions; maintains the RDD lineage and when an action is performed on an RDD it packages the code from its lineage and sends it to the executors to perform the computation.

The executors are slave processes responsible for running the code received from the driver and for providing in-memory caching.

Similarly like on the Section 2.5.1, we exemplify using pseudocode how Spark can be used to count the number of word occurrences in a large collection of documents.

```
RDD[String] textFile       = read("/path/to/input")
RDD[String] words          = textFile.flatMap(line => line.split(" "))
RDD[String,Integer] init   = words.map(word => (word, 1))
RDD[String,Integer] count  = init.reduceByKey((x,y) => x + y)
count.saveAsTextFile("/path/to/output")
```

In the above driver code, we create four RDDs: `textFile, words, init` and `count`. The `textFile` RDD refers to the set of strings representing the lines from all the input documents. The `words` RDD is derived from `textFile` after the `flatMap` transformation. This transformation splits each line into individual words, hence each string in `words` RDD is a single word. Next, the `map` transformation associates to each word the number of its occurrences (initially 1) and produces a key-value pair RDD `init`. Finally, the `count` RDD is obtained after applying the `reduceByKey` transformation on the `init` RDD and passing the sum operator (+) that is applied to all values with the same keys. The final RDD contains unique words for keys and the number of occurrences for the values.

Part II

SOLOIST Satisfiability

CHAPTER 3

Decision procedure based on CLTLB($\mathcal{D}$)

3.1 Overview

In Section 2.2.2 we showed how, under certain assumptions, SOLOIST can be translated into LTL, thus guaranteeing its decidability based on well-known results in temporal logic. However, this translation was only a proof of concept and was not meant to guarantee efficiency if one would apply any LTL-based verification techniques. The SOLOIST operators can specify very complex quantitative properties whose semantics can be captured *efficiently* only by using a more expressive logic. In this chapter we show how SOLOIST can be encoded into CLTLB($\mathcal{D}$) and thus effectively reducing the decision procedure for SOLOIST to the decision procedure for CLTLB($\mathcal{D}$) that is provided by ZOT, as noted in Section 2.3.

We chose CLTLB($\mathcal{D}$) as the target of our translation since it has an efficient decision procedure and it supports the definition of arithmetical constraints over a set of integer variables (also called counters); as we will detail in Section 3.2, these counters allow a compact, intuitive and easy-to-verify translation.

In Section 3.3 we show how this translation can be used in the context of trace checking. We express the problem of trace checking of SOLOIST

properties in terms of bounded satisfiability checking (BSC) of CLTLB($\mathcal{D}$) and rely on the BSC procedure for metric temporal logic [105] implemented in ZOT.

In Chapter 5 we show how the approach can be applied for trace checking properties of service compositions [34]. We focus on requirements containing quantitative properties involving aggregate operations on events occurring in a given time window, like the average response time of a certain operation provided by a partner service.

3.2 Translation

The key point in defining the translation from SOLOIST to CLTLB($\mathcal{D}$) is to bridge the gap between the semantics of SOLOIST based on *timed* ω-words, where the temporal information is denoted by an integer time-stamp, and the one of CLTLB($\mathcal{D}$), where the temporal information is implicitly defined by the integer position in an ω-word. The two temporal models can be transformed into each other. Here we are interested in pinpointing, in a CLTLB($\mathcal{D}$) ω-word, only the positions that correspond to actual time-stamps in a SOLOIST timed ω-word. These timestamps correspond to instants where some event actually occurs. To do so, we add to the set Π a special propositional symbol e, which is true in each position corresponding to a "valid" time-stamp in the timed ω-word; a "valid" time-stamp is one where at least an event, represented by a propositional symbol, occurs. An example of this conversion is shown in Fig. 2.3, where a timed ω-word is depicted in the timeline at the top and its equivalent ω-word corresponds to the timeline at the bottom; notice the special symbols $\neg e$ that hold in positions in the ω-word which do not correspond to a "valid" time-stamp in the timed ω-word. Hereafter, when displaying ω-words, we will omit the symbol e from positions in the timeline, since its presence can be implied by the presence of other propositional symbols in the same position in the timeline.

To define the translation from SOLOIST to CLTLB($\mathcal{D}$) we consider, without loss of expressiveness, only formulae in positive normal form, i.e., where negation may only occur on atoms (see, for example, [104]). First, we extend the syntax of the language by introducing a dual version for each operator in the original syntax, except for the $\mathfrak{C}^K_{\bowtie n}, \mathfrak{U}^{K,h}_{\bowtie n}, \mathfrak{M}^{K,h}_{\bowtie n}, \mathfrak{D}^K_{\bowtie n}$ modalities[1]: the dual of $\wedge$ is $\vee$; the dual of U_I is "*Release*" R_I: $\phi \mathsf{R}_I \psi \equiv \neg(\neg\phi \mathsf{U}_I \neg\psi)$; the dual of S_I is "*Trigger*" T_I: $\phi \mathsf{T}_I \psi \equiv \neg(\neg\phi \mathsf{S}_I \neg\psi)$. A for-

[1] A negation in front of one of the $\mathfrak{C}^K_{\bowtie n}, \mathfrak{U}^{K,h}_{\bowtie n}, \mathfrak{M}^{K,h}_{\bowtie n}, \mathfrak{D}^K_{\bowtie n}$ modalities becomes a negation of the relation denoted by the $\bowtie$ symbol, hence no dual version is needed for them.

mula is in *positive normal form* if its alphabet is $\{\wedge, \vee, \mathsf{U}_I, \mathsf{R}_I, \mathsf{S}_I, \mathsf{T}_I, \mathfrak{C}^K_{\bowtie n}, \mathfrak{U}^{K,h}_{\bowtie n}, \mathfrak{M}^{K,h}_{\bowtie n}, \mathfrak{D}^K_{\bowtie n}\} \cup \Pi \cup \bar{\Pi}$, where $\bar{\Pi}$ is the set of formulae of the form $\neg p$ for $p \in \Pi$.

3.2.1 Translation of boolean and temporal formulae

We can now illustrate the translation ρ from SOLOIST formulae to CLTLB($\mathcal{D}$). For the propositional ($\neg$, $\wedge$ and $\vee$) and temporal part (U_I, S_I, R_I and T_I) of SOLOIST the translation is straightforward:

$$\begin{aligned}
\rho(p) &\equiv p, p \in \Pi \\
\rho(\neg p) &\equiv \neg p, p \in \Pi \\
\rho(\phi \wedge \psi) &\equiv \rho(\phi) \wedge \rho(\psi) \\
\rho(\phi \vee \psi) &\equiv \rho(\phi) \vee \rho(\psi) \\
\rho(\phi \mathsf{U}_I \psi) &\equiv (\neg e \vee \rho(\phi)) \mathsf{U}_I (e \wedge \rho(\psi)) \\
\rho(\phi \mathsf{S}_I \psi) &\equiv (\neg e \vee \rho(\phi)) \mathsf{S}_I (e \wedge \rho(\psi)) \\
\rho(\phi \mathsf{R}_I \psi) &\equiv (e \wedge \rho(\phi)) \mathsf{R}_I (\neg e \vee \rho(\psi)) \\
\rho(\phi \mathsf{T}_I \psi) &\equiv (e \wedge \rho(\phi)) \mathsf{T}_I (\neg e \vee \rho(\psi))
\end{aligned}$$

In the subsequent sections we focus on the translation of the $\mathfrak{C}^K_{\bowtie n}$, $\mathfrak{U}^{K,h}_{\bowtie n}$, $\mathfrak{M}^{K,h}_{\bowtie n}$ and $\mathfrak{D}^K_{\bowtie n}$ modalities.

3.2.2 Translation of the $\mathfrak{C}$ modality

The $\mathfrak{C}$ modality expresses a bound on the number of occurrences of a certain event in a given time window; it comes natural to use the counters available in CLTLB($\mathcal{D}$) for the translation. Indeed, for each sub-formula of the form $\mathfrak{C}^K_{\bowtie n}(\chi)$, we introduce a counter c_χ, constrained by a set of CLTLB($\mathcal{D}$) axioms, detailed below. Informally, these axioms define the value of c_χ such that at each time position it captures the number of occurrences of event χ seen in the past:

A1) $c_\chi = 0$
A2) $\mathsf{G}((e \wedge \chi) \rightarrow \mathsf{X}(c_\chi) = c_\chi + 1)$
A3) $\mathsf{G}((\neg e \vee \neg \chi) \rightarrow \mathsf{X}(c_\chi) = c_\chi)$

Axiom A1 initializes the counter to zero. Axiom A2 states that if there is an occurrence of a valid event χ, (denoted by $e \wedge \chi$) the value of the counter c_χ in the next time instant is increased by one with respect to the value at the current time instant. Axiom A3 refers to the opposite situation, when either there is no occurrence of the event χ or the time instant is not valid (i.e., e does not hold in that time instant). In this case, the value of the

counter in the next time instant must have the same value as in the current time instant. Both axioms A2 and A3 have to hold at every time instant, so they are in the scope of a *globally* temporal operator.

We can calculate the exact number of occurrences by subtracting the values of the counter at the appropriate time instants; we explain this through the example in Fig. 3.2, which depicts a short trace of length 21 and the values assumed by the counter c_χ (in the first row) at each time instant, as determined by the axioms. In the example, to evaluate the formula $\mathfrak{C}^K_{>1}(\chi)$ with $K = 11$ at time instant $t = 16$, we subtract from the value of the counter c_χ at time instant $t + 1 = 17$ (since we want to consider a possible occurrence of χ at time instant t) the value of the counter at time instant 6 (i.e., $t-(K-1) = 16-(11-1)$, which is 11 time instants in the past with respect to time instant $t+1$); these values are enclosed in the figure with diamond markers. The value resulting from the subtraction $6 - 1 = 5$ is then compared to the specified bound ($5 > 1$). In symbols, this can be written as $\mathsf{X}(c_\chi) - \mathsf{Y}^{10}(c_\chi) > 1$ evaluated at time instant t. This intuition is captured by the following CLTLB($\mathcal{D}$) formula, which generalizes the translation of a SOLOIST sub-formula of the form $\mathfrak{C}^K_{\bowtie n}(\chi)$:

$$\rho\left(\mathfrak{C}^K_{\bowtie n}(\chi)\right) \equiv \mathsf{X}(c_\chi) - \mathsf{Y}^{K-1}(c_\chi) \bowtie n$$

Notice that the axioms are conjuncted with the resulting translation of the SOLOIST formula, thus effectively constraining the behavior of all the counters of type c_χ.

3.2.3 Translation of the $\mathfrak{U}$ modality

The translation of the $\mathfrak{U}$ modality is defined in terms of the $\mathfrak{C}$ modality; it can then be defined as follows:

$$\rho\left(\mathfrak{U}^{K,h}_{\bowtie n}(\phi)\right) \equiv \rho\left(\mathfrak{C}^{\lfloor \frac{K}{h} \rfloor \cdot h}_{\bowtie n \cdot \lfloor \frac{K}{h} \rfloor}(\phi)\right)$$

This translation ignores the tail subinterval of the $\mathfrak{U}$ modality, which is consistent with the SOLOIST semantics [36].

3.2.4 Translation of the $\mathfrak{M}$ modality

To translate the $\mathfrak{M}$ modality we rely on the $\mathfrak{C}$ modality. The translation of a formula of the form $\mathfrak{M}^{K,h}_{<n}(\phi)$ is defined as: $\rho\left(\mathfrak{M}^{K,h}_{<n}(\phi)\right) \equiv$

$$\left(\bigwedge_{m=0}^{\lfloor \frac{K}{h} \rfloor - 1} \mathsf{Y}^{m \cdot h}(\ \rho(\mathfrak{C}^h_{<n}\phi))\right) \wedge \left(\mathsf{Y}^{\lfloor \frac{K}{h} \rfloor \cdot h}(\ \rho(\mathfrak{C}^{(K \bmod h)}_{<n}\phi))\right)$$

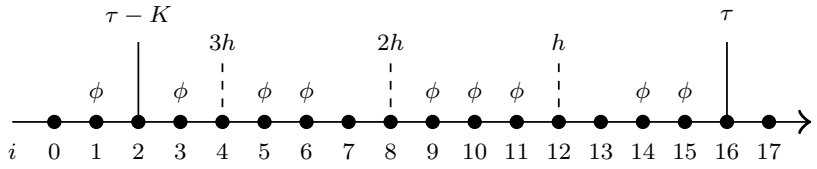


Figure 3.1: *Sample trace showing the time window and the observation subintervals considered for the evaluation of the $\mathfrak{M}^{14,4}_{\geq 1}(\phi)$ formula at time instant τ =16*

For a formula of the form $\mathfrak{M}^{K,h}_{>n}(\phi)$ we have: $\rho\left(\mathfrak{M}^{K,h}_{>n}(\phi)\right) \equiv$

$$\left(\bigvee_{m=0}^{\lfloor \frac{K}{h} \rfloor - 1} \mathsf{Y}^{m \cdot h}(\ \rho(\mathfrak{C}^{h}_{>n}\phi))\right) \vee \left(\mathsf{Y}^{\lfloor \frac{K}{h} \rfloor \cdot h}(\ \rho(\mathfrak{C}^{(K \bmod h)}_{>n}\phi))\right)$$

The formula decomposes the computation of the maximum number of occurrences of the event $(e \wedge \phi)$ by suitably combining constraints on the number of occurrences of the event in each observation interval h within the time window K. The other cases of the operator $\bowtie$ can be defined in a similar way.

Fig. 3.1 shows an example trace of length 18. We evaluate the formula $\mathfrak{M}^{14,4}_{\geq 3}(\phi)$ at time instant $\tau = 16$. The vertical solid lines delimit the time window of length $K = 14$; the dashed lines delimit the adjacent non-overlapping observation subintervals of length $h = 4$. The $\mathfrak{M}$ modality formula is translated into a disjunction of four $\mathfrak{C}$ modality formulae each referring to a different subinterval. The first three ($\lfloor \frac{K}{h} \rfloor = \lfloor \frac{14}{4} \rfloor = 3$) formulae have the form $\mathfrak{C}^{4}_{\geq 3}(\phi)$ and are evaluated at time instants $16(= 16 - 0 \cdot 4)$, $12(= 16 - 1 \cdot 4)$ and $8(= 16 - 2 \cdot 4)$. The fourth formula (corresponding to rightmost disjunct defined in the translation ρ) has the form $\mathfrak{C}^{2}_{\geq 3}(\phi)$ and is evaluated at time instant $4(= 16 - \lfloor \frac{14}{4} \rfloor \cdot 4)$. We can conclude that, the formula $\mathfrak{M}^{14,4}_{\geq 3}(\phi)$ holds at time instant $\tau = 16$ since formula $\mathfrak{C}^{4}_{\geq 3}(\phi)$ holds at time instant 12 and renders the disjunction true.

3.2.5 Translation of the $\mathfrak{D}$ modality

The $\mathfrak{D}$ modality expresses a bound on the average distance between the occurrences of pairs of events in a given time window. We consider only (sub)formulae of the $\mathfrak{D}$ modality that refer to one pair, like $\mathfrak{D}^{K}_{\bowtie n}(\phi, \psi)$.

Events, corresponding to atomic propositions in SOLOIST, can occur multiple times in a trace; when we refer to a specific occurrence of an event

ϕ at a time instant τ, we denote this as $\phi_{|\tau}$. Clearly, a pair of events (ϕ, ψ) may also have multiple instances in a trace. We call a pair of the form $(\phi_{|i}, \psi_{|j})$ an *instance* if there is an occurrence of event ϕ at time instant i and an occurrence of event ψ at time instant j, with $i < j$. We call such instance *open* at time instant τ if $i \leq \tau < j$. Otherwise, the instance is *closed* at time instant τ. The *distance* of a closed (pair) instance is $j - i$; for an open pair at time instant τ, the distance is $\tau - i$. A time window of length K defined for a $\mathfrak{D}$ modality (sub-)formula evaluated at time instant τ is bounded by the time instants $\tau + 1$ and $\tau - K + 1$. For a certain trace, we say that a $\mathfrak{D}$ modality (sub-)formula for a pair of events (ϕ, ψ) has a *left-open* pair in the trace if there is an open instance of (ϕ, ψ) at time instant $\tau - K + 1$ in the trace; similarly, we say that the (sub-)formula has a *right-open* pair in the trace if there is an open instance of (ϕ, ψ) at time instant $\tau + 1$ in the trace. The translation has then to take into account four distinct cases, depending on whether a $\mathfrak{D}$ modality (sub-)formula contains either (left- and/ or right-) open pairs or none.

As done in the case of the $\mathfrak{C}$ modality, the translation is based on CLTLB($\mathcal{D}$) counters. For each sub-formula of the form $\mathfrak{D}^{K}_{\bowtie n}(\phi, \psi)$, we introduce five counters, namely:

- $g_{\phi,\psi}$: this binary counter assumes value 1 in the time instants following an occurrence of ϕ and it is reset to 0 after an occurrence of ψ. It acts as a flag denoting the time instants during which the event pair instance is open;

- $h_{\phi,\psi}$: in each time instant, this counter contains the number of previously-seen closed pair instances. It is increased after every occurrence of ψ;

- $s_{\phi,\psi}$: at each time instant, the value of this counter corresponds to the sum of distances of all previously occurred pair instances. It is increased at every time instant when either $g_{\phi,\psi} = 1$ holds or ϕ occurs;

- $a_{\phi,\psi}$: this counter keeps track of the sum of the distances of all previously occurred closed pair instances;

- $b_{\phi,\psi}$: this counter has the values that will be assumed by counter $s_{\phi,\psi}$ at the next occurrence of ψ (more details below).

Counters $a_{\phi,\psi}$, $b_{\phi,\psi}$, and $h_{\phi,\psi}$ are directly used in the translation of the $\mathfrak{D}$ modality (sub-)formulae, while counters $g_{\phi,\psi}$ and $s_{\phi,\psi}$ are helper counters, used to determine the values of the other counters. These five counters are constrained by the following axioms:
A4) $g_{\phi,\psi} = 0 \wedge h_{\phi,\psi} = 0 \wedge a_{\phi,\psi} = 0 \wedge s_{\phi,\psi} = 0$

A5) $(\mathsf{X}(b_{\phi,\psi}) = b_{\phi,\psi})\mathsf{W}(e \wedge \psi)$
A6) $\mathsf{G}\,((e \wedge \phi \wedge \neg\psi) \rightarrow (\mathsf{X}(g_{\phi,\psi}) = 1 \wedge \mathsf{X}(s_{\phi,\psi}) = s_{\phi,\psi} + 1 \wedge \mathsf{X}(h_{\phi,\psi}) = h_{\phi,\psi} \wedge \mathsf{X}(a_{\phi,\psi}) = a_{\phi,\psi}))$
A7) $\mathsf{G}\,((e \wedge \psi \wedge \neg\phi) \rightarrow (\mathsf{X}(g_{\phi,\psi}) = 0 \wedge \mathsf{X}(h_{\phi,\psi}) = h_{\phi,\psi} + 1 \wedge \mathsf{X}(a_{\phi,\psi}) = s_{\phi,\psi} \wedge \mathsf{X}(s_{\phi,\psi}) = s_{\phi,\psi} \wedge b_{\phi,\psi} = s_{\phi,\psi} \wedge \mathsf{X}((\mathsf{X}(b_{\phi,\psi}) = b_{\phi,\psi})\mathsf{W}(e \wedge \psi))))$
A8) $\mathsf{G}\,((\neg e \vee (\neg\phi \wedge \neg\psi)) \rightarrow (\mathsf{X}(g_{\phi,\psi}) = g_{\phi,\psi} \wedge \mathsf{X}(h_{\phi,\psi}) = h_{\phi,\psi} \wedge \mathsf{X}(a_{\phi,\psi}) = a_{\phi,\psi} \wedge (g_{\phi,\psi} = 1 \rightarrow \mathsf{X}(s_{\phi,\psi}) = s_{\phi,\psi} + 1) \wedge (g_{\phi,\psi} = 0 \rightarrow \mathsf{X}(s_{\phi,\psi}) = s_{\phi,\psi})))$
A9) $\mathsf{G}\,((e \wedge \phi \wedge \psi) \rightarrow (\mathsf{X}(g_{\phi,\psi}) = g_{\phi,\psi} \wedge \mathsf{X}(h_{\phi,\psi}) = h_{\phi,\psi} + 1 \wedge \mathsf{X}(a_{\phi,\psi}) = a_{\phi,\psi} \wedge \mathsf{X}(s_{\phi,\psi}) = s_{\phi,\psi} \wedge \mathsf{X}((\mathsf{X}(b_{\phi,\psi}) = b_{\phi,\psi})\mathsf{W}(e \wedge \psi))$

Axiom A4 initializes all counters except counter $b_{\phi,\psi}$, which will assume values determined by counter $s_{\phi,\psi}$. Axiom A5 states that the value of counter $b_{\phi,\psi}$ will stay the same in all the time instants until the first occurrence of ψ. Notice that we use the W modality ("weak until"), to deal with traces without occurrences of ψ. Axiom A6 determines the next time instant value of the following counters, upon occurrence of a ϕ and absence of a ψ event (denoted by $e \wedge \phi \wedge \neg\psi$): counter $g_{\phi,\psi}$ is set to 1; counter $s_{\phi,\psi}$ is incremented by 1; counters $h_{\phi,\psi}$ and $a_{\phi,\psi}$ are constrained not to change in the next time instant. Axiom A7 determines how the counters are updated when a ψ event occurs and a ϕ event does not: counter $g_{\phi,\psi}$ is set to 0; counters $b_{\phi,\psi}$, $\mathsf{X}a_{\phi,\psi}$, and $\mathsf{X}s_{\phi,\psi}$ are set to be equal to $s_{\phi,\psi}$. Moreover, a formula equivalent to axiom A5 holds in the next time instant, forcing the value of $b_{\phi,\psi}$ to stay the same in all the following time instants until the next occurrence of ψ. Axiom A8 covers the cases either when there are no valid events or when neither ϕ nor ψ occur. In these cases the values of counters $g_{\phi,\psi}$, $h_{\phi,\psi}$, and $a_{\phi,\psi}$ are constrained to stay the same, while counter $b_{\phi,\psi}$ is unconstrained. As for counter $s_{\phi,\psi}$, we need to distinguish two separate cases: when the pair instance is open (denoted by $g_{\phi,\psi} = 1$), counter $s_{\phi,\psi}$ is incremented by 1, otherwise it stays the same. Axiom A9 handles the case when both events ϕ and ψ hold, by incrementing counter $h_{\phi,\psi}$ by 1 and constraining the value of counter $b_{\phi,\psi}$ in the same way like axiom A7. The values of the other counters are constrained to stay the same.

As said above, the $b_{\phi,\psi}$ counter keeps the values that will be assumed by counter $s_{\phi,\psi}$ at the next occurrence of ψ. The value assumed by both counters $a_{\phi,\psi}$ and $b_{\phi,\psi}$ originates from counter $s_{\phi,\psi}$, as enforced by axiom A7. Axioms A6 and A8 make sure the value of $s_{\phi,\psi}$ is propagated in the future via counter $a_{\phi,\psi}$, while axiom A7 enables the propagation of this value in the past via counter $b_{\phi,\psi}$. We elaborate this through an example: Fig. 3.2 represents a short trace with event ψ occurring at time instants 5, 14, and 19. Axiom A5 enforces equality between successive values of counter $b_{\phi,\psi}$

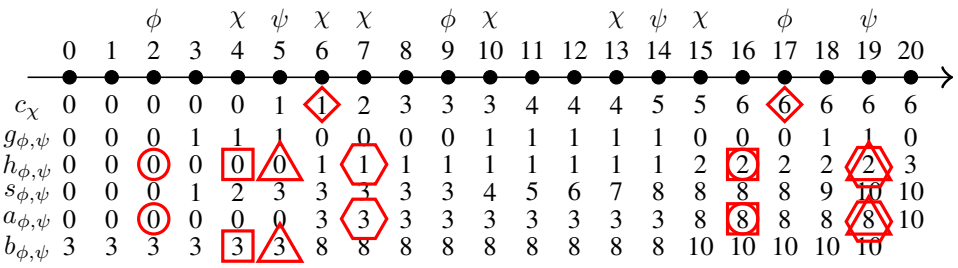


Figure 3.2: *Sample trace showing the counters used for the translation of the $\mathfrak{C}$ and $\mathfrak{D}$ modalities*

at adjacent time instants until the first occurrence of ψ (time instants 0–5). Additional equalities (of the same type) on the values of counter $b_{\phi,\psi}$ are enforced by axiom A7 (time instants 6–14 and 15–19). The same axiom also determines equality between the values of the $s_{\phi,\psi}$ and $b_{\phi,\psi}$ counters upon an occurrence of ψ (time instants 5, 14 and 19).

The translation $\rho\left(\mathfrak{D}^K_{\bowtie n}(\phi,\psi)\right)$ is defined as:

$$\begin{aligned}\texttt{if}^2\ (\mathsf{Y}^{K-1}(g_{\phi,\psi})=1)\quad &\texttt{then}\ \left(\tfrac{\mathsf{X}(a_{\phi,\psi})-\mathsf{Y}^{K-1}(b_{\phi,\psi})}{\mathsf{X}(h_{\phi,\psi})-\mathsf{Y}^{K-1}(h_{\phi,\psi})-1}\bowtie n\wedge Z_1\right)\\ &\texttt{else}\ \left(\tfrac{\mathsf{X}(a_{\phi,\psi})-\mathsf{Y}^{K-1}(a_{\phi,\psi})}{\mathsf{X}(h_{\phi,\psi})-\mathsf{Y}^{K-1}(h_{\phi,\psi})}\bowtie n\wedge Z_2\right)\end{aligned}$$

The condition $\mathsf{Y}^K(g_{\phi,\psi}) = 1$ checks whether the time window contains an open pair instance on its left bound. Since the semantics of the $\mathfrak{D}$ modality considers only closed pairs within the time window to compute the average distance, open pairs must be ignored both on the left and on the right bound of the time window. There is no need to differentiate between the cases when there is a right-open pair, since counter $a_{\phi,\psi}$ only considers distances between closed pair instances. The numerator of the fraction in both the `then` and `else` branches denotes the total distance, while the denominator corresponds to the number of pair instances considered for computing the total distance. Propositions Z_1 and Z_2 are respectively $\mathsf{X}(h_{\phi,\psi}) - \mathsf{Y}^{K-1}(h_{\phi,\psi}) \neq 1$ and $\mathsf{X}(h_{\phi,\psi}) - \mathsf{Y}^{K-1}(h_{\phi,\psi}) \neq 0$; due to these disjuncts the $\mathfrak{D}$ modality evaluates to true when there are no closed pairs in the time window K. Axioms A4, A5, A6, A7, A8, A9 are conjuncted with the resulting translation and added as constraints that hold at the initial time instant of the trace.

An example of the use of counters to evaluate a formula with the $\mathfrak{D}$ modality is shown in Fig. 3.2, which depicts a simple trace and the values

[2] "`if` A `then` B `else` C" can be written as $(A \wedge B) \vee (\neg A \wedge C)$

assumed by the counters $g_{\phi,\psi}$, $h_{\phi,\psi}$, $s_{\phi,\psi}$, $a_{\phi,\psi}$, and $b_{\phi,\psi}$ at each time instant, as determined by the axioms. We notice that there are three instances of the (ϕ, ψ) pair. If we evaluate the formula $\mathfrak{D}^{14}_{\bowtie n}(\phi, \psi)$ at time instant 15, the two pair instances $(\phi_{|2}, \psi_{|5})$ and $(\phi_{|9}, \psi_{|14})$, considered to compute the average distance, are closed. The left-hand side (lhs) of the comparison operator ($\bowtie$) is evaluated using the values of counters $a_{\phi,\psi}$ and $h_{\phi,\psi}$ at time instants 16 and 2 (enclosed in a circle in the figure), resulting in $\frac{8}{2} = 4$. When the same formula is evaluated at time instant 18, the portion of the trace considered contains both a left-open $(\phi_{|2}, \psi_{|5})$ pair and a right-open $(\phi_{|17}, \psi_{|19})$ one. The lhs of the comparison operator is evaluated using the values of counters $a_{\phi,\psi}$, $b_{\phi,\psi}$, and $h_{\phi,\psi}$ at time instants 19 and 5 (enclosed in a triangle in the figure); its value is $\frac{5}{1} = 5$. Now consider the formula $\mathfrak{D}^{12}_{\bowtie n}(\phi, \psi)$. When evaluated at time instant 15, it has a left-open pair $(\phi_{|2}, \psi_{|5})$. The values of the counters $a_{\phi,\psi}$, $b_{\phi,\psi}$, and $h_{\phi,\psi}$ considered to compute the lhs of the comparison operator are those at time instants 16 and 4 (enclosed in a square in the figure); the lhs evaluates to $\frac{5}{1} = 5$. If the same formula is evaluated at time instant 18, we find only a right-open pair $(\phi_{|17}, \psi_{|19})$. The lhs of the comparison operator is evaluated using the value of counters $a_{\phi,\psi}$ and $h_{\phi,\psi}$ considered at time instants 19 and 7 (enclosed in a hexagon in the figure); its value is $\frac{5}{1} = 5$.

3.3 Implementation

The translation described in the previous section has been implemented in Common Lisp as a plugin[3] of the ZOT verification toolset [105] translating SOLOIST formulae into CLTLB($\mathcal{D}$). ZOT supports satisfiability checking of CLTLB($\mathcal{D}$) formulae by means of SMT solvers. A plugin-based architecture makes it easy to extend ZOT to support more expressive languages using CLTLB($\mathcal{D}$) as a core, and to output code for the different dialects of various SMT solvers.

Figure 3.3 shows a relevant subset of the ZOT architecture (presented in its entirety in Section 2.3). The architecture is extended to support SOLOIST by introducing another layer called *soloist* that exposes the SOLOIST operators to be used in the layer above. For example formula $\mathsf{G}(p \rightarrow \mathfrak{C}^{5}_{>0}(q))$ can be written in the Zot script layer as:

```
(-G- (-> (-P- p) (-C- 5 > 0 (-P- q))))
```

As you can see, ZOT adopts prefix notation for all its operators and every atomic formula ϕ is denoted as `(-P- `ϕ`)`. The *soloist* layer also performs the translation of the operators it exposes, into CLTLB($\mathcal{D}$) and invokes the

[3] https://github.com/fm-polimi/zot

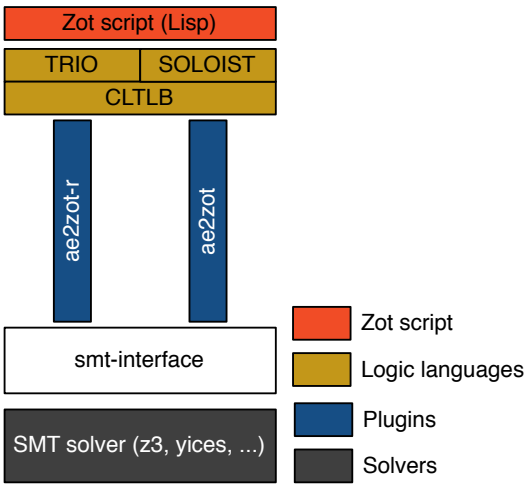


Figure 3.3: *The extended ZOT architecture*

cltlb layer below to perform satisfiability checking. The *cltlb* layer invokes the *ae2zot* plugin to translate the CLTLB($\mathcal{D}$) formula into an appropriate SMT dialect, then uses the *smt-interface* layer to invoke an SMT solver of choice and then parse its output.

We now give a rundown of the translation steps applied to an example, to provide a glimpse of the implementation of our SMT-based trace checking algorithm. These steps and the example are also sketched in Figure 3.4 where: the first row shows (a fragment of) the example input trace and the SOLOIST formula to verify on the trace; the second row shows how the input trace is transformed from timed ω-word to ω-word, the translation of the input formula and the definition of the counter constraints as described in Section 3.2.

Let us consider the problem of performing trace checking of the formula $\phi \equiv \mathfrak{C}^5_{<3}(p)$ over the trace H of length 7 depicted in Figure 3.4; the formula is evaluated at time instant 5. As described in Section 3.2.2, our plugin translates the SOLOIST formula ϕ into CLTLB($\mathcal{D}$) as $\rho(\phi) \equiv \mathsf{X}(c_p) - \mathsf{Y}^4(c_p) < 3$, where c_p is a counter. The behavior of this counter is constrained by the conjunction of axioms A1, A2, and A3, defined as $\mathcal{C}_{c_p} \equiv (c_p = 0) \wedge \mathsf{G}((e \wedge p) \rightarrow \mathsf{X}(c_p) = c_p + 1) \wedge \mathsf{G}((\neg e \vee \neg p) \rightarrow \mathsf{X}(c_p) = c_p)$. In the next step ZOT translates the input CLTLB($\mathcal{D}$) formula $\neg(\mathsf{X}^5(\rho(\phi))) \wedge \mathcal{C}_{c_p}$ to SMT dialect and calls the SMT solver with the final translated formula conjuncted with the trace as input. Notice that the formula ϕ is negated; hence, it is satisfied by trace H if the SMT solver returns *unsat*. The expo-

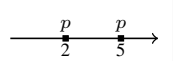
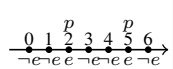
	Trace	Formula	Counter constraints
SOLOIST		$\mathfrak{C}^5_{<3}(p)$	n/a
CLTLB($\mathcal{D}$)		$\mathsf{X}(c_p) - \mathsf{Y}^4(c_p) < 3$	$(c_p = 0)$ (A1) $\wedge$ $\mathsf{G}((e \wedge p) \rightarrow \mathsf{X}(c_p) = c_p + 1)$ (A2) $\wedge$ $\mathsf{G}((\neg e \vee \neg p) \rightarrow \mathsf{X}(c_p) = c_p)$ (A3)

Figure 3.4: *Example of the translation from SOLOIST to CLTLB($\mathcal{D}$)*

nent 5 in the term $\mathsf{X}^5(\rho(\phi))$ is determined by the evaluation of the formula fixed at time instant 5. For the the translation from CLTLB($\mathcal{D}$) to the input language of the SMT solver, please refer to [105], as the details are out of the scope of this work.

3.4 Complexity

This section gives an estimate on the complexity of the translation of SOLOIST to CLTLB($\mathcal{D}$) and an intuition on the complexity of the satisfiability problem for SOLOIST.

Let us consider a SOLOIST formula ϕ of length $|\phi| = \lambda$ (see Section 2.2), trace T of length H and let μ be the maximum constant occurring in the SOLOIST formula ϕ and trace T. More precisely $\mu = \mathsf{max}\{\{i \mid i \in I_1 \cup \ldots \cup I_\lambda\} \cup \{K_1, \ldots K_\lambda\} \cup \{n_1 \ldots n_\lambda\} \cup \{h_1 \ldots h_\lambda\} \cup \{\tau_1, \ldots \tau_H\}\}$ where I_i represent all the intervals, K_i represent all the time windows, n_i represent all the bounds and h_i represent the observational subintervals occurring in the SOLOIST formula and τ_i are the timestamps. The size of the SOLOIST formula is then $\mathcal{O}(\lambda \mathsf{log}(\mu))$. We demonstrate that length of the CLTLB($\mathcal{D}$) formula obtained as a translation of ϕ is $\mathcal{O}(\lambda\mu)$, i.e., the translation is PSPACE in the length of the formula λ and EXPSPACE in the size of the constants used in the formula $\mathsf{log}(\mu)$.

The translation function ρ, introduces a constant length formula for every type of SOLOIST formula. Atomic formulae and boolean operators ($\neg$, $\wedge$ and $\vee$) are not changed. Temporal operators (U_I, S_I,R_I, T_I) are translated into single formula with additional conjunct and disjunct in the subformulae, however this is still constant length. Translation ρ introduces a counter c_ϕ for every atom ϕ occurring in a $\mathfrak{C}$ modality. In the worst case $\mathcal{O}(\lambda)$ counters can be introduced. Notice that subformulae occurring in aggregate modalities are restricted only to be atomic. The translation of the

$\mathfrak{C}$ includes both its CLTLB($\mathcal{D}$) translation and the additional axioms that define the behavior of the counter. However the length of these formulae is constant. We remark that we use a direct encoding of the exponent K in formulae of the form Y^K or X^K, both in the case of arithmetical temporal terms and of boolean formulae. The direct encoding of the exponent allows us to avoid expanding it into nested Y or X formulae and therefore the length of the translation of the $\mathfrak{C}$ modality is $\mathcal{O}(\lambda)$. Similar reasoning can be applied to the other modalities. For $\mathfrak{M}$ modality we reuse the translation of the $\mathfrak{C}$ modality $\lfloor \frac{K}{h} \rfloor$ times, therefore its translation is $\mathcal{O}(\lambda\mu)$.

The size of the trace after encoding the timing implicitly is $\mathcal{O}(\mu)$, as each time instant is going to be encoded as a separate position up to the maximum timestamp. Each position can contain all the atoms in the worst case. Encoding of the trace considers only atoms that occur in the SOLOIST formula, therefore at most λ of them. Therefore the complete size of the translated formula is still $\mathcal{O}(\mu \cdot \lambda)$.

Satisfiability of CLTLB($\mathcal{D}$) [26] is PSPACE in the length of the CLTLB($\mathcal{D}$) formula and EXPSPACE in the size of the constants used in the formula. Let μ_c be the largest constant occurring in the translated CLTLB($\mathcal{D}$) formula $\rho(\phi)$ and λ_c its length, i.e., $|\rho(\phi)| = \lambda_c$. According to the translation ρ we have that μ_c is $\mathcal{O}(\mu)$ and λ_c is $\mathcal{O}(\lambda \cdot \mu)$ therefore satisfiability of SOLOIST is EXPSPACE.

Finally, we claim that satisfiability of SOLOIST is EXPSPACE-complete. EXPSPACE hardness can be obtained by reducing the problem of satisfiability of MTL with point-based semantics (which is known to be EXPSPACE-complete [6]) to the SOLOIST satisfiability problem. Translation of any MTL formula to SOLOIST is trivial, since MTL is a strict fragment of SOLOIST.

CHAPTER 4

Decision procedure based on QF-EUFIDL

4.1 Overview

The translation of SOLOIST to CLTLB($\mathcal{D}$) paves the way for an efficient decision procedure for SOLOIST. A decision procedure for a language, in turn, allows for many different use cases including trace checking, as shown in Section 3.3. However, a drawback of the translation presented in the previous chapter is the way it handles timing information in the traces. Namely, when bridging the gap between the different models of CLTLB($\mathcal{D}$) and SOLOIST (ω-words and *timed* ω-words, respectively) the explicit value of a timestamp is encoded implicitly, as the position in the ω-word. For example, a timed ω word $w = (\{a\}, 1)((\{a\}, 3)(\{\}, 5)(\{a\}, 8))^\omega$ is encoded as $w' = \{a, e\}\{\}(\{a, e\}\{\}\{e\}\{\}\{\}\{a, e\})^\omega$, where atom e pinpoints the original positions from w and, hence, occurs at positions 1, 3, 5 and 8 in w'. This means that using the translation based on CLTLB($\mathcal{D}$) for trace checking a very *sparse* trace would, in the worst case, generate exponential number of positions with respect to the size of the maximal value of the timestamp in the trace. We say that a trace is *sparse* [30] if the number of time instants in which events occur[1] is much lower than the total time

[1] Also called *valid* time instants

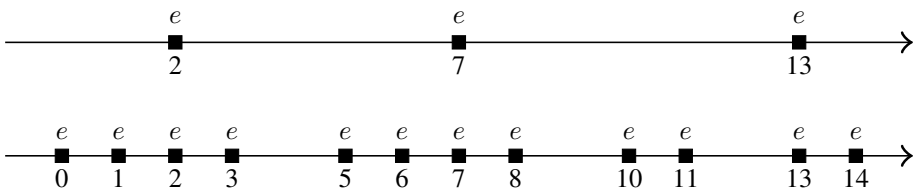


Figure 4.1: *Example of a sparse (above) and dense (below) trace modeled as timed words*

span of the trace (i.e. the difference between the last and the first timestamp). Figure 4.1 shows two example traces modeled as timed words over alphabet $\{e\}$. The first trace is considered sparse with respect to the second trace; it contains only 3 valid time instants (with timestamps 2, 7 and 13) and spans over a total of 15 time instants. Considering sparse traces is motivated by our experience in verification the interactions between service-based applications [34]. For example, the log used for the Business Process Intelligence Challenge 2012 (BPIC 2012) [116] was taken from a Dutch Financial Institute contains 13087 traces, whose average number of time instants in which events occur is 20.0347: this represents (on average) the 0.003% of the total number of time instants.

To solve this problem, we chose QF-EUFIDL as the target language for a new translation tailored to handle sparse traces more efficiently. The key idea behind the translation is to keep the representation of the SOLOIST model compact and adapt the translation of its operators to account for the timestamps in the model. The detailed translation of SOLOIST to QF-EUFIDL and its complexity is described in Section 4.2. Section 4.3 shows how the translation is incorporated into the ZOT's plugin architecture and Section 4.4 discusses the complexity of the translation. Finally, in Chapter 5 we show how the approach can be applied for trace checking properties of service compositions and we compare it to the translation based on CLTLB($\mathcal{D}$).

4.2 Translation

As shown in Section 2.2 SOLOIST can be seen as MTL over discrete time, enriched with aggregate modalities. The decision procedure for MTL over discrete time [105] can be efficiently performed by reducing semantics of U_I and S_I to suitable propositional formulae which take advantage from the information about the metric over time defined by I. In [105], however, authors consider ω-words as models for MTL formulae without timestamps.

Therefore, the temporal structure required to translate the semantics of a formula such as $\top \mathsf{U}_{[10,10]}\phi$ is at least as long as ten discrete positions, because no timing information is available from the model. In this paper, we devise a new way to represent information about timing constraints defined in metric temporal modalities (including the aggregate ones); this is an improvement on the method proposed in [105]. The encoding presented afterwards is an extension of the one defined in [25], which allows one to capture timed ω-words. As a consequence, models do not require as many discrete positions as needed to build the discrete temporal structure in [105], because the measure of time distances is realized through arithmetical variables that store how much time elapses among consecutive discrete positions. Intuitively, by adding an arithmetical variable $\tau \in \mathbb{N}$ measuring the elapsed time, formula $\top \mathsf{U}_{[10,10]}\phi$ holds at position i if, for instance, at position $i+1$, ϕ holds and the time τ elapsed between position i and position $i+1$ is equal to 10. To realize this counting mechanism with variables and arithmetical operators, we require a language that incorporates arithmetics, hence our choice of QF-EUFIDL as the target language of our encoding.

We use the QF-EUFIDL structure with $f^{\mathbb{Z}}$ containing only unary functions of the form $f : \mathbb{Z}_0^+ \to \mathbb{Z}$. Each function represents arithmetical variable used in the encoding. To simplify our presentation we introduce a set P containing boolean functions of the form $p : \mathbb{Z}_0^+ \to \{\top, \bot\}$; each of them represents a *predicate* whose value is defined over a nonnegative integer domain. Since QF-EUFIDL supports only atoms, the boolean functions are just a syntactic sugar for an enumeration of atoms. Using this QF-EUFIDL structure we can define a finite representation of models of SOLOIST formulae. Since our structure is ordered, let $0, 1, 2, \ldots, H$ be a finite linear order, with H corresponding to the length of the finite prefix of the timed ω-word satisfying a SOLOIST formula. The linear order represents a temporal structure and since it is a subset of the domain of both the predicates from P and the functions from F, we can interpret them as having "time dependent" values and hence they can model boolean and arithmetical entities that change their values over time.

In the encoding, we use the notation $[\![X]\!]$ to denote any predicate in P representing a boolean entity X. We denote with $|X|$ an arithmetical variable in F representing an arithmetical entity X. We use $[\![X]\!]_i$ and $|X|_i$ as a shorthand for $[\![X]\!](i)$ and $|X|(i)$, respectively. The truth of $[\![X]\!]_i$ is interpreted as entity X holding at time instant i in an execution trace (or, equivalently, a timed word).

We assume SOLOIST formulae to be in *positive normal form* (PNF). The PNF of a formula is an equivalent formula where negation may only

occur on atoms, i.e., atomic propositions (see [104]). PNF can be obtained by propagating the negation towards the atoms, by means of converting a negated operator into its dual version and negating its operand(s). To do so, we introduce the connective $\vee$, dual of $\wedge$, as well as the dual versions of all temporal modalities. The dual of U_I is "*Release*" R_I: $\phi \mathsf{R}_I \psi \equiv \neg(\neg\phi \mathsf{U}_I \neg\psi)$; the dual of S_I is "*Trigger*" T_I: $\phi \mathsf{T}_I \psi \equiv \neg(\neg\phi \mathsf{S}_I \neg\psi)$[2]. A negation in front of one of the $\mathfrak{C}^K_{\bowtie n}, \mathfrak{U}^{K,h}_{\bowtie n}, \mathfrak{M}^{K,h}_{\bowtie n}, \mathfrak{D}^K_{\bowtie n}$ modalities becomes a negation of the relation denoted by the $\bowtie$ symbol, hence no dual version is needed for them.

Let Φ be a SOLOIST formula in PNF. Its encoding is a set of QF-EUFIDL constraints over the predicates from P and functions from F. We introduce a predicate $[\![\varphi]\!]$ for each subformula φ of Φ.

4.2.1 Translation of boolean and temporal formulae

We first define the constraints for timing information. As defined in Sect. 2, the temporal structure contains an integer timestamp. An arithmetical variable $|\tau|$ denotes the absolute time at positions $i = 0 \ldots H$. Let $\mathcal{C}_{time}$ be the conjunction of the following constraints:

Position i	Timing information	Description
$0 \ldots H-1$	$\lvert\tau\rvert_i < \lvert\tau\rvert_{i+1}$	strict monotonicity

(4.1)

Next, we define constraints for atomic propositions and propositional operators; their conjunction is denoted as $\mathcal{C}_{prop}$ (where $\leftrightarrow$ stands for a double implication):

Position i	Propositional operators	Description
$0 \ldots H$	$[\![p]\!]_i \leftrightarrow p(i)$	atomic propositions
$0 \ldots H$	$[\![\neg p]\!]_i \leftrightarrow \neg p(i)$	negation
$0 \ldots H$	$[\![\phi \wedge \psi]\!]_i \leftrightarrow [\![\phi]\!]_i \wedge [\![\psi]\!]_i$	conjunction

(4.2)

Notice that for any sub-formula of the form $\phi \wedge \psi$ in a SOLOIST formula Φ we add in the resulting encoding, instances of formulae from the third row of (4.2). This encoding completely conforms to the one in [38].

As for the modality U_I, we add to the encoding, for any subformula of

[2]Note that the strict semantics of U_I and S_I preserve the duality of R_I and T_I also on finite words.

the form[3] $\phi \mathsf{U}_{(a,b)} \psi$ in Φ, the following formulae, denoted as $\mathcal{C}_{temp-until}$:

Position i	Temporal operator	Description
$0 \dots H-1$	$[\![\phi \mathsf{U}_{(a,b)} \psi]\!]_i \leftrightarrow \bigvee_{k=i+1}^{H} ([\![\psi]\!]_k \wedge$ $a < \lvert\tau\rvert_k - \lvert\tau\rvert_i \wedge \lvert\tau\rvert_k - \lvert\tau\rvert_i < b \wedge \bigwedge_{p=i+1}^{k-1} [\![\phi]\!]_p)$	"Until"
H	$[\![\phi \mathsf{U}_{(a,b)} \psi]\!]_H \leftrightarrow \bot$	"Until" at position H

(4.3)

This is a straightforward encoding of the semantics of the "*Until*" operator. The disjunction in the first row represents a case split on all possible future time instants with respect to i. For each such time instant k a conjunction is created with $[\![\psi]\!]_k$ stating that ψ subformula has to hold at time instant k; moreover, ϕ needs to hold in all instants from $i+1$ to $k-1$, i.e., $\bigwedge_{p=i+1}^{k-1} [\![\phi]\!]_p$. Formula $(a < |\tau|_k - |\tau|_i) \wedge (|\tau|_k - |\tau|_i < b)$ enforces the timing constraint of the $\mathsf{U}_{(a,b)}$ modality, i.e., if $\tau_k - \tau_i \in (a, b)$.

The case for the S_I modality is similar to the above. For any sub-formula of the form S_I in Φ we add to the encoding the following formulae, denoted as $\mathcal{C}_{temp-since}$:

Position i	Temporal operator	Description
0	$[\![\phi \mathsf{S}_{(a,b)} \psi]\!]_0 \leftrightarrow \bot$	"Since" at position 0
$1 \dots H$	$[\![\phi \mathsf{S}_{(a,b)} \psi]\!]_i \leftrightarrow \bigvee_{k=0}^{i-1} ([\![\psi]\!]_k \wedge$ $a < \lvert\tau\rvert_i - \lvert\tau\rvert_k \wedge \lvert\tau\rvert_i - \lvert\tau\rvert_k < b \wedge \bigwedge_{p=k+1}^{i-1} [\![\phi]\!]_p)$	"Since"

(4.4)

The conjunction of all formulae from $\mathcal{C}_{temp-until}$ and $\mathcal{C}_{temp-since}$ is denoted as $\mathcal{C}_{temp}$.

4.2.2 Translation of the $\mathfrak{C}$ modality

The $\mathfrak{C}$ modality expresses a bound on the number of occurrences of a certain event in a given time window; in the encoding, it comes natural to use arithmetical variables as counters of the events. For each subformula of the form $\mathfrak{C}^K_{\bowtie n}(\phi)$, we add an arithmetical variable $|c_\phi|$ to F, constrained with the following formulae:

Position i	$\mathfrak{C}$ modality constraints	Description
0	$\lvert c_\phi\rvert_0 = 0$	initialization
$0 \dots H-1$	$[\![\phi]\!]_i \rightarrow (\lvert c_\phi\rvert_{i+1} = (\lvert c_\phi\rvert_i + 1))$	ϕ occurs at i
$0 \dots H-1$	$\neg[\![\phi]\!]_i \rightarrow (\lvert c_\phi\rvert_{i+1} = \lvert c_\phi\rvert_i)$	ϕ does not occur at i

(4.5)

The constraint in the first row initializes the arithmetical variable to zero at time instant 0. The following H constraints (in the second row) force

[3] A closed interval $[a, b]$ over $\mathbb{N}$ can be expressed as an open one of the form $(a-1, b+1)$.

$|c_\phi|$ to increase by 1 at time instant $i+1$, if ϕ occurs at time instant i. The last H constraints from the third row refer to the opposite situation: when there is no occurrence of the event ϕ at time instant i, the value of $|c_\phi|_{i+1}$ is constrained to have the same value as $|c_\phi|_i$. Let us denote, for a $\mathfrak{C}$ modality that has ϕ as a sub-formula, the conjunction of these constraints as $\mathcal{C}_{c-cons}(\phi)$. Besides $\mathcal{C}_{c-cons}(\phi)$, we add to the encoding, for each $i = 0 \dots H-1$, the following constraints, denoted as $\mathcal{C}_{c-form}(\phi)$:

$$[\![\mathfrak{C}^K_{\bowtie n}(\phi)]\!]_i \leftrightarrow \bigvee_{z=0}^{min\{i,K\}} |c_\phi|_{i+1} - |c_\phi|_{i-z} \bowtie n \wedge |\tau|_i - |\tau|_{i-z-1} \geq K \wedge |\tau|_i - |\tau|_{i-z} < K$$

This formula characterizes each time instant i of the temporal structure in which the $\mathfrak{C}$ modality is true. We have presented a simplified version of the formula here, therefore any value of arithmetical variable before position 0 is considered to be 0. The disjunction is a case split for each position z in the past with respect to the current position i. Notice that, if $K > i$ we need to consider all previous positions in the temporal structure; otherwise, it is enough to consider K previous time instants, since in the worst case all timestamps can increase by at least one. Each case is a conjunction where sub-formula $|\tau|_i - |\tau|_{i-z-1} \geq K \wedge |\tau|_i - |\tau|_{i-z} < K$ determines the correct position on the left side of the time window, while $|c_\phi|_{i+1} - |c_\phi|_{i-z} \bowtie n$ checks that the $\mathfrak{C}$ modality holds in the considered time window.

4.2.3 Translation of the $\mathfrak{U}$ modality

To simplify the presentation, we express the $\mathfrak{U}$ modality in terms of the $\mathfrak{C}$ one, based on this definition: $\mathfrak{U}^{K,h}_{\bowtie n}(\phi) \equiv \mathfrak{C}^{\lfloor \frac{K}{h} \rfloor \cdot h}_{\bowtie n \cdot \lfloor \frac{K}{h} \rfloor}(\phi)$, which can be derived from the semantics in Fig. 2.2. Therefore, we can reuse the translation of $\mathfrak{C}$ modality to encode the semantics of $\mathfrak{U}$ modality into QF-EUFIDL.

4.2.4 Translation of the $\mathfrak{M}$ modality

As for the $\mathfrak{M}$ modality, for each subformula of the form $\mathfrak{M}^{K,h}_{\bowtie n}(\phi)$, we introduce the same arithmetical variable $|c_\phi|$ and the constraint $\mathcal{C}_{c-cons}(\phi)$ (now denoted $\mathcal{C}_{m-cons}(\phi)$) as for the $\mathfrak{C}$ modality. Additionally, we add arithmetical variables $|p_0| \dots |p_{\lfloor \frac{K}{h} \rfloor + 1}|$ to the set F for each $\mathfrak{M}$ modality sub-formula of Φ. The encoding of the $\mathfrak{M}$ modality depends on the operator $\bowtie$; for example, when the comparison operator is "$<$" we have the following constraints, denoted $\mathcal{C}_{m-form}(\phi)$:

$$[\![\mathfrak{M}^{K,h}_{<n}(\phi)]\!]_i \leftrightarrow \bigwedge_{y=0}^{\lfloor \frac{K}{h} \rfloor} \Big(\bigvee_{z=0}^{min\{i,h\cdot(y+1)\}} (|p_{y+1}|_i = |c_\phi|_{i+1} - |c_\phi|_{i-z} \wedge |p_{y+1}|_i - |p_y|_i < n \wedge |\tau|_i - |\tau|_{i-z-1} > (y+1)\cdot h \wedge |\tau|_i - |\tau|_{i-z} \leq (y+1)\cdot h) \Big) \wedge |p_0|_i = 0$$

In this formula, in each conjunct y we perform a case split, similar to the case of the $\mathfrak{C}$ modality, but with a different time window: $(y+1) \cdot h$. We assign the result of counting to the variable $|p_{y+1}|$ in each conjunct. Therefore, values $|p_0|_i \dots |p_{\lfloor \frac{K}{h} \rfloor + 1}|_i$ contain the number of occurrences of ϕ in time windows $0, h, 2h, \dots, \lfloor \frac{K}{h} \rfloor \cdot h, K$ with respect to position i, respectively. With subformula $|p_{y+1}|_i - |p_y|_i < n$, we check that in each observation subinterval with respect to i there is a bounded number of occurrences. The other cases of $\bowtie$ can be defined in a similar way.

4.2.5 Translation of the $\mathfrak{D}$ modality

The $\mathfrak{D}$ modality expresses a bound on the average distance between the occurrences of a pair of events in a given time window. Since events can occur multiple times in the temporal structure, a pair of events (ϕ, ψ) may have multiple instances. We call a pair of the form $([\![\phi]\!]_i, [\![\psi]\!]_j)$ *an instance* if there is an occurrence of event ϕ at time instant i and an occurrence of event ψ at time instant j, with $i < j$. We call such instance *open* at time instant q if $i \leq q < j$. Otherwise, the instance is *closed* at time instant q. The *distance* of a closed pair instance is $j - i$; for an open pair at time instant q, the distance is $q - i$. A time window defined for a $\mathfrak{D}^K_{\bowtie n}(\phi, \psi)$ (sub-)formula evaluated at time instant q is bounded by the time instants $q + 1$ and $q - K + 1$. It has a *left-open* (respectively, *right-open*) pair in position q of a temporal structure, if there is an open instance of (ϕ, ψ) at time instant $q - K + 1$ (respectively, $q + 1$). Depending on whether a $\mathfrak{D}$ modality (sub-)formula contains either (left- and/ or right-) open pairs or none, there are four distinct cases to take into account for the encoding.

For each subformula of the form $\mathfrak{D}^K_{\bowtie n}(\phi, \psi)$, we add to F five arithmetical variables:

- $|g_{\phi,\psi}|$: it assumes value 1 in the time instants following an occurrence of ϕ and is reset to 0 after an occurrence of ψ. It acts as a flag denoting the time instants during which the event pair instance is open.
- $|h_{\phi,\psi}|$: in each time instant, it contains the number of previously seen closed pair instances. It is increased after every occurrence of ψ.
- $|s_{\phi,\psi}|$: At each time instant, its value corresponds to the sum of distances of all previously occurred pair instances. It is increased after every time instant when either $|g_{\phi,\psi}|$ is 1 or ϕ holds.
- $|a_{\phi,\psi}|$: it keeps track of the sum of the distances of all previously occurred closed pair instances.

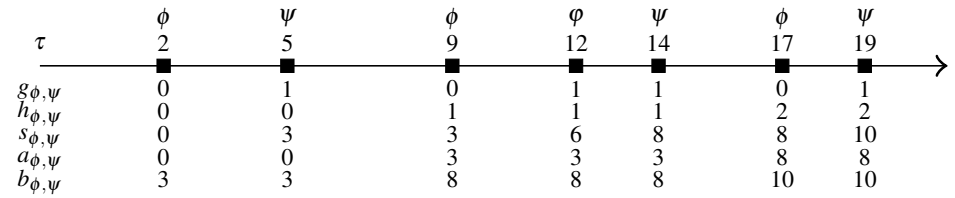


Figure 4.2: *Example of trace for the $\mathfrak{D}$ modality, with the corresponding arithmetical variables used in the encoding*

- $|b_{\phi,\psi}|$: it has the values that will be assumed by variable $|s_{\phi,\psi}|$ at the next occurrence of ψ (more details below).

Variables $|a_{\phi,\psi}|$, $|b_{\phi,\psi}|$, and $|h_{\phi,\psi}|$ are directly used in the encoding of the $\mathfrak{D}$ modality (sub-)formulae, while variables $|g_{\phi,\psi}|$ and $|s_{\phi,\psi}|$ are helper variables, used to determine the values of the other variables. Figure 4.2 shows a portion of a trace and the values assumed by these variables: the uppermost row shows instants where *atoms* ϕ, ψ, and φ hold; the second row shows the value of $|\tau|$ at each time instant; the other rows show the values of the variables at each time instant.
For each $\mathfrak{D}^K_{\bowtie n}(\phi, \psi)$ modality sub-formula we define the set of constraints $\mathcal{C}_{d-cons}(\phi, \psi)$:

Position i	$\mathfrak{D}$ modality constraints	Description
0	$\lvert g_{\phi,\psi}\rvert_0 = 0 \wedge \lvert h_{\phi,\psi}\rvert_0 = 0 \wedge \lvert a_{\phi,\psi}\rvert_0 = 0 \wedge \lvert s_{\phi,\psi}\rvert_0 = 0$	variable initialization
0	$[\![B_{eq}]\!]_0$	$\lvert b_{\phi,\psi}\rvert$ initialization
$0 \ldots H-1$	$[\![\phi]\!]_i \rightarrow (\lvert g_{\phi,\psi}\rvert_{i+1} = 1 \wedge \lvert s_{\phi,\psi}\rvert_{i+1} = \lvert s_{\phi,\psi}\rvert_i + (\lvert\tau\rvert_{i+1} - \lvert\tau\rvert_i) \wedge \lvert h_{\phi,\psi}\rvert_{i+1} = \lvert h_{\phi,\psi}\rvert_i \wedge \lvert a_{\phi,\psi}\rvert_{i+1} = \lvert a_{\phi,\psi}\rvert_i)$	ϕ occurs at i
$0 \ldots H-1$	$[\![\psi]\!]_i \rightarrow (\lvert g_{\phi,\psi}\rvert_{i+1} = 0 \wedge \lvert h_{\phi,\psi}\rvert_{i+1} = \lvert h_{\phi,\psi}\rvert_i + 1 \wedge \lvert a_{\phi,\psi}\rvert_{i+1} = \lvert s_{\phi,\psi}\rvert_i \wedge \lvert s_{\phi,\psi}\rvert_{i+1} = \lvert s_{\phi,\psi}\rvert_i \wedge \lvert b_{\phi,\psi}\rvert_i = \lvert s_{\phi,\psi}\rvert_i \wedge [\![B_{eq}]\!]_{i+1})$	ψ occurs at i
$0 \ldots H-1$	$\neg[\![\phi]\!]_i \wedge \neg[\![\psi]\!]_i \rightarrow (\lvert g_{\phi,\psi}\rvert_{i+1} = \lvert g_{\phi,\psi}\rvert_i \wedge \lvert h_{\phi,\psi}\rvert_{i+1} = \lvert h_{\phi,\psi}\rvert_i \wedge \lvert a_{\phi,\psi}\rvert_{i+1} = \lvert a_{\phi,\psi}\rvert_i \wedge (\lvert g_{\phi,\psi}\rvert_i = 1 \rightarrow \lvert s_{\phi,\psi}\rvert_{i+1} = \lvert s_{\phi,\psi}\rvert_i + (\lvert\tau\rvert_{i+1} - \lvert\tau\rvert_i)) \wedge \lvert g_{\phi,\psi}\rvert_i = 0 \rightarrow \lvert s_{\phi,\psi}\rvert_{i+1} = \lvert s_{\phi,\psi}\rvert_i))$	neither ϕ nor ψ occurs at i

(4.6)

The formula in the first row of (4.6) initializes all variables at time instant 0 except $|b_{\phi,\psi}|$. In the second row we introduce a new predicate $[\![B_{eq}]\!]$; it has

the following constraints:

Position i	$[\![B_{eq}]\!]$ predicate constraints	Description
$0 \dots H-1$	$[\![B_{eq}]\!]_i \leftrightarrow [\![\psi]\!]_i \vee ((\lvert b_{\phi,\psi}\rvert_{i+1} = \lvert b_{\phi,\psi}\rvert_i) \wedge [\![B_{eq}]\!]_{i+1})$	propagation of value of $\lvert b_{\phi,\psi}\rvert$
H	$[\![B_{eq}]\!]_H \leftrightarrow \top$	last state constraint

(4.7)

These constraints force the values of the variables $\lvert b_{\phi,\psi}\rvert_i$ to stay the same in all the consecutive time instants until the first occurrence of ψ or until the end of the trace; the second constraint in (4.7) deals with traces without occurrences of ψ.

The third constraint in (4.6) determines the value of variables in the next time instant, upon occurrence of an event ϕ at time instant i. Variable $\lvert g_{\phi,\psi}\rvert_{i+1}$ is set to 1; variable $\lvert s_{\phi,\psi}\rvert_{i+1}$ is incremented by $\lvert\tau\rvert_{i+1} - \lvert\tau\rvert_i$ with respect to value of the variable $\lvert s_{\phi,\psi}\rvert_i$; variables $\lvert h_{\phi,\psi}\rvert_{i+1}$ and $\lvert a_{\phi,\psi}\rvert_{i+1}$ are constrained not to change with respect to value of their counterparts at time instant i. The fourth constraint determines how the variables are updated when an event ψ occurs at time instant i: variable $\lvert g_{\phi,\psi}\rvert_{i+1}$ is set to 0; variables $\lvert b_{\phi,\psi}\rvert_i$, $\lvert a_{\phi,\psi}\rvert_{i+1}$, and $\lvert s_{\phi,\psi}\rvert_{i+1}$ are set to be equal to $\lvert s_{\phi,\psi}\rvert_i$. Moreover, $[\![B_{eq}]\!]_{i+1}$ is constrained to hold, forcing values of $\lvert b_{\phi,\psi}\rvert_j$ to stay the same in all the consecutive time instants $j > i$, until the next occurrence of ψ. The constraints in the fifth row of (4.6) cover the cases when neither ϕ nor ψ occur at time instant i. In these cases the values of variables $\lvert g_{\phi,\psi}\rvert_{i+1}$, $\lvert h_{\phi,\psi}\rvert_{i+1}$, and $\lvert a_{\phi,\psi}\rvert_{i+1}$ are constrained to have the same value as in their counterparts at i, variable $\lvert b_{\phi,\psi}\rvert_{i+1}$ is unconstrained, while for $\lvert s_{\phi,\psi}\rvert_{i+1}$ we need to distinguish two separate cases. If the last event of the pair is ϕ (denoted by $\lvert g_{\phi,\psi}\rvert_i = 1$), then value of $\lvert s_{\phi,\psi}\rvert_{i+1}$ is $\lvert s_{\phi,\psi}\rvert_i$ incremented by $\lvert\tau\rvert_{i+1} - \lvert\tau\rvert_i$, otherwise it is just $\lvert s_{\phi,\psi}\rvert_i$.

For any sub-formula of the form $\mathfrak{D}^K_{\bowtie n}\{(\phi,\psi)\}$ evaluated at time instant i, we add to the encoding the constraint $\mathcal{C}_{d-form}(\phi,\psi)$:

$$
\begin{aligned}
[\![\mathfrak{D}^K_{\bowtie n}(\phi,\psi)]\!]_i \leftrightarrow \quad & \bigvee_{z=0}^{min\{i,K\}} \big((\texttt{if}^4\ (\lvert g_{\phi,\psi}\rvert_{i-z} = 1) \quad \texttt{then}\ (\tfrac{\lvert a_{\phi,\psi}\rvert_{i+1} - \lvert b_{\phi,\psi}\rvert_{i-z}}{\lvert h_{\phi,\psi}\rvert_{i+1} - \lvert h_{\phi,\psi}\rvert_{i-z-1}} \bowtie n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ (\tfrac{\lvert a_{\phi,\psi}\rvert_{i+1} - \lvert a_{\phi,\psi}\rvert_{i-z}}{\lvert h_{\phi,\psi}\rvert_{i+1} - \lvert h_{\phi,\psi}\rvert_{i-z}} \bowtie n)) \\
& \wedge \lvert\tau\rvert_i - \lvert\tau\rvert_{i-z-1} \geq K \wedge \lvert\tau\rvert_i - \lvert\tau\rvert_{i-z} < K\big)
\end{aligned}
$$

In the above formula, the outer disjunction considers all positions that are z time instants in the past with respect to i (i.e., $i - z$) and checks, for each of them, if they fit into the time window using the $\lvert\tau\rvert_i - \lvert\tau\rvert_{i-z-1} \geq K \wedge \lvert\tau\rvert_i - \lvert\tau\rvert_{i-z} < K$ formula. If one position does, the rest of the formula considers whether there is an *open* (ϕ,ψ) pair instance at that position

[4] "$\texttt{if}\ A\ \texttt{then}\ B\ \texttt{else}\ C$" can be written as $(A \wedge B) \vee (\neg A \wedge C)$

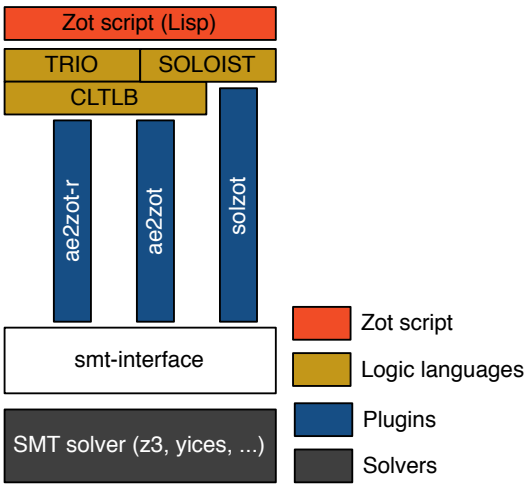


Figure 4.3: *The extended ZOT architecture*

which is captured by the $|g_{\phi,\psi}|_{i-z} = 1$ formula. In such a case, we compute the total delay between all pair instances within the time window by subtracting variable $|b_{\phi,\psi}|$ from $|a_{\phi,\psi}|$ at the appropriate positions. Since the value of $|b_{\phi,\psi}|$ at each position contains the value of $|s_{\phi,\psi}|$ at the position of the next occurrence of ψ, we effectively ignore the delay of the left-open pair. Otherwise, we use variable $|a_{\phi,\psi}|$, since it contains the delay from the last *closed* pair instance. Fractions in this formula are used for the sake of clarity, however the actual formula conforms to IDL due to the fact that n is a constant and $\frac{A}{B} = n$ can be written as $A = \underbrace{B + B + \ldots + B}_{\text{n times}}$.

The final QF-EUFIDL formula obtained from the encoding of the input SOLOIST formula Φ is the following conjunction of (possibly empty) formulae, which is supplied to the SMT solver: $[\![\Phi]\!]_0 \wedge \mathcal{C}_{time} \wedge \mathcal{C}_{prop} \wedge \mathcal{C}_{temp} \wedge \mathcal{C}_c \wedge \mathcal{C}_m \wedge \mathcal{C}_d$, where $\mathcal{C}_c \leftrightarrow \mathcal{C}_{c-cons} \wedge \mathcal{C}_{c-form}$, $\mathcal{C}_m \leftrightarrow \mathcal{C}_{m-cons} \wedge \mathcal{C}_{m-form}$ and $\mathcal{C}_d \leftrightarrow \mathcal{C}_{d-cons} \wedge \mathcal{C}_{d-form}$.

4.3 Implementation

The translation presented in Section 4.2 has been implemented in Common Lisp as another plugin[5] of the ZOT verification toolset [105]. By virtue of the new plugin ZOT can translate SOLOIST formulae directly to QF-EUFIDL.

[5] https://github.com/fm-polimi/zot

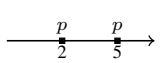
	Trace	Formula	Counter constraints
SOLOIST		$\mathfrak{C}^5_{<3}(p)$	n/a
QF-EUFIDL	Trace: $[\![p]\!]_0 \wedge \lvert\tau\rvert_0 = 2 \wedge$ $[\![p]\!]_1 \wedge \lvert\tau\rvert_1 = 5$ Timing information: $\lvert\tau\rvert_0 < \lvert\tau\rvert_1 \wedge$ $\lvert\tau\rvert_1 < \lvert\tau\rvert_2$	$[\![\mathfrak{C}^5_{<3}(p)]\!]_0 \leftrightarrow \lvert c_p\rvert_1 - \lvert c_p\rvert_0 < 3 \wedge$ $\lvert\tau\rvert_0 \geq 5 \wedge$ $[\![\mathfrak{C}^5_{<3}(p)]\!]_1 \leftrightarrow (\lvert c_p\rvert_2 - \lvert c_p\rvert_0 < 3 \wedge$ $\lvert\tau\rvert_1 \geq 5 \wedge \lvert\tau\rvert_1 - \lvert\tau\rvert_0 < 5) \vee$ $(\lvert c_p\rvert_2 - \lvert c_p\rvert_1 < 3 \wedge \lvert\tau\rvert_1 - \lvert\tau\rvert_0 \geq 5)$	$\lvert c_p\rvert_0 = 0 \wedge$ $[\![p]\!]_0 \rightarrow (\lvert c_p\rvert_1 = (\lvert c_p\rvert_0 + 1)) \wedge$ $[\![p]\!]_1 \rightarrow (\lvert c_p\rvert_2 = (\lvert c_p\rvert_1 + 1)) \wedge$ $\neg[\![p]\!]_0 \rightarrow (\lvert c_p\rvert_1 = \lvert c_p\rvert_0) \wedge$ $\neg[\![p]\!]_1 \rightarrow (\lvert c_p\rvert_2 = \lvert c_p\rvert_1) \wedge$

Figure 4.4: *Example of the translation from SOLOIST to QF-EUFIDL*

Figure 4.3 shows the part of the ZOT architecture affected by the extension. We have introduced a plugin called *solzot*, to facilitate the direct translation from SOLOIST to QF-EUFIDL. The plugin is used only by the *soloist* layer and it relies on the functionality provided by *smt-interface* layer to invoke an SMT solver of choice and parse its output.

We now provide an example of the translation applied to the same example from Section 4.3, to provide an intuition of its implementation and our SMT-based trace checking algorithm. Figure 4.4 shows an example trace and a SOLOIST formula to verify, in the first row; the second row shows how the input trace is encoded in QF-EUFIDL with timing information constraints, the translation of the SOLOIST formula $\mathcal{C}_{c-form}$ and the constraints on the arithmetical variable $|c_p|$ labeled as $\mathcal{C}_{c-cons}$ in Section 4.2.2.

In the example we want to trace check formula $\mathfrak{C}^5_{<3}(p)$ at time instant 5 of the trace that has only two positions ($H = 2$). As you can see in the second row, the trace is represented as a formula constraining the values of the arithmetical variable $|\tau|$ at positions 0 to $H - 1$. The strict monotonicity is also enforced using the timing information constraints. For the $\mathfrak{C}$ modality we need to introduce one arithmetical variable $|c_p|$. The constraints defining the behavior of $|c_p|$ are in the rightmost cell of the second row. The second that the third axiom are introduced for every position from 0 to $H - 1$.

The $\mathfrak{C}$ modality formula itself is translated into series of constraints for every position in the trace. At the position 0 the predicate $[\![\mathfrak{C}^5_{<3}(p)]\!]_0$ is true iff there is less than 3 occurrences of the event p at position 0 (calculated by the term $|c_p|_1 - |c_p|_0$) and the time window of the $\mathfrak{C}$ modality refers to the actual instants in the trace (captured by formula $|\tau|_0 \geq 5$). The modality does not hold at position 0 since $|\tau|_0$ is less than 5. At position 1 we have two disjunctive cases: if the leftmost point of the time window of the formula is between positions 0 and 1 or before position 0. In the

first case the formula $|\tau|_1 \geq 5 \wedge |\tau|_1 - |\tau|_0 < 5$ characterizes that the leftmost point of the time window is between positions 0 and 1. Formula $|c_p|_2 - |c_p|_1 < 3$ states that the time window contains less than 3 occurrences of the event p. The second case is similar to the formula for position 0 only applied for position 1. The final formula checked by the SMT solver is the conjunction of the constraints from the second row with the trace and the formula $\neg[\![\mathfrak{C}^5_{<3}(p)]\!]_1$ that evaluates the formula at time instant 5 and make the SMT solver produce *unsat* verdict if the trace satisfies the formula.

4.4 Complexity

We provide an estimation of the size of the QF-EUFIDL formula corresponding to a temporal or aggregating modality of SOLOIST. Although the syntactic complexity of the translation is already known in the case of standard LTL temporal modalities (e.g., [38]), we still provide a measure for U_I and S_I, since we rely on an ad-hoc encoding.

Let us consider first $\phi \mathsf{U}_I \psi$; the case for $\phi \mathsf{S}_I \psi$ is similar. At position $0 \leq i \leq H$, the formula in (4.3) has size $\mathcal{O}(H-i)^2$. We have then $\sum_{i=0}^{H} \mathcal{O}(H-i)^2 < \mathcal{O}(H^3)$.

Let μ be the maximum constant occurring in the SOLOIST formula and in the trace. One variable $|c_\phi|$ is required for all formulae $\mathfrak{C}^K_{\bowtie n}(\phi)$ with the same argument ϕ. In the worst case, we introduce one variable for each one. At position $0 \leq i \leq H$, formula $[\![\mathfrak{C}^K_{\bowtie n}(\phi)]\!]_i$ has size $\mathcal{O}(i)$. We have then $\sum_{i=0}^{H} \mathcal{O}(\log(\mu) i) < \mathcal{O}(\log(\mu) H^2)$. The $\mathfrak{U}$ modality is defined through $\mathfrak{C}$ and, therefore, inherits the same syntactic complexity.

Encoding of formula $\mathfrak{M}^{K,h}_{\bowtie n}(\phi)$ requires one variable $|c_\phi|$. We can reuse variable c_ϕ if in the original SOLOIST formula there are $\mathfrak{M}$ formulae or $\mathfrak{C}$ formulae with the same argument ϕ. Moreover, for each $\mathfrak{M}$ we need also $\lfloor \frac{K}{h} \rfloor + 2$ arithmetical variables $|p_0| \ldots |p_{\lfloor \frac{K}{h} \rfloor + 1}|$. In the worst case, we introduce $\lfloor \frac{K}{h} \rfloor + 3$ variables for each formula $\mathfrak{M}^{K,h}_{\bowtie n}(\phi)$. At position $0 \leq i \leq H$, formula $[\![\mathfrak{M}^{K,h}_{\bowtie n}(\phi)]\!]_i$ has size $\mathcal{O}(\log(\mu) \frac{K}{h} \cdot i)$. We have then $\sum_{i=0}^{H} \mathcal{O}(\log(\mu) \frac{K}{h} i) < \mathcal{O}(\log(\mu) \frac{K}{h} H^2)$.

The set of formulae translating $\mathfrak{D}$ is defined by the conjunction of formulae in (4.6) and (4.7) in addition to constraint $\mathcal{C}_{d-form}$. For each formula $\mathfrak{D}$ we introduce five variables related to the pair (ϕ, ψ). The size of formulae in (4.6) and in (4.7) is $\mathcal{O}(H)$. Constraint $\mathcal{C}_{d-form}$ requires a more careful analysis; notice that its size depends on the parameter n because of the way formula $\frac{a}{b} < n$ is expanded. At position $0 \leq i \leq H$, formula $[\![\mathfrak{D}^K_{\bowtie n}(\phi, \psi)]\!]_i$ has size $\mathcal{O}(\log(\mu) i n)$. Then, the complexity for $\mathfrak{D}$ is obtained

by $\sum_{i=0}^{H} \mathcal{O}(\log(\mu)in) < \mathcal{O}(\log(\mu)nH^2)$.

The size of the QF-EUFIDL encoding of a SOLOIST formula of length λ is $\mathcal{O}(\lambda \log(\mu)(H^3 + \frac{K}{h}H^2 + nH^2))$, as the number of sub-formulae is polynomial in λ, whereas the size of the encoding of a trace is $\mathcal{O}(\log(\mu)H)$. In the worst-case for $K = \mu, n = \mu$ and $h = 1$, the overall size of the QF-EUFIDL formula encoding a SOLOIST formula is bounded by $\mathcal{O}(\lambda \log(\mu)\mu H^3)$.

Given that the initial formula size is in $\mathcal{O}(\lambda \log(\mu))$ the encoding is polynomial in length of the trace H, linear in the length of the formula λ and exponential in the size of the constants occurring in the formula μ. Note that, as we stated in Section 2.4, the complexity of the satisfiability problem for QF-EUFIDL is NP-complete. Therefore the overall complexity of the satisfiability problem of SOLOIST is NEXPSPACE, since we check the encoded QF-EUFIDL formula of exponential size with respect to the initial SOLOIST formula. According to the Savitch's theorem [111] a NEXPSPACE problem is equivalent to an EXPSPACE problem, therefore we obtain same complexity results for SOLOIST satisfiability (EXPSPACE-complete) as reported in Section 3.4, since hardness can again be demonstrated by reducing satisfiability of MTL with point-based semantics [6] to satisfiability of SOLOIST.

CHAPTER 5

Evaluation & Application

5.1 Overview

We evaluated the effectiveness of our approach by investigating the following research questions:

RQ1: *Can the proposed trace checking procedures, based on CLTLB($\mathcal{D}$) and QF-EUFIDL, handle traces more efficiently than the procedure [85] based on the translation [36] to LTL?* (Section 5.2)

RQ2: *How do the proposed trace checking procedures scale with respect to the various parameters (e.g., the length of the trace, the length of the time window K) involved in SOLOIST trace checking?* (Section 5.3)

RQ3: *How do the two proposed trace checking procedures compare?* (Section 5.4)

RQ4: *Can our approach to trace checking be applied in a real setting?* (Sections 5.5, 5.6 and Chapter 6)

All traces used for the evaluation were synthesized with the Process Log Generator (PLG) tool [41], starting from a model of a realistic service composition (the "Order Booking" business process [102] distributed with the Oracle SOA Suite [103]), comprising 37 activities, of which 16 were interactions with external services (*invoke*, *receive*, *reply* activities). This

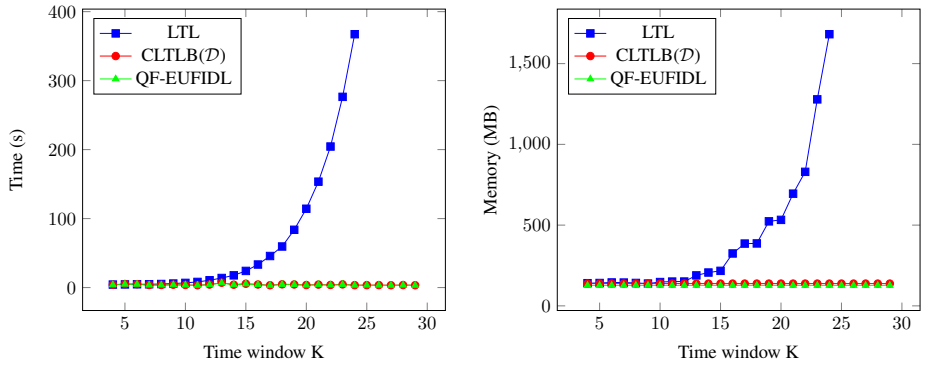


Figure 5.1: *Comparison between LTL- and CLTLB(𝒟)- and QF-EUFIDL-based encoding*

model was defined by specifying the workflow structure, the duration of each synchronous *invoke* activity, the branching probabilities, and the error rates. Other activities (e.g., *receive*) were given 0 as duration; branching was used to create loops and simulate the behavior of the *pick* activity.

For each run of the trace checker, we recorded the memory usage, the translation time, and the SMT verification time.

The evaluation was performed on a PC equipped with a 2.0GHz Intel Core i7-2630QM processor, running GNU/Linux Ubuntu 12.10 64bit, with 2GB RAM allocated for the verification tool. We used the *Z3* [52] SMT solver v. 4.3.1. For each experiment we have set a time limit to be 5 minutes and memory limit to 2GB. Each point shown in the plots, represents an average value of 10 trace check runs on traces of the same length.

5.2 Comparison with the LTL-based translation

To address RQ1: *Can the proposed trace checking procedures, based on CLTLB(𝒟) and QF-EUFIDL, handle traces more efficiently than the procedure [85] based on the translation [36] to LTL?*, we synthesized a sample history trace of length 30 and the SOLOIST formula $\mathfrak{C}^{K}_{>2}(r)$, which checks whether there have been more than two occurrences of the *replay* activity (denoted simply as the event r) within the last K time units. As we have shown in Section 2.2.2 the drawback of the translation from [36] is that the size of the output formula depends on parameters K, h and n. Therefore, if we vary the length of the time window K progressively between 2 and 30 we could see how this impacts the performance of the trace checking procedures of all the three translations. The formula was always evaluated at the

last time instant of the trace. Figure 5.1 shows the time and memory usage of the trace checking procedures based on all the three translations. The results show that with the increase of the time window K procedure based on LTL uses significantly more time and memory. The translation from [36] is inefficient and produces a large formula that results in a considerable increase in time and memory usage of the trace checking procedure; the two new translations proposed in this thesis address these issues, paving the way for a more efficient verification.

5.3 Scalability

To address RQ2: *How do the proposed trace checking procedures scale with respect to the various parameters (e.g., the length of the trace, the length of the time window K) involved in SOLOIST trace checking?*, we considered both translations and synthesized traces of various length in order to check the scalability of both approaches over the same traces. We also considered the following parameters:

Trace length. It represents the length of the synthesized trace and the bound given to the SMT solver. The length of each synthesized trace depends on the duration of the activities invoking an external service as well as on the branching probabilities of the loop(s) in the process.

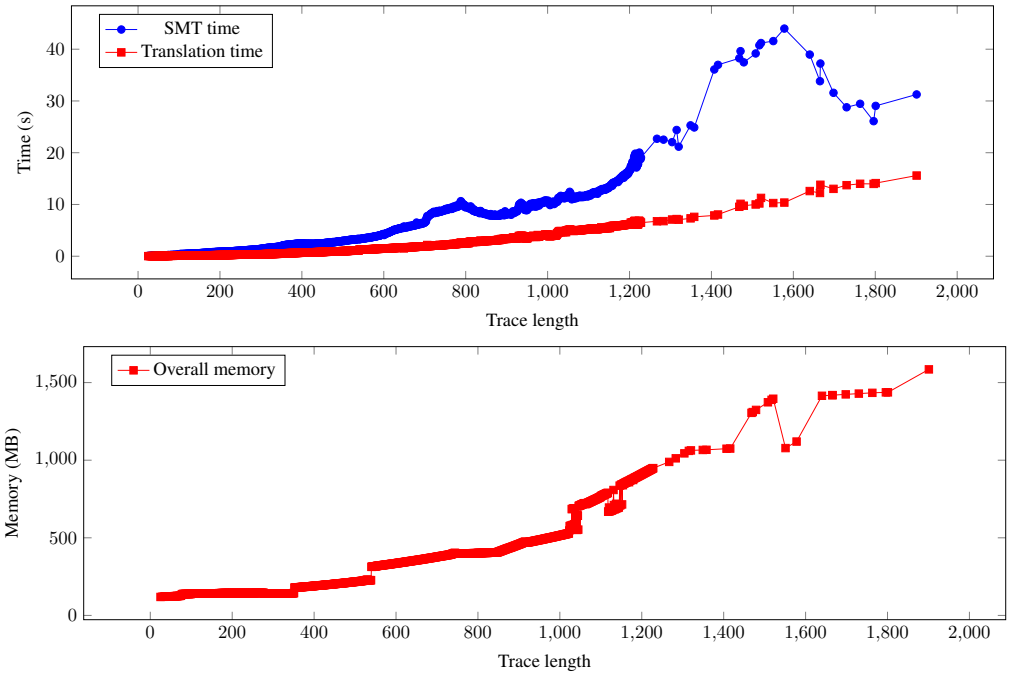
Length of the time window. It is used in the aggregate modalities; it corresponds to the K parameter.

Bound of the comparison operator. It is used in the aggregate modalities; it corresponds to the n parameter.

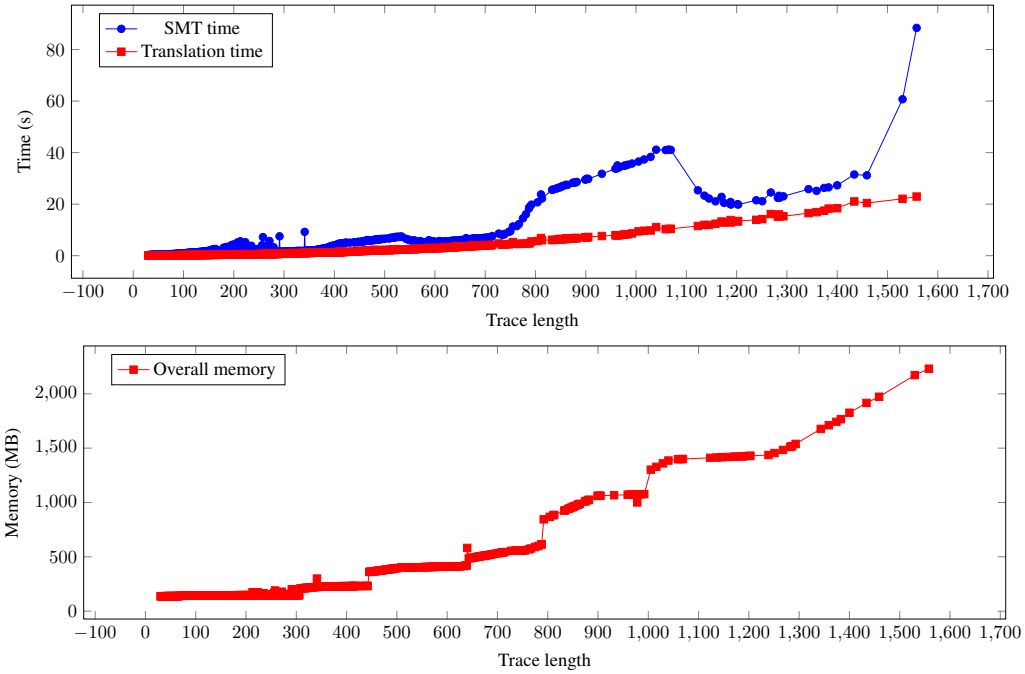
We present only the results of the evaluation done for the $\mathfrak{C}$ and $\mathfrak{D}$ modalities, since they are the keystones of the translations. We synthesized 20000 different traces, of variable length between 10 and 2000. We checked the following properties on them: $\mathfrak{C}^K_{>n}(p)$, and $\mathfrak{D}^K_{>n}(p, q)$, with propositions p and q corresponding to the start and end events of *CreditCardAuthorization* service invocation of the *Order Booking* business process.

Evaluation of the CLTLB($\mathcal{D}$)-based procedure

First we present the measured performance for the trace checking procedure based on CLTLB($\mathcal{D}$) in the Figures 5.2 and 5.3 which contain four plots each. Each pair of adjacent plots are grouped and the upper plot shows the time in seconds taken for the translation and the one taken by the SMT solver, while the lower plot shows the overall memory usage in megabytes. Figures 5.2a and 5.2b present the results of trace checking each of the two properties mentioned above on the synthesized traces classified according

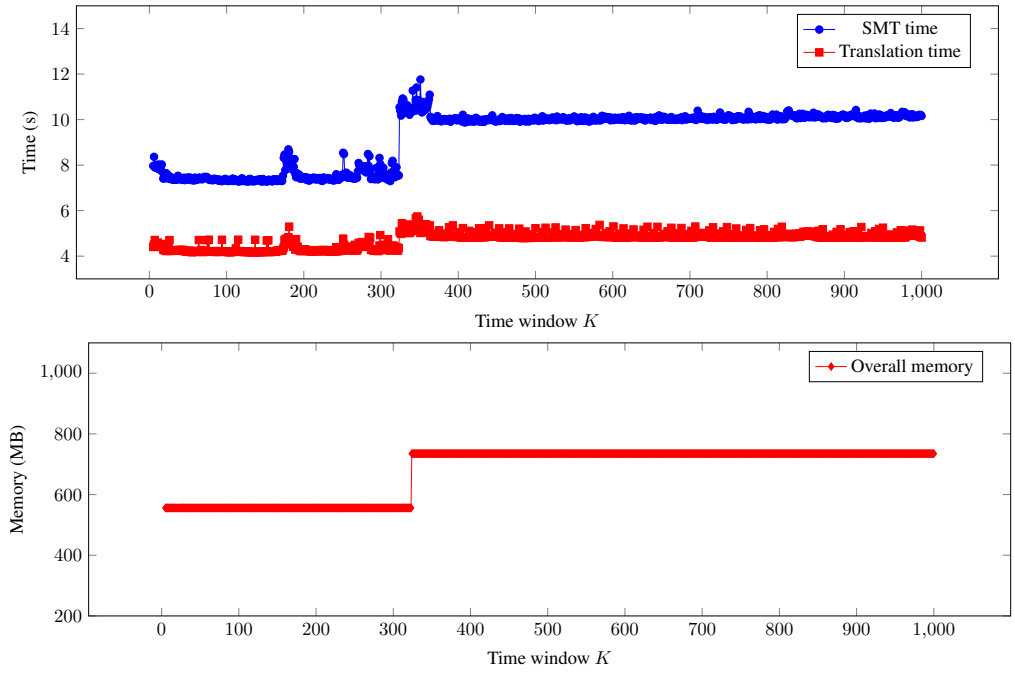


(a) *Time and memory scalability of* $\mathfrak{C}^{100}_{>5}(p)$ *modality with respect to trace length* H

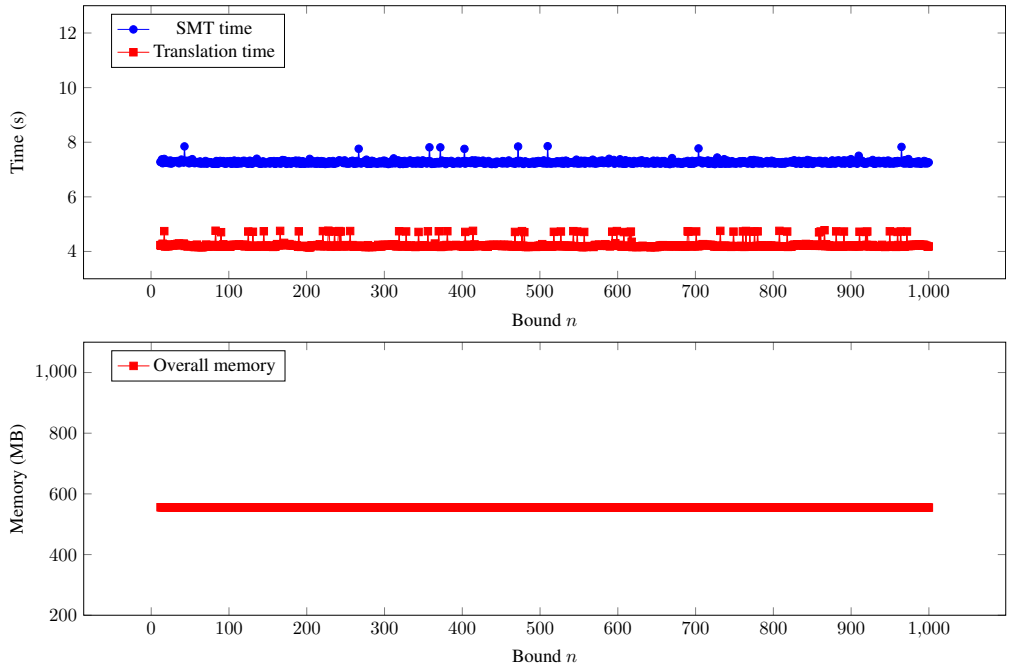


(b) *Time and memory scalability of* $\mathfrak{D}^{100}_{>5}(p, q)$ *modality with respect to trace length* H

Figure 5.2: *Scalability of CLTLB(*$\mathcal{D}$*)-based trace checking wrt trace length* H

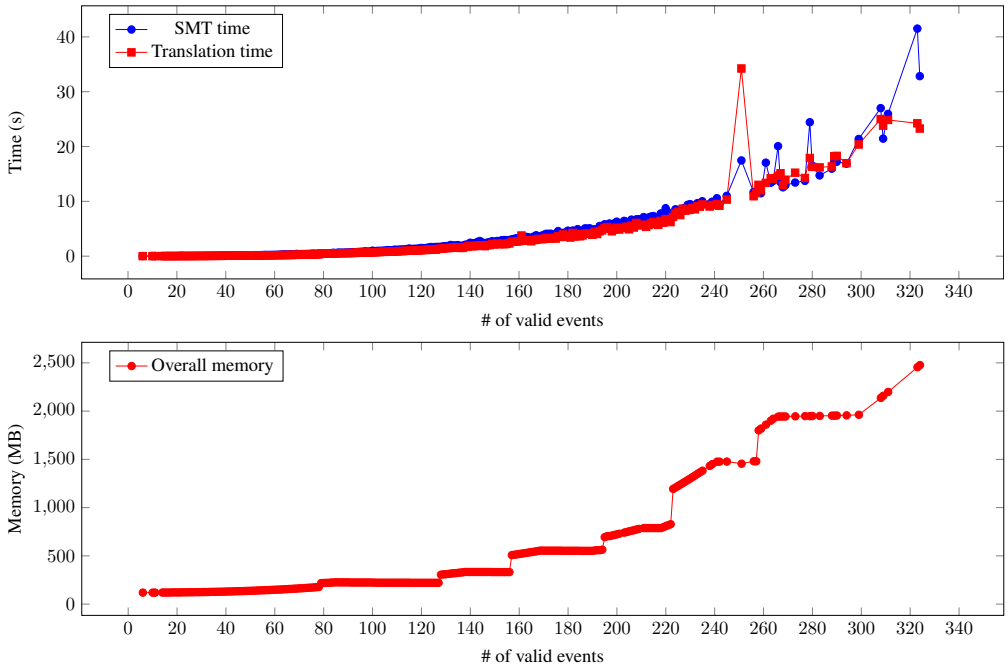


(a) *Time and memory scalability of* $\mathfrak{C}^{K}_{>5}(p)$ *modality with respect to time window* K

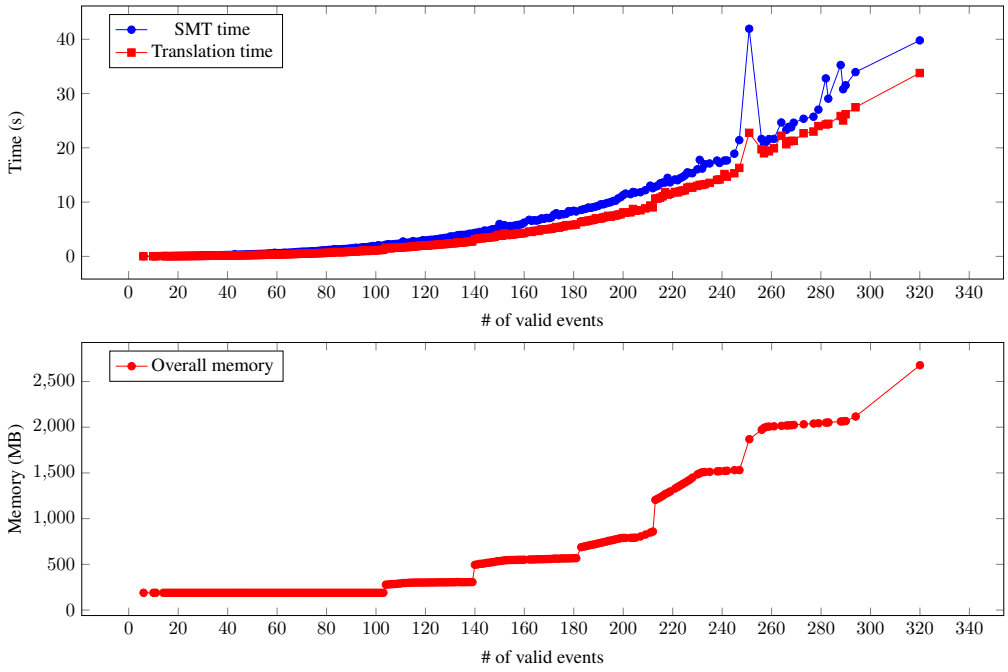


(b) *Time and memory scalability of* $\mathfrak{C}^{100}_{>n}(p)$ *modality with respect to the bound* n

Figure 5.3: *Scalability of CLTLB(*$\mathcal{D}$*)-based trace checking wrt parameters* K *and* n

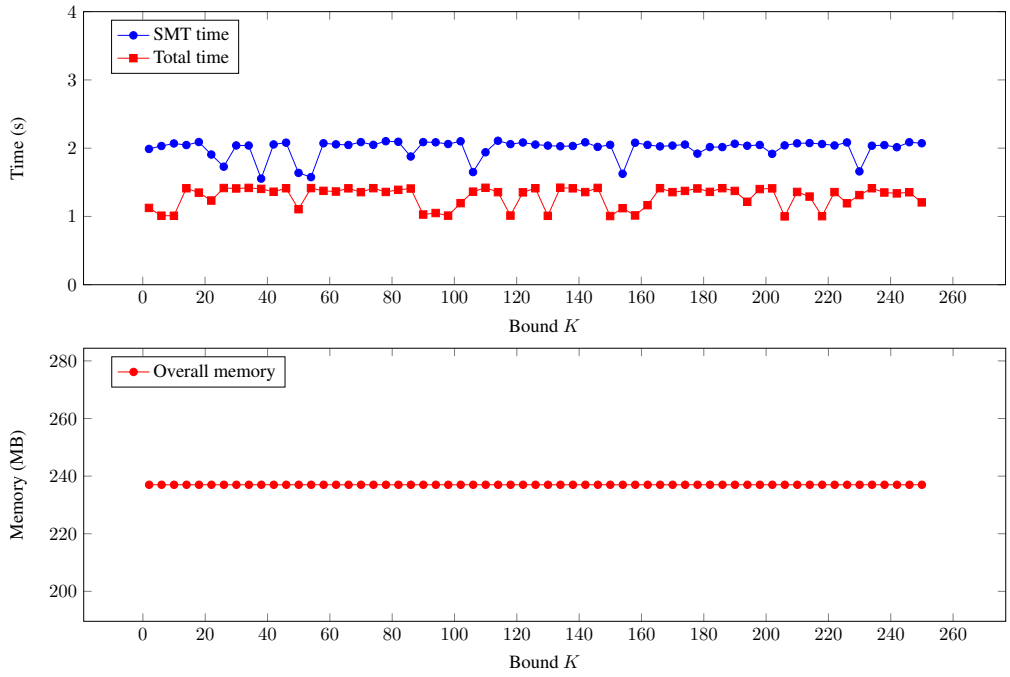


(a) *Time and memory scalability of* $\mathfrak{C}$ *modality with respect to number of valid events*

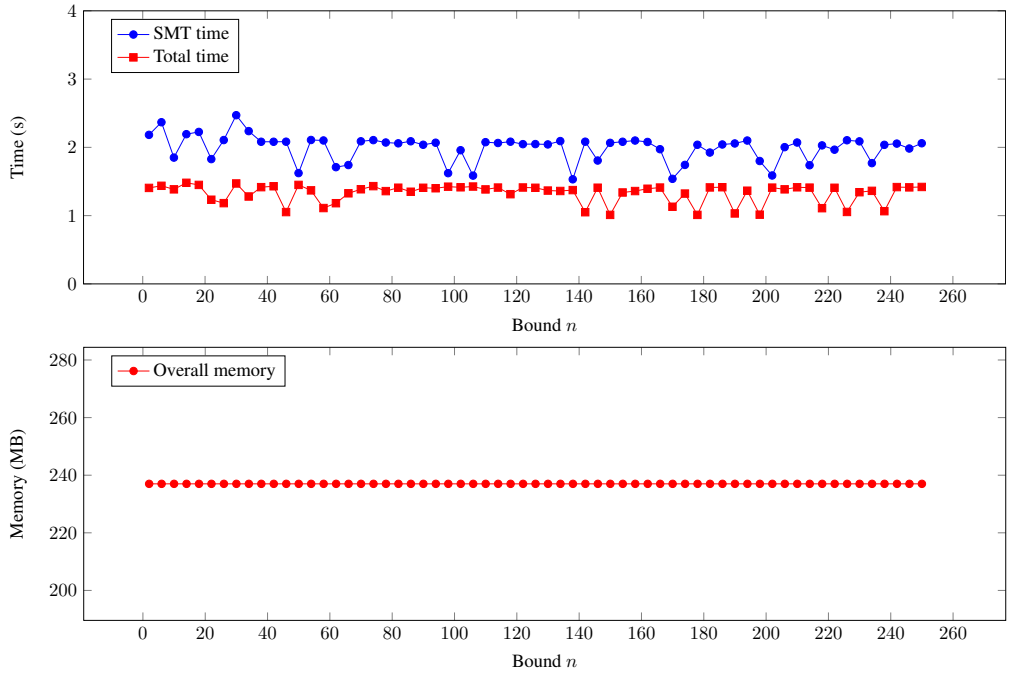


(b) *Time and memory scalability of* $\mathfrak{D}$ *modality with respect to number of valid events*

Figure 5.4: *Scalability of QF-EUFIDL-based trace checking wrt number of valid events*



(a) *Time and memory scalability of $\mathfrak{C}$ modality with respect to time window K*



(b) *Time and memory scalability of $\mathfrak{D}$ modality with respect to the bound n*

Figure 5.5: *Scalability of QF-EUFIDL-based trace checking wrt parameters K and n*

to their length; we fixed $K = 100$ and $n = 5$ for the both modalities. The plots provide an intuition of the growth rate of the resources usage with respect to the length of the input trace. The memory usage for the respective properties yields a very similar plot. The memory dedicated for the evaluation of the $\mathfrak{C}$ modality was exhausted at 2000 time instances, requiring 2GB of memory and 40 seconds to solve. For the evaluation of the properties with the $\mathfrak{D}$ modality, the maximum number of time instances manageable before exhausting the preset memory limit was 1600. The lower value with respect to the $\mathfrak{C}$ modality is due to the linear multivariate constraints introduced in the translation of the $\mathfrak{D}$ modality; these constraints are harder to solve than the univariate ones used for the $\mathfrak{C}$ modality. As for the scalability with respect to the other parameters, namely the length of the time window K and the bound of the comparison operator n, we notice that they do not affect the resource usage, and only introduce some non-deterministic noise in the SMT solver time. Results are similar for all the modalities thus we focus only on $\mathfrak{C}$ modality in Figures 5.3a (for K) and 5.3b (for n). The plots show the time and memory usage with respect to the variation of each of these two parameters when checking formulae over traces of length fixed to 1000; the upper two plots refer to checking formula $\mathfrak{C}^{K}_{>5}(p)$, while the lower two refer to checking formula $\mathfrak{C}^{100}_{>n}(p)$.

Evaluation of the QF-EUFIDL-based procedure

To evaluate the QF-EUFIDL-based encoding, we have performed the same experiments for both modalities. The only difference is that, in this case, we classify the traces according to the number of valid time instants, as it corresponds to the length H of the temporal structure of QF-EUFIDL. The plots in Figures 5.4 and 5.5 show the performance of QF-EUFIDL-based trace checking procedure using the same traces[1] and formulae as for the evaluation of the CLTLB($\mathcal{D}$)-based procedure. Figures 5.4a and 5.4b show quadratic increase in time and memory usage with respect to the number of valid time instants, as anticipated in Sect. 4.4. Similarly like for CLTLB($\mathcal{D}$), the plots in Figures 5.5a and 5.5b show that parameters K and n do not affect the computational time and space. Although the complexity analysis states that the size of the encoding for the $\mathfrak{D}$ modality linearly depends on n, the evaluation showes that in the actual implementation this does not happen, since the SMT decision procedure natively supports multiplication of terms by a constant. This allows us to write a more concise encoding for $\mathfrak{D}$ modality in $\mathcal{O}(H^2)$.

[1] sorted differently

5.4 Comparison

To address RQ3: *How do the two proposed trace checking procedures compare?* we considered what are the differences in their respective encoding. Namely, CLTLB($\mathcal{D}$) encodes timing implicitly using the positions of its two-part model, while QF-EUFIDL uninterpreted functions to encode timing information explicitly. Hence, the model of QF-EUFIDL is more compact, however its encoding of the SOLOIST modalities is more complex, as it needs to explicitly take into account the timing information.

As you can see from Figure 5.2, QF-EUFIDL-based approach can support the checking of traces containing up to 300 valid time instants, using up to 2GB of memory. With the same memory limit, the CLTLB($\mathcal{D}$)-based encoding could support traces with up to 2000 time instants (both valid and non-valid). The number of non-valid time instants in the trace does not affect the scalability of the QF-EUFIDL-based encoding. In other words, QF-EUFIDL-based encoding can deal with traces of arbitrary length, with varying degrees of sparseness, and still use up to 2GB of memory if the trace contains at most 300 valid time instants. The number and type of supported instants are hard constraints for both approaches, and any trace that exceeds them cannot be verified. However, one may devise heuristics such as abstraction [47] or partial order reduction [115] that could transform the trace in linear time to conform to the constraints.

As suggested earlier, the main difference between the two approaches is in how they handle traces with different *degrees of sparseness*. In Section 4.1 we defined the degree of sparseness informally, now we give its precise definition. Let ξ be the number of valid time instants in a trace, i.e., the instants in which at least one event occurs. This number corresponds to the number of positions in a timed word modeling the trace. Let ν denote the number of non-valid time instants, i.e., those where no event occurs. Notice that, in timed words, these events are abstracted away by using timestamps. We can use the total length of a trace $\xi + \nu$ to compute the degree of sparseness as $\varsigma = \frac{\xi}{\xi+\nu}$. For example, if we consider traces from Figure 4.1 their degree of sparseness is 0.2 and 0.8 (or 20% and 80%) respectively.

We compared the performance of the two approaches by classifying the synthesized traces according to their degree of sparseness before performing comparative trace checking runs based on both encodings. Figure 5.6 shows the results of this comparison, in terms of time and memory usage: the blue line shows the scalability of the approach based on CLTLB($\mathcal{D}$), while the seven red lines correspond to the QF-EUFIDL-based approach

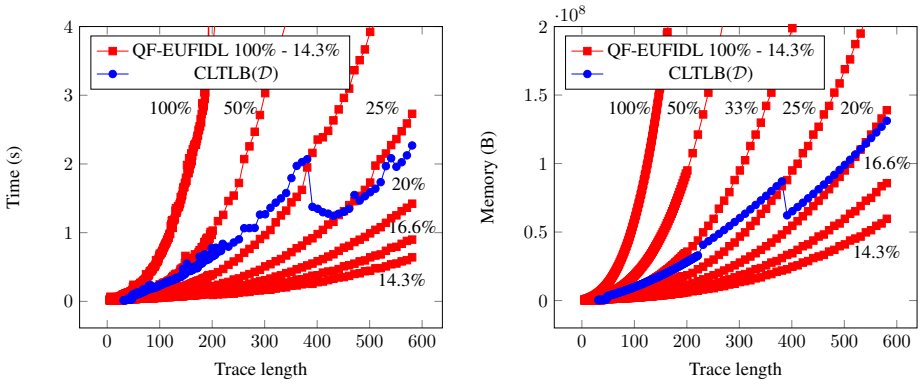


Figure 5.6: *Tradeoff between the trace checking approach based on CLTLB(D) [85] and the one based on the QF-EUFIDL encoding, with respect to the degree of sparseness of the trace*

applied to traces with different degrees of sparseness (100%, 50%, 33%, 25%, 20%, 16.6%, and 14.3%, from left to right, respectively). The results show that the QF-EUFIDL-based encoding is more efficient than the CLTLB($\mathcal{D}$)-based one, only when the degree of sparseness of input traces is less than 25%.

5.5 Formalization of Quantitative properties

To address RQ4: *Can our approach to trace checking be applied in a real setting?*, we consider in this section if SOLOIST can be used to formalize quantitative properties of systems. In the next section we apply SOLOIST decision procedures and perform trace checking of real traces.

To exemplify the use of SOLOIST for formalization we consider a variant of the *ATMFrontEnd* business process example from the JBoss jBPM distribution. We present its simplified description in BPEL, depicted in Figure 5.7 using the (visually intuitive) notation introduced in [10].

The process ATMFrontEnd starts when the *receive* activity `logOn` processes a message from the *SessionManager* service. This starts a customer session: the process verifies whether the customer holds a valid account at the bank, by invoking the `checkAccess` operation of the *BankAccount* service. If the latter identifies the customer, a loop is started to manage the customer's requests sent via the *UserInteraction* service. The `customerMenu` *pick* activity, contained in the body of the loop, may receive four kinds of possible requests: three of them (`getBalance`,

`deposit`, `withdraw`) are forwarded to the corresponding operations of the *BankAccount* service; the `logOff` request terminates the loop, closing the customer session.

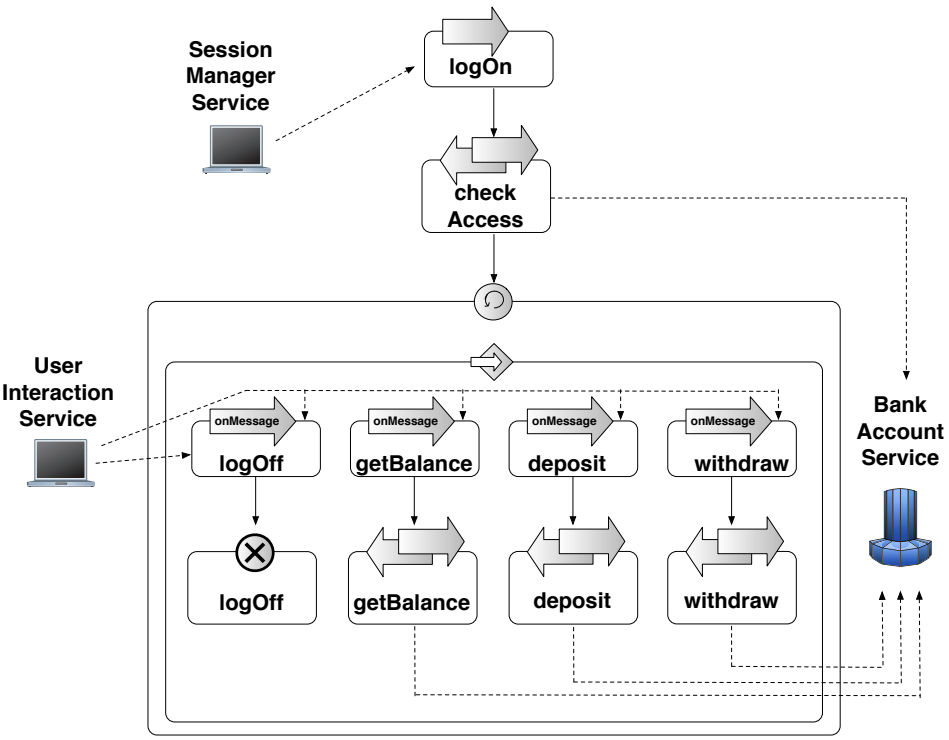


Figure 5.7: ATMFrontEnd *business process*

To annotate a BPEL process with SOLOIST, we denote the execution of each activity with a predicate symbol. Activities that involve a data exchange (e.g., an *invoke*) are modeled as non-nullary predicates, where the arguments correspond to the variables by which the input (output) messages can be represented. Synchronous *invoke* activities are actually modeled with two predicates, corresponding to the start and the end of the invocation; these are denoted with the "_start" and "_end" suffixes, respectively.

Below we list some examples of quantitative properties expressed first in natural language and then with SOLOIST; more details on the features of the language are available in [36]. All properties are under the scope of an implicit universal temporal quantification as in *"In every process run, ..."*; we assume the time units to be in seconds.

QP1: WithdrawalLimit
The number of withdrawal operations performed within 10 minutes

before customer logs off is less than or equal to the allowed limit (assumed to be 3, for example). This property is expressed as: $\mathsf{G}(logOff \rightarrow \mathfrak{C}^{600}_{\leq 3}(withdraw))$.

QP2: CheckAccessAverageResponseTime
The average response time of operation `checkAccess` provided by the *BankAccount* service is always less than 5 seconds within any 15 minute time window. This property is expressed as: $\mathsf{G}(\mathfrak{D}^{900}_{<5}\{(checkAccess\_start, checkAccess\_end)\})$.

QP3: MaxNumberOfBalanceInquiries
The maximum number of balance inquiries is restricted to at most 2 per minute within 10 minutes before customer session ends. This property is expressed as: $\mathsf{G}(logOff \rightarrow \mathfrak{M}^{600,60}_{\leq 2}(getBalance))$.

Notice that we express time in seconds and use propositions, as the content of the messages exchanged between the services is not important in the properties above.

5.6 Application on Real Traces

To address RQ4: *Can our approach to trace checking be applied in a real setting?*, we have also applied our approach also to a real application, a service composition called ACME BOT [88], whose monitoring data are available[2] as part of the "S-Cube Use Case Repository". We reconstructed 9796 execution traces, based on the monitoring data of the corresponding service composition instances. On each of these traces, we performed trace checking with respect to properties based on $\mathfrak{C}$, $\mathfrak{D}$ and $\mathfrak{M}$ modalities similar to ones in the previous section. In the first case, trace checking took on average 0.672s with a standard deviation of 0.035s and used on average 125.7MB of memory with 0.476MB standard deviation; for the checks with the $\mathfrak{D}$ modality, it took on average 0.813s with 0.032s standard deviation and used on average 127.7MB of memory with 0.476MB standard deviation; finally for the $\mathfrak{M}$ modality, trace checking took on average 1.335s with 0.032s standard deviation and used on average 163.3MB with 0.476MB standard deviation. On average, each trace had 31.5 valid time instants and a total length of 39341.3; the average degree of sparseness was then 0.08%. This example shows that our approach can efficiently check properties of realistic systems.

[2] `http://scube-casestudies.ws.dei.polimi.it/index.php/`.

CHAPTER 6

Case study: Cloud-based Elastic Systems

6.1 Overview

Cloud computing has become a practical solution to manage and leverage IT resources and services. Cloud platforms offer several benefits, among which the ability to access resources or service applications offered as (remote) services, available on-demand and on-the-fly, and billed according to a pay-per-use model.

Cloud providers offer resources and services at three different layers: at the *Software-as-a-Service (SaaS)* layer, users can remotely access full-fledged software applications; at the *Platform-as-a Service (PaaS)* layer, one finds a development platform, a deployment and a run-time execution environment, which is used to run user-provided code in sandboxes hosted on cloud-based premises; at the *Infrastructure-as-a-Service (IaaS)* the user can access computing resources such as virtual machines, block storage, firewalls, load balancers, or networking I/O.

This chapter focuses on the IaaS layer, and assumes, without loss of generality, that resources offered at this level are virtual machines. In particular, we consider *cloud-based elastic systems*. Elasticity [77] of computing systems is a quantitative property defined [93] by the (US) National

Institute of Standards and Technology (NIST) as:

> "[Elasticity is the] capability to rapidly and elastically provision [resources], in some cases automatically, to quickly scale out, and rapidly release [resources] to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time."

Application providers exploit resource elasticity at run time to balance the trade-offs between the quality of service (QoS), the input workload, and the operational costs. The main goal is, when confronted with fluctuating workloads, to maintain the QoS at an adequate level while minimizing the costs. In particular, when the input workload escalates over the current system capacity, the acquisition of new resources prevents *under*-provisioning, and allows the system to maintain an adequate QoS, even though operational costs increase. On the other hand, when the input workload decreases below the current system capacity, releasing some resources prevents *over*-provisioning, and contributes to reducing the costs while still providing an adequate QoS.

The behavior (in terms of dynamically scaling up and down resource allocation) of a cloud-based elastic system depends on the combination of many factors, such as the input workload and infrastructure costs. Techniques for designing such systems in terms of these factors can highly benefit from the research in quantitative properties and their verification.

From the point of view of specification and verification the three main open issues are:1) how to specify the desired elastic behavior of these systems; 2) how to check whether they manifest or not such an elastic behavior; 3) how to identify when they depart from the intended behavior.

In order to provide a more comprehensive answer to the RQ4 from Chapter 5, this chapter presents a detailed case study of formalization and verification of the behavior of cloud-based elastic systems, by characterizing the properties related to elasticity, resource management, and quality of service using an extension of SOLOIST and then using CLTLB($\mathcal{D}$)- and QF-EUFIDL-based trace checking procedures for the verification. The rest of the chapter is organized as follows: Section 6.2 provides an overview of cloud-based elastic systems, describing how they operate. Section 6.3 introduces SOLOIST$^{\mathcal{A}}$, an extension of SOLOIST used for the formalization of the quantitative properties of cloud-based elastic systems. Section 6.4 formally defines some general aspects of resources used in cloud-based systems. Section 6.5 illustrates the formalization of the properties that we have considered. Section 6.6 reports on checking some of the properties on

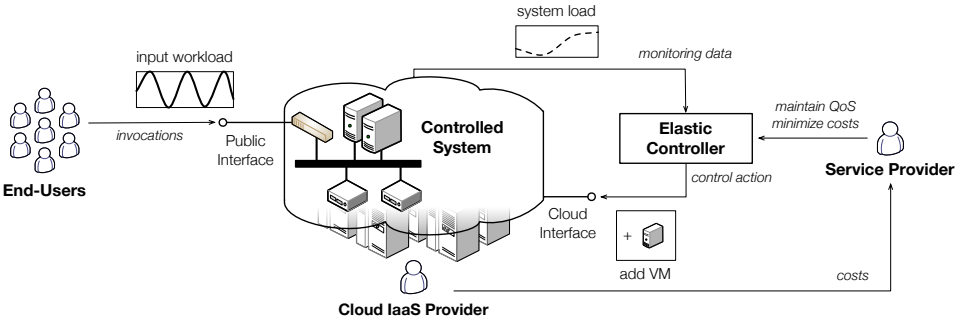


Figure 6.1: *High level architecture/view of cloud-based elastic systems.*

realistic execution traces.

6.2 Cloud-based Elastic Systems

Two hallmarks of cloud computing are the ability to dynamically manage the allocation of resources in the system and the pay-per-use billing model. In particular, these traits characterize *cloud-based elastic systems*, which are systems that can dynamically adjust their resources allocation to maintain a predefined/suitable level of QoS, in spite of fluctuating input workloads, while minimizing running costs. The key aspect of cloud-based elastic systems is their ability to *adapt* at run time, in response to a change in the operating conditions (e.g., a spike in the number of input requests). In this context, adaptation means trying to manage the allocation of resources so that they match the capacity required to properly sustain the input workload. In other words, cloud-based elastic systems aim to prevent both over-provisioning (allocating more resources than required) and under-provisioning (allocating fewer resource than required).

The behavior of an elastic system can be intuitively described as follows. Consider the case in which there is an increase in the load of a system, which might lead to the saturation of system resources, causing a degradation of the QoS perceived by end-users. To avoid the saturation, an elastic system *stretches*, i.e., its capacity is scaled up by allocating additional resources (acquired from a cloud infrastructure); the load can then be spread over a bigger set of resources. Conversely, when the system load decreases, some resources might become under-utilized, hence unnecessarily expensive. To reduce costs, an elastic system *contracts*, i.e., its capacity is scaled down by deallocating a portion of the allocated resources, which are then released back to the cloud infrastructure.

Cloud-based elastic systems usually implement the closed-loop archi-

tecture shown in Figure 6.1, where an *elastic controller* monitors the actual system (i.e., the *controlled system*) and determines its adaptation. End-users send their requests, which constitute the input workload of the elastic system, through its public interface. Notice that the workload may fluctuate because of seasonality in the users' demand or some unexpected events such as a flash-crowd (i.e., the appearance of a web site on a popular blog or news column, determining an exponential spike of the requests to the server).

The *controlled system* responds to end-user requests by implementing the business logic of the application. It is deployed onto a cloud infrastructure provided by a dedicated IaaS provider, and constituted by a set of cooperating virtual machines. The controlled system implements also the logic to change the allocation of resources (i.e., virtual machines) and adjust the capacity of the system; these are essential functionalities to enable an elastic behavior. The controlled system is also characterized by two attributes that constrain, respectively, the minimum and the maximum number of allocated resources. The former corresponds to the minimal amount of resources that must be always allocated to guarantee the provision of the application functionalities to end-users. As for the latter, it sets an upper bound for the maximum amount of allocated resources, beyond which scaling the system is not cost-effective anymore.

The elastic controller periodically monitors the operating parameters of the controlled system (e.g., the *system load*) and determines the control actions to be executed to perform adaptation. The controller implements the logic that tries to fulfill the high-level goals (e.g., minimizing running costs while delivering a certain level of QoS) specified by the service provider that operates the elastic system. The control actions that the controller can issue are *scale-up* and *scale-down*, which correspond to instantiating and terminating virtual machines, respectively. These actions are sent to the controlled system through its cloud interface, which plays the role of the controller actuator. Notice that executing these actions might take a non-negligible time, which is called actuation delay. The cloud interface propagates the control actions issued by the controller to the cloud IaaS provider, which performs the physical allocation/deallocation of virtual machines. The cloud IaaS provider tracks the total resource usage accumulated by each service provider, who is then billed for the cost of running the system.

6.3 SOLOIST$^{\mathcal{A}}$

This section presents SOLOIST$^{\mathcal{A}}$, an extension of SOLOIST with arithmetical atomic formulae that allows us to formalize the properties of cloud-based elastic systems. SOLOIST$^{\mathcal{A}}$ introduces the concept of *variables* similar to counters and arithmetical variables in CLTLB($\mathcal{D}$) and QF-EUFIDL. Intuitively, using this extension one can define values of variables in different adjacent positions by defining *arithmetical constraints*. For example, we can express a property like "whenever an event S occurs, in the next position, variable z must be incremented by 1 with respect to the value at the current position". Recall that the model of SOLOIST formulae is a timed word that is a sequence of timestamped events, like: $(\emptyset, 1)\ (\{Q\}, 3)\ (\emptyset, 6)\ (\{R\}, 10)\ (\emptyset, 15)\ldots$. Each pair contains a set of events and a timestamp. The events correspond to observations we make about the behavior of a system at a particular absolute time represented by the timestamp. Typically we associate all the positions in the sequence to time instants where the behavior of the system changes. For example, at the second position Q occurs, the elapsed time is 3.

To account for the addition of variables we need to extend this model to consider their values in each observation. For instance, consider the following sequence: $(\{Q\}, \{2\}, 3)\ (\emptyset, \{2\}, 4)\ (\{R\}, \{1\}, 5)\ (\{Q\}, \{2\}, 15)$ $(\{R\}, \{0\}, 20)$. In each position we have a triple: the first element is a set of events occurring in that position (e.g., Q or R); the second is a set of values of variables (e.g., modeling the number of pending jobs inside the system); the third element is the timestamp. Using the timestamps, all positions in the sequence correspond to time instants where the behavior of the system changes. For example, at the first position Q occurs, the elapsed time is 3 and the number of pending jobs is 2. Position 3 captures the fact that at time 5 the system replies (denoted with R), completing only one of the two pending jobs, thus one job remains. Position 4 denotes that at time 15 another query (this time equipped with one job) occurs; the number of pending jobs is incremented accordingly. The last position indicates that at time 20 the system generates another reply and, by that time, all the pending jobs are done (the corresponding variable is equal to 0).

In the rest of this section, we formally define SOLOIST$^{\mathcal{A}}$. Atomic formulae in SOLOIST$^{\mathcal{A}}$ can now also be *constraints* over a structure $(\mathbb{Z}, =, (<_d)_{d\in\mathbb{Z}})$, where binary relation$(<_d)_{d\in\mathbb{Z}}$ is introduced in Section 4.

The syntax of the *terms* used in the constraints, called *arithmetic tempo-*

$$
\begin{array}{lll}
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \alpha_1 \sim \alpha_2 & \text{iff} & \pi_i(\alpha_1) \sim \pi_i(\alpha_2) \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} p & \text{iff} & p \in \sigma_i \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \neg\phi & \text{iff} & ((\sigma,\pi,\tau),i) \not\models_{\mathcal{A}} \phi \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \phi \wedge \psi & \text{iff} & ((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \phi \wedge ((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \psi \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \phi \mathsf{S}_I \psi & \text{iff} & \text{for some } j < i, \tau_i - \tau_j \in I, ((\sigma,\pi,\tau),j) \models_{\mathcal{A}} \psi \\
 & & \text{and for all } k, j < k < i, ((\sigma,\pi,\tau),k) \models_{\mathcal{A}} \phi \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \phi \mathsf{U}_I \psi & \text{iff} & \text{for some } j > i, \tau_j - \tau_i \in I, ((\sigma,\pi,\tau),j) \models_{\mathcal{A}} \psi \\
 & & \text{and for all } k, i < k < j, ((\sigma,\pi,\tau),k) \models_{\mathcal{A}} \phi \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \mathfrak{C}^K_{\bowtie n}(\phi) & \text{iff} & c(\tau_i - K, \tau_i, \phi) \bowtie n \text{ and } \tau_i \geq K \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \mathfrak{U}^{K,h}_{\bowtie n}(\phi) & \text{iff} & \frac{c(\tau_i - \lfloor \frac{K}{h} \rfloor h, \tau_i, \phi)}{\lfloor \frac{K}{h} \rfloor} \bowtie n \text{ and } \tau_i \geq K \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \mathfrak{M}^{K,h}_{\bowtie n}(\phi) & \text{iff} & \max\left\{ \bigcup_{m=0}^{\lfloor \frac{K}{h} \rfloor} \{c(lb(m), rb(m), \phi)\} \right\} \bowtie n \text{ and } \tau_i \geq K \\
((\sigma,\pi,\tau),i) \models_{\mathcal{A}} \mathfrak{D}^K_{\bowtie n}(\phi,\psi) & \text{iff} & \frac{\sum_{(s,t)\in d(\phi,\psi,\tau_i,K)}(\tau_t - \tau_s)}{|d(\phi,\psi,\tau_i,K)|} \bowtie n \text{ and } \tau_i \geq K \text{ and } d(\phi,\psi,\tau_i,K) \neq \emptyset
\end{array}
$$

where $c(\tau_a, \tau_b, \phi) = |\{s \mid \tau_a < \tau_s \leq \tau_b \text{ and } ((\sigma,\pi,\tau),s) \models_{\mathcal{A}} \phi\}|$, $lb(m) = \max\{\tau_i - K, \tau_i - (m+1)h\}$, $rb(m) = \tau_i - mh$, and $d(\phi,\psi,\tau_i,K) = \{(s,t) \mid \tau_i - K < \tau_s \leq \tau_i \text{ and } ((\sigma,\pi,\tau),s) \models_{\mathcal{A}} \phi,$ $t = \min\{u \mid \tau_s < \tau_u \leq \tau_i, ((\sigma,\pi,\tau),u) \models_{\mathcal{A}} \psi\}\}$

Figure 6.2: *Semantics of SOLOIST$^{\mathcal{A}}$ defined as an extension of the semantics of SOLOIST.*

ral terms (hereafter simply called *terms*) is defined as:

$$\alpha := c \mid x \mid \mathsf{Y}(x) \mid \mathsf{X}(x)$$

where $c \in \mathbb{Z}$ is a constant, $x \in V$ is a variable, Y is the *arithmetical previous* temporal operator, and X is the *arithmetical next* temporal operator. The temporal operators are applied to terms, and they refer to the value of that term in the previous (Y) and in the next (X) position in the sequence, i.e., the corresponding discrete position.

The syntax of SOLOIST$^{\mathcal{A}}$ formula ψ is extended with term expression $\alpha \sim \alpha$ and thus defined as follows:

$$\psi ::= \boldsymbol{\alpha \sim \alpha} \mid p \mid \neg\psi \mid \psi \wedge \psi \mid \psi \mathsf{U}_I \psi \mid \psi \mathsf{S}_I \psi \mid \mathfrak{C}^K_{\bowtie n}(\psi) \mid \mathfrak{U}^{K,h}_{\bowtie n}(\psi) \mid$$
$$\mathfrak{M}^{K,h}_{\bowtie n}(\psi) \mid \mathfrak{D}^K_{\bowtie n}(\psi,\psi),$$

where the relation $\sim$ belongs to $\{=, (<_d)_{d\in\mathbb{Z}}\}$ and all other concepts are defined in Section 2.2. Notice that we denote a SOLOIST$^{\mathcal{A}}$ formula with a non-terminal ψ, while we retain ϕ as the non-terminal of SOLOIST formulae.

Hereafter, we use quantifiers ($\forall$ and $\exists$) and parameterized propositions over finite sets as a shorthand for representing a group of constraints. For example, given the set $A = \{1,2,3\}$ and the parameterized proposition $p(\cdot)$, the formula $\forall a \in A : p(a)$ is a shorthand for $p_1 \wedge p_2 \wedge p_3$, where

$p_1, p_2, p_3 \in \Pi$. We omit the definition of the set when it is clear from the context.

The formal semantics of SOLOIST$^{\mathcal{A}}$ formulae can be defined as follows. Let τ be a *timed sequence* and σ *word* over the alphabet 2^{Π} (see Section 2.1.1). We additionally define $\pi = \pi_0\pi_1 \dots \pi_{|\pi|-1}$ be a sequence of evaluations $\pi_i : V \to \mathbb{Z}$ used to pinpoint the value of variables at each time position. We denote the value of x at position i with $\pi_i(x)$ and the value $\pi_{i+|\alpha|}(x_\alpha)$ with $\pi_i(\alpha)$, where x_α is the variable in V occurring in term α, if any. Given a time instant $i \geq 0$ and a structure (σ, π, τ), we define the satisfaction relation $(\sigma, \pi, \tau), i \models_{\mathcal{A}} \psi$ for SOLOIST$^{\mathcal{A}}$ formulae as shown in Figure 6.2. A formula $\psi \in$ SOLOIST$^{\mathcal{A}}$ is *satisfiable* if there exists a triple (σ, π, τ) such that $(\sigma, \pi, \tau), 0 \models_{\mathcal{A}} \psi$; in this case, we say that (σ, π, τ) is a *model* of ψ, with (σ, τ) being the *timed propositional model* and (π, τ) being the *arithmetic model*. Also, we use SOLOIST$^{\mathcal{A}}$ over finite words (i.e., for trace checking), thus we retain decidability.

In order to provide a decision procedure for SOLOIST$^{\mathcal{A}}$ language we need to extend our translations to support arithmetical terms. We considered both CLTLB($\mathcal{D}$) and QF-EUFIDL translation for extension and chose QF-EUFIDL since the language supports arithmetical variables natively. Although CLTLB($\mathcal{D}$) supports counters they cannot be directly used to encode arithmetical constraints from SOLOIST$^{\mathcal{A}}$, since they are not defined over timed words. To be able to capture semantics of SOLOIST$^{\mathcal{A}}$, the CLTLB($\mathcal{D}$) needs to support a concept of a freezing operator and use it to compare values of counters at arbitrary positions, however formal proof for the lack of expressiveness of CLTLB($\mathcal{D}$) is still an open issue.

Therefore, we focus on the (more intuitive) extension of the QF-EUFIDL-based encoding by introducing an arithmetical variable $|v|$ for every term v in SOLOIST$^{\mathcal{A}}$ formula and we also introduce constraints $\mathcal{C}_{\mathcal{A}}$ that define the appropriate semantics of the SOLOIST$^{\mathcal{A}}$ terms as shown in Figure 6.3.

The first row defines the values of the variables. The second and third rows define constraints for the equality and integer difference relations as a straightforward mapping to the same relations in QF-EUFIDL. The forth and fifth row define semantics of the arithmetical next operator, by associating the value of the variable $|\mathsf{X}(v)|$ at position i to the value of the variable $|v|$ at position $i + 1$. The value of the variable $|\mathsf{X}(v)|$ at the last position is 0 by convention. Encoding of the arithmetical previous operator is defined in a similar way.

The final QF-EUFIDL formula obtained from the encoding of the input SOLOIST$^{\mathcal{A}}$ formula Φ is now the following conjunction extended with $\mathcal{C}_{\mathcal{A}}$ constraints: $[\![\Phi]\!]_0 \wedge \boldsymbol{\mathcal{C}_{\mathcal{A}}} \wedge \mathcal{C}_{time} \wedge \mathcal{C}_{prop} \wedge \mathcal{C}_{temp} \wedge \mathcal{C}_c \wedge \mathcal{C}_m \wedge \mathcal{C}_d$.

Position i	Propositional operators	Description
$0 \dots H$	$\|v\|_i = \pi_i(v)$	arithmetical variables
$0 \dots H$	$[\![v = w]\!]_i \leftrightarrow \|v\|_i = \|w\|_i$	equality
$0 \dots H$	$[\![v <_d w]\!]_i \leftrightarrow \|v\|_i <_d \|w\|_i$	integer difference
$0 \dots H-1$	$\|\mathsf{X}(v)\|_i = \|v\|_{i+1}$	"arithmetical next" operator
H	$\|\mathsf{X}(v)\|_H = 0$	"arithmetical next" operator at position H
$1 \dots H$	$\|\mathsf{Y}(v)\|_i = \|v\|_{i-1}$	"arithmetical previous" operator
0	$\|\mathsf{X}(v)\|_0 = 0$	"arithmetical previous" operator at position H

Figure 6.3: *Extended encoding for the SOLOIST$^{\mathcal{A}}$ terms*

6.4 Modeling Resources of Cloud-based Systems

At the basis of cloud-based service provisioning there is the possibility of accessing remote resources. As anticipated in Section 6.1, in this paper we consider elastic behaviors with respect to the management and adaptation of resources offered at the IaaS level, in particular virtual machines; hereafter, by slightly abusing the terminology, we will refer to resources and virtual machines interchangeably. In the rest of this section we introduce some useful notation and express in SOLOIST$^{\mathcal{A}}$ some general aspects that characterize the lifecycle of resources in a cloud-based system.[1] These aspects will then be *assumed* to hold across the formalization of the properties of cloud-based elastic systems in the next section.

We model the total resources in use by a system at a certain time instant by means of a non-negative integer variable $R \in V$. We use constants R_{min} and R_{max} to denote the minimum and maximum number of resources that the system can allocate. At any time, the amount of resources allocated to a system must be within these limits. We capture this constraint on resource allocation with the following formula:

$$\mathsf{G}(R_{min} \leq R \ \wedge \ R \leq R_{max}) \qquad (\mathcal{M}_{bound})$$

which states that number of resources R is bounded throughout the entire execution of the cloud-based system.

We use a non-negative integer variable $L \in V$ to denote the current load of the system, expressed in terms of required resources. We assume

[1] We use the term "cloud-based system" instead of the one "cloud-based elastic system" used elsewhere in the paper, since the aspects described here for modeling the resources can be assumed to hold for every kind of cloud-based system, not necessarily with an elastic behavior.

that each cloud-based system has always enough resources to support the current load (the *manageable load* assumption), as stated below:

$$\mathsf{G}(L \leq R_{max}) \qquad (\mathcal{M}_{load})$$

Virtual Machine Lifecycle

At the IaaS layer, users create virtual machines to host their applications; each virtual machine is uniquely identified by an ID. Let R_{max} be a positive integer parameter representing the maximum number of virtual machines that can be allocated by a system. The set of valid virtual machines IDs is then defined as $ID = \{0, \ldots, R_{max} - 1\}$.

To model the events characterizing the lifecycle of a virtual machine, we represent them as parameterized propositions. We use $M_{start}(\cdot)$ to denote a request to instantiate a new virtual machine; conversely, we use $M_{stop}(\cdot)$ to denote a shut-down request. After receiving the request to instantiate a new virtual machine, the cloud infrastructure starts the actual instantiation process by allocating the physical resources and booting the OS: the end of the OS boot is denoted with proposition $M_{boot}(\cdot)$. The actual event in which the user application is ready to process the input is denoted with $M_{ready}(\cdot)$. Similarly, in the case of a shut-down request, we denote the actual termination of the virtual machine (following the shut-down request) with the proposition $M_{end}(\cdot)$. The order of occurrence of these events has to match the one defined by the lifecycle of a virtual machine: M_{start}–M_{boot}–M_{ready}–M_{stop}–M_{end}. We state this constraint by AND-ing the following formulae $\mathcal{M}_{sb}$, $\mathcal{M}_{br}$, $\mathcal{M}_{rs}$, $\mathcal{M}_{se}$:

$$\begin{aligned}\forall id : \mathsf{G}(M_{start}(id) \rightarrow ((&\neg M_{start}(id) \wedge \neg M_{ready}(id) \\ &\wedge \neg M_{stop}(id) \wedge \neg M_{end}(id)) \\ &\mathsf{U} M_{boot}(id)))\end{aligned} \qquad (\mathcal{M}_{sb})$$

$$\begin{aligned}\forall id : \mathsf{G}(M_{boot}(id) \rightarrow ((&\neg M_{boot}(id) \wedge \neg M_{start}(id) \\ &\wedge \neg M_{stop}(id) \wedge \neg M_{end}(id)) \\ &\mathsf{U} M_{ready}(id)))\end{aligned} \qquad (\mathcal{M}_{br})$$

$$\forall id : \mathsf{G}(M_{ready}(id) \rightarrow((\neg M_{ready}(id) \wedge \neg M_{start}(id) \wedge \neg M_{boot}(id) \wedge \neg M_{end}(id)) \mathsf{U} M_{stop}(id))) \qquad (\mathcal{M}_{rs})$$

$$\forall id : \mathsf{G}(M_{stop}(id) \rightarrow((\neg M_{stop}(id) \wedge \neg M_{ready}(id) \wedge \neg M_{start}(id) \wedge \neg M_{boot}(id)) \mathsf{U} M_{end}(id))) \qquad (\mathcal{M}_{se})$$

All the formulae above follow the same structure. For example, in the case of Formula $\mathcal{M}_{sb}$, we state that after a request to instantiate a certain virtual machine (M_{start}), all other requests for the same machine (events M_{start}, M_{ready}, M_{stop}, M_{end}) cannot occur until the OS boot ends (event M_{boot}).

In addition, we require that the lifecycle of a virtual machine starts with event M_{start}:

$$\forall id : \mathsf{G}(M_{end}(id) \rightarrow \mathsf{P}(M_{start}(id))) \qquad (\mathcal{M}_{start})$$

The formula states that event M_{end} is always preceded by event M_{start}, for any id.

Finally, we specify that the transition from event $M_{start}(\cdot)$ to event $M_{boot}(\cdot)$ might take some finite time, bounded by the parameter T_{cd} defined by each single provider. This requirement is expressed as:

$$\forall id : \mathsf{G}(M_{start}(id) \rightarrow \mathsf{F}_{(0,T_{cd})}(M_{boot}(id)))^2 \qquad (\mathcal{M}_{bad})$$

The formula states that after receiving a request to allocate a new virtual machine, the boot process has to complete within T_{cd} time units.

6.5 Properties of Cloud-Based Elastic Systems

In this section we present concepts and properties that can be used to characterize relevant behaviors of cloud-based elastic systems. The concepts and the properties have been selected and derived based on our research experience in the field, especially matured within EU-funded projects like RESERVOIR [98] and CELAR [43]. The presentation is divided in three groups: elasticity, resource management, and quality of service.

As previously remarked, for the proposed formalization of properties, we assume that the concepts described in the previous section must always hold, i.e., if $\mathcal{C}_{system}$ is the conjunction of all formulae described in Section 6.4, we consider execution traces that satisfy $\mathcal{C}_{system}$.

[2] A closed interval $[a, b]$ over $\mathbb{N}$ can be expressed as an open one of the form $(a-1, b+1)$, $a \geq 1$.

6.5.1 Elasticity

As we have seen in Section 6.2, elastic systems are supposed to dynamically adapt in reaction to fluctuations in the input workload, by changing their computing capacity. In the case of cloud-based elastic systems, the adaptation is performed either by increasing the computing capacity with the allocation of additional resources, or by decreasing the capacity by releasing a portion of those resources.

Recall that we model the resources currently in use by a system with a non-negative integer variable R. Since elastic systems usually start with a minimal allocation of resources, at the beginning we set the value of variable R equal to R_{min}:

$$R = R_{min} \qquad (\mathcal{M}_{init})$$

A cloud-based elastic system must change its resources allocation according to the decisions made by the elastic controller. These decisions result in the requests M_{start} and M_{stop}, to allocate or deallocate resources, which ultimately determine the amount of resources that are actually in use by the system, which we model with the variable R. The value of R has to change when either an M_{start} or M_{stop} event occurs. We set an arithmetic constraint on the value of R with the following formulae:

$$\forall id : \mathsf{G}(M_{start}(id) \rightarrow R = \mathsf{Y}(R) + 1) \qquad (\mathcal{M}_i)$$
$$\forall id : \mathsf{G}(M_{stop}(id) \rightarrow R = \mathsf{Y}(R) - 1) \qquad (\mathcal{M}_d)$$

The first formula states that after receiving the request to instantiate a new virtual machine the number of resources in use by the system must be increased; conversely, the second formula states that when there is a request to shut down a virtual machine, the number of resources must be decreased.

We also require that changes to the allocation of resource should happen only as a consequence of an M_{start} or M_{stop} request, triggered by the elastic controller.[3] This requirement is expressed as a constraint on changes to the value of R with the following formulae:

$$\exists id : \mathsf{G}(R = \mathsf{Y}(R) + 1 \rightarrow M_{start}(id)) \qquad (\mathcal{M}_{ix})$$
$$\exists id : \mathsf{G}(R = \mathsf{Y}(R) - 1 \rightarrow M_{stop}(id)) \qquad (\mathcal{M}_{dx})$$

We explicitly require that the value of R must not change if neither M_{start} nor M_{stop} occurs with the formula:

$$\mathsf{G}((\forall id : \neg M_{start}(id) \wedge \neg M_{stop}(id)) \leftrightarrow R = \mathsf{Y}(R)) \qquad (\mathcal{M}_{eq})$$

[3] We assume that the elastic controller is the only component that can issue the M_{start} and M_{stop} requests.

After formalizing resource change over time, we introduce the concepts of *eagerness* and *sensitivity*, which capture dynamic aspects of an elastic behavior, such as the speed of adaptation and the minimum variation in the load that triggers an adaptation.

Eagerness informally refers to the speed of the reaction of a system upon a change of the load. It captures the fact that elastic systems must adapt to changes in the workload in a timely manner. We introduce parameter T_e to represent the maximum amount of time within which an elastic system must react to change in the load.

Sensitivity informally refers to the minimum change in the load that should trigger adaptation. It captures the fact that different elastic systems may react to different load intensities and variations. Sensitivity prevents the system from adapting to small, transient changes in the load, possibly creating unnecessary costs. We model the sensitivity of a system with the parameter Δ. This parameter defines a range over the currently measured load: if the load stays within this range, then adaptations are not necessary and will not be triggered. The parameter Δ can assume values from the $[0, R_{max})$ interval. The case $\Delta = 0$ identifies an elastic system that reacts to any change in the load. Under the "manageable load" assumption (see Formula $\mathcal{M}_{load}$), the case $\Delta = R_{max}$ would identify a system that is totally insensitive to the load, i.e., a system that does not adapt. In our understanding this is not an elastic system, and we do not allow this behavior.

We introduce an auxiliary variable, $L_a \in V$, to model eagerness and sensitivity. This variable accumulates the change in the value of the load L. The behavior of L_a is constrained as shown below:

$$L_a = 0 \qquad (\mathcal{P}_{rt1})$$

$$\mathsf{G}((-\Delta \leq L_a \leq \Delta) \rightarrow \mathsf{X}(L_a) = L_a + \mathsf{X}(L) - L) \qquad (\mathcal{P}_{rt2})$$

The first formula initializes the value of L_a to zero. The second formula constraints the value of L_a to change only if its value stays within the threshold defined by the parameter Δ; the value of L_a is incremented according to the difference of the values of the system load L in two consecutive time positions ($\mathsf{X}(L) - L$).

We can now characterize the behavior of a system when scaling up and down occur in terms of variables R and L_a:

$$\mathsf{G}((L_a > \Delta) \rightarrow (\mathsf{X}(L_a) = \mathsf{X}(L) - L \wedge \mathsf{F}_{(0,T_e]}(\mathsf{X}(R) > R))) \qquad (\mathcal{P}_{rt3})$$

$$\mathsf{G}((L_a < -\Delta) \rightarrow (\mathsf{X}(L_a) = \mathsf{X}(L) - L \wedge \mathsf{F}_{(0,T_e]}(\mathsf{X}(R) < R))) \qquad (\mathcal{P}_{rt4})$$

The two formulae express the possible changes in the number of allocated resources: either an increase (denoted with the arithmetical constraint

$\mathsf{X}(R) > R$ in $\mathcal{P}_{rt3}$) or a decrease (denoted with $\mathsf{X}(R) < R$ in $\mathcal{P}_{rt4}$). In both cases, the adaptation is triggered when the value of L_a (the accumulated change of the load) exceeds the threshold defined by parameter Δ. Moreover, the number of resources R is constrained to change within the temporal bound T_e. Furthermore, the formulae constrain also the value of L_a in the next instant, by "resetting" the value accumulated so far.

Plasticity. A distinctive characteristic of elastic systems is their ability to release resources when the load decreases. In particular, when the load drops to zero an elastic system must be able to deallocate all its resources within a reasonable time and return to the minimal configuration of resource allocation. We call systems that do not show this behavior *plastic*; a plastic system is a system that cannot return to its minimal configuration after increasing the number of resources. Ideally, a truly elastic system should never show a plastic behavior.

We introduce parameters T_{p_1} and T_{p_2}. Parameter T_{p_1} indicates for how long the system needs to experience no load, before deallocating all the resources; it is useful to avoid reacting to transient and short-term changes in the load. The other parameter T_{p_2} represents the maximum time the system has to react if a complete deallocation of resources is needed. The following formula characterizes a system that is *not* plastic:

$$\mathsf{G}(\mathsf{G}_{(0,T_{p_1}]}(L = 0) \rightarrow \mathsf{F}_{(0,T_{p_2}]}R = R_{min}) \qquad (\mathcal{P}_{pl})$$

It states that, for all time positions, if a load is equal to zero in a time range bounded by T_{p_1}, then the number of resources will return to its minimal configuration within T_{p_2}. The violation of Formula ($\mathcal{P}_{pl}$) is a sufficient condition for a system to be plastic.

6.5.2 Resource Management

There is a variety of valid elastic behaviors that our model allows. In this section, we list some properties that can be used to better characterize these behaviors. All the properties described in this section focus on resource management, that is, how resources are allocated and deallocated by the system.

Precision identifies how good is the elastic system in allocating and deallocating the right number of resources with respect to variation in the load. In other terms, precision constrains the amount of resources that system is allowed to over- or under-provision. We capture precision by means of parameter ϵ and Formula $\mathcal{P}_{div}$, under the "manageable load" assumption:

$$\mathsf{G}(|R - L| < \epsilon) \qquad (\mathcal{P}_{div})$$

The formula[4] states that during the course of system execution the overall difference between the load and the resources allocation cannot differ more than the specified amount ϵ. This parameter should be defined by the designer of the cloud-based application depending on its requirements.

Oscillation. An elastic system that repeatedly allocates and deallocates resources even when the load stays stable is said to *oscillate.* Oscillations may appear as a consequence of the discrete nature of resources allocation in combination with poorly-designed conditions that trigger adaptation. For example, an elastic controller might try to allocate an average capacity of 1.5 virtual machines by switching between the allocation of one virtual machine and two virtual machines. Despite oscillations being a valid elastic behavior, they might impact on the running costs of the system. We characterize *non-oscillating* behaviors with the following formulae:

$$\mathsf{G}(\mathsf{X}(R) > R \rightarrow \mathsf{P}_{(0,T_e]}(\mathsf{X}(L) > L)) \qquad (\mathcal{P}_{po1})$$
$$\mathsf{G}(\mathsf{X}(R) < R \rightarrow \mathsf{P}_{(0,T_e]}(\mathsf{X}(L) < L)) \qquad (\mathcal{P}_{po2})$$

The formulae constrain the increase (decrease) of the number of resources only in correspondence with an increase, respectively an decrease, of the load. The formulae use the eagerness parameter (T_e) to limit the observation of load variations in the past (expressed with $\mathsf{P}_{(0,T_e]}$). If the controller performs an adaptation not "justified" by a change in the load, it will violate the property captured by the formulae above.

Resource Thrashing. Elastic systems may present opposite adaptations in a very short time; for example, a system may scale up, and then, right after finishing the adaptation, it can start to scale down. This situation is commonly known as *resource thrashing*. In other words, resource thrashing is a temporary, yet very quick, oscillation in the allocation of resources. In the case of a resource thrashing situation, the resources that are impacted by the adaptation generally do not perform any useful work, yet they contribute to an increment of the running costs. Resource thrashing is parametrized by a minimum time T_{rtx} allowed between an increase and a decrease in the number resources. This time is usually defined by the designer of the cloud-based application, after taking into account the actuation delay of the controller. For a system *not* manifesting a resource trashing condition, the

[4]For clarity, in the formula we use the metric $|\cdot|$, which does not belong to the SOLOIST$^{\mathcal{A}}$ syntax. A SOLOIST$^{\mathcal{A}}$ compliant formulation can be obtained by applying the following rule: $|a| \sim b \equiv (a \sim b \wedge a \geq 0) \vee (-a \sim b \wedge a < 0)$, where $\sim$ is a relational operator.

following formulae should hold:

$$\mathsf{G}(R < \mathsf{X}(R) \rightarrow \neg \mathsf{F}_{(0,T_{rtx}]}(R > \mathsf{X}(R))) \qquad (\mathcal{P}_{rtx1})$$
$$\mathsf{G}(R > \mathsf{X}(R) \rightarrow \neg \mathsf{F}_{(0,T_{rtx}]}(R < \mathsf{X}(R))) \qquad (\mathcal{P}_{rtx2})$$

The formulae constrain the occurrence of opposite adaptations to happen after a minimum amount of time T_{rtx}.

Cool-down period is a strategy adopted by designers to achieve a bounded number adaptations over a period of time. It is used to prevent the controller from adapting faster than the time needed for the actual actuation on the cloud-based system. The controller is required to *freeze* for a given amount of time and let the system stabilize after an adaptation. We consider a system unstable if it is in the process of adaptation; this is represented by proposition A. In the following formulae

$$\mathsf{G}\left(\forall id : \begin{pmatrix} \neg M_{ready}(id) \mathsf{S} M_{start}(id) \\ \vee \\ \neg M_{end}(id) \mathsf{S} M_{stop}(id) \end{pmatrix} \rightarrow A\right) \qquad (\mathcal{M}_{ai})$$

$$\mathsf{G}\left(\exists id : A \rightarrow \begin{pmatrix} \neg M_{ready}(id) \mathsf{S} M_{start}(id) \\ \vee \\ \neg M_{end}(id) \mathsf{S} M_{stop}(id) \end{pmatrix}\right) \qquad (\mathcal{M}_{adp})$$

we yield the proposition A true whenever an adaptation is currently in progress: either event $M_{start}(id)$ or $M_{stop}(id)$ were issued, but no M_{ready} (respectively M_{end}) event is observed. This notions are expressed using the *"Since"* (S) modality. We can then use proposition A to express the fact that the controller needs to wait for all recently allocated resources to be ready before performing a new adaptation. This can be represented as a constraint on R to not change when A holds:

$$\mathsf{G}(A \rightarrow \mathsf{Y}(R) = R) \qquad (\mathcal{P}_{cdp})$$

Bounded concurrent adaptations. Sometimes forcing the controller not to react during adaption can be considered a very rigid policy. We can relax this requirement by allowing the controller a fixed number of actions during the adaptation. This property can be viewed as a generalization of the previous one, where the fixed number of actions during adaption was one. To formalize this property we rely on formulae $\mathcal{M}_{ai}$ and $\mathcal{M}_{adp}$ to distinguish the time positions during which an adaptation of the system occurs. The constant M_a represents the maximum number of allowed actions for the controller (either allocations or deallocations) while the system is

in the unstable state. In the formalization, we also introduce an additional variable c_a that counts how many overlapping adaptations occur.

$$c_a = 0 \qquad (\mathcal{P}_{bca1})$$
$$\mathsf{G}((\mathsf{Y}(R) \neq R) \rightarrow \mathsf{X}(c_a) = c_a + 1) \qquad (\mathcal{P}_{bca2})$$
$$\mathsf{G}((A \wedge \mathsf{Y}(R) = R) \rightarrow \mathsf{X}(c_a) = c_a) \qquad (\mathcal{P}_{bca3})$$
$$\mathsf{G}(\neg A \rightarrow \mathsf{X}(c_a) = 0) \qquad (\mathcal{P}_{cdp3})$$
$$\mathsf{G}(c_a < M_a) \qquad (\mathcal{P}_{bca4})$$

Formula ($\mathcal{P}_{bca1}$) initializes the variable c_a at position 0. Formulae ($\mathcal{P}_{bca2}$) and ($\mathcal{P}_{bca3}$) update c_a. Formula ($\mathcal{P}_{bca2}$) increases c_a when there is a change in the number of resources ($\mathsf{Y}(R) \neq R$), while formula ($\mathcal{P}_{bca3}$) propagates the current value of c_a during adaptation (hence, A is required to hold in its antecedent). When the system is not adapting (denoted by $\neg A$) we reset the value of c_a to zero, expressed in ($\mathcal{P}_{cdp3}$). Finally, we constrain the value of c_a to be less then M_a over the whole execution trace.

Bounded resource usage. The running costs of elastic systems can be constrained by specifying properties that apply on the whole set of resources in use. For example, we can specify a constraint on the absolute amount of resources in use by the system, as done in Section 6.4 with Formula $\mathcal{M}_{bound}$. We can also specify time-dependent constraints that temporarily bound the maximum number of resources to certain predefined levels. Time dependent constraints are useful if the budget allocated to the elastic system is very limited, and one must guarantee that the system will run for a given period of time. If the budget is exhausted while the system is still running, the infrastructure abruptly stops and deallocates all the virtual machines. To avoid this situation, an elastic system might need to limit the use of resources beyond a certain threshold, for a specified time interval. We call this requirement *bounded resource usage* and define two parameters to characterize it: a stricter resource bound R_{tmax} with $R_{tmax} < R_{max}$, that represents the temporary new threshold for allocating resources, and a time bound T_{bru}, within which resources above the threshold R_{tmax} should be released. This requirement is expressed as follows:

$$\mathsf{G}(R > R_{tmax} \rightarrow \mathsf{F}_{(0,T_{bru}]}(R \leq R_{tmax})) \qquad (\mathcal{P}_{bru})$$

The formula states that whenever the controller allocates more resources than allowed by the temporary threshold, it needs to release them within T_{bru} time.

6.5.3 Quality of Service

As any other computer system, elastic systems must provide some predefined level of QoS; however, cloud-based elastic systems introduce a new way to enforce them with respect to non-elastic systems. In the rest of this section, we describe some properties that determine the QoS perceived for a cloud-based system.

Bounded QoS degradation. Implementing a system adaptation, such as scaling up or down resources, may incur in non-trivial operations inside the system. Component synchronization, registration, data replication and data migration are just the most widely known examples. During systems adaptation it may happen that the system shows a degraded QoS. Elastic systems may be required to limit this amount of QoS degradation.

Assuming that the level of quality of service is measurable with a single value, we model the normally-required QoS limit with the parameter c. In addition, we define the parameter d to model the reduced (degraded) bound on the value of QoS ($c \geq d$). We formalize the requirements on *bounded degradation during adaptation* as follows:

$$\mathsf{G}(A \rightarrow Q > d) \wedge \mathsf{G}(\neg A \rightarrow Q > c) \qquad (\mathcal{P}_{bqos})$$

The formula above says that the threshold on the normally-required QoS level c should be satisfied only when the system is not performing any adaptation. Instead, during adaptation, the relaxed value d for the QoS is enforced.

Bounded actuation delay. The performance of an elastic system is greatly impacted by the reaction time of its controller, and a controller with a slow reaction time may determine non-effective elastic systems, because adaptations are triggered too late. However, even though the controller triggers an adaptation in time, the system could still be non-effective if the resources take too long to be ready. For this reason, we constrain the actuation delay of the controller with the following formulae:

$$\forall id : \mathsf{G}(M_{boot}(id) \rightarrow \mathsf{F}_{(0,T_{ad})}(M_{ready}(id))) \qquad (\mathcal{P}_{bad})$$
$$\forall id : \mathsf{G}(M_{stop}(id) \rightarrow \mathsf{F}_{(0,T_{ad})}(M_{end}(id))) \qquad (\mathcal{P}_{bad'})$$

We introduce a temporal bound, denoted by parameter T_{ad}, in Formula ($\mathcal{P}_{bad}$) (respectively, ($\mathcal{P}_{bad'}$)) between occurrences of the $M_{boot}(id)$ and $M_{ready}(id)$ events (respectively, $M_{stop}(id)$ and $M_{end}(id)$). Intuitively, the time needed for the application to be ready to serve requests must be less than T_{ad}.

Table 6.1: *Time T(s) and Memory M(MB) evaluation data of the Elastic Doodle service*

	Trace			RT	PL	CDP
ID	Events	Time span (s)	Max resources	T(s)/M(MB)	T(s)/M(MB)	T(s)/M(MB)
T1	15	1102	2	1.44/120.1	1.20/117.7	2.29/126.2
T2	43	635	4	2.83/135.3	1.47/121.8	1.42/121.5
T3	29	641	3	1.77/131.5	1.21/117.7	1.62/126.2
T4	17	499	3	1.20/116.7	1.27/116.0	1.38/115.9
T5	44	644	3	2.94/135.4	1.45/122.1	1.45/121.7

6.6 Trace checking

We evaluated our formalization of the relevant properties of cloud-based systems by determining whether it could be effectively applied to check this class of properties over execution traces of realistic applications. In particular, we performed trace checking of some of the properties described in Section 6.5 over system execution traces. Our goal was to evaluate the resource usage (execution time and memory) of the trace checking procedure for the different types of properties and if the properties are violated by our example cloud-based elastic system.

6.6.1 Methodology

Traces were obtained from the execution of an instance of the "Elastic Doodle" service (see below), deployed over a private OpenStack cloud infrastructure. To trigger elastic behaviors in the application, we created several input workloads, fluctuating according to sine waves, squared waves, and sawtooth patterns. We configured the monitoring tools of the application to create a set of execution traces, each of them containing the timestamped events corresponding to the allocation and deallocation of virtual machines. We leveraged the AUToCLES tool [70] to automate the execution of multiple runs of the system with different input workloads, and to perform data collection.

The following properties were selected for verification over the generated traces: (RT) Resource thrashing; (PL) Plasticity; (CDP) Cool down period.

We used the ZOT verification toolset enhanced with our two plugins for trace checking. We translated the traces collected from the execution of the "Elastic Doodle" service into a SOLOIST$^{\mathcal{A}}$ formula, where each occurrence of a virtual machine allocation or deallocation event is mapped onto an atomic proposition holding at the corresponding timestamp. The logical

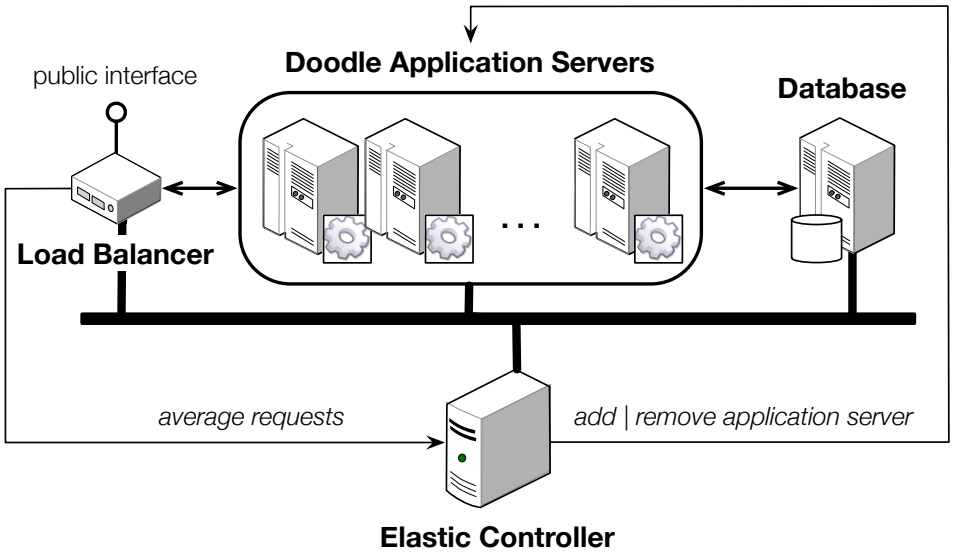


Figure 6.4: *High-level architecture of the "Elastic Doodle" service.*

conjunction of this formula, of the formulae presented in Section 6.4, and of the SOLOIST$^{\mathcal{A}}$ version of each of the properties to verify was then provided as input to ZOT. For each verification run, we recorded the memory usage and the SMT verification time.

6.6.2 The Elastic Doodle

The "Elastic Doodle" service is an open-source clone of the popular Internet calendar tool of the same name. We extended the original code base to support elastic behaviors, by including both the adaptation logic and the elastic controller. We also added advanced monitor capabilities.

Figure 6.4 shows the high-level architecture of this elastic service. The system is organized as a n-tier system with a *load-balancer* that exposes the service endpoint to end-users on one side, and forwards client requests to a lineup of *application servers* on the other side. The application servers interact with a shared *database* that acts as the storage/persistency tier of the system. The middle tier has an elastic behavior: instances of the applications server composing this tier can be dynamically added and removed. This elastic behavior is determined by the *controller*, which periodically (e.g., every ten seconds) reads the monitored data and decides on the next resource allocation strategy using a rule-based approach. The following two rules are in place:

scale-up: if the average number of requests per running application server in the last minute is over a certain maximum threshold, a new instance of application server is allocated; the controller stops its execution for one minute;

scale-down: if the average number of requests per running application server in the last minute is below a certain minimum threshold, a running instance of application server is deallocated; the controller stops its execution for two minutes.

6.6.3 Results and Discussion

Our trace checking procedure processes the logs generated by the Elastic Doodle service and filters only significant events like M_{start}, M_{boot}, M_{ready}, M_{stop}, M_{end} and any change in the value of the load assigned to L. These events were conjuncted into a formula representing the trace. Let us define a formula $(\mathcal{M}_R)$ as a conjunction of all formulae that define the behavior of the variable R: $(\mathcal{M}_{init}) \wedge (\mathcal{M}_{load}) \wedge (\mathcal{M}_i) \wedge (\mathcal{M}_d) \wedge (\mathcal{M}_{ix}) \wedge (\mathcal{M}_{dx}) \wedge (\mathcal{M}_{eq})$. Similarly, we define $(\mathcal{M}_A)$ as the conjunction of formulae that define the behavior of proposition A: $(\mathcal{M}_{ai}) \wedge (\mathcal{M}_{adp})$.

For the property (RT) we perform bounded satisfiability checking of the formula $\mathcal{F}_{RT}$= $(\mathcal{M}_R) \wedge \neg ((\mathcal{P}_{rtx1}) \wedge (\mathcal{P}_{rtx2}))$ over the traces. We conjunct the term $(\mathcal{M}_R)$ in $\mathcal{F}_{RT}$ because resource thrashing formula relies on variable R. We choose T_{rtx} to be 50 seconds. For the property (PL) we check formula $\mathcal{F}_{PL}$=$(\mathcal{M}_R) \wedge \neg (\mathcal{P}_{pl})$. Plasticity formula also uses variable R, hence the conjunct $(\mathcal{M}_R)$. We choose T_{p_1} to be 3 minutes and T_{p_2} 30 seconds. Finally, for the property (CDP) we check $\mathcal{F}_{CDP}$=$(\mathcal{M}_R) \wedge (\mathcal{M}_A) \wedge \neg (\mathcal{P}_{cdp})$. Since cool down period formula relies both on variable R and proposition A we conjunct their definitions with $\mathcal{F}_{CDP}$.

Notice that in $\mathcal{F}_{RT}$, $\mathcal{F}_{PL}$ and $\mathcal{F}_{CDP}$ we are negating the property formulae. This is done because we perform satisfiability checking and the models of the formulae in fact represent counterexamples of the properties.

We chose five traces with different values for the number of significant events, the time span, and the maximum number of resources being allocated during execution. We report time and memory results for the verification of each property RT, PL and CDP in Table 6.1. Besides the size of the formula, the parameters that affect the time and memory of the verification procedure are the number of significant events in the traces and the number of resources allocated during the execution.

The trace checking procedure confirmed that traces T1, T3 and T5 satisfy all three properties. Trace T2 violates property RT due to deallocation of a resource 30 seconds after its allocation; it also violates property PL since not all resources are deallocated in the last 3 minutes. Trace T4 violates both properties RT and CDP because of a single (wrong) decision made by the elastic controller: it deallocated a resource while another resource was still initializing, within 50 seconds.

The results of the evaluation suggest that checking the proposed properties formalized in SOLOIST$^{\mathcal{A}}$ over realistic execution traces is feasible, since the time needed for executing the checking is small (1.66 seconds on average) and the amount of memory required is reasonable (123MB on average). Thus, we answer the RQ4 from Chapter 5.

CHAPTER 7

State of the Art

This work lies in the wider area of research on verification of SBAs; reader may refer to various surveys [11, 40, 42, 110], illustrating approaches both for design-time and for run-time verification of functional and QoS properties. The rest of this section focuses on existing work on trace checking and verification of quantitative properties specified in languages similar to SOLOIST, as well as other attempts at developing specification languages for expressing quantitative properties.

Trace checking of quantitative properties

Finkbeiner et al. [66] describe an approach to collect statistics over run-time executions. They extend LTL to return values from a trace and use them to compute aggregate properties of the trace. However, the specification language they use to describe the statistics to collect provides only limited support for timing information. For example, compared to SOLOIST, it cannot express properties on a certain subset of an execution trace. Furthermore, their evaluation algorithm relies on the formalism of algebraic alternating automata. These automata are manually built from the specification; thus making frequent changes to the property error-prone.

In reference [18] authors define an extension of metric first-order temporal logic (called MFOTL$_\Omega$) which supports aggregate modalities from a set Ω. This language is very similar to SOLOIST and it supports any aggregate modality defined as a mapping from multi-sets of values from some domain D to $\mathbb{Q} \cup \{\bot\}$. A finite multi-set is mapped to a rational number while infinite multi-set is mapped to $\bot$. MFOTL$_\Omega$ is strictly more expressive than the propositional version of SOLOIST considered in this thesis. Languages also differ in the way the aggregate modalities are defined: MFOTL$_\Omega$ expresses aggregate properties over the values of the parameters of first-order relations from the language signature, while SOLOIST expresses aggregate properties on the occurrences of propositions in the temporal structure ($\mathfrak{C}$, $\mathfrak{U}$ and $\mathfrak{M}$ modalities), as well as on the sequences of timestamps ($\mathfrak{D}$ modality). In order to encode a SOLOIST formula into MFOTL$_\Omega$, one needs to extend relation signatures with two parameters: one assuming a constant value 1 and the other assuming the value of the timestamp at each position of the trace. Former can be used to aggregate on the occurrences of relations, while latter on the timestamps.

In reference [22], authors introduce a specification language $PTLTL^{FO}$ (past time linear temporal logic with first-order (guarded) quantifiers) with a counting quantifier. It is used for expressing policies that can categorize the behavioral patterns of a user based on its transaction history. The counting quantifier counts the occurrences of an event from the beginning of the trace until the position of evaluation. The difference with the $\mathfrak{C}$ modality of SOLOIST is that there is no timing information: this means one cannot specify the exact part of the trace the modality should consider.

In reference [51], de Alfaro proposes pTL and pTL* as probabilistic extensions of CTL and CTL*. These new languages include a new modality $\mathfrak{D}$ that expresses the bound on the average time between events. This is achieved by using an instrumentation clock that keeps track of the elapsed time from the beginning of the computation until the first occurrence of a specified event. To this end, the extended pTL formulae are evaluated on an instrumented timed probabilistic Markov decision process. Notice that the $\mathfrak{D}$ modality used in [51] differs from the one we introduced here, since it computes the time passed before the first occurrence of an event, averaged over the different computations of the underlying Markov decision process.

In [55] the authors define a temporal extension of the OCL specification language [72] limited, however, to expressing only common properties classified within the well-known property specification patterns [60,73,82]. The authors translate their extension of the language into the original OCL fragment in order to reuse the existing OCL checkers to perform trace

checking of real time properties.

Du et al. [58] propose an extension of past-time linear temporal logic (PTLTL) using an aggregate operator called *counting quantifier*, similar to the $\mathfrak{C}$ modality proposed in this thesis. It is able to express the number of times some sub-policy is satisfied in the past. The authors define a fragment of the proposed specification language that counts only values with periodic occurrence and therefore can be monitored using constant space. The authors propose an algorithm that tracks only the finite set of equivalence classes for the periodic values and although the language does not support timing information and other possible aggregate operators, it presents an interesting contribution that can be considered orthogonal to our work.

In [107] the authors propose *counting fluents* as a generalization of propositional fluents used in the fluent linear temporal logic. Counting fluents represent abstract states in event-based systems whose values depend on the particular events that occurred in the execution of the system. The concept of a counting fluent is very similar to the *counters* (and *arithmetical variables*) used to encode SOLOIST modalities, however counters can be used to encode any arbitrary complex operator and are not constrained to increase only when particular events occur.

This work is also related to approaches for SAT/SMT-based trace checking and bounded model checking, which is usually done over properties expressed in conventional temporal logics. For example, the SAT-based approach for bounded model checking proposed in [119] verifies Metric Temporal Logic (MTL) properties of discrete timed automata. SMT-based techniques like those proposed in [27, 28, 81] deal with verification of MTL over real-valued words.

Specification of quantitative properties

Dustdar et al. [59] describe the main principles of elasticity in the context of elastic processes.

There have been few proposals for modeling and formalizing elastic properties in the literature so far. Herbst et al. [77] highlight the need for a precise definition of elasticity in the context of cloud computing. Similarly to the modeling approach followed in this paper, the authors characterize the degree of elasticity in terms of speed of adaptation and precision of adaptation. Starting from basic concepts such as adaptation, demands and capacity the authors define a set of properties to describe the elastic behavior of cloud-based systems. However, these descriptions are informal; the paper only described a set of metrics for measuring system elasticity.

Islam et al. [79] provide a quantitative definition of elasticity using financial terms, taking the point of view of a customer of an elastic system who wants to measure the elasticity provided by the system. The authors measure the financial penalty for systems under-provisioning (due to SLA violations) and over-provisioning (unnecessarily costs) using a reference benchmark suite to characterize the system elasticity. Several critical situations identified in [79] have been reported/discussed in this paper. However, Islam and coauthors provide only an informal description for the properties.

A formal definition and modeling of system plasticity is provided in [69]. The authors model elastic systems by means of state transition systems where transitions are associated with probabilities of switching between states, i.e., different resources allocations, as they are observed in the system run. Plasticity is identified when the model has transitions corresponding to scaling up but lacks (some) transitions corresponding to scaling down. The authors use the proposed model to define an automated procedure for the generation of test cases that expose plastic behaviors of cloud-based elastic systems.

The work from [44, 45, 75, 76] is a part of larger research agenda that aims at providing a complete quantitative generalization of formal languages and their decision procedures. In particular in [44] the authors define nested weighted automaton that serves as an operational formalization of quantitative properties.

Part III

Scalable Trace Checking

CHAPTER 8

Distributed and Parallel Trace Checking

8.1 Overview

Software systems have become more complex, distributed, and increasingly reliant on third-party functionality as discussed in Section 1.1. The dynamic behavior of such systems makes traditional design-time verification approaches unfeasible, because they cannot analyze all the behaviors that can emerge at run time, thus there is a need for *runtime* or *post-mortem* verification techniques. This chapter focuses on the *post-mortem* technique called *trace checking*. To perform trace checking one must first collect and store relevant *execution traces* produced by the system and then check them *offline* against the system specifications.

The volume of the execution traces gathered for modern systems increases continuously as systems become more and more complex. For example, an hourly page traffic statistics for Wikipedia articles collected over a period of seven months amounts to $320\mathrm{GB}$ of data [4]. Execution traces easily get very large, depending on the 1) running time captured by the logging infrastructure (e.g., traces are captured over weeks, months or even years); 2) the complexity of the systems the log refers to (e.g., several virtual machines running on a cloud-based infrastructure); 3) the types of

events recorded (e.g., high-level events like service invocations or low-level events like method calls); and 4) the granularity of the captured events (e.g., every second or millisecond).

This huge volume of trace data challenges the scalability of current trace checking tools [17, 63, 78, 108, 114], which are centralized and use sequential algorithms to process the trace. Some log analyzers that process data streams [49] or perform data mining [117] also partially solve this problem, because of the limited expressiveness of the specification language they support. Indeed, the analysis of a trace may require checking for complex quantitative properties, which can refer to specific sequence of events, conditioned by the occurrence of other event sequence(s), possibly with additional constraints on the distance among events, on the number of occurrences of events, and on various aggregate values (e.g., average response time). As we discussed in Section 2.2, SOLOIST specification language can express these kind of properties.

The recent advent of cloud computing has made it possible to process large amount of data on networked commodity hardware, using a distributed model of computation. One of the most prominent programming models for distributed, parallel computing is *MapReduce* [53]. The MapReduce model allows developers to process large amount of data by breaking up the analysis into independent tasks, and performing them in parallel on the various nodes of a distributed network infrastructure, while exploiting, at the same time, the locality of the data to reduce unnecessary transmission over the network. However, porting a traditionally-sequential algorithm (like trace checking) into a parallel version that takes advantage of a distributed computation model like MapReduce is a non-trivial task as trace checking is inherently a sequential task.

The main contribution we present in this chapter is an algorithm that exploits the MapReduce programming model to check large execution traces against requirements specifications written in SOLOIST. The algorithm exploits the structure of a SOLOIST formula to parallelize its evaluation, with significant gain in time. The algorithm's iterative execution flow is inspired by the algorithm from [12] developed to check LTL specifications.

An in-depth description of the algorithm based on MapReduce is given in Section 8.2. Section 8.3 describes how the algorithm can be ported to the Spark framework [120, 121] in order to obtain better performance.

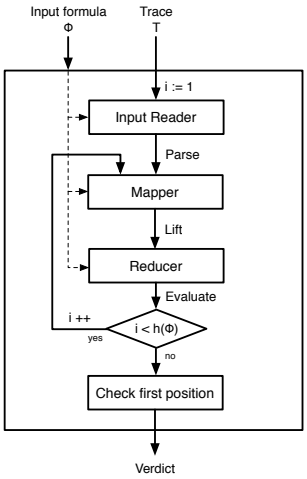


Figure 8.1: *High-level overview of the trace checking algorithm*

8.2 Trace checking with MapReduce

In this section presents the trace checking algorithm that uses MapReduce framework to check a trace in a distributed and parallel manner. The main difference between the version presented in this section and in [33], where algorithm is initially presented, is in how the timing information is distributed. Instead of using the globally-shared associative list data structure [33], we make the timing information part of the (input, intermediate and output) tuples, since it is more elegant and does not change the performance of the algorithm.

The algorithm takes as input a non-empty execution trace T and an SOLOIST formula Φ and provides a verdict whether the trace satisfies the formula. Figure 8.1 shows a high-level overview of the algorithm's control flow. The trace is modeled as a timed word, i.e., we have $T = (\sigma, \tau)$. We call $p_i = (\sigma_i, \tau_i)$ an element of the trace T at position i. It contains the set of atoms $\sigma_i \subseteq \Pi$ that hold at position i and an integer timestamp τ_i. We assume that the execution trace is saved in the distributed file system of the cluster on which the distributed algorithm is executed. This is a realistic assumption since in a distributed setting it is possible to collect logs, as long as there is a total order among the timestamp induced by some clock synchronization protocol.

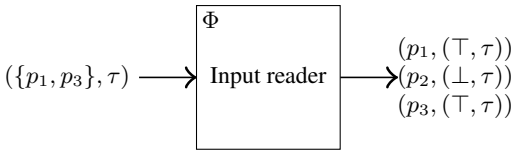
The trace checking algorithm processes the trace iteratively, through a sequence of MapReduce executions. The number of MapReduce iterations is equal to the height of the SOLOIST formula Φ. The first MapReduce

iteration parses the input trace from the distributed file system, applies the `map` and `reduce` functions and passes the output (a set of tuples) to the next iteration. Each subsequent iteration receives the set of tuples from the respective previous iteration in the expected internal format, thus parsing is performed only in the first iteration. A subsequent iteration l (where $1 < l \leq h(\Phi)$) receives the set of tuples from the iteration $l-1$. The set of tuples contains all the positions where the subformulae of Φ of height $l-1$ hold. Note that the trace itself is a similar set, containing all the positions where the atoms (with a height 1) hold. Based on the set it receives, the l-th iteration can then calculate all the positions where the subformulae of height $l+1$ hold. Note that, atoms have a height 1 and therefore checking an atomic formula needs no MapReduce iterations. Each iteration consists of three phases: 1) *read phase* that reads and splits the input; 2) *map phase* that associates each formula with its superformula; and 3) *reduce phase* that applies the semantics of the appropriate subformula of Φ. The final set of tuples represents the all the positions where the input SOLOIST formula holds, thus to produce the verdict it is only a matter of checking if it holds in the first position.

8.2.1 Read phase

```
function INPUT READER^Φ(T_k[])
  for all (σ, τ) ∈ T_k[] do
    for all p ∈ sub_a(Φ) do
      output(p, (p ∈ σ, τ))
    end for
  end for
end function
```

(a) *Input Reader algorithm*



(b) *Block diagram of the Input reader*

Figure 8.2: *Input reader algorithm (*sub_a *is defined in Section 2.2)*

The input reader component of the MapReduce framework is used in this phase; this component can process the input trace in a parallel way. The trace saved in a distributed file system is split into several blocks (usually 64MB in size), replicated (usually 3 times) and distributed evenly among the nodes. The MapReduce framework exploits this block-level parallelization both during the read and map phases. For example, the default block size of the Hadoop deployment is 64MB, which means that a 1GB trace is split in 16 parts and can be potentially processed using 16 parallel readers and mappers. However, if we execute the algorithm on 3 nodes with 4 cores each, we could process up to 12 blocks in parallel. The input reader is used

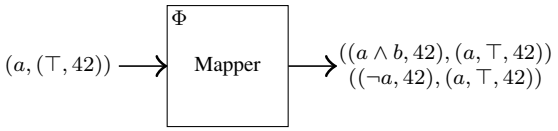
only in the first iteration and can be seen as a parser that converts the trace into a uniform internal representation that is used in the subsequent iterations. As shown in Fig. 8.2a, the k-th instance of the input reader handles the k-th block T_k of the trace T. For each element (σ, τ) in T_k and every atom p occurring in the SOLOIST formula Φ, the reader outputs a key-value pair of the form $(p, (p \in \sigma, \tau))$. The key is the atomic p itself, while the value is a pair consisting of the truth value of p at time τ (obtained by evaluating the expression $p \in \sigma$) and the timestamp τ.

Figure 8.2b is a block diagram of the input reader showing on a high level how it transforms an example element $(\{p_1, p_3\}, \tau)$. Suppose that atoms p_1, p_2 and p_3 occur in the SOLOIST formula to be checked. The input reader outputs a tuple for each atom, namely tuples $(p_1(\top, \tau)), (p_2(\bot, \tau))$ and $(p_3(\top, \tau))$. The key of each tuple is the atom itself and the value is a pair consisting of the truth value of the atom and the timestamp τ.

8.2.2 Map phase

function MAPPER$^{\Phi}((\phi, (v, \tau)))$
  for all $\psi \in \mathsf{sup}_{\Phi}(\phi)$ **do**
    output$((\psi, \pm\tau), (\phi, v, \tau))$
  end for
end function

(a) *Mapper algorithm*



(b) *Block diagram of a Mapper*

Figure 8.3: *Mapper algorithm (*sup_{Φ} *is defined in Section 2.2)*

Each tuple generated by an input reader is passed to a mapper on the same node. Mappers associate the formula in the tuple with all its super-formulae in Φ. For example, given $\Phi = (a \wedge b) \vee \neg a$, if the input reader returns a tuple $(a, (\top, 42))$, the mapper will associate it with formulae $a \wedge b$ and $\neg a$, outputting the tuples $(a \wedge b, (a, \top, 42))$ and $(\neg a, (a, \top, 42))$. As shown in Fig. 8.3a, the mapper receives tuples in the form $(\phi, (v, \tau))$ from the input reader and outputs all tuples of the form $((\psi, \pm\tau), (\phi, v, \tau))$ where $\psi \in \mathsf{sup}_{\Phi}(\phi)$ (see Section 2.2 for the definition of sup).

Notice that the key of the intermediate tuples emitted by the mapper has two parts: this type of key is called a *composite key* and it is used to perform the so called *secondary sorting* of the intermediate tuples. As explained in Section 2.5.1 MapReduce framework partitions the intermediate tuples using the partition function and sorts each partition using the comparison operator for each key. Secondary sorting is a technique that allows sorting intermediate tuples not only by the key, but also "by value" and it's achieved

by introducing a composite key and redefining the partition function and the comparison operator. In our case, we perform secondary sorting based on the timestamp τ added to the composite key with either a positive or a negative sign; the sign depends on the type of superformula ψ. If ψ is a future-time temporal operator then a negative sign is used so that the trace is sorted in the reverse order. Otherwise, the trace is sorted in the increasing value of the timestamp. To get an intuition on how the comparison operator is implemented consider the following Java code:

```
@Override
public int compareTo(CompositeKey that) {
     int result = this.key.compareTo(that.key);
     if(result == 0) {
          result = this.time.compareTo(that.time);
     }
     return result;
}
```

The code above ensures that the tuples with the same key are compared according to the value of the timestamps. Therefore the reducers will receive the intermediate tuples in the order of the increasing (or decreasing) value of timestamps. The partition function is defined to consider only the formula in the composite key, so that every reducer will receive tuples with a different formula. Secondary sorting greatly decreases the memory used by the reducer.

8.2.3 Reduce phase

In this phase, each reducer receives tuples with the same formula in the composite key and process them in parallel. The reducers exploit the information produced by the mappers to determine the truth values of the superformula at each position, i.e., reducers apply the appropriate SOLOIST semantics of the appropriate formula. The total number of reducers running in parallel at the l-th iteration is the minimum between the number of subformulae with height l in the input formula Φ and number of available reducers[1]. Each reducer calls an appropriate reduce function depending on the type of formula used as key in the received tuple.

In the rest of this section we present the reduce algorithms for each SOLOIST operator and modality. We also introduce two auxiliary functions checkIteration and updateQueue in Figure 8.4. Function checkIteration (see Algorithm 8.4a) checks if the formula ψ received as a key by a reducer can be handled in the iteration l. As explained earlier at iteration l only

[1]This depends on the configuration of the cluster. Typically, it is the number of nodes in the cluster multiplied by the number of cores available on each node.

```
1  function CHECKITERATION((T, ψ, l))
2    if (h(ψ) > l + 1) then
3      for all (φ, v, τ) ∈ T do
4        output(φ, v, τ)
5      end for
6      return ⊥
7    end if
8    if (h(ψ) = l + 1) return ⊤ end if
9    return ⊥
10 end function
```

(a) *Disjunction*

```
1  function UPDATEQUEUE(((φ, v, τ), win, I, f))
2    if (v) win.enqueue((φ, v, τ)) end if
3    while |τ − win.first.τ| ∉ [0, 0] ⊎ I do
4      f(win.dequeue())
5    end while
6  end function
```

(b) *Conjunction*

Figure 8.4: *Auxiliary functions used by the reducer algorithms*

formulae with height $l + 1$ can be checked, therefore the function returns true, that signals the reducer to proceed with processing the tuples. In other cases the function returns false. Any tuple received *before* the appropriate iteration needs to be retransmitted. This is done in lines 3–5 using a for loop and traversing the intermediate values denoted with variable T. T is a data structure provided by the MapReduce framework for representing intermediate tuples. Traversing T is done by fetching each tuple in a *lazy* fashion and never materializing all the tuples completely in memory. Retransmission of prematurely received tuples is done in order to make sure that they are processed in the appropriate iteration. For example, if we consider a formula $(a \mathsf{U} b) \wedge c$, in the first iteration the mapper would generate tuples of the form $((a \mathsf{U} b, \tau), (a, v, \tau))$ and $((a \mathsf{U} b, \tau), (b, v, \tau))$ for all τ, as well as tuples of the form $(((a \mathsf{U} b) \wedge c, \tau), (c, v, \tau))$. These tuples cannot be handled in the first iteration since the value of the formula $a \mathsf{U} b$ is still not computed. Therefore tuples of the form (c, v, τ) must be retransmitted and considered again in the second iteration. Function updateQueue is a function that updates a queue data structure passed as a parameter win. It is used by the reducer functions for metric temporal operators and aggregate modalities, since implementing their semantics requires storage. As shown in Algorithm 8.4b, updateQueue also takes as parameters a tuple (ϕ, v, τ), an interval I and a generic function f. If value v is true, tuple (ϕ, v, τ) is inserted at the end of the queue win (line 2). After that, in the while loop (lines 3–5) all the tuples from the head of the queue are removed, if their timestamp is not within the interval $[0, 0] \uplus I$ from the timestamp τ. Interval $[0, 0] \uplus I$ is a convex union[2] of intervals $[0, 0]$ and I. Function updateQueue applies the function f to every removed tuple producing a side effect defined by the caller. In other words, updateQueue maintains a

[2] A convex union of intervals is defined as a convex hull of the union of the intervals. For example, convex union of [0,3] and [5,6] is [0,6].

queue of selected events ϕ, with certain truth value v, whose timestamps are in the interval $[0,0] \biguplus I$ from τ. We chose this particular function because it supports a vide range of queue operations that can be used to implement semantics of various temporal formulae.

```
1  function REDUCER_l^¬(ψ, T)
2    if (checkIteration(ψ, T, l))
3    then
4    for all (φ, (v, τ)) ∈ T do
5      output(ψ, (¬v, τ))
6    end for
7    end if
8  end function
```

(a) *Negation operator*

```
1  function REDUCER_l^∧(ψ, T)
2    if (checkIteration(ψ, T, l))
     then
3      τ_prev ← 0, val ← ⊤
4      for all (φ, v, τ) ∈ T do
5        if (τ = τ_prev) val ← val∧v
6        else output(ψ, (val, τ_prev))
7             val ← ⊤
8        end if
9        τ_prev ← τ
10     end for
11     output(ψ, (val, τ_prev))
12   end if
13 end function
```

(b) *Conjunction operator*

```
1  function REDUCER_l^∨(ψ, T)
2    if (checkIteration(ψ, T, l))
     then
3      τ_prev ← 0, val ← ⊥
4      for all (φ, v, τ) ∈ T do
5        if (τ = τ_prev) val ← val∨v
6        else output(ψ, (val, τ_prev))
7             val ← ⊥
8        end if
9        τ_prev ← τ
10     end for
11     output(ψ, (val, τ_prev))
12   end if
13 end function
```

(c) *Disjunction operator*

Figure 8.5: *Reducer algorithms for boolean operators*

Boolean operators

Figure 8.5 shows reducer algorithms for negation, conjunction and disjunction boolean operators.

Negation. When the key refers to a negation superformula, the reducer simply emits a tuple with a negated value. For example if the reducer receives a tuple $(\neg a, (a, \bot, 42))$ it emits a tuple $(\neg a, \top, 42)$. Algorithm. 8.5a shows how output tuples are emitted.

Conjunction. We extend the binary $\wedge$ operator defined in Section 2.2 to any positive arity; this extension does not change the language but improves the conciseness of the formulae and the efficiency of the algorithm, since we reduce the height of the formulae. For example, conjunction $a \wedge b \wedge c \wedge d$ is represented as a single conjunction with 4 subformulae and has height equal to 2, rather than 4. This effectively reduces the number of iterations.

Algorithm 8.5b processes all the tuples sequentially. Since the incoming tuples are sorted by their timestamps, if k is the arity of the conjunction formula, the reducer will receive k tuples with the same timestamp before the timestamp value changes. Therefore, it is enough to track when the value of the received timestamp changes (using a τ_{prev} variable) and record a conjunction of all truth values v with the same timestamp. This is done in line 5 and the partial conjunction of the values is saved in the val variable. As soon as the timestamp changes, the reducer can emit the value of val

variable as the truth value of the conjunction at the time instant τ_{prev} (line 6). The variable val is reset to true (line 7) and the process is continues.

Disjunction. Algorithm 8.5c shows the reducer function for disjunction. It is a dual implementation of Algorithm 8.5b.

```
1  function REDUCER_l^{U,S}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      val ← ⊥
4      for all (φ1, (v1, τ)), (φ2, (v2, τ)) ∈ T do
5        output(ψ, (val, τ))
6        val ← v2 ∨ (v1 ∧ val)
7      end for
8    end if
9  end function
```

(a) *Non-metric* U *and* S *operators*

```
1  function REDUCER_l^{U_I,S_I}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      val ← ⊥, val_m ← ⊥, win ← ∅
4      for all (φ1, (v1, τ)), (φ2, (v2, τ)) ∈ T do
5        if (¬v1) win.removeAll end if
6        updateQueue((φ2, v2, τ), win, I, \x → ())
7        val_m ← |τ − win.first.τ| ∈ I
8        output(ψ, (val ∧ val_m, τ))
9        val ← v2 ∨ (v1 ∧ val)
10     end for
11   end if
12 end function
```

(b) *Metric* U_I *and* S_I *operators*

```
1  function REDUCER_l^{F,P}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      val ← ⊥
4      for all (φ, v, τ) ∈ T do
5        output(ψ, (val, τ))
6        val ← v ∨ val
7      end for
8    end if
9  end function
```

(c) *Non-metric* F *and* P *operators*

```
1  function REDUCER_l^{F_I,P_I}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅
4      for all (φ, v, τ) ∈ T do
5        updateQueue((φ, v, τ), win, I, \x → ())
6        val ← |τ − win.first.τ| ∈ I
7        output(ψ, (val, τ))
8      end for
9    end if
10 end function
```

(d) *Metric* F_I *and* P_I *operators*

```
1  function REDUCER_l^{G,H}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      val ← ⊤
4      for all (φ, v, τ) ∈ T do
5        output(ψ, (val, τ))
6        val ← v ∧ val
7      end for
8    end if
9  end function
```

(e) *Non-metric* G *and* H *operators*

```
1  function REDUCER_l^{G_I,H_I}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅
4      for all (φ, v, τ) ∈ T do
5        updateQueue((φ, ¬v, τ), win, I, \x → ())
6        val ← |τ − win.first.τ| ∈ I
7        output(ψ, (¬val, τ))
8      end for
9    end if
10 end function
```

(f) *Metric* G_I *and* H_I *operators*

Figure 8.6: *Reducer algorithms for temporal operators*

Temporal operators

Figure 8.6 shows the reducer algorithms for *"Until"*, *"Since"*, *"Eventually"*, *"Eventually in the Past"*, *"Globally"* and *"Historically"*[3]. Each algorithm has two variants: non-metric (shown on the left) and metric (shown on the right). Non-metric operators have intervals of the form $[0, +\infty)$, therefore

[3] Also known and "Always in the past" or "Globally in the Past".

algorithms are simpler, as they do not need to consider the timing constraints. Metric operators have arbitrary intervals. We present only the future-time operators, while the past-time operators have the same implementation are their future counterparts. As discussed in Section 8.2.2, the only difference is in the way the trace has been sorted in the shuffle and sort phase of the MapReduce framework.

Until operator. Algorithm 8.6a implements the reducer function for the non-metric "Until" operator. The reducer expects the trace to be sorted in the reverse order, such that the first tuple it receives contains the last timestamp of the trace. The semantics of the "Until" operator is implemented in a straightforward manner using its fix-point definition. Variable val is initialized to false, since the value of the "Until" operator is false in the last position of the trace (see Figure 2.2). Each subsequent value is calculated by iterating through the tuples and considering two tuples at a time, $(\phi_1, (v_1, \tau))$ and $(\phi_2, (v_2, \tau)$ corresponding to the left and right subformula respectively. Line 6, implements the fix-point semantics of "Until" operator using the current truth values of the left (variable v_1) and the right (variable v_2) subformula and the truth value of the "Until" formula from the previous iteration (variable val).

Reducer algorithm for the metric "Until" operator shown in Algorithm 8.6b is inspired by the algorithm from [20] for the "Since" temporal operator, interpreted using point-based MTL semantics. Conceptually, the algorithm is similar to the non-metric version, with the addition of the code that checks the timing constraints. The timing constraints are checked by utilizing a queue (denoted as variable win). The queue is updated using the updateQueue function such that it contains only tuples that refer to the right subformula ϕ_2, with timestamps in the $[0, 0] \uplus I$ interval from the current timestamp τ. To output a tuple with a positive truth value, two conditions need to be satisfied: ϕ_2 must hold at a time instant in the interval I relative to the current timestamp and and ϕ_1 must continuously hold from the current instant until the instant where ϕ_2 holds. The first condition is stored in the variable val_m: it suffices to check if the tuple from the head of the queue has a timestamp in the interval I. Variable val stores the second condition, computed in the same way as in the non-metric algorithm. In order to simplify the code, we omit a possible memory optimization present in [20]. It prescribes to keep in the queue only the most recent tuple that satisfies the timing constraints. Although we omit it here, our prototype tool [84] implements this optimization.

Eventually operator. Algorithm 8.6c implements the reducer function for non-metric "Eventually" operator. It uses an auxiliary boolean variable

val that is initialized to false. It loops through all the tuples received in T, already sorted (by the *shuffle and sort* phase of the MapReduce framework) in descending order of timestamp values. The algorithm updates, for each tuple, the variable *val* with a disjunction of its old value (i.e., from the previous iteration) and a truth value from each current tuple. Variable *val* is emitted in the output tuple as the truth value of $\mathsf{F}\phi$.

Algorithm 8.6d implements the metric version of the reducer function for "Eventually" temporal operator. It uses an additional variable *win* as a queue to keep track of all the tuples with positive truth value that fall in the convex union of the intervals $[0, 0]$ and I. This is ensured by the updateQueue function. The final truth value of $\mathsf{F}_I\phi$ depends on whether the queue *win* contains a tuple with a timestamp that is in the interval I.

Globally operator. Both non-metric and metric versions of "Globally" operator are dual of the corresponding "Eventually" operator. Algorithms 8.6e and 8.6f are, therefore, implemented in a similar way. Algorithm 8.6e initializes val to true, and computes the truth value using the conjunction operator. The only difference introduced in Algorithm 8.6f is that the queue *win* keeps track of all the tuples with negative truth value via the updateQueue function with the parameter v negated. Hence, the truth value of "Globally" operator depends on whether the queue *win* contains a tuple in the interval I that is a witness to its violation (therefore variable val is negated in line 7).

Aggregating modalities

Algorithms implementing the reducer functions for the aggregating modalities are shown in Figure 8.7. For the aggregating modalities the trace is ordered in the increasing value of the timestamp, since they are past-time operators.

$\mathfrak{C}$ **modality.** The reduce function for the $\mathfrak{C}$ modality is outlined in the Algorithm 8.7a. To correctly determine if $\mathfrak{C}$ modality holds, we need to keep track of all the tuples in the past time window $[0, K)$. We use queue win and update it with updateQueue function, such that it contains all the tuples with positive truth value and with timestamps in the past time window $[0, K)$ with respect to the current timestamp. To determine the truth value, the size of the queue is compared to n according to the $\bowtie$ comparison operator.

$\mathfrak{U}$ **modality.** To simplify the presentation, we express the $\mathfrak{U}$ modality in terms of the $\mathfrak{C}$ one, based on this definition: $\mathfrak{U}^{K,h}_{\bowtie n}(\phi) \equiv \mathfrak{C}^{\lfloor \frac{K}{h} \rfloor \cdot h}_{\bowtie n \cdot \lfloor \frac{K}{h} \rfloor}(\phi)$, demonstrated in previous sections. Therefore, we can reuse the algorithm

```
1  function REDUCER_l^{C,K,⋈n}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅
4      for all (φ, v, τ) ∈ T do
5        f ← (\x → ())
6        updateQueue((φ, v, τ), win, [0, K), f)
7        output(ψ, (win.size ⋈ n, τ))
8      end for
9    end if
10 end function
```

(a) $\mathfrak{C}$ *modality*

```
1  function REDUCER_l^{M,K,⋈n}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅
4      for all (φ, (v, τ)) ∈ T do
5        f ← (\x → ())
6        updateQueue((φ, v, τ), win, [0, K), f)
7        m ← win.map(\x → ⌊(τ − x.τ)/h⌋).maxEqual
8        output(ψ, (m ⋈ n, τ))
9      end for
10   end if
11 end function
```

(b) $\mathfrak{M}$ *modality*

```
1  function REDUCER_l^{D,K,⋈n}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅, dist ← ∅
4      for all (φ1, (v1, τ)), (φ2, (v2, τ)) ∈ T do
5        if (win.size ≠ 0 ∧ v2) then
6          dist.enqueue(τ − win.last.τ)
7        end if
8        f ← (\x → if (x.φ = φ1) dist.dequeue() end if)
9        updateQueue((φ, v, τ), win, [0, K), f)
10       output(ψ, (dist.size ≠ 0 ∧ dist.sum/dist.size ⋈ n, τ))
11     end for
12   end if
13 end function
```

(c) $\mathfrak{D}$ *modality*

Figure 8.7: *Reducer algorithms for aggregating modalities*

from $\mathfrak{C}$ modality to also perform distributed trace checking of $\mathfrak{U}$ modality.

$\mathfrak{M}$ modality. Algorithm 8.7b shows how the tuples are emitted for the $\mathfrak{M}$ modality. Similarly to the $\mathfrak{C}$ modality, we use queue win and updateQueue function to keep track of the all the tuples with positive truth value in the $[0, K)$ time window in the past. To get the maximum number of occurrences of the subformula for all the subintervals of length h we use a map function to convert all the timestamps in the queue win into subinterval indexes from the range $[0, \ldots \lfloor \frac{K}{h} \rfloor]$. For example, timestamps that get converted to index 0 belong to the subinterval closes to the current timestamp τ. The maximum value assigned to variable max is the maximum number of indexes with the same value. The final truth value is computed by comparing variable max with n.

$\mathfrak{D}$ modality. The reduce function for the $\mathfrak{D}$ modality is shown in Algorithm 8.7c. To implement the semantics of the $\mathfrak{D}$ modality we use two queues. Queue win keeps track of the tuples of both subformulae with timestamps in the $[0, K)$ interval from the current timestamp τ. To update the queue win, updateQueue function is called for either of the two subformulae ϕ_1 or ϕ_2 (represented just as ϕ) with their respective truth values

(represented as v). Queue $dist$ is a queue of integers each representing a timestamp difference between adjacent ϕ_1 and ϕ_2 subformula currently in the queue win. As stated before, we assume that the two subformulae occurring the $\mathfrak{D}$ modality alternate in the trace, starting with ϕ_1. Therefore, we enqueue a new element to queue $dist$ every time ϕ_2 is true at the current instant and the queue win is not empty (lines 5 and 6). We remove an element from the head of the $dist$ queue every time a tuple with subformula ϕ_1 is removed from the win queue. This is done with the f closure passed to the updateQueue function. Finally, the average distance is calculated as $\frac{dist.\mathsf{sum}}{dist.\mathsf{size}}$ if queue $dist$ is nonempty.

Example: Let us now use our algorithm to evaluate formula $\Phi = \mathsf{F}_{[3,7]}(p)$ on the following trace (represented as a timed word): $(\{p\}, 1)$, $(\{p\}, 2)$, $(\{q\}, 4)$, $(\{p, q\}, 6)$, $(\{p, q\}, 8)$, $(\{q\}, 9)$, $(\{q\}, 10)$. In the *read phase* the algorithm parses the trace in parallel and creates the input tuples for the *map phase*. From the first element $(\{p\}, 1)$ the *InputReader* creates only $(p, (\top, 1))$ tuple since Φ refers only to the atom p. Tuples $(p, (\top, 1))$, $(p, (\top, 2))$, $(p, (\bot, 4))$, $(p, (\top, 6))$, $(p, (\top, 8))$, $(p, (\bot, 9))$, $(p, (\bot, 10))$ are thus received by *map phase*. The *Mapper* associates the formulae from the input tuples with their superformulae. In the case of the tuple $(p, (\top, 1))$ it generates only tuple $(\mathsf{F}_{[3,7]}(p), (p, \top, 1))$ since $\mathsf{F}_{[3,7]}(p)$ is the only superformula of p. *Reduce phase*, therefore, receives tuples $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 10)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 9)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 8)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 6)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 4)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 2)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 1)))$ shuffled and sorted in a descending order of their timestamps. Since all the tuples have the same key only one reducer is needed. The reducer applies the Algorithm 8.6d and outputs truth values of $\mathsf{F}_{[3,7]}(p)$ for every position in the trace: $(\mathsf{F}_{[3,7]}(p), (\bot, 10))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 9))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 8))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 6))$, $(\mathsf{F}_{[3,7]}(p), (\top, 4))$, $(\mathsf{F}_{[3,7]}(p), (\top, 2))$, $(\mathsf{F}_{[3,7]}(p), (\top, 1))$.

Based on the final list of tuples, we can conclude that the formula holds, since it has a positive truth value at the first time instant.

8.3 Implementation

We have implemented both of our distributed trace checking algorithms in the MTLMAPREDUCE tool, which is publicly available [84]. The tool is implemented in Java and uses the Apache Spark framework [120, 121], which supports iterative MapReduce applications in a better way than Apache Hadoop [8, 118] (see Section 2.5.2). Algorithms presented in Section 8.2 are the basis of an older version of the tool implemented in MapReduce. This section describes how the algorithms are adapted for Spark frame-

work. The pseudocode has explicit type annotation to provide the reader with the precise signatures of the various methods.

The iterative execution of the algorithm is implemented as the part of its main loop (shown in Algorithm 8.8a). As discussed in Section 2.5.2, Spark framework expresses computation as a set of transformations and actions performed on resilient distributed datasets (RDDs). The function in Algorithm 8.8a take two parameters: a path to the trace, saved in a distributed file system and a SOLOIST formula. Line 2 creates an RDD "trace", a collection of strings corresponding to the lines in the trace file[4]. At this point no computation is performed yet, since the read function only creates an RDD and associates it with a path. Line 3 corresponds to the read phase: transformation flatMap is invoked on the "trace" RDD and a Reader object is passed as a parameter. Reader class (shown in Algorithm 8.8b) implements a function that takes a string as an input, parses it and executes the inner loop of the Reader function from Algorithm 8.2a to produce a set of input tuples of the from $(\phi, (v, \tau))$. A tuple is produced for each atom in the SOLOIST formula. The tuple has type $(\mathsf{Formula}, \mathsf{MapValue})$, where MapValue is an alias for $(\mathsf{Boolean}, \mathsf{Long})$. The method call of the Reader object is called by the Spark framework on each member of the "trace" RDD and the resulting tuples are in the "formulae" RDD. The algorithm performs a number of iterations equal to the height of the SOLOIST formula (lines 4–9). The map phase is executed in line 5 by performing a flatMap transformation and passing the Mapper object. Mapper class implements the inner loop of the mapper function from Algorithm 8.3a. It produces tuples of the form $((\psi, \pm\tau), (\phi, v, \tau))$ annotated with type $(\mathsf{CompKey}, \mathsf{ReduceValue})$, where CompKey is an alias for $(\mathsf{Formula}, \mathsf{Long})$ and ReduceValue is an alias for $(\mathsf{Formula}, \mathsf{Boolean}, \mathsf{Long})$. Next, shuffle and sort phase is implemented with two transformations provided my the Spark framework: repartitionAndSortWithinPartitions and groupByKey. RDD "grouped" contains tuples partitioned according to the formula in their composite key. Each partition is sorted according to the timestamp in the tuples. In line 8 the flatMap transformation implements the reduce phase. Instance of the Reducer class (shown in Algorithm 8.8d) calls the accept method of the abstract Formula class (shown in Algorithm 8.8f) and passes the iterator with all the value tuples. Each SOLOIST operator is implemented as a separate class that inherits from the Formula class and overrides the accept method to return its own particular Iterable object. Each formula return an iterable object that implements its semantics. Algorithm 8.8f shows an implementation of the iterable object for the Negation operator.

[4]We assume that trace has one timestamp and a set of events per line

```
1  function SPARKTRACECHECKING(p: String, Φ: Formula): Boolean
2    trace : RDD[String] ← read(p)
3    formulae : RDD[(Formula,MapValue)] ← trace.flatmap(new Reader(Φ))
4    for i ∈ {1, 2, ... h(Φ)} do
5      mapped : RDD[(CompKey, ReduceValue)] ← formulae.flatmap(new  Mapper(Φ))
6      sorted : RDD[(CompKey, ReduceValue)] ← mapped.repartitionAndSortWithinPartitions()
7      grouped : RDD[(CompKey, Iterable[ReduceValue])] ← sorted.groupByKey()
8      formuale : RDD[(Formula, MapValue)] ← sorted.flatmap(new Reducer(i))
9    end for
10   return formulae.take(1).value.v
11 end function
```

(a) *Main loop of the algorithm*

```
1  class READER extends PAIRFLATMAPFUNCTION
2    field FORMULA: Formula
3    method CALL(s: String): Iterable[(Formula, MapValue)]
4      //Implements the inner loop of the reader function from Algorithm 8.2a
5    end method
6  end class
```

(b) *Reader class*

```
1  class MAPPER extends PAIRFLATMAPFUNCTION
2    field FORMULA: Formula
3    method CALL(t: (Formula, MapValue)):  Iterable[(CompKey, ReduceValue)]
4      //Implements the inner loop of the mapper function from Algorithm 8.3a
5    end method
6  end class
```

(c) *Mapper class*

```
1  class REDUCER extends PAIRFLATMAPFUNCTION
2    field FORMULA: Formula
3    field ITERATION: Integer
4    method CALL(ts: (CompKey, Iterable[ReduceValue])):  Iterable[(Formula, MapValue)]
5      ts.compKey.φ.accept(ts.values.iterator())
6    end method
7  end class
```

(d) *Reducer class*

```
1  abstract class FORMULA
2    abstract method VALUES: ACCEPT(Iterator[ReduceValue])):  Iterable[(Formula, MapValue)]
3  end class
```

(e) *Formula class*

```
1  class NEGATION extends FORMULA
2    method ACCEPT(values: Iterator[ReduceValue])):  Iterable[(Formula, MapValue)]
3      return new Iterable[(Formula, MapValue)]()
4            field ITERATOR: Iterator[ReduceValue] = values
5            method NEXT(): (Formula, MapValue)
6              tuple ← iterator.next()
7              output(this, (¬tuple.v, tuple.τ))
8            end method
9        end anonymous class
10   end method
11 end class
```

(f) *Negation formula class*

Figure 8.8: *Implementation of the* MTLMAPREDUCE *tool in Spark*

CHAPTER 9

Lazy semantics for MTL and Optimizations

9.1 Overview

A time interval specified by a metric temporal formula determines a portion of the trace that needs to be considered to decide if the formula is true in a single position of the trace. Depending on the particular MTL formula that is being checked, in the worst case this process needs to be repeated for every position in the trace.[1]

Generally speaking, trace checking algorithms scan a trace and buffer the events that satisfy the temporal constraints of the formula. The buffer is incrementally updated as the trace is scanned and the algorithms incrementally provide verdicts for the positions for which they have enough information (to determine the verdict). The buffer updates consist of adding the newly scanned positions of the trace and removing positions with timestamps not in the time interval relative to the timestamp of the latest scanned position. Online algorithms are constrained to scan the trace in order in which it is being produced. They immediately provide a verdict for the currently scanned position in case of the past temporal operators. For the future temporal operators, however, they have to delay the verdict until they

[1] For example, if a "*Globally*" temporal operator is used.

have seen enough of the trace. Offline algorithms do not need to do this, as they assume that the trace is complete and can scan it backwards when evaluating future operators.

The lower-bound for memory complexity of trace checking algorithms is known to be exponential in the numeric constants occurring in the MTL formula encoded in binary [114]. Therefore the strategy of buffering events creates a memory scalability issue for trace checking algorithms. This issue also affects distributed and parallel solutions, including the one we introduced in Chapter 8. Specifically, the memory scalability of a trace checking algorithm on a single cluster node depends exponentially on the numeric constants defining the bounds of the time intervals in the MTL formula to be checked.

In this chapter we address this memory scalability issue by proposing a trace checking algorithm that exploits new semantics for MTL, called *lazy* semantics [31].

Unlike traditional *point-based* semantics [83], our lazy semantics can evaluate both temporal formulae and boolean combinations of temporal-only formulae *at any arbitrary* time instant, while it evaluates atomic propositions only at time-stamped positions of the trace. We propose lazy semantics because it possesses certain properties that allow us to decompose any MTL formula into an equivalent MTL formula where the upper bound of all time intervals of its temporal operators is limited by some constant. This decomposition plays a major role in the context of (distributed) trace checking of formulae with large time intervals. In practice, if we want to check a formula with a large time interval, applying the decomposition entails an equivalent formula, with smaller time intervals. This new formula can be checked in a more memory efficient way by using our new trace checking algorithm, which applies lazy semantics.

Motivating example

Let us present an example to motivate the need for lazy semantics.

Consider again the formula $\Phi = \mathsf{F}_{[3,7]}(p)$ and its evaluation on the following trace: $(\{p\}, 1)$, $(\{p\}, 2)$, $(\{q\}, 4)$, $(\{p, q\}, 6)$, $(\{p, q\}, 8)$, $(\{q\}, 9)$, $(\{q\}, 10)$. The timed word, shown in Figure 9.1, is defined over the set of atoms $\Pi = \{p, q\}$; its length is 7 and it spans over 10 time units. The first two rows in the picture represent its atoms and time-stamps; the last two rows show, respectively, the evaluation of subformula p and formula $\mathsf{F}_{[3,7]}(p)$ using point-based semantics. As shown in the last row of Figure 9.1, according to point-based semantics, formula $\mathsf{F}_{[3,7]}(p)$ holds at time

Atoms:	$\{p\}$	$\{p\}$		$\{q\}$		$\{p,q\}$		$\{p,q\}$	$\{q\}$	$\{q\}$
Time-stamps:	1	2		4		6		8	9	10
Time instants:	1	2	3	4	5	6	7	8	9	10
p	$\top$	$\top$		$\bot$		$\top$		$\top$	$\bot$	$\bot$
$\mathsf{F}_{[3,7]}(p)$	$\top$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$

Figure 9.1: *Evaluation of formula* $\Phi = \mathsf{F}_{[3,7]}(p)$.

instants 1, 2 and 4.

For a formula of the form $\mathsf{F}_{[a,b]}(p)$, the algorithm needs to buffer, in the worst case (i.e., in case there exists an element at every time instant), at most $b + 1$ elements. For example, to evaluate formula $\mathsf{F}_{[3,7]}(p)$ at time instant 2, in the worst case the algorithm will buffer 8 elements, i.e., all the elements whose time-stamp ranges from 2 to 9. The elements with time-stamps ranging from 6 to 9 satisfy the time interval constraint of the formula; the others are kept for the evaluation of the formula at subsequent positions. Let us assume that the execution infrastructure could only store 5 elements in the buffer, because of the limited available memory. The worst-case requirement of keeping 8 elements in the buffer would then be too demanding for the infrastructure, in terms of memory scalability. To lower the memory requirement for the buffer we would need a formula with a smaller time interval and expressing the same property as Φ. In other words, one might ask whether there is an MTL formula equivalent to Φ with all the intervals bounded by the constant 4 (and thus requiring to store at most 4+1=5 elements in the buffer).

Let us consider formula $\Phi' = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$: a naïve and intuitive interpretation might lead us to think that it defines the same property as Φ. Roughly speaking, instead of checking if p eventually occurs within the entire $[3, 7]$ time interval, Φ' checks if p either occurs in the $[3, 4]$ interval (as specified by subformula $\mathsf{F}_{[3,4]}(p)$) or in the interval $[0, 3]$ when evaluated exactly 4 time instants in the future (as specified by subformula $\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$). Figure 9.2 shows the evaluation of formula Φ' over the same trace used in Figure 9.1. As you can see, formula Φ' does not have the same evaluation as Φ on the same trace. More specifically, at time instant 1 Φ' is false while Φ is true (see the values circled in both figures). By analyzing the evaluation of Φ', one can notice that subformula $\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$ at time instant 1 refers to the value of $\mathsf{F}_{[0,3]}(p)$ at time instant 5, which does not have a corresponding element in the trace. If there was an element at time instant 5, $\mathsf{F}_{[0,3]}(p)$ would be true since p holds at time instant 6.

The above example shows that the evaluation of temporal subformulae according to point-based semantics depends on the existence of certain elements in the trace. It also shows that point-based semantics is not suitable to support the intuitive decomposition of MTL formulae into equivalent ones with smaller time intervals, like the one from Φ to Φ' shown above. We maintain that this constitutes a limitation for the application of point-based semantics in the context of trace checking. Therefore, in this paper we propose a new, alternative semantics for MTL, called lazy semantics.

The main feature of lazy semantics is that it evaluates temporal formulae and boolean combinations of temporal-only formulae *at any arbitrary* time instant, regardless of the existence of the corresponding elements in the trace. The existence of the elements is only required when evaluating atoms. This feature allows us to decompose any MTL formula into an equivalent MTL formula in which the upper bound of all time intervals of its temporal operators is limited by some constant. Such a decomposition can be used as a preprocessing step of a trace checking algorithm, which can then run in a more memory-efficient way.

In the following sections we first introduce lazy semantics (Section 9.2) and formalize the notion of the decomposition exemplified above (Section 9.3). Afterwards, in Section 9.4 we describe the modifications to our trace checking algorithm from Chapter 8, required to preprocess the formula and support lazy semantics.

9.2 Lazy Semantics for MTL

The following example shows an anomalous case of $\mathsf{MTL}_{\mathsf{P}}$ semantics that lazy semantics for MTL (denoted as $\mathsf{MTL}_{\mathsf{L}}$ semantics) intends to remedy. Consider a timed word $w = (\sigma, \tau) = (\{q\}, 1)(\{p\}, 7)$ and two MTL for-

Atoms:	$\{p\}$	$\{p\}$		$\{q\}$		$\{p,q\}$		$\{p,q\}$	$\{q\}$	$\{q\}$
Time-stamps:	1	2		4		6		8	9	10
Time instants:	1	2	3	4	5	6	7	8	9	10
p	$\top$	$\top$		$\bot$		$\top$		$\top$	$\bot$	$\bot$
$\mathsf{F}_{[3,4]}(p)$	$\bot$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$
$\mathsf{F}_{[0,3]}(p)$	$\top$	$\top$		$\top$		$\top$		$\top$	$\bot$	$\bot$
$\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$	$\bot$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$
Φ'	$\bot$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$

Figure 9.2: *Evaluation of formula* $\Phi' = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$.

$$
\begin{aligned}
(\sigma,\tau,t) &\models_L p \text{ iff } \exists i.(0 \le i < |\sigma| \text{ and } t = \tau_i \text{ and } p \in \sigma_i)\\
(\sigma,\tau,t) &\models_L \neg\phi \text{ iff } (\sigma,\tau,t) \not\models_L \phi\\
(\sigma,\tau,t) &\models_L \phi \vee \psi \text{ iff } (\sigma,\tau,t) \models_L \phi \text{ or } (\sigma,\tau,t) \models_L \psi\\
(\sigma,\tau,t) &\models_L \phi \mathsf{U}_I \psi \text{ iff } \exists t'.(t' \ge t \text{ and } t' - t \in I \text{ and}\\
(\sigma,\tau,t') &\models_L \psi \text{ and } \forall t''.(t < t'' < t' \text{ and } \exists i.(0 \le i < |\sigma| \text{ and}\\
&\qquad t'' = \tau_i) \text{ then } (\sigma,\tau,t'') \models_L \phi))
\end{aligned}
$$

Figure 9.3: $\mathsf{MTL_L}$ *semantics on timed words.*

mulae $\psi_1 = \mathsf{F}_{=6}p$ and $\psi_2 = \mathsf{F}_{=3}\mathsf{F}_{=3}p$. The intuitive meaning of the two formulae is the same: p holds 6 time units after the origin, i.e., at time-stamp 7. However, when evaluated on w using the $\mathsf{MTL_P}$ semantics, the two formulae have different values: ψ_1 correctly evaluates to true, but ψ_2 to false. Indeed, in ψ_2 the outermost $\mathsf{F}_{=3}$ subformula is trivially false, because there is no position that is exactly 3 time instants in the future with respect to the origin. The two formulae, instead, are equivalent over the $\mathsf{MTL_L}$ semantics, where they both evaluate to true. Indeed, this is true also over signal-based semantics [39]; however, signals are not very practical for monitoring and trace checking, which usually operate on logs that are best modeled as a sequence of individual time-stamped observations, i.e., timed words.

$\mathsf{MTL_L}$ semantics. $\mathsf{MTL_L}$ semantics on timed words is given in Figure 9.3, in terms of the satisfaction relation $\models_L$, with respect to a timed word (σ,τ) and a time instant $t \in \mathbb{R}^+$; p is an atom and ϕ and ψ are MTL formulae. An MTL formula ϕ, when interpreted over $\mathsf{MTL_L}$ semantics, defines a timed language $L_L(\phi) = \{(\sigma,\tau)|(\sigma,\tau,0) \models_L \phi\}$. The main difference between $\mathsf{MTL_P}$ and $\mathsf{MTL_L}$ semantics is that $\mathsf{MTL_P}$ evaluates formulae only at positions i of a timed word, while $\mathsf{MTL_L}$ inherits a feature of signal-based semantics, namely it may evaluate (non-atomic) formulae at any possible time instant t, even if there is no time-stamp equal to t. For example, according to the $\mathsf{MTL_P}$ semantics, an "*Until*" formula $\phi \equiv \psi_1 \mathsf{U}_I \psi_2$ evaluates to false in case there are no positions in the interval I, due to the existential quantification on j (see Figure 2.2). Conversely, over the $\mathsf{MTL_L}$ semantics, the evaluation of ϕ depends on the evaluation of ψ_2. If the latter is an atom then formula ϕ also evaluates to false, because of the existential quantifier in the $\mathsf{MTL_L}$ semantics of atoms. However, if ψ_2 is a temporal formula or a boolean combination of temporal-only formulae (e.g., other "*Until*" formulae), it will be evaluated in the part of the timed word that satisfies the interval of ϕ. Hereafter we refer to the MTL formulae interpreted over the $\mathsf{MTL_L}$ semantics as "$\mathsf{MTL_L}$ formulae"; similarly, "$\mathsf{MTL_P}$

formulae" are MTL formulae interpreted over the $\mathsf{MTL_P}$ semantics.

Let $\mathbb{M}(\Pi)$ be the set of all formulae that can be derived from the MTL grammar shown in Section 2.1.1, using Π as the set of atoms. We show that any language $L_P(\phi)$ defined using some $\mathsf{MTL_P}$ formula ϕ can be defined using an $\mathsf{MTL_L}$ formula obtained after applying the translation $l2p : \mathbb{M}(\Pi) \rightarrow \mathbb{M}(\Pi)$ to ϕ, i.e., $L_p(\phi) = L_L(l2p(\phi))$ for any ϕ. The $l2p$ translation is defined as follows:

$$l2p(p) \equiv p, p \in \Pi; \qquad l2p(\phi \vee \psi) \equiv l2p(\phi) \vee l2p(\psi)$$
$$l2p(\neg\phi) \equiv \neg l2p(\phi); \qquad l2p(\phi \mathsf{U}_I \psi) \equiv l2p(\phi) \mathsf{U}_I (\varphi_{act} \wedge l2p(\psi))$$

where $\varphi_{act} \equiv a \vee \neg a$ for some $a \in \Pi$.

The goal of $l2p$ is to prevent the occurrence of *direct nesting* of temporal operators, i.e., to avoid the presence of (sub)formulae like $\mathsf{F}_{=3}\mathsf{F}_{=3}p$. As discussed in the example above, nested temporal operators are interpreted differently over the two semantics. Direct nesting is avoided by rewriting the right argument of every "*Until*" (i.e., the "existential" component of "*Until*"). The argument is conjuncted with a formula φ_{act} that evaluates to true (over both semantics) if there exists a position in the underlying timed word; otherwise φ_{act} evaluates to false. To explain this intuition, let us evaluate φ_{act} over a timed word (σ, τ) over the alphabet $\Pi = \{a\}$. Over point-based semantics, $(\sigma, \tau, i) \models_P \varphi_{act} \equiv (\sigma, \tau, i) \models_P a \vee \neg a$ is true for any position i, since either a belongs to σ_i or not. However, the same does not hold for lazy semantics. According to lazy semantics, $(\sigma, \tau, t) \models_L \varphi_{act}$ is true only in those time instants t for which there exists i such that $\tau_i = t$ and therefore exists the corresponding σ_i (to which a can belong or not).

Lemma 1. *Given an MTL formula ϕ and a timed word $\omega = (\sigma, \tau)$, for any $i \geq 0$, the following equivalence (modulo $l2p$ translation) holds: $(\sigma, \tau, i) \models_P \phi$ iff $(\sigma, \tau, \tau_i) \models_L l2p(\phi)$.*

Proof. The lemma is proved by structural induction on formula ϕ. Let γ be an MTL formula. The inductive hypothesis is $(\sigma, \tau, i) \models_P \gamma$ iff $(\sigma, \tau, \tau_i) \models_L l2p(\gamma)$.

1. base case γ is $p \in \Pi$.
   $(\sigma, \tau, i) \models_P p$ iff $p \in \sigma_i$ iff $\exists i.(0 \leq i < |\sigma| \wedge \tau_i = \tau_i \wedge p \in \sigma_i)$ iff $(\sigma, \tau, \tau_i) \models_L p$. Then, we obtain $(\sigma, \tau, \tau_i) \models_L l2p(p)$, by definition of $l2p$.
2. γ is $\neg\phi$.
   $(\sigma, \tau, i) \models_P \neg\phi$ iff $(\sigma, \tau, i) \not\models_P \phi$ iff, by inductive hypothesis, $(\sigma, \tau, \tau_i) \not\models_L l2p(\phi)$ iff, by definition of $l2p$, $(\sigma, \tau, \tau_i) \not\models_L \neg l2p(\neg\phi)$ iff $(\sigma, \tau, \tau_i) \models_L l2p(\neg\phi)$.

3. γ is $\phi \wedge \psi$.
   $(\sigma, \tau, i) \models_P \phi \wedge \psi$ iff $(\sigma, \tau, i) \models_P \phi$ and $(\sigma, \tau, i) \models_P \psi$ iff, by inductive hypothesis, $(\sigma, \tau, \tau_i) \models_L l2p(\phi)$ and $(\sigma, \tau, \tau_i) \models_L l2p(\psi)$ iff, $(\sigma, \tau, \tau_i) \models_L l2p(\phi) \wedge l2p(\psi)$ iff, by definition of $l2p$, $(\sigma, \tau, \tau_i) \models_L l2p(\phi \wedge \phi)$.
4. γ is $\phi \mathsf{U}_I \psi$.
   $(\sigma, \tau, i) \models_P \phi \mathsf{U}_I \psi$ iff
   $\exists j.(i \leq j < |\sigma| \wedge \tau_j - \tau_i \in I \wedge (\sigma, \tau, j) \models_P \psi \wedge \forall k.(i < k < j \rightarrow (\sigma, \tau, k) \models_P \phi)$ iff, by inductive hypothesis,
   $\exists j.(i \leq j < |\sigma| \wedge \tau_j - \tau_i \in I \wedge (\sigma, \tau, \tau_j) \models_L l2h(\psi) \wedge \forall k.(i < k < j \rightarrow (\sigma, \tau, \tau_k) \models_L l2p(\phi))$ iff
   $\exists t_j.(t_i \leq t_j < t_{|\sigma|} \wedge t_j - t_i \in I \wedge (\sigma, \tau, t_j) \models_L (\varphi_{act} \wedge l2h(\psi)) \wedge \forall t_k.(t_i < t_k < t_j \wedge \exists i.(0 \leq i < |\sigma| \wedge t_k = \tau_i) \rightarrow (\sigma, \tau, t_k) \models_L l2p(\phi))$ iff
   $(\sigma, \tau, t_i) \models_L l2p(\phi) \mathsf{U}_I (\varphi_{act} \wedge l2p(\psi))$ iff, by definition of $l2p$
   $(\sigma, \tau, t_i) \models_L l2p(\psi \mathsf{U}_I \psi)$.

The lemma is proved by considering $\gamma = \phi$. □

Theorem 1. *Any timed language defined by an* $\mathsf{MTL_P}$ *formula can be defined by an* $\mathsf{MTL_L}$ *formula over the same alphabet.*

Proof. By Lemma 1, for $i = 0$. □

Notice that the translation $l2p$ defines a syntactic MTL fragment where temporal or boolean combination of temporal-only operators cannot be directly nested. In this fragment $\mathsf{MTL_P}$ and $\mathsf{MTL_L}$ formulae define the same languages. However, if we consider the complete definition of MTL, without syntactic restrictions, the class of timed languages defined by $\mathsf{MTL_L}$ formulae strictly includes the class of languages defined by $\mathsf{MTL_P}$ formulae. In other words, MTL interpreted over lazy semantics is *strictly* more expressive than MTL interpreted over point-based semantics; this result is established by the following theorem.

Theorem 2. *There exists a timed language defined by some* $\mathsf{MTL_L}$ *formula that cannot be defined by any* $\mathsf{MTL_P}$ *formula.*

Proof. Consider the language of timed words $L = \{(\sigma, \tau) \mid \exists i \exists j (i \leq j \wedge (\sigma, \tau, i) \models_L b \wedge (\sigma, \tau, j) \models_L c \wedge \tau_j \leq 2)\}$. L is defined by the $\mathsf{MTL_L}$ formula $\Phi = \Phi_1 \vee \Phi_2 \vee \Phi_3$, where $\Phi_1 = (\mathsf{F}_{(0,1)} b) \wedge (\mathsf{F}_{[1,2]} c) \vee (\mathsf{F}_{(0,1]} b) \wedge (\mathsf{F}_{(1,2]} c)$; $\Phi_2 = \mathsf{F}_{(0,1]}(b \wedge \mathsf{F}_{(0,1]} c)$ and $\Phi_3 = \mathsf{F}_{(0,1]}((\mathsf{F}_{(0,1)} b) \wedge (\mathsf{F}_{[1,1]} c))$. L cannot be defined by any $\mathsf{MTL_P}$ formula (see reference [39], proposition 6). □

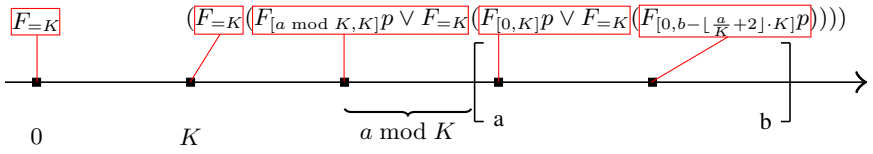


Figure 9.4: *Example of the $\mathcal{L}_K$ decomposition of the formula $\mathsf{F}_{[a,b]}p$*

9.3 Parametric Decomposition

In this section we show that lazy semantics allows for a parametric decomposition of MTL formulae into MTL formulae where the upper bound of all intervals of the temporal operators is limited by some constant K (the parameter of the decomposition). This structural characteristic will then be used in the trace checking algorithm presented thereafter.

We first introduce some notation and show some properties of lazy semantics that will be used to prove the correctness of the decomposition. We define the operator $\oplus$ over intervals in $\mathbb{R}$ with endpoints in $\mathbb{N}$ such that $I \oplus J = \{i + j \mid \forall i \in I \text{ and } j \in J\}$.

Lemma 2. *For any timed word (σ, τ) and $t \geq 0$,*

$$(\sigma, \tau, t) \models_L \mathsf{F}_I \mathsf{F}_J \phi \textit{ iff } (\sigma, \tau, t) \models_L \mathsf{F}_{I \oplus J} \phi. \tag{9.1}$$

Proof. We show two cases. In the first one, both F formulae have left and right-closed intervals. The second one considers all the other combinations.

$\mathsf{F}_{[a,b]}(\mathsf{F}_{[c,d]}\phi)$**:**
$(\sigma, \tau, t) \models_L \mathsf{F}_{[a,b]}(\mathsf{F}_{[c,d]}\phi)$ iff
$\exists t'.(t' \geq t \ \wedge t' - t \in [a, b] \wedge (\sigma, \tau, t') \models_L \mathsf{F}_{[c,d]}\phi$ iff
$\exists t'.(t' \geq t \ \wedge t' \in [t + a, t + b] \wedge (\sigma, \tau, t') \models_L \exists t''.(t'' \geq t' \ \wedge t'' \in [t' + c, t' + d] \wedge (\sigma, \tau, t'') \models_L \phi)$ iff
$\exists t'.(t' \geq t \ \wedge (\sigma, \tau, t') \models_L \exists t''.(t'' \geq t' \ \wedge t'' \in [t + a + c, t + b + d] \wedge (\sigma, \tau, t'') \models_L \phi))$ iff
$\exists t''.(t'' \geq t \ \wedge t'' \in [t + a + c, t + b + d] \wedge (\sigma, \tau, t'') \models_L \phi)$ iff
$(\sigma, \tau, t) \models_L \mathsf{F}_{I \oplus J}\phi$.

$\mathsf{F}_{\langle a,b\rangle}(\mathsf{F}_{\langle c,d\rangle}\phi)$**:**
$(\sigma, \tau, t) \models_L \mathsf{F}_{\langle a,b\rangle}(\mathsf{F}_{\langle c,d\rangle}\phi)$ iff
$\exists t'.(t' \geq t \ \wedge t' - t \in \langle a, b\rangle \wedge (\sigma, \tau, t') \models_L \mathsf{F}_{\langle c,d\rangle}\phi$ iff
$\exists t'.(t' \geq t \ \wedge t' \in \langle t + a, t + b\rangle \wedge (\sigma, \tau, t') \models_L \exists t''.(t'' \geq t' \ \wedge t'' \in \langle t' + c, t' + d\rangle \wedge (\sigma, \tau, t'') \models_L \phi)$ iff

$\exists t'.(t' \geq t \ \wedge (\sigma, \tau, t') \models_L \exists t''.(t'' \geq t' \ \wedge t'' \in (t+a+c, t+b+d) \wedge (\sigma, \tau, t'') \models_L \phi))$ iff
$\exists t''.(t'' \geq t \ \wedge t'' \in (t+a+c, t+b+d) \wedge (\sigma, \tau, t'') \models_L \phi)$ iff
$(\sigma, \tau, t) \models_L \mathsf{F}_{I \oplus J} \phi$.

□

Corollary 1. *For any timed word* (σ, τ) *and* $t, N \geq 0$*,*

$$(\sigma, \tau, t) \models_L \mathsf{F}^N_{=K} \phi \textit{ iff } (\sigma, \tau, t) \models_L \mathsf{F}_{=K \cdot N}. \tag{9.2}$$

Lemma 3. *For any timed word* (σ, τ) *and* $t \geq 0$*,*

$$(\sigma, \tau, t) \models_L \mathsf{F}_I \phi \vee \mathsf{F}_J \phi \textit{ iff } (\sigma, \tau, t) \models_L \mathsf{F}_{I \cup J} \phi, \textit{ if } I \cap J \neq \emptyset.$$

Proof. We prove the lemma for the case of $I = (a, b)$, $J = (c, d)$, as we can always rewrite intervals as left-right open ones. The case $\mathsf{F}_{[0,b)} \phi$ becomes $\mathsf{F}_{(1,b)} \phi \vee \phi$. The case for unbounded intervals is similar. By $I \cap J \neq \emptyset$ we have both $c + 1 < b$ and $a + 1 < d$ which entails $c < b$ and $a < d$. Therefore, we have that $min\{a, b, c, d\} = min\{a, c\}$ and $max\{a, b, c, d\} = max\{b, d\}$.
$(\sigma, \tau, t) \models_L \mathsf{F}_I \phi \vee \mathsf{F}_J \psi$ iff
$(\sigma, \tau, t) \models_L \mathsf{F}_I \phi$ or $(\sigma, \tau, t) \models_L \mathsf{F}_J \phi$ iff
$\exists t'.(t' \geq t \ \wedge t' - t \in (a, b) \wedge (\sigma, \tau, t') \models_L \mathsf{F}_{(a,b)} \phi$ or $\exists t'.(t' \geq t \ \wedge t' - t \in (c, d) \wedge (\sigma, \tau, t') \models_L \mathsf{F}_{(c,d)} \phi$ iff,
$\exists t'.(t' \geq t \wedge (t' - t \in (a, b) \wedge (\sigma, \tau, t') \models_L \phi \vee t' - t \in (c, d) \wedge (\sigma, \tau, t') \models_L \phi)$ iff, as $a < b$ and $c < d$,
$\exists t'.(t' \geq t \ \wedge t' - t \in (min\{a, c\}, min\{b, d\}) \wedge (\sigma, \tau, t') \models_L \phi$ iff
$(\sigma, \tau, t) \models_L \mathsf{F}_{(min\{a,c\}, max\{b,d\})} \phi$ iff
$(\sigma, \tau, t) \models_L \mathsf{F}_{I \cup J} \phi$. □

Hereafter, we focus on bounded MTL formulae, i.e., formulae where intervals are always finite. Notice that it is this class of formulae that causes memory scalability issues in trace checking algorithms. We present the parametric decomposition by referring to the bounded "*Eventually*" operator. The bounded "*Until*" and "*Globally*" operators can be expressed in terms of the bounded "*Eventually*" operator using the usual equivalences; moreover, we remark that the decomposition does not affect atoms and is applied recursively to boolean operators. We use angle brackets (symbols "$\langle$" and "$\rangle$") in the definition of the decomposition to cover all four possible cases of open (denoted with round brackets) and closed (denoted with square brackets) intervals; the definition is valid for any instantiation of the symbols as long as they are consistently replaced on the right-hand side.

The *decomposition* $\mathcal{L}_K$ of MTL formulae with respect to *parameter* K is the translation $\mathcal{L}_K : \mathbb{M}(\Pi) \to \mathbb{M}(\Pi)$ such that $\mathcal{L}_K(\mathsf{F}_{\langle a,b\rangle}\phi) =$

$$\begin{cases} \mathsf{F}_{\langle a,b\rangle}\mathcal{L}_K(\phi) & , b \leq K \\ \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \bmod K, b-\lfloor \frac{a}{K} \rfloor \cdot K\rangle}\mathcal{L}_K(\phi)) & , K < b \leq \lfloor \frac{a}{K} + 1 \rfloor \cdot K \\ \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \bmod K, K]}\mathcal{L}_K(\phi) \vee & , b > \lfloor \frac{a}{K} + 1 \rfloor \cdot K \\ \quad \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{=K}(\mathcal{D}_F(\mathcal{L}_K(\phi), K, b - \lfloor \frac{a}{K} + 1 \rfloor \cdot K))) & \end{cases} \tag{9.3}$$

where

$$\mathcal{D}_F(\psi, K, h) = \begin{cases} \mathsf{F}_{[0,h\rangle}\psi & , h \leq K \\ \mathsf{F}_{[0,K]}\psi \vee \mathsf{F}_{=K}(\mathcal{D}_F(\psi, K, h-K)) & , h > K \end{cases} \tag{9.4}$$

The decomposition $\mathcal{L}_K$ considers three cases depending on the values of a, b, and K. In the first case we have $b \leq K$, which means that the upper bound of the temporal interval $[a, b]$ in the input formula is smaller than K, therefore no decomposition is needed. The other two cases consider input formulae where $b > K$. The second case is characterized by $b \leq \lfloor \frac{a}{K} + 1 \rfloor \cdot K \equiv b \leq \lfloor \frac{a}{K} \rfloor \cdot K + K$. The decomposition yields a formula of the form $\mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\alpha)$, where $\alpha = \mathsf{F}_{[a \bmod K, b-\lfloor \frac{a}{K} \rfloor \cdot K]}\mathcal{L}_K(\phi)$ is equivalent to the input formula $\mathsf{F}_{[a,b]}(\phi)$ evaluated at time instant $\lfloor \frac{a}{K} \rfloor \cdot K$. Notice that according to Corollary 1, the argument α in $\mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\alpha)$ is evaluated at time instant $\lfloor \frac{a}{K} \rfloor \cdot K$. The third case is characterized by $b > \lfloor \frac{a}{K} + 1 \rfloor \cdot K$.

We illustrate the decomposition of $\mathsf{F}_{[a,b]}p$ with $p \in \Pi$ by referring to the example in Figure 9.4, where the black squares divide the timeline into segments of length K. We refer to each position in the timeline pinpointed by a black square as a K-position. The big brackets enclose the interval $[a, b]$ relative to time instant 0. Moreover, we assume some values for a and K such that $\lfloor \frac{a}{K} \rfloor = 2$; hence, in the figure the position of a in the timeline is between the marks corresponding to $2K$ and $3K$. The application of $\mathcal{L}_K(\mathsf{F}_{[a,b]}p)$ returns the formula $F_{=K}(F_{=K}(F_{[a \bmod K,K]}p \vee F_{=K}(F_{[0,K]}p \vee \ F_{=K}(F_{[0,b-\lfloor \frac{a}{K}+2 \rfloor \cdot K]}p))))$, which is shown above the timeline, spanning through its length such that each subformula (highlighted in red) is written above the corresponding K-position where it is evaluated. Since $\lfloor \frac{a}{K} \rfloor = 2$ there are two subformulae of the form $\mathsf{F}_{=K}$ evaluated in the first two K-positions. Unlike the previous case, the interval $[a, b]$ is too big to allow for rewriting the input formula into another formula with a single F operator with bounded length. Hence, we use three subformulae:

1) $\mathsf{F}_{[a \bmod K,K]}p$ evaluated at the third K-position, 2) $\mathsf{F}_{[0,K]}p$ evaluated at the fourth K-position, and 3) $\mathsf{F}_{[0,b-\lfloor \frac{a}{K}+2 \rfloor \cdot K]}p$ evaluated at the fifth K-position; the last two subformulae are obtained from the definition of $\mathcal{D}_F$. Notice that if K is set to be equal to one, the $\mathcal{L}_K$ decomposition boils down to the reduction of MTL to LTL.

Theorem 3. *Given an* $\mathsf{MTL}_{\mathsf{L}}$ *formula* ϕ*, a timed word* (σ, τ) *and a positive constant* K*, we have that:*

$$(\sigma, \tau, 0) \models_L \phi \textit{ iff } (\sigma, \tau, 0) \models_L \mathcal{L}_K(\phi) \tag{9.5}$$

and the upper bound of every bounded interval in all temporal subformulae of $\mathcal{L}_K(\phi)$ *is less than or equal to* K*.*

Proof. We can prove this statement by showing that $\mathcal{L}_K(\phi)$ can always be rewritten back to ϕ using lemmata 2 and 3. Let us preform structural induction on the $\mathsf{MTL}_{\mathsf{L}}$ formula ϕ. The inductive hypothesis is $(\sigma, \tau, i) \models_L \theta$ *iff* $(\sigma, \tau, i) \models_L \mathcal{L}_K(\theta)$. Then, the theorem is proved by choosing $\theta = \phi$ and $i = 0$. In the proof we extensively use the following properties $\lfloor \frac{a}{K} + 1 \rfloor \cdot K = \lfloor \frac{a}{K} \rfloor \cdot K + K$ denoted with (*); $b - \lfloor \frac{b}{K} \rfloor \cdot K = b \mod K$ denoted with (**); and $\lfloor n + \epsilon \rfloor = n$, for $n \in \mathbb{N}$ and $\epsilon \in [0, 1)$ denoted with (***).

1. Base cases are the atoms which are not affected by the translation.
2. Same holds for boolean connectives.
3. Let $\theta = \mathsf{F}_{\langle a,b \rangle}(\phi)$. We need to consider three cases.
   (a) $[b \leq K]$: $(\sigma, \tau, i) \models_L \mathsf{F}_{\langle a,b \rangle}(\phi)$ iff $(\sigma, \tau, i) \models_L \exists j.(j - i \in \langle a, b \rangle$ and $(\sigma, \tau, j) \models_L \phi))$ which is, by inductive hypothesis, $(\sigma, \tau, i) \models_L \exists j.(j - i \in \langle a, b \rangle$ and $(\sigma, \tau, j) \models_L \mathcal{L}_K(\phi))$ iff $(\sigma, \tau, i) \models_L \mathsf{F}_{\langle a,b \rangle}(\mathcal{L}_K(\phi))$ which is, by definition of $\mathcal{L}_K$, $(\sigma, \tau, i) \models_L \mathcal{L}_K(\mathsf{F}_{\langle a,b \rangle}(\phi))$. Since $b \leq K$ the right-hand side bound is less then or equal to K.
   (b) $[K < b \leq \lfloor \frac{a}{K} + 1 \rfloor \cdot K]$: Identically to (a), we have $(\sigma, \tau, i) \models_L \mathsf{F}_{\langle a,b \rangle}(\phi)$ iff $(\sigma, \tau, i) \models_L \mathsf{F}_{\langle a,b \rangle}(\mathcal{L}_K(\phi))$ by inductive hypothesis. The interval is not bounded by K as $K < b$. By property (**), we get $(\sigma, \tau, i) \models_L \mathsf{F}_{\langle a \bmod K + K \cdot \lfloor \frac{a}{K} \rfloor, b - K \cdot \lfloor \frac{a}{K} \rfloor + K \cdot \lfloor \frac{a}{K} \rfloor \rangle} \mathcal{L}_K(\phi)$ and, by Lemma 2, we obtain $(\sigma, \tau, i) \models_L \mathsf{F}_{=K \cdot \lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \bmod K, b - \lfloor \frac{a}{K} \rfloor \cdot K \rangle} \mathcal{L}_K(\phi))$. By Corollary 1 the formula can be rewritten into $(\sigma, \tau, i) \models_L \mathsf{F}^{\lfloor \frac{a}{K} \rfloor}_{=K}(\mathsf{F}_{\langle a \bmod K, b - \lfloor \frac{a}{K} \rfloor \cdot K \rangle} \mathcal{L}_K(\phi))$ and, then, by definition of $\mathcal{L}_K$ we obtain $(\sigma, \tau, i) \models_L \mathcal{L}_K(\phi)$. By property (*) and the case assumption is $b \leq \lfloor \frac{a}{K} + 1 \rfloor \cdot K$ we have that $b - \lfloor \frac{a}{K} \rfloor \cdot K \leq K$ therefore the right-hand side bound of the interval is less than or equal to K.

(c) $[b > \lfloor \frac{a}{K} + 1 \rfloor \cdot K]$: Identically to (b) we have
$(\sigma, \tau, i) \models_L \mathsf{F}_{\langle a,b \rangle}(\phi)$ iff $(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, b - \lfloor \frac{a}{K} \rfloor \cdot K \rangle} \mathcal{L}_K(\phi))$.
Since $b > \lfloor \frac{a}{K} + 1 \rfloor \cdot K$ then $b - \lfloor \frac{a}{K} \rfloor \cdot K > K$. Let $n = \lfloor \frac{b}{K} \rfloor - \lfloor \frac{a}{K} \rfloor$, we can use Lemma 3 to write
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} (\phi) \vee \mathsf{F}_{[K,2K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{[2K,3K]} \mathcal{L}_K(\phi) \vee \ldots \vee \mathsf{F}_{[(n-1)K,nK]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{[nK, b - \lfloor \frac{a}{K} \rfloor \cdot K \rangle} \mathcal{L}_K(\phi))$
then, by Lemma 2 and property (*), we get
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{[K,2K]} \mathcal{L}_K(\phi) \vee \ldots \vee \mathsf{F}_{[(n-2)K,(n-1)K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{[(n-1)K, b - \lfloor \frac{a}{K} + 1 \rfloor \cdot K \rangle} \mathcal{L}_K(\phi)))$.
The Lemma 2 is applied n times until we get
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \ldots \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathsf{F}_{[0, b - \lfloor \frac{a}{K} + n \rfloor \cdot K \rangle} \mathcal{L}_K(\phi))) \ldots))))$.
According to properties (**) and (***) the value $b - \lfloor \frac{a}{K} + n \rfloor \cdot K = b \mod K$, which is strictly less than K.
By definition of $\mathcal{D}_F$ (base case) we write
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \ldots \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathcal{D}_F(\mathcal{L}_K(\phi), K, b - \lfloor \frac{a}{K} + n \rfloor \cdot K))) \ldots))))$.
By definition of $\mathcal{D}_F$ (recursive case) we write
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \ldots \vee \mathsf{F}_{=K}(\mathcal{D}_F(\mathcal{L}_K(\phi), K, b - \lfloor \frac{a}{K} + n \rfloor \cdot K + K)) \ldots))))$.
By property (*) we write
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathsf{F}_{[0,K]} \mathcal{L}_K(\phi) \vee \ldots \vee \mathsf{F}_{=K}(\mathcal{D}_F(\mathcal{L}_K(\phi), K, b - \lfloor \frac{a}{K} + n - 1 \rfloor \cdot K)) \ldots))))$.
We apply definition of $\mathcal{D}_F$ (recursive case) and property (*) $n-1$ times to get
$(\sigma, \tau, i) \models_L \mathsf{F}_{=K}^{\lfloor \frac{a}{K} \rfloor}(\mathsf{F}_{\langle a \mod K, K]} \vee \mathsf{F}_{=K}(\mathcal{D}_F(\mathcal{L}_K(\phi), K, b - \lfloor \frac{a}{K} + 1 \rfloor \cdot K)))$.
Finally, we apply the definition of $\mathcal{L}_K$ to obtain
$(\sigma, \tau, i) \models_L \mathcal{L}_K(\mathsf{F}_{\langle a,b \rangle}(\phi))$.

□

9.4 Trace checking Lazy semantics

In this section we only detail the modifications (emphasized with grey boxes in Fig. 9.5) applied to the original algorithm presented in Chapter 8.2 to support MTL_L semantics. Although the original algorithm was designed to perform trace checking of properties written in SOLOIST, here we consider only its MTL subset as the optimization applies only to temporal op-

erators [31].

The algorithm takes as input a non-empty execution trace T and an MTL formula Φ and provides a verdict, indicating whether the trace satisfies the formula or not. Before the algorithm is used we assume that the execution infrastructure, i.e., the cluster of machines, is configured and running. We also assume that one can easily estimate through experimentation K_{cluster}, which is the largest time interval bound that can be used in a formula without triggering memory saturation in the cluster. This bound depends on the memory configuration of the node in the cluster with the least amount of memory available. Once we have this information, we can preprocess the input formula Φ, leveraging the theoretical results of Section 9.3. If the temporal operators in Φ have bounded intervals less than K_{cluster}, we apply the unmodified version of the original algorithm [33], which evaluates formulae over point-based semantics. Otherwise, we have to transform the original formula into an equivalent one that can be checked in a memory-efficient way. This transformation is achieved by first interpreting the input formula Φ over lazy semantics: to preserve its meaning, we apply the $l2p$ transformation. Afterwards, given the parameter K_{cluster}, we rewrite the formula using the $\mathcal{L}_{K_{\mathsf{cluster}}}$ decomposition (i.e., the $\mathcal{L}_K$ decomposition instantiated with parameter K_{cluster}) and obtain the formula $\Phi_L^{K_{\mathsf{cluster}}} = \mathcal{L}_{K_{\mathsf{cluster}}}(l2p(\Phi))$. Thanks to Theorem 3, this formula contains intervals no greater than K_{cluster} and is equivalent to Φ.

The basic flow of the algorithm is unchanged and each iteration still consists of read, map and reduce phase.

Read phase modifications. Read phase remains unmodified.

Map phase modifications. To support the lazy semantics, the algorithm needs to consider all the positions in the trace where we want to evaluate the temporal operators. If any of these positions did not exist in the trace then the original algorithm would evaluate a formula to false (see the example in Section 9.1). However, to support the lazy semantics, we do not need to introduce a position in the trace for each time instant: we know a priori that only formulae of the form $\mathsf{F}_{=K}$—explicitly introduced by the $\mathcal{L}_K$ decomposition— may be evaluated incorrectly if the appropriate positions are missing in the trace (see Section 9.1). Therefore, we modify the algorithm for the mapper (see Fig. 9.5a) to introduce one position at $\tau + K$ only when the parent formula ψ is a subformula of the form $\mathsf{F}_{=K}$; this condition is captured by the $lazy()$ predicate. The emitted tuple contains the tuple $(\varphi_{act}, \bot, \tau + K)$ as its value. The mapper is stateless and cannot check if a tuple at time instant $\tau + K$ already exists: it is the reducer's responsibility to discard a tuple if it has a duplicate.

```
1  function MAPPER^Φ((φ, (v, τ)))
2    for all ψ ∈ sup_Φ(φ) do
3      output(ψ, (φ, v, τ))
       if lazy(ψ) then
         output(ψ, (φ_act, ⊥, τ + K))
       end if
4    end for
5  end function
```

(a) *Mapper algorithm*

```
1  function CHECKDUP(T)
2    r ← T.next()
3    a ← T.peek()
4    if r.φ ≠ φ_act then
5      if (a.φ = φ_act ∧ a.τ = r.τ) T.next() end if
6    else
7      if (a.φ ≠ φ_act ∧ a.τ = r.τ) r ← T.next()
8      end if
9    end if
10   return r
11 end function
```

(b) checkDup *function*

```
1  function REDUCER_l^{F_I, P_I}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅
4      for all (φ, v, τ) ∈ checkDup(T) do
5        updateQueue((φ, v, τ), win, I, \x → ())
6        val ← |τ − win.first.τ| ∈ I
7        output(ψ, (val, τ))
8      end for
9    end if
10 end function
```

(c) *Reducer algorithm for operator* F

```
1  function REDUCER_l^{G_I, H_I}(ψ, T)
2    if (checkIteration(ψ, T, l)) then
3      win ← ∅
4      for all (φ, v, τ) ∈ checkDup(T) do
5        updateQueue((φ, ¬v, τ), win, I, \x → ())
6        val ← |τ − win.first.τ| ∈ I
7        output(ψ, (¬val, τ))
8      end for
9    end if
10 end function
```

(d) *Metric* G_I *and* H_I *operators*

Figure 9.5: *Mapper and Reducer algorithm changes. (*sup_Φ *is defined in Section 2.2)*

Reduce phase modifications. All reducer algorithms are thus modified to discard all duplicates of tuple $(\varphi_{act}, \bot, \tau)$ by means of the checkDup function as exemplified in line 4 of Algorithm 9.5c and 9.5d. We chose to present the modified algorithms for "Eventually" and "Globally" operators since they are directly used in the parametric decomposition. The other algorithms have been modified in the same way, but we omit them for brevity. The function checkDup (implemented in Algorithm 9.5b) simply checks two adjacent tuples, if one of them has φ_{act} as a key, and both have the same timestamp the $(\varphi_{act}, \bot, \tau)$ tuple is discarded.

Examples of application of the algorithm. Let us use our algorithm to evaluate the formula Φ from Section 9.1 on the same trace using $\mathsf{MTL_P}$ semantics. In the *read* phase the algorithm parses the trace in parallel and creates the input tuples for the *map* phase. From the first element $(\{p\}, 1)$ the *Input Reader* creates only the tuple $(p, (\top, 1))$ since Φ refers only to atom p. Tuples $(p, (\top, 1))$, $(p, (\top, 2))$, $(p, (\bot, 4))$, $(p, (\top, 6))$, $(p, (\top, 8))$, $(p, (\bot, 9))$, $(p, (\bot, 10))$ are thus received by the *map* phase. The *Mapper* associates the formulae from the input tuples with their superformulae. In the case of tuple $(p, (\top, 1))$ it generates only tuple $(\mathsf{F}_{[3,7]}(p), (p, \top, 1))$ since $\mathsf{F}_{[3,7]}(p)$ is the only superformula of p. The *Reduce* phase, therefore, receives tuples $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 10)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 9)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 8)))$,

l	Atoms: / Time-stamps:	$\{p\}$ 1	$\{p\}$ 2		$\{q\}$ 4		$\{p,q\}$ 6		$\{p,q\}$ 8	$\{q\}$ 9	$\{q\}$ 10				
	Time instants:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	p	$\top$	$\top$		$\bot$		$\top$		$\top$	$\bot$	$\bot$				
1	$\mathsf{F}_{[3,4]}(p)$	$\top$	$\top$		$\top$	$\top$	$\bot$		$\bot$	$\bot$	$\bot$				
	$\mathsf{F}_{[0,3]}(p)$	$\top$	$\top$		$\top$	$\top$	$\top$		$\top$	$\bot$	$\bot$				
	φ_{act}					$\bot$	~~$\bot$~~		~~$\bot$~~		~~$\bot$~~		$\bot$	$\bot$	$\bot$
2	$\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$	$\top$	$\top$		$\top$	$\bot$	$\bot$		$\bot$	$\bot$	$\bot$		$\bot$	$\bot$	$\bot$
3	$\mathcal{L}_4(l2p(\Phi))$	(⊤)	(⊤)		(⊤)	$\top$	(⊥)		(⊥)	(⊥)	(⊥)		$\bot$	$\bot$	$\bot$

Figure 9.6: *Evaluation of the* $\mathcal{L}_4(l2p(\Phi)) = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$ *formula over* MTL_L *semantics.*

$(\mathsf{F}_{[3,7]}(p), (p, (\top, 6)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 4)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 2)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 1)))$, all shuffled and sorted in descending order of their time-stamps. Since all the tuples have the same key, only one reducer is needed. The reducer applies the algorithm shown in Figure 9.5c and outputs the truth values of $\mathsf{F}_{[3,7]}(p)$ for every position in the trace:
$(\mathsf{F}_{[3,7]}(p), (\bot, 10))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 9))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 8))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 6))$, $(\mathsf{F}_{[3,7]}(p), (\top, 4))$, $(\mathsf{F}_{[3,7]}(p), (\top, 2))$, $(\mathsf{F}_{[3,7]}(p), (\top, 1))$. Notice that the boolean values in the tuples correspond to the values in Figure 9.1 (row #4).

Assuming again that the memory requirement of keeping 8 positions is too demanding for our infrastructure we can now use parametric decomposition and lazy semantics to limit the upper bound of the interval in Φ to $K = 4$. We obtain formula $\mathcal{L}_4(l2p(\Phi)) = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$.

Let us evaluate formula $\mathcal{L}_4(l2p(\Phi))$ on the same trace from Section 9.1 over MTL_L semantics. Table 9.6 shows the truth values of the emitted tuples for every evaluated subformulae of $\mathcal{L}_4(l2p(\Phi))$. Since $h(\mathcal{L}_4(l2p(\Phi))) = 4$ the algorithm performs three iterations (whose index is indicated in the left-most column l). The truth values of the subformulae from the different iterations are separated by the horizontal dashed lines. In the first iteration the trace is parsed to obtain the truth values of atom p. After that, two reducers in parallel calculate the truth values of the $\mathsf{F}_{[0,3]}(p)$ and $\mathsf{F}_{[3,4]}(p)$ subformulae. In the second iteration the *Mapper* emits the additional φ_{act} tuples since the superformula is of the form $\mathsf{F}_{=4}$. The reducer evaluating formula $\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$ receives the tuples with the evaluation of $\mathsf{F}_{[0,3]}(p)$ and φ_{act}. The φ_{act} tuples with the crossed truth values are discarded because of the already existing $\mathsf{F}_{[0,3]}(p)$ tuples shown in the row above. Finally, in the third

iteration we can see that the truth values $\mathcal{L}_4(l2p(\Phi))$ (circled in Figure 9.6) are the same (at all time instants in common) as the truth values of Φ shown in Figure 9.1.

CHAPTER 10

Evaluation & Application

10.1 Overview

In this section we report on the evaluation of our tool, in terms of scalability and time/memory tradeoffs and compare it to other available trace checking tools. More specifically, we evaluate our new trace checking algorithm by answering the following research questions:

RQ1: *How does the proposed algorithm scale with respect to the size of the trace and the height of the formula?* (Section 10.2)

RQ2: *How does the proposed algorithm scale with respect to the size of the time interval used in the formula to be checked?* (Section 10.2)

RQ3: *How does our the trace checking tool perform compared to state-of-the-art tools?* (Section 10.3)

RQ4: *What are the time/memory tradeoffs of the proposed algorithm with respect to the decomposition parameter K?*(Section 10.4)

RQ5: *How do the different values of the decomposition parameter K affect the size and the height of the decomposed formula?* (Section 10.5)

To evaluate our approach, we used six *t2.micro* instances from the Amazon EC2 cloud-based infrastructure with a single CPU core and 1 GB of memory each. We used the standard configuration for the HDFS distributed

file system and the YARN data operating system. HDFS block size was set to $64\,\mathrm{MB}$ and block replication was set to 3. YARN was configured to allocate containers with memory between $512\,\mathrm{MB}$ and $1\,\mathrm{GB}$ with 1 core. We have used a cluster provisioning tool [71] that enables consistent deployment of any number of nodes on the Amazon EC2 cloud-based infrastructure and can be easily used to replicate the results. For Hadoop service provisioning we have used Apache Ambari [3] and Hue [1] to interact with Hadoop services easily. For each experiment we have set a time limit to be 5 minutes and memory limit to $1\,\mathrm{GB}$ per machine. Each point shown in the plots, represents an average value of 10 trace check runs on traces of the same length.

Measuring the actual memory usage of user-defined code in Spark-based applications requires to distinguish between the memory usage of the Spark framework itself and the one of user-defined code. This step is necessary since the framework may use the available memory to cache intermediate data to speedup computation. Hence, to measure the memory usage of the auxiliary data structures used by our algorithm (e.g., the *win* queue), we instrumented the code. This instrumentation, which has a negligible overhead, monitors the memory usage of the algorithm's data structures and reports the maximum usage for each run.

For the evaluation described in the next two subsections, we used *synthesized* traces. By using synthesized traces, we are able to control in a systematic way the factors, such as the trace length and the frequency of events, that impact on the time and memory required for checking a specific type of formula. In particular, we evaluated our approach by triggering the worst-case scenario, in terms of memory scalability, for our trace checking algorithm. Such scenario is characterized by having the auxiliary data structures used by the algorithm always at their maximum capacity. To synthesize the traces, we implemented a trace generator program that takes as parameters the desired trace length n and the number $m \geq 3$ of events (i.e., atoms) per trace element. The program generates a trace with n trace elements, such that the i-th element (with $0 \leq i \leq n-1$) has $\tau_i > 0$ as time-stamp value and $\tau_i < \tau_{i+1}$ for all $0 \leq i \leq n-2$. Each trace element has between 3 and m events denoted as $\{e_1, \ldots, e_m\}$, where $e_1 = p_1$ is present at every trace element; events $e_2 = p_2$ and $e_3 = p_3$ are generated to alternate starting with p_2 and the other $m-3$ events are randomly selected from the set $\{p_4, \ldots, p_m\}$ using a uniform distribution. Events p_2 and p_3 are useful for evaluating performance of the $\mathfrak{D}$ modality since its semantics assumes alternating events.

10.2 Scalability

To address RQ1: *How does the proposed algorithm scale with respect to the size of the trace and the height of the formula?*, we considered 4 formulae, with different heights:

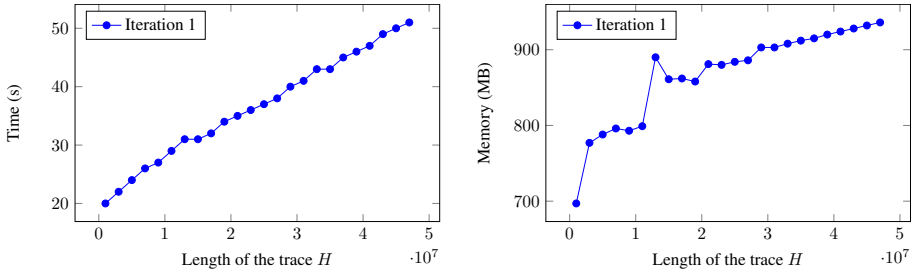
P1: $\mathfrak{C}_{<10}^{50000000}(p_1)$,

P2: $\mathfrak{D}_{<10}^{50000000}(p_2, p_3)$,

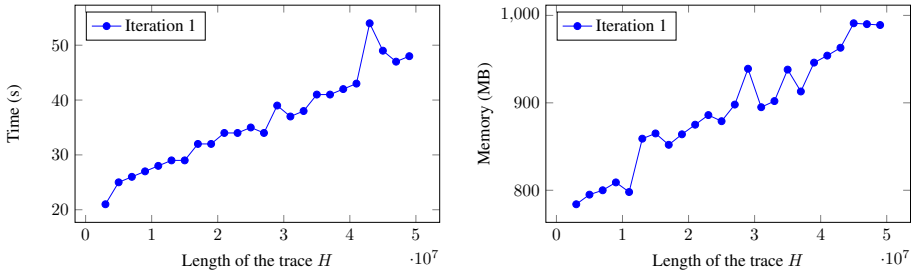
P3: $(a_4 \wedge (p_5 \wedge p_6))\mathsf{U}_{(50,1000000)}((p_5 \wedge p_6) \vee p_5)$

P4: $\exists j \in [0,9]\ \forall i \in [0,8] : \mathsf{G}_{(50,50000000)}(p_{4+z} \rightarrow \mathsf{X}_{(50,1000000)}(p_{5+z}))$

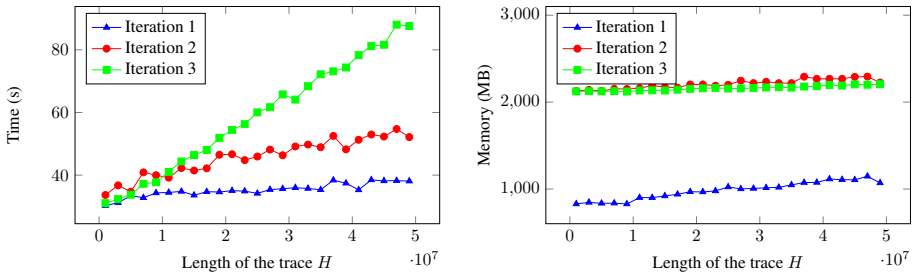
where $z = \frac{(i+j)(i+j+1)}{2} + i$. In P3 and P4, the $\forall$ and $\exists$ quantifiers are used just as a shorthand notation to predicate on finite number of atoms: for example, $\forall i \in \{1,2,3\} : a_i$ is equivalent to $a_1 \wedge a_2 \wedge a_3$. The expression z is utilized to enumerate all the pairs i and j, so that we can avoid having two subscripts for the atoms. To check the scalability of the algorithm with respect to the size of the trace, we used our trace generation tool described above to synthesize traces of lengths varying from 1 million to 50 million with an increment of 2 million. We chose $m = 100$ distinct events for each time instant. Hence, we evaluated our algorithm on traces with up to 5 billion events. The average time span of the trace is 347.2 days, if timestamps are incremented by one at each instant and time granularity is one second. Figure 10.1 shows the time and the total memory utilized across all the nodes used by the algorithm to check the four formulae on the synthesized traces. The time and memory utilization is broken down by iterations. Formulae $\mathfrak{C}_{<10}^{50000}(p_1)$ and $\mathfrak{D}_{<10}^{50000}(p_2, p_3)$ need one iteration to be evaluated (shown in Figure 10.1a and Figure 10.1b). In both cases, the time taken to check the formula increases linearly with respect to the trace length; time increases because reducers need to process more tuples. As for the linear increase in memory usage, for modalities $\mathfrak{C}$ and $\mathfrak{D}$ reducers have to keep track of all the tuples in the window of length K time units larger than the trace itself, so the more time instants there are in the trace, more tuples must be buffered by the win queue, with a consequent increase in memory usage. Checking the other two formulae (shown in Figure 10.1c and Figure 10.1d), requires more iterations because of their larger height. Also in this case, the time taken by each iteration tends to increase as the length of the trace increases, especially in the iterations where the metric temporal operators G and X are considered. Notice the increase of the overall time and memory usage from Figure 10.1c to



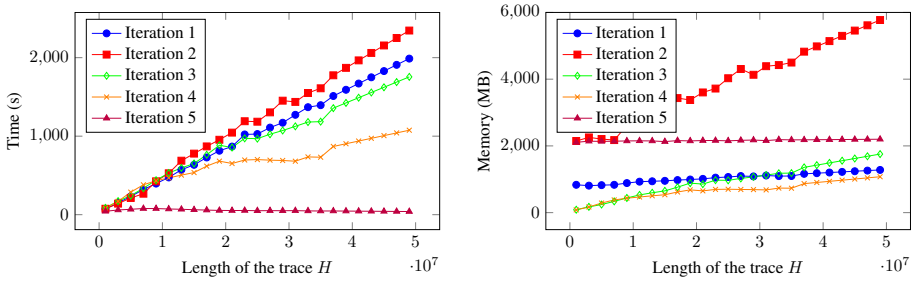
(a) *Formula:* $\mathfrak{C}_{<10}^{50000000}(p_1)$



(b) *Formula:* $\mathfrak{D}_{<10}^{50000000}(p_2, p_3)$



(c) *Formula:* $(a_4 \wedge (p_5 \wedge p_6))\mathsf{U}_{(50,1000000)}((p_5 \wedge p_6) \vee a_5)$



(d) *Formula:* $\exists j \in [0,9]\ \forall i \in [0,8] : \mathsf{G}_{(50,50000000)}(p_{4+z} \rightarrow \mathsf{X}_{(50,1000000)}(p_{5+z})$ *for* $z = \frac{(i+j)(i+j+1)}{2} + i$

Figure 10.1: *Time (left) and memory (right) scalability of the algorithm*

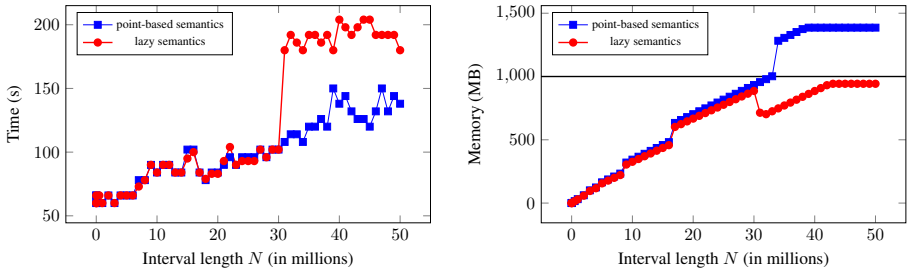
Figure 10.1d: this is due to the expansion of the quantifiers in formula $\exists j \in [0,9]\ \forall i \in [0,8] : \mathsf{G}_{(50,50000000)}(p_{4+z} \rightarrow \mathsf{X}_{(50,1000000)}(p_{5+z}))$ that creates many formulae that can be checked in parallel.

To address RQ2: *How does the proposed algorithm scale with respect to the size of the time interval used in the formula to be checked?*, we evaluate the memory usage of the algorithm for different sizes of the time interval used in the MTL formula to be checked. As discussed in Section 9.4, the largest time interval manageable in a cluster depends on the memory configuration of the node in the cluster with the least amount of memory available. Hence, we evaluate the memory usage on a single node by using formulae of height 1; nevertheless, the map phase is still executed in parallel. We consider the two metric formulae $\mathsf{F}_{[0,N]}p_1$ and $\mathsf{G}_{[0,N]}p_0$, parametrized by the value N of the bound of their time interval. Formula $\mathsf{F}_{[0,N]}p_1$ refers to atom p_1; notice that our trace generator guarantees that p_1 is present in every trace element. Formula $\mathsf{G}_{[0,N]}p_0$ refers to atom p_0; we configured our trace generator so that event p_0 is absent in all trace elements. These two formulae exercise the trace checking algorithm in its worst-case. Indeed, according to line 4 in Figure 9.5c, the reducer for F_I buffers all the elements where atom p_1 is true; hence, when checking formula $\mathsf{F}_{[0,N]}p_1$, at any point in time the queue *win* will be at its maximum capacity. Dually, when checking formula $\mathsf{G}_{[0,N]}p_0$, the absence of the event p_0 from the trace will force the algorithm to maintain the queue *win* at its maximal capacity (line 4 in Figure 9.5d). Notice that a particular heuristic for formulae $\mathsf{F}_{[0,N]}p_1$ and $\mathsf{G}_{[0,N]}p_0$ would simply limit their evaluation to the first N positions. However, here we consider the most general case, in which formulae can be arbitrarily nested. This case requires to evaluate every subformula in every position of the trace.

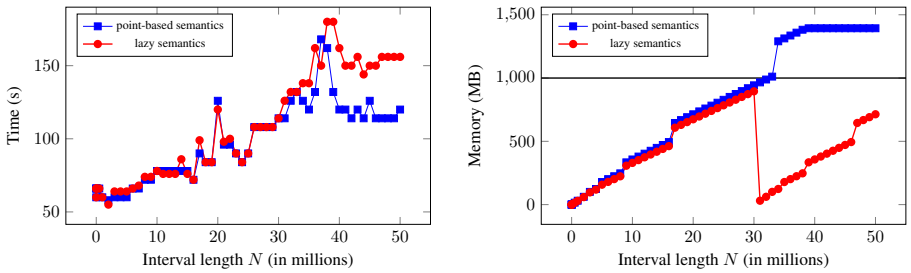
We used our trace generation tool to synthesize ten traces, with length set to $n = 50\,000\,000$ and number of event set to $m = 20$ events; the average size of each trace, before saving it in the distributed file system, is $3.2\,\mathrm{GB}$. We make sure that the trace generator creates a trace element for every time instant in the trace, i.e., $\tau_i = i$ in order to stress the memory capacity of the algorithms' data structures. These traces and the other artifacts used for the evaluation are available on the tool web site [84]. Plots in Figures 10.2a and 10.2b show the execution time and the memory usage required to check, respectively, formula $\mathsf{F}_{[0,N]}p_1$ and $\mathsf{G}_{[0,N]}p_0$, instantiated with different values of parameter N. Each data point is obtained by running the algorithm over the ten synthesized traces and averaging the results. The plots colored in blue show the average time and memory usage of our previous algorithm [33], which applies $\mathsf{MTL}_{\mathsf{P}}$ semantics. The plots colored

Table 10.1: *Average processing time per tuple for the four properties.*

	Property 1		Property 2		Property 3		Property 4	
	SOLOIST	LTL	SOLOIST	LTL	SOLOIST	LTL	SOLOIST	LTL
Number of tuples	16,121	55,009	24,000	119,871	215,958	599,425	1,747,360	4,987,124
Time per event (μs)	1.172	19	1.894	21	3.707	14	7.200	30



(a) *Execution time (left) and memory usage (right) to check* $\mathsf{F}_{[0,N]}q$



(b) *Execution time (left) and memory usage (right) to check* $\mathsf{G}_{[0,N]}p$

Figure 10.2: *Comparison between our two proposed trace checking algorithms.*

in red represent the runs of our new algorithm that applies $\mathsf{MTL}_{\mathsf{L}}$ semantics and decomposes all the formulae with time interval N strictly greater than $30\,000\,000$. The decomposition parameter $K = 30\,000\,000$ is the maximal value that our infrastructure can support.

We answer RQ2 by observing the trend in the red plots of Figures 10.2a and 10.2b: the proposed algorithm can check, on very large traces, formulae that use very large time intervals (up to $50\,000\,000$), using at most 1GB of memory and taking a reasonable time (at most 200s).

10.3 Comparison

To address RQ3: *How does our the trace checking tool perform compared to state-of-the-art tools?*, we need a baseline for comparison. Among the

non-distributed, non-parallel trace checking tools for MTL, we selected the MONPOLY [16] tool, which was the best performing tool supporting MTL in the "offline monitoring" track of the first international Competition on Software for Runtime Verification [14] (CSRV 2014). MonPoly, when executed on the traces described above, produced a stack overflow error[1]; hence we could not use it for comparison. Among *distributed and parallel approaches* there are tools supporting LTL [12, 21] and MTL [15, 33] specification languages. The only publicly available tool supporting LTL is *DecentMon* [21], however when invoked with a trace of length 1 million as input, it produces a segmentation fault, therefore we could not use it for the comparison. Among the tools supporting MTL only our tool [33, 84] is publicly available therefore we compare and discuss the two algorithms developed for point-based and lazy semantics respectively. We also try to replicate the experimental setting from [12] that inspired our algorithm, in order to provide an idea how our implementation compares to one in [12].

To answer RQ3, the plots show that the proposed algorithm is more scalable in terms of memory usage than the algorithm from [33]. Indeed, for the evaluation of both formulae, the latter exhausts the memory bound of $1\mathrm{GB}$ when the time interval N is higher than $30\,000\,000$. Nevertheless, the proposed algorithm is on average 1.35x slower that the previous algorithm [33] when the time interval N is higher than $30\,000\,000$. This additional time is needed to process the new formula obtained through the $\mathcal{L}_K$ decomposition.

To compare our approach to the one presented in [12], which focuses on trace checking of LTL properties using MapReduce we consider formulae from the LTL fragment included in SOLOIST. Although the focus of our work was on implementing the semantics of SOLOIST temporal and aggregate operators, we also introduced some improvements in the LTL layer of SOLOIST with respect to the algorithm from [12]. First, we exploited composite keys and secondary sorting as provided by the MapReduce framework to reduce the memory used by reducers. We also extended the binary $\wedge$ and $\vee$ operators to support any positive arity and therefore reducing the height of large conjunctions and disjunctions.

We compared the two approaches by checking the following formulae:

P1: $\mathsf{G}(\neg p_4)$;

P2: $\mathsf{G}(p_4 \rightarrow \mathsf{X}(p_5))$;

P3: $\forall i \in \{0 \ldots 8\} : \mathsf{G}(p_{4+i} \rightarrow \mathsf{X}(p_{5+i}))$; and

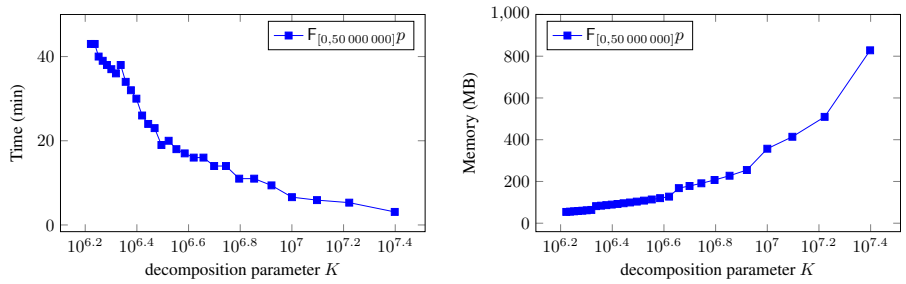
P4: $\exists j \in \{0 \ldots 9\}\ \forall i \in \{0 \ldots 8\} : \mathsf{G}(p_{4+z} \rightarrow \mathsf{X}(p_{5+z}))$.

[1] To replicate the error refer to the Replication package section at [84]

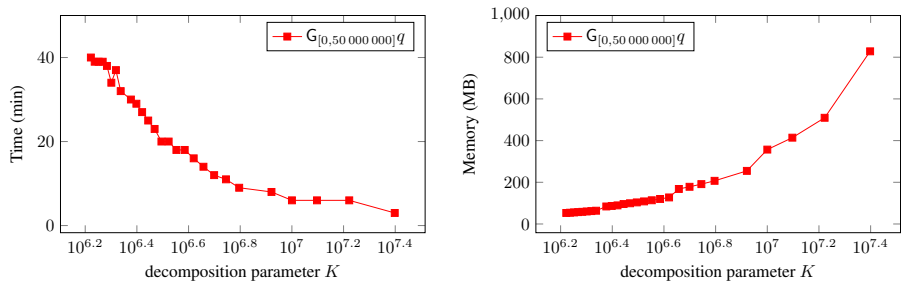
The height of these formulae are 3, 4, 5 and 6, respectively. This admittedly gives our approach a significant advantage since in [12] the restriction for the $\wedge$ and $\vee$ operators to have an arity fixed to 2 results in a larger height for formulae 3 and 4. For the comparison, we have synthesized traces with length, ranging from 1000 to 100000 time instants, with up to 100 events per time instant. With this configuration, a trace can contain potentially up to 10 million events. We chose to have up to 100 events per time instant to match the configuration proposed in [12], where there are 10 parameters per formula that can take 10 possible values. We generated 500 traces. The time needed by our algorithm to check each of the four formulae, averaged over the different traces, was 6.28, 8.89, 16.14 and 42.53 seconds, respectively. We do not report the time taken by the approach proposed in [12] since the article does not report any statistics from the run of an actual implementation, but only metrics determined by a simulation. Table 10.1 shows the average number of tuples generated by the algorithm for each formulae. The number of tuples is calculated as the sum of all input tuples for mappers at each iterations in a single trace checking run. The table also shows the average time needed to process a single event in the trace. This time is computed as the total processing time divided by the number of time instants in the trace, averaged over the different trace checking runs. The SOLOIST column refers to the data obtained by running our algorithm, while the LTL column refers to data reported in [12], obtained with a simulation. Our algorithm performs better in terms of processing time.

10.4 Tradeoff

To address RQ4: *What are the time/memory tradeoffs of the proposed algorithm with respect to the decomposition parameter K?*, we evaluate the execution time and the memory usage of the algorithm for different values of parameter K. As suggested in Section 10.3, the parametric decomposition used with lazy semantics leads to a reduced memory usage, but increases the execution time. In this section we dig into and generalize this result by investigating the time/memory tradeoffs of our algorithm, with respect to the decomposition parameter K (RQ4). We consider formulae $\mathsf{G}_{[0,50\,000\,000]}q$ and $\mathsf{F}_{[0,50\,000\,000]}p$ and perform trace checking using different decomposition parameter K. These formulae are processed using the $\mathcal{L}_K$ decomposition, with values of K that are taken from the set $V = \{\frac{5 \cdot 10^7}{i} \mid i = 2, 3, 4, \ldots\}$. As the set V is potentially infinite, we set a threshold of one hour on the execution time.



(a) *Execution time and memory usage when varying parameter K for "Eventually" formula*



(b) *Execution time and memory usage when varying parameter K for "Globally" formula*

Figure 10.3: *Time/memory tradeoffs for the proposed trace checking algorithm.*

The plots in Figure 10.3 show the execution time and the memory usage to check the two formulae. Each data point is obtained by running the algorithm over the ten synthesized traces and averaging the results. The value of K is represented in both plots on the x-axis using the logarithmic scale. The smallest value of K that satisfies the execution time threshold is $1\,666\,666$ (obtained from set V with $i = 30$); for this value of K the algorithm used 54.14MB of memory and took 43 minutes to complete. The plots show that using a lower value for K decreases the memory footprint of the algorithm. However, a lower value for K also yields a longer execution time for the algorithm. This longer execution time is due to the fact that a lower value for K increases the size (and the height) of the formula obtained after applying the L_K decomposition. The increased height of the decomposed formula triggers more iterations of the algorithm, yielding longer execution times. We answer RQ 4 by stating that there is a tradeoff between time and memory, determined by the value of parameter K. A good balance between these two factors can be achieved when K is set to the largest possible value supported by the infrastructure: in this way, it is possible to reduce the size of the decomposed formula without incurring

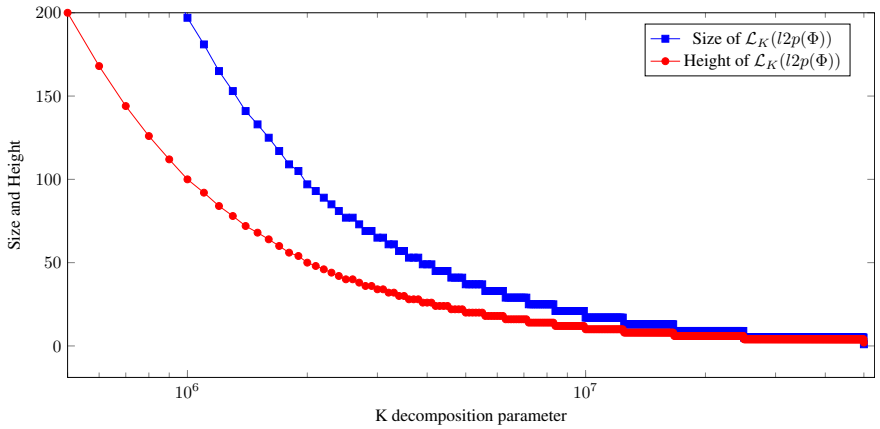


Figure 10.4: *Size and height of* $\mathsf{G}_{50,000,000}\mathsf{p}_0$ *decomposed with different parameter* K.

a longer execution time for the algorithm. Nevertheless, our algorithm is completely parametric in K, allowing engineers to tune the algorithm to be either more time- or more memory-intensive, depending on their needs.

10.5 Size of the decomposed formula

Regarding RQ 5: *How do the different values of the decomposition parameter* K *affect the size and the height of the decomposed formula?*, Figure 10.4 shows how the size (blue plot) and height (red plot) of formula $\Phi = \mathsf{G}_{50,000,000}\mathsf{p}_0$ changes when decomposed with different values of K. The values of K range from $100,000$ to $50,000,000$ with an increment of $100,000$. The plot shows that both the size and the height of the decomposed formula increase when K decreases. This means that the choice of K should be the largest possible value supported by the infrastructure, in order to reduce the size of the decomposed formula and in turn the running time of the algorithm.

CHAPTER 11

State of the Art

The approach presented in this thesis is strictly related to work done in the areas of distributed and parallel trace checking/run-time verification and of alternative semantics for metric temporal logics.

Trace checking/run-time verification

Several approaches for trace checking and run-time verification and monitoring of temporal logic specifications have been proposed in the last decade. The majority of them (see, for example, [17, 63, 78, 108, 114]) are centralized and use sequential algorithms to process the trace (or, in online algorithms, the stream of events). As mentioned in Section 10.3, the centralized, sequential nature of these algorithms does not allow them either to process large traces or properties containing very large time bounds. In the last years there have been approaches for trace checking [15] and runtime verification [21,92,108] that rely on some sort of parallelization. These approaches mostly focus on splitting the traces based on the data they contain, rather than on the structure of the formula. They adopt first-order relations with finite domains or infinite domains with finite representations as the events in the trace. The trace can then be split into several unrelated par-

titions based on the terms occurring in the relations. This work considers these approaches as *orthogonal* to the one presented in this thesis that focuses on the scalability with respect to the temporal dimension, rather than the data dimension.

As stated in Chapter 8, the algorithms for the MTL temporal operators are inspired by ones proposed in [20]. The main differences are that in the case of our algorithms no recursive call to evaluate subformulae is needed, since arbitrary nesting of the formulae is handled by the MapReduce iterative procedure. The additional support for future-time temporal operators is possible since we propose an *offline* procedure.

As for the specific application of MapReduce for trace checking, an iterative algorithm for LTL is proposed in [12]. Similarly to the algorithm presented in this thesis the algorithm in [12] performs iterations of MapReduce jobs depending on the height of the formula to check. However, it does not address the issue of memory consumption of the reducers. Moreover, the whole trace is kept in memory during the reduce phase, making the approach unfeasibile for very large traces.

Distributed computing infrastructures and/or programming models have also been used for other verification problems. Reference [89] proposes a distributed algorithm for performing *model checking* of LTL *safety properties* on a network of interconnected workstations. By restricting the verification to safety properties, authors can easily parallelize a bread-first search algorithm. Reference [23] proposes a parallel version of the well-known fixed-point algorithm for CTL model checking. Given a set of states where a certain formula holds and a transition relation of a Kripke structure, the algorithm computes the set of states where the superformula of a given formula holds though a series of MapReduce iterations, parallelized over the different predecessors of the states in the set. The set is computed when a fixed-point of a predicate transformer is reached as defined by the semantics of each specific CTL modality.

Alternative semantics for metric temporal logics

The work closest to our lazy semantics is the one in [57], which proposes an alternative MTL semantics, used to prove that signal-based semantics is more expressive than point-based semantics over finite words. Despite the similarity between the two semantics, the definition of the *Until* operator over our lazy semantics is more practical for the purpose of trace checking, since it requires the left subformula of an *Until* operator to hold in a finite number of positions. Reference [48] revised the model parametric semantics (MPS) of the TRIO temporal logic [96], in order to overcome

counterintuitive behaviors of bounded temporal operators on a finite temporal domain. The proposal shares the same intuition behind our definition of lazy semantics, but overall the two semantics are quite different (in particular, in the interpretation of bounded and unbounded temporal operators). MPS defines a delayed evaluation, similar to our lazy semantics, used to evaluate temporal operators outside a finite temporal domain.

Part IV

Epilogue

CHAPTER 12

Summary and Conclusions

Most traditional software engineering techniques have dealt with systems that were assumed to live in a closed and controlled environment. However, software engineering has shifted towards a type of software that is characterized by a different set of assumptions that take into account the dynamic nature of the environment in which the software executes; therefore, we say that this new kind of software is embedded in an *open world* and we call it *open-world* software. The new assumptions are the following: software requirements are subject to change; software development and provisioning are decentralized and involve multiple stakeholders belonging to different organizations; systems are thus assembled out of components that provide a specific functionality; bindings among components are established dynamically (at run-time) and may vary to accommodate changes that support the evolution of the software as well as the environment with which the system interacts. Finally, the physical deployment of the system is typically performed on a cloud-based infrastructure that provides virtualized and distributed computing resources shared among many users. Thus the infrastructure on which the software runs is also subject to uncontrollable change.

The dynamic behavior of open-world software asks for verification tech-

niques that complement the design-time approaches, because the behavior that one wants to verify emerges only at run time. This puts forward techniques like *trace checking* as a viable complementary choice for verifying open-world software.

We focus on the verification of quantitative properties that can be seen as *constraints* on quantifiable values from a system execution. Currently, there is no consolidated research into verification of quantitative properties of open-world software. This thesis addresses the issue of verification of open-world software in the context of quantitative properties, stated more precisely with the following research goal:

> *"To study quantitative properties of systems occurring in practice and provide a practical and scalable approach to verification, driven by the selected specification language suitable to express such properties."*

The rest of this chapter summarizes the contributions of the thesis in Section 12.1 and points out limitations of the approach and possible future research directions in Section 12.2.

12.1 Contributions

We chose SOLOIST as the baseline specification language used to express a wide range of quantitative properties. SOLOIST is an extension of metric temporal logic (MTL) therefore it is able to specify both functional and non-functional properties of systems. Additionally, its extension allows specification of aggregating behavior of systems. Since the original language is undecidable due to the first-order quantification, we restricted our analysis to its propositional fragment.

Contribution 1 - Decision procedure for SOLOIST

We have implemented two efficient decision procedures for SOLOIST that make use of state-of-the-art SMT solver. The implementation is a translation that reduces the problem of SOLOIST satisfiability to satisfiability of a particular logic supported by the SMT solver theories. The main difference between the two implemented procedures is the target logic of the translation: CLTLB($\mathcal{D}$) and QF-EUFIDL respectively. An efficient decision procedure provides a general framework for building a SOLOIST verification suite that supports many verification use cases. We exploit the implemented decision procedures for SOLOIST to perform trace checking and showcase how the two decision procedures can be used complementarily.

Contribution 2 - Scalable trace checking of SOLOIST

The main requirements of algorithms for trace checking logics based on MTL is that they need to scale with respect to two crucial dimensions: the length of the trace and the size of the time interval of the formula to be checked. To address the former issue, we propose a distributed and parallel trace checking algorithm that can take advantage of modern cloud computing and programming frameworks like MapReduce and Spark. We address the latter issue by proposing an alternative semantics for MTL, called *lazy* semantics. Lazy semantics possesses certain properties that allow us to decompose any MTL formula into an equivalent MTL formula with all time intervals of its temporal operators limited by some constant. This decomposition plays a major role in the context of (distributed) trace checking of formulae with large time intervals.

Contribution 3 - Specifying quantitative properties

In the process of specifying quantitative properties of systems we have encountered many complex cases where SOLOIST is not expressive enough. Therefore we extended SOLOIST with arithmetical constrains (SOLOIST$^{\mathcal{A}}$) that allow us to express complex quantitative properties of cloud-based elastic systems, like *elasticity* or *resource thrashing*. Another contribution towards specifying quantitative properties is *lazy* semantics. Besides allowing us to optimize our distributed trace checking algorithm, we believe that lazy semantics makes the process of specifying system properties using MTL more intuitive.

12.2 Limitations and Future work

The work presented in this thesis has limitations and they represent interesting open issues that can be basis for future research directions.

Regarding the particular choice of the specification language, we note that this thesis relies on an informal definition of quantitative properties and the choice of specification language is driven by particular quantitative properties encountered in practice. The domain of service-based and cloud-based elastic systems represent typical instances of open-world software. Although we established that SOLOIST with provided extension could capture all the properties of interest encountered in practice, one could still raise a concern if it is appropriate to express any quantitative property. To answer this question a comprehensive study is needed to identify the complete spectrum of relevant quantitative properties and to precisely state the

adequacy of SOLOIST to express them.

Regarding the use of the decision procedure of SOLOIST for trace checking, one might raise a concern if this is an efficient approach. Indeed, satisfiability is a harder and a more general problem than trace checking and we show in Sections 3.4 and 4.4 that the complexity of SOLOIST satisfiability is EXPSPACE-complete. However, this is only an upper bound on the complexity of SOLOIST trace checking. Solvers typically have a much easer time producing a verdict for a formula that has a trace explicitly encoded and we show in the evaluation section that in practice this approach is very efficient given its worst case complexity. Furthermore, the SOLOIST decision procedure provides us with an opportunity for rapid development of prototype tools that enable other verification use cases. As part of the future work other verification use cases can be explored such as model checking or runtime verification. A possible research direction may explore operational models equivalent to SOLOIST and use them to model open-world software. In that case the SOLOIST decision procedure can be reused to perform model checking of quantitative properties expressed in SOLOIST.

Chapter 8 shows a particular strategy for parallel execution of SOLOIST trace checking — using the formula structure. However, there are different orthogonal strategies that can be applied to perform trace checking in a parallel manner. Namely, the trace to be checked can be split into several partitions based on the data it contains or based on time it spans [15]. In the former case each partition needs to contain data that is not related to any data in the other partitions based on the formula to be checked, while in the latter case partitions contain portions of the trace bounded by overlapping time intervals. The main idea is to obtain partitions of the trace that can be checked independently. Applying these techniques would improve the parallelization of the current approach and this remains to be investigated in the future.

Regarding the parametric decomposition optimization introduced in Chapter 9, one might state that it does not apply for the full fragment of SOLOIST. Indeed, parametric decomposition is designed as an optimization for real-time specifications that can be expressed with MTL. It remans to be investigated if similar reasoning can be applied to decompose SOLOIST aggregating modalities into ones with smaller time windows.

As stated in Section 9.4 the optimal choice of value of K depends on the particular configuration of the cluster. In practice we estimate the value of K offline, before running the algorithm through experimentation on the cluster. Choosing the optimal value of K online, during the algorithm's execution remains a very interesting future research direction.

Bibliography

[1] Cloudera Inc, Hue. gethue.com, visited on June 2015.

[2] Google API for android. `https://beep.metid.polimi.it/`.

[3] The ASF, Apache Ambari. ambari.apache.org, visited on October 2015.

[4] Wikipedia page traffic statistics. `http://aws.amazon.com/datasets/2596`.

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

[6] Rajeev Alur and Thomas A. Henzinger. Real-time logics: complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, May 1993.

[7] Tony Andrews et al. Business Process Execution Language for Web Services, Version 1.1, 2003.

[8] Apache Software Foundation. Hadoop MapReduce. `http://hadoop.apache.org/mapreduce/`.

[9] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, Oct 2006.

[10] Luciano Baresi, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, 2007.

[11] Luciano Baresi and Elisabetta Di Nitto, editors. *Test and Analysis of Web Services*. Springer, 2007.

[12] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. MapReduce for Parallel Trace Validation of LTL Properties. In *Proceedings of RV 2012*, volume 7687 of *LNCS*, pages 184–198. Springer, 2013.

[13] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret H. Smith. An entry point for formal methods: Specification and analysis of event logs. In *Proceedings of the Workshop on Formal Methods for Aerospace, FMA 2009, Eindhoven, The Netherlands*, pages 16–21, 2009.

[14] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In *Proceedings of RV 2014*, volume 8734 of *LNCS*, pages 1–9. Springer, 2014.

Bibliography

[15] David Basin, Germano Caronni, Sarah Ereth, Matùš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In *Proceedings of RV 2014*, volume 8734 of *LNCS*, pages 31–47. Springer, 2014.

[16] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monpoly: Monitoring usage-control policies. In *Proceedings of RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364, 2011.

[17] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monitoring data usage in distributed systems. *IEEE Trans. Softw. Eng.*, 39(10):1403–1426, 2013.

[18] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. In *Proceedings RV'13*, volume 8174 of *LNCS*, pages 40–58. Springer, 2013.

[19] David Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.

[20] David Basin, Felix Klaedtke, and Eugen Zălinescu. Algorithms for monitoring real-time properties. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 260–275. Springer, 2012.

[21] Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. In *Proceedings of FM 2012*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.

[22] Andreas Bauer, Rajeev Goré, and Alwen Tiu. A first-order policy language for history-based transaction monitoring. In *Proceedings of ICTAC '09*, volume 5684 of *LNCS*, pages 96–111. Springer, 2009.

[23] Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Distributed CTL model checking in the cloud. Technical Report 1310.6670, Cornell University, October 2013.

[24] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87. ACM, 2000.

[25] Marcello M. Bersani, Achille Frigeri, Angelo Morzenti, Matteo Pradella, Matteo Rossi, and Pierluigi San Pietro. Bounded reachability for temporal logic over constraint systems. In *Proceedings of TIME'10*, pages 43–50. IEEE Computer Society, 2010.

[26] Marcello M. Bersani, Achille Frigeri, Angelo Morzenti, Matteo Pradella, Matteo Rossi, and Pierluigi San Pietro. Constraint LTL satisfiability checking without automata. *CoRR*, abs/1205.0946, 2012.

[27] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. Deciding continuous-time metric temporal logic with counting modalities. In *Proceedings of RP 2013*, volume 8169 of *LNCS*, pages 70–82. Springer, 2013.

[28] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. On the satisfiability of metric temporal logics over the reals. In *Proceedings of AVOCS'13*, 2013.

[29] Marcello Maria Bersani, Domenico Bianculli, Schahram Dustdar, Alessio Gambi, Carlo Ghezzi, and Srđan Krstić. Towards the formalization of properties of cloud-based elastic systems. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-oriented Systems (PESOS 2014), co-located with ICSE 2014, Hyderabad, India*. ACM, June 2014.

[30] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. In *Proceedings of FASE 2014*, volume 8411, pages 276–290. Springer, April 2014.

[31] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using MapReduce. 2016.

[32] Domenico Bianculli. *Open-world Software: Specification, Verification, and Beyond*. PhD thesis, Università della Svizzera italiana, July 2012.

[33] Domenico Bianculli, Carlo Ghezzi, and Srđan Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In *Proceedings of SEFM 2014*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.

[34] Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Offline trace checking of quantitative properties of service-based applications. In *Proceedings of the 7h International Conference on Service Oriented Computing and Application (SOCA 2014), Matsue, Japan*. IEEE, November 2014.

[35] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proceedings of ICSE 2012*, pages 968–976. IEEE Computer Society, 2012.

[36] Domenico Bianculli, Carlo Ghezzi, and Pierluigi San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *Proceedings of FACS'12*, volume 7684 of *LNCS*, pages 55–72. Springer, 2013.

[37] Domenico Bianculli, Mehdi Jazayeri, and Mauro Pezzè, editors. *Matinée with Carlo Ghezzi - from Programming Languages to Software Engineering*. CreateSpace, June 2012.

[38] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(15), 2006.

[39] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. *Information and Computation*, 208(2):97 – 116, 2010.

[40] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing & verification in service-oriented architecture: A survey. *Softw. Test. Verif. Reliab.*, 2012.

[41] Andrea Burattin and Alessandro Sperduti. PLG: A framework for the generation of business process models and their execution logs. In *BPM Workshops*, volume 66 of *LNBIP*, pages 214–219. Springer, 2011.

[42] Gerardo Canfora and Massimiliano Di Penta. Service oriented architectures testing: a survey. In *ISSSE 2006–2008*, volume 5413 of *LNCS*, pages 78–105. Springer, 2009.

[43] CELAR Project. Description of Work. `http://www.celarcloud.eu`, 2012.

[44] K. Chatterjee, T.A. Henzinger, and J. Otop. Nested weighted automata. In *Logic in Computer Science (LICS), 2015 30th Annual ACM/IEEE Symposium on*, pages 725–737, July 2015.

[45] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. *ACM Trans. Comput. Logic*, 11(4):23:1–23:38, July 2010.

[46] Boris Cherkassky and Andrew Goldberg. Negative-cycle detection algorithms. In *Algorithms - ESA 1996*, volume 1136 of *LNCS*, pages 349–363. Springer, 1996.

[47] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 154–169. Springer, 2000.

[48] Alberto Coen-Porisini, Matteo Pradella, and Pierluigi San Pietro. A finite-domain semantics for testing temporal logic specifications. In *Proceedings of FTRTFT 1998*, pages 41–54. Springer, 1998.

Bibliography

[49] Gianpaolo Cugola and Alessandro Margara. Complex event processing with T-REX. *J. Syst. Softw.*, 85(8):1709–1728, August 2012.

[50] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Zohar Manna. Lola: runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, pages 166–174, June 2005.

[51] Luca de Alfaro. Temporal logics for the specification of performance and reliability. In *Proceedings of STACS'97*, volume 1200 of *LNCS*, pages 165–176. Springer, 1997.

[52] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[53] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[54] Stéphane Demri and Deepak D'Souza. An automata-theoretic approach to constraint LTL. *Inf. Comput.*, 205(3):380–415, March 2007.

[55] Wei Dou, Domenico Bianculli, and Lionel C. Briand. OCLR: A more expressive, pattern-based temporal extension of OCL. In *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, pages 51–66, 2014.

[56] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27:758–771, 1980.

[57] Deepak D'Souza and Pavithra Prabhakar. On the expressiveness of MTL in the pointwise and continuous semantics. *International Journal on Software Tools for Technology Transfer*, 9(1):1–4, 2007.

[58] Xiaoning Du, Yang Liu, and Alwen Tiu. Trace-length independent runtime monitoring of quantitative policies in ltl. In *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 231–247. Springer, 2015.

[59] S. Dustdar, Y. Guo, B. Satzger, and Hong-Linh Truong. Principles of elastic processes. *IEEE Internet Computing*, 15(5):66–71, Sept 2011.

[60] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of FMSP '98*, pages 7–15. ACM, 1998.

[61] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[62] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.

[63] Peter Faymonville, Bernd Finkbeiner, and Doron Peled. Monitoring parametric temporal logic. In *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *LNSC*, pages 357–375. Springer, 2014.

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

[65] Antonio Filieri. *Model based verification and adaptation of software systems@ runtime*. PhD thesis, Politecnico di Milano, 2013. PhD Thesis.

[66] Bernd Finkbeiner, Sriram Sankaranarayanan, and HennyB. Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27:253–274, 2005.

[67] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, March 2004.

[68] Carlo A Furia and Paola Spoletini. Mtl satisfiability over the integers. Technical report, Technical Report 2008.2, DEI, Politecnico di Milano, 2008.

[69] Alessio Gambi, Antonio Filieri, and Schahram Dustdar. Iterative test suites refinement for elastic computing systems. In *Proceedings of ESEC/SIGSOFT FSE 2013*, pages 635–638. ACM, 2013.

[70] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Automated testing of cloud-based elastic systems with AUToCLES. In *Proceedings of ASE 2013*, pages 714–717. IEEE, 2013.

[71] Giovanni Paolo Gibilisco and Srđan Krstić. InstaCluster: Building A Big Data Cluster in Minutes. *CoRR*, abs/1508.04973, 2015.

[72] Object Management Group. Object Constraint Language. `http://www.omg.org/spec/OCL/ISO/19507/`, Last accessed on October 2015.

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

[74] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.

[75] Thomas A. Henzinger. From boolean to quantitative notions of correctness. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 157–158, 2010.

[76] Thomas A. Henzinger. Quantitative reactive modeling and verification. *Computer Science - Research and Development*, 28(4):331–344, 2013.

[77] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of ICAC 2013*, pages 23–27. USENIX, 2013.

[78] Hsi-Ming Ho, Joel Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In *Runtime Verification*, volume 8734 of *LNCS*, pages 178–192. Springer, 2014.

[79] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of ICPE 2012*, pages 85–96. ACM, 2012.

[80] Hans W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, USA, 1968.

[81] Roland Kindermann, Tommi A. Junttila, and Ilkka Niemelä. Bounded model checking of an MITL fragment for timed automata. *CoRR*, abs/1304.7209, 2013.

[82] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of ICSE '05*, pages 372–381. ACM, 2005.

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

[84] Srđan Krstić. MTL-MapReduce. `https://bitbucket.org/krle/mtlmapreduce`.

[85] Srđan Krstić. Verification of quantitative properties of service-based applications. Master's thesis, Politecnico di Milano, December 2012.

[86] Srđan Krstić. Quantitative properties of software systems: Specification, verification, and synthesis. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Doctoral Symposium*, pages 1–4. ACM, June 2014.

[87] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for UTVPI constraints. In *Frontiers of Combining Systems*, pages 168–183, 2005.

Bibliography

[88] P. Leitner, W. Hummer, and S. Dustdar. A Monitoring Data Set for Evaluating QoS-Aware Service-Based Systems. In *Proceedings of PESOS 2012*, pages 67–68, 2012.

[89] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proceedings of SPIN 1999*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

[90] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[91] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In *Proceedings of Logics of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, 1985.

[92] Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister. Parallelized runtime verification of first-order LTL specifications, 2014. Technical report.

[93] Peter Mell and Timothy Grace. The NIST Definition of Cloud Computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, September 2011. NIST Special Publication 800-145.

[94] Claudio Menghi. *Dealing with Incompleteness in Automata-based Model Checking*. PhD thesis, Politecnico di Milano, 2015. PhD Thesis.

[95] Bertrand Meyer. *Agile! - The Good, the Hype and the Ugly*. Springer, 2014.

[96] Angelo Morzenti, Dino Mandrioli, and Carlo Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14:521–573, October 1992.

[97] Aouatef Mrad, Samatar Ahmed, Sylvain Hallé, and Èric Beaudet. Babeltrace: A collection of transducers for trace validation. In *Proceedings of RV 2012*, volume 7687 of *LNCS*, pages 126–130. Springer, 2013.

[98] Syed Naqvi and Philippe Massonet. RESERVOIR - a european cloud computing project. *ERCIM News*, 2010(83):35, 2010.

[99] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[100] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[101] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[102] Oracle. SOA Demo: Order Booking. `http://docs.oracle.com/cd/E28271_01/dev.1111/e10224/fod_hi_level_fod.htm#CIHDGIED`, Last accessed on October 2015.

[103] Oracle. SOA Suite. `http://www.oracle.com/us/products/middleware/soa/suite/overview/index.html`, Last accessed on October 2015.

[104] Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *Proceedings of ESEC-FSE '07*, pages 312–320. ACM, 2007.

[105] Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. Bounded satisfiability checking of metric temporal logic specifications. *ACM Trans. Softw. Eng. Methodol.*, 22(3):20:1–20:54, July 2013.

[106] V. R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

[107] G. Regis, R. Degiovanni, N. D'Ippolito, and N. Aguirre. Specifying event-based systems with a counting fluent temporal logic. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 733–743, May 2015.

[108] Grigore Rosu and Feng Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1), 2012.

[109] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE 1987, pages 328–338. IEEE, 1987.

[110] Gwen Salaün. Analysis and verification of service interaction protocols - a brief survey. In *Proceedings of TAV-WEB 2010*, volume 35 of *EPTCS*, pages 75–86, 2010.

[111] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[112] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226. ACM, 1978.

[113] Giordano Tamburrelli. *Specification and Verification of Quality of Service in Open-World Systems*. PhD thesis, Politecnico di Milano, 2010. PhD Thesis.

[114] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electr. Notes Theor. Comput. Sci*, 113:145–162, 2005.

[115] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 1–22, 1989.

[116] B.F.; van Dongen. BPI challenge 2012, 2012.

[117] H.M.W. Verbeek, JoosC.A.M. Buijs, BoudewijnF. Dongen, and WilM.P. Aalst. XES, XESame, and ProM 6. In *Proceedings CAISE 2010*, volume 72 of *LNBIP*, pages 60–75. Springer, 2011.

[118] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[119] Bozena Wozna-Szczesniak and Andrzej Zbrzezny. Checking MTL properties of discrete timed automata via bounded model checking. In *CS&P*, volume 1032, pages 469–477. CEUR-WS.org, 2013.

[120] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12. USENIX Association, 2012.

[121] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of HotCloud 2010*. USENIX, 2010.