FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Monitoring of data-centric business rules and processes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Rahmen des Erasmus Mundus Studiums

Computational logic

eingereicht von

Gil Vegliach
Matrikelnummer 1126067

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Prof. Dr. Helmut Veith
Mitwirkung: Dr. Andreas Bauer

Wien, 07.02.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Monitoring of data-centric business rules and processes

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.)

in

Computational logic

by

Gil Vegliach
Registration Number 1126067

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:	Prof. Dr. Helmut Veith
External advisor:	Dr. Andreas Bauer

Vienna, 07.02.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gil Vegliach
Viale Ippodromo 2/1, 34139, Trieste

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I am sincerely and heartily grateful to my Nicta advisor Andreas Bauer and the PhD student Jan-Christoph Küster for all the ideas, discussions, insight, and research experience they have imbued me with throughout the writing of my thesis. I would like to express my gratitude to my TUW advisor Helmut Veith for his availability and his time. I also wish to thank my family, my girlfriend and all my friends for the support they provided at Nicta, through my studies and in my personal life.

Gil Vegliach

Abstract

A first-order temporal logic is introduced and argued to be suitable for modelling business processes and security policies for Android. The monitoring problem in such logic is introduced and argued to be undecidable under specific and reasonable assumptions underlying any proper monitor. Although no one can hope for a complete monitoring algorithm, a construction based on a novel automata model is depicted and its correctness demonstrated. Concrete examples taking root in the business process and Android framework are laid out and shown in detail. A digression on the model checking problem displays the difference of our approach from propositional LTL and other work.

Kurzfassung

In dieser Arbeit wird eine temporale Logik erster Ordnung eingeführt und argumentiert, dass diese als geeignet für die Laufzeitverifikation von Geschäftsprozessen und Android Sicherheitsrichtlinien ist. Desweiteren wird das “Monitoring Problem” in einer solchen Logik vorgestellt und dessen Unentscheidbarkeit unter angemessenenen Annahmen, was ein geeigneter Monitor ist, gezeigt. Obwohl kein vollständiger Algorithmus zur Lösung dieses Problems existieren kann, wird eine Konstruktion auf einem neuartigen Automaten Modell basierend dargestellt und dessen Richtigkeit bewiesen. Konkrete Beispiele für Geschäftsprozess- und Android Szenarien werden im Detail erläutert und diskutiert. Ein Exkurs zum “Model Checking Problem” zeigt den Unterschied unseres Ansatzes zu propositionalem LTL und andere Arbeiten.

Contents

Introduction

Motivation and aim of the work

The Provably Correct Business Rules and Processes project at Nicta [1, 3, 2] develops techniques to formally model business rules and processes in order to facilitate automated reasoning about such artefacts. Unlike related approaches, PCBRP emphasises data in rules and processes, making it possible not only to express operations on data, but also to reason about them. The formal foundations are laid upon a first-order CTL* logic whose domain values are represented by generic JSON objects (Java Script Object Notation). Processes, in turn, are modelled as guarded, labelled transition systems whose states define operations on data objects. Reasoning tasks supported by PCBRP included model checking of process models against business rules (i.e. FO-CTL* specifications), constructing processes from process fragments (i.e. temporal planning), and theorem proving. The last is undertaken in combination with a natural language front-end that allows users to query data using a dialect of structured natural language.

What had been missing was a formalism to monitor runtime assertions on concrete execution paths of business processes. Such assertions describe business artefacts' proprieties that processes have to maintain in order to retain compliance, a key issue in BPM. Techniques implemented before this thesis were meant to be more a design-time tool; heavily relying on model checking and theorem proving, they were not adapt to monitoring.

Another source of motivation for this thesis comes from previous work transporting runtime verification onto the Android platform [13]. Android [7] is an open-source software-stack for mobile devices, initially developed by Android Inc. and later purchased by Google in 2005. Made up of utilities, some connecting middleware and an entire operating system running a linux 2.6 kernel, Android is the open-source alternative to other glamour platforms based on iOS and Symbian OS. Java applications, or in jargon "apps", are distributed through an online marketplace called Google Play [5], open to everybody previous registration, with the option of placing a small fee per download or monetize the software through adverts included in the apps. Easiness of development and a large market share attracted many hackers to code malicious software (e.g. AndroidOS.FakePlayer [8]), and consequently defensive measures to counter-attack the spread of malware were explored [19, 20, 16].

The previous paper [13] showed that LTL is a suitable framework for monitoring on Android, but it fundamentally lacks expressiveness in dealing with properties handling complex data, key point in fine security policies. This thesis extends that work from the theoretical point of view, developing tools for a new, more powerful, future implementation.

Structure of the work

The thesis is laid out as follows: in Chapter 1 the general idea is gently introduced, the methodology explained and compared with other existing works. Chapter 2 sets basic definitions and notations, syntax and semantics of $\mathrm{LTL}^{\forall}$. Chapter 3 proves useful theoretical properties of $\mathrm{LTL}^{\forall}$, leading later to undecidability of monitoring. Chapter 4 depicts a new suitable model of automata and shows its correctness. Chapter 5 finally deals with the monitoring problem, demonstrates its theoretical insolubility yet devises a practical algorithm. Chapter 6 reduces $\mathrm{LTL}^{\forall}$ model checking to LTL model checking. Chapter 7 gives practical examples both in the business processes and in the Android context. Lastly, Chapter 8 wraps up obtained results and discusses future work.

CHAPTER 1

Monitoring in brief

This chapter defines the idea of monitoring and sets the context of our work. We will define our concept of monitoring, its underlying assumptions, and compare related work.

1.1 The idea

Monitoring a system means to watch a system for violations or satisfactions of predetermined policies. A system is anything with an observable runtime behaviour, systematically channelled in a long series of seen events, or, in jargon, the trace. The trace is thus evaluated by a monitor, an algorithm encoding the gist of a policy, that, processing event by event, determines the satisfaction or violation of the policy itself; should neither of these cases happen, the algorithm keeps running. Policies, initial specifications and starting block in building of a monitor, are specified in a formal language, a machine processable and precise idiom capable of expressing temporal patterns such as "some event cannot happen until some other event" or "this event has never to happen".

Figure 1.1 illustrates all the elements above-mentioned plus some details pertinent only to our approach. For example, the monitor exploits an automaton, a newly invented special kind turned into an algorithm that encapsulates the meaning of a policy. The system has a logical inner stratum called evaluation, place accessible from the monitor where all built-in and user-defined functions are evaluated. Included are common arithmetic operations such as additions and exponentiation, string manipulations such as concatenation and string-number conversions, and all user defined functions along with functions' domains. The monitor can use functions as it likes, but it cannot extract and store information such as functions' domains (longer arrow in "reading", Figure 1.1). The evaluation had to be placed in the system itself instead of say in the monitor because of a twofold argument: first, practically speaking it is being performed inside the system, therefore moving it outside would be just unnecessary and ultimately slower; second, domains and consequently functions are infinite entities and therefore not suitable to be inputs of the theoretical monitoring problem, a decision problem.

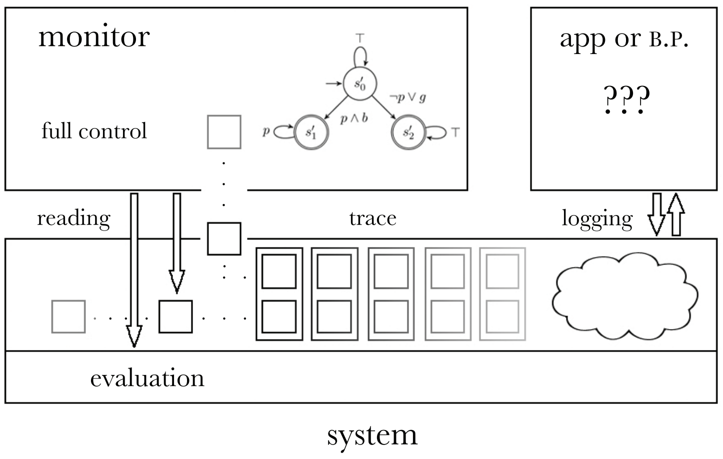

Figure 1.1: monitoring

A similar problem concerns the system status: as a monitor need evaluate only some of infinitely many system functions, it also need consider only some of possibly too many occurred events. The abstraction letting this happen is, in fact, the trace, whose events are further split in two mini-events ("trace", Figure 1.1): the first part is what our monitor can manipulate fully, shown as a square entering the monitor's body ("full control", Figure 1.1), e.g. the mobile number of a just sent text; the second part is what our monitor can test for truth but not save or manipulate directly (shorter arrow in "reading", Figure 1.1), e.g. the infinite predicate "multiple of 3" and the question "is this number a multiple of 3?". Both event's parts are evaluated using the evaluation, the monitor being syntactical in its nature and the meaning being given by the system.

The last section of the diagram is the logging of events and building of the trace ("logging", Figure 1.1). An example of such a design was already given in the Android context in [13] and here is presented another one in the business rules framework. However, this part is interesting mostly from the point of view of the implementation and only marginally mentioned here: as long as the system provides our monitor with faithful information about an application or business process, our monitor behaves correctly. Then the question whether the information really models the application's behaviour really boils down to the expertise of the designer who structures the logging system.

1.2 Approach and problem statement

This sections describes sketchily our approach and our designing choices, discussion carried out throughout the whole thesis.

The technique here exploited is that of runtime verification in a suitable first-order linear temporal logic. Runtime verification is a dynamic verification technique where only one system run is tested against a specification, often during execution and at a low cost. In this respect, it differs from static techniques, such as theorem proving and model checking [10], because no prior knowledge of the system nor a down-scaled model are needed, its general speed is usually higher, although certain mathematical adherence to policies cannot be guaranteed. Runtime verification is therefore often thought as a complimentary approach to static verification, used when others are inapplicable because of size or speed or when an very high level of reliability is desired but not yet achieved, for example model checking has been used with abstraction to simplify the model.

Linear temporal logic (LTL) has been introduced by Pnueli [30] as a specification language for reactive and concurrent systems, and become ever since the de facto standard both in static verification techniques and lightweight runtime formalisms [12]. It lets users specify temporal patterns of events a system must compel to, or some sort of recovery action will be triggered, possibly preventing further damage.

It has extensively been studied [27,32,15] although it lacks expressiveness about fine-grained details of its policies, the ability to state properties among data in formulae, an ability only a first-order formalism may grant (cf. [12]). One example is [13] where in the context of Android apps, propositional LTL could not express the simple property "the phone number of every outgoing SMS has to be in the users contact list".

This thesis wants to fill this gap fulfilling the following requirements:

- the monitor must be online
- the monitor must has a monotonic semantics
- the monitor must have an as-early-as-possible semantics
- the monitor must be able to handle data (first-order)
- the monitor must fit in the Provably Correct Business Rules and Processes NICTA's project
- the monitor must be an extension to previous work set in [13]

The requirements define our concept of a *real monitor* and we are going to briefly introduce them. A monitor is online if it is able to process a trace on-the-fly, event by event; conversely, it is called offline if it need record the whole trace. As both business processes and Android apps are very dynamical entities, reporting a problem once a run has been terminated and damage has been caused could be too late for recovery: just think about products delivered to the wrong address and the relative loss of money, or a virus which deletes sensitive information from a mobile.

The semantics is said to be monotonic if whenever a value is returned it is never to be changed, otherwise it is said non-monotonic. There are plenty of respectable examples of non-monotonic semantics, for example using stationary traces [32] i.e. the last event is endlessly repeated, or just adapting LTL semantics to finite words [27]: to see the problem consider a formula $\Diamond p$ and suppose no p has been seen so far. The monitor would thus return $\bot$ but this value will change into $\top$ as soon as a p turns up.

The semantics is said as-early-as-possible if the monitor returns $\top$ or $\bot$ as soon as it can: for instance if the formula to be monitored is $\bigcirc \bigcirc \bot$, the value $\bot$ is returned at the first event. We will see that adding this requirement makes the theoretical problem undecidable, therefore any practical correct implementation cannot fulfil it.

The monitor must also be able to handle data, otherwise many techniques above mentioned could be used: furthermore data is involved in the other two projects in the requirements.

1.3 Related work

Born to guide model checkers in Java programs [23], runtime verification has evolved over the years into its own broad field within the formal methods community. Pioneer work has been set out by Klaus Havelund[1] and Grigore Rosu[2] in [24, 26, 27, 32, 23], and earlier similar approaches had been tested by Vardi and Wolper [35].

This section is a quick review of other approaches along with comparisons to our work. Each paragraph describes one approach that is relevant to the context and it is independent from the others.

Klaus Havelund and Grigore Rosu in [24] (also cf. [25]) monitored Java program executions through term rewriting rules implemented in Maude, a system and reflective language for term rewriting and equational logic developed by University of Illinois at Urbana Campaign [4]. The logic used is future LTL with finite trace semantics in which the last event is endlessly repeated to fill up an infinite trace. Monitoring is performed generating events from special instrumented bytecode derived from specifications (Jtrek), that also define rules to rewrite the LTL formula. The approach is basically what we called progression in [13] and has the same advantages and disadvantages: in addition, being propositional in nature, it lacks the ability to express relations among atoms, and therefore to model complex data terms.

The same two authors kept working in the same framework on a past temporal logic, still using finite trace semantics and considering only safety properties [27]. They therein argue that past LTL is "more convenient for specifying certain properties", since past LTL is usually more succinct than its future counterpart although they have been shown expressiveness equivalent by Gabbay [21]; also, for convenience of notation, they specify special (redundant) monitoring operators taken from [28]. Two monitoring algorithms are provided, the former being an *in*line rewriting-based module employing Maude, similar to their previous work, the latter being an offline "synthesising software" algorithm, basically using dynamic programming to remember semantics of all subformulae, event by event while the trace is processed forwardly. The latter algorithm is exceptionally fast, $\mathcal{O}(|\varphi| \cdot |u|)$ to process the whole trace u and the formula φ; however, we want to point out it is intrinsically different from our approach because it is propositional, it monitors only safety properties and not monitorable properties, and it suffers from non-monotonicity (events cannot be added in general to a finite trace without violating

[1]NASA Ames Research Center

[2]Department of Computer Science, University of Illinois at Urbana-Champain

semantics, e.g. $\Diamond p$ is $\bot$ in a finite trace u without p's, but in our case it would be just ?, as some p could be seen later).

A work similar to ours is by Hallé and Villemaire [22]. The authors describe a first-order logic, LTL-FO$^+$, suitable to model message-based workflows, structures where input and output are messages composed of data elements, such as business processes, XML-based web service interactions and programming language method calls. Three concrete examples are given: a holiday location finder, a user-controlled lightpaths scenario [17], a car rental system and related properties specified. Their logic allows quantification over data fields but it is only a subset of ours: in particular it lacks the ability to handle complex data terms, atoms different from equalities between variables and/or constants, and evaluation of infinite predicates by a system (what will be called R-atoms). The monitoring algorithm described in Section 4 introduces the concept of a watcher, a finite-state automaton with an outcome, similar to [15], reading event after event and evaluating the prefix semantics of a formula. The evaluation is split in a now-part and a next-part, the now-part evaluated by a tableaux procedure called *spawn* (different from our usage of spawning automata), and the next-part passed over, remembering all runs so far visited and accepting in case one run accepts, rejecting if all runs rejects, otherwise an inconclusive result is returned. Although the many similarities, Hallé and Villemaire's approach still suffers from the problems of progression: as the automaton is not computed in advance, semantic properties of the formulae cannot be used and, for example, $\bigcirc \bigcirc \bot$ cannot be told to be unsatisfiable from the beginning, whereas in [15] the monitor would be only a $\bot$-trap. The authors also point out in Section 4.2 that a discussion on finite trace semantics of LTL-FO$^+$ could be adapted from [14], and here we would like to carry out such a discussion in detail, since our logic is just a superset of LTL-FO$^+$.

The main paper our work is based on is by Bauer, Leucker and Schallhart [15]. The authors describe two logics, future propositional LTL and a timed counterpart, distinguishing runtime verification from model checking and monitoring. The concept of online monitoring is introduced (a monitor running along with the system and watching over it), the 3-value semantics LTL$_3$ defined (true, false, inconclusive), an as-early-as-possible minimal automata-based online monitoring algorithm constructed and its complexity studied. Our work intends to extend [15] to first order logic setting. An initial attempt has been tried in [13] where the framework is tested on the Android platform as an anti-malware mechanism: a minimal logging component injected in the Android framework generates the trace monitored by a top-stack Java application using progression as monitoring algorithm. Although a sound proof of concept, progression cannot use structure in formulae, for instance cannot detect that $\bigcirc \bigcirc \bot$ is unsatisfiable before processing three events. Therefore the motivation for trying the automata-based approach, along with an in-depth study of the logical foundations that clarifies formal properties such as quantification over the trace linked to finite models and undecidability.

CHAPTER 2

Notations and definitions

This chapter sets out basic definitions and notations in full detail. Most of the material is intended as a reference, steering the reader through the following chapters. Most of the material is also similar to what already found in the literature (cf. [12]), but the keen reader would note the differences in the universal quantifier's syntax and semantics: such small details will lead to plenty of interesting properties, both from the theoretical logic aspect and from the rather practical monitoring point of view. Quantification over a finite event in a trace is indeed a well thought out design choice, as it avoids problems looping over infinite structures, as well as a theoretical choice aimed at enforcing important properties.

2.1 Syntax

Definition 1 (Signature)**.** *Given a countably infinite set of variables V, a signature Γ is a triple (C, F, Q), where C is countable set of constant symbols, F is a countable set of function symbols, and Q is a countable set of relation symbols, union of two disjoint sets P and R. Variables will be typically denoted by letters $x_1, x_2, \ldots$ or $x, y, z, \ldots$.*

Relation symbols $p \in P$ will be allowed syntactically in quantifiers and atoms, and semantically interpreted to finite relations, whereas relation symbols $r \in R$ will be allowed syntactically only in atoms, and will not have any restriction on their sematical interpretation.

Definition 2 (Terms)**.** *Let Γ be a signature (C, F, Q), let V be a countably infinite set of variables and let Σ be the alphabet $C \cup F \cup \{(\} \cup \{)\} \cup \{,\}$. The set of terms over Γ is the smallest subset T of Σ^* verifying the following conditions:*

- $C \subset T$
- $V \subset T$

- *if $f \in F$, f has arity n, and $t_1, \ldots, t_n \in T$, then $f(t_1, \ldots, t_n) \in T$.*

Definition 3 (Formulae)**.** *Let Γ be a signature, T be the set of terms over Γ, and Σ be the alphabet $T \cup Q \cup \{\wedge, \neg, \mathbf{U}, \circ, \forall, (,), :, ., \} \cup \{, \}$. The set of temporal formulae over Γ, denoted by LTL$^\forall$, is the smallest subset F of Σ^* verifying the following conditions:*

- *if t_1, t_2 are terms, then $t_1 = t_2 \in F$*
- *if $q \in Q$ and $t_1, \ldots, t_n \in T$, then $q(t_1, \ldots, t_n) \in F$*
- *if $\varphi \in F$, then $\neg\varphi \in F$.*
- *if $\varphi_1, \varphi_2 \in F$, then $(\varphi_1 \wedge \varphi_2) \in F$*
- *if $\varphi \in F$, $p \in P$ and p has arity n, then $\forall(x_1, \ldots, x_n){:}p.\ \varphi \in F$*
- *if $\varphi \in F$ then $\bigcirc\varphi \in F$*
- *if $\varphi_1, \varphi_2 \in F$, then $\varphi_1 \mathbf{U} \varphi_2 \in F$.*

Syntactic sugar for $\vee, \Rightarrow, \Diamond, \Box$ is defined as follows:

$$\begin{array}{rcl}
\exists x{:}p.\ \psi & = & \neg\forall x{:}p.\ \neg\psi \\
\varphi_1 \vee \varphi_2 & = & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \Rightarrow \varphi_2 & = & \neg(\varphi_1 \wedge \neg\varphi_2) \\
\Diamond\varphi & = & \top\mathbf{U}\varphi \\
\Box\varphi & = & \neg(\top\mathbf{U}\neg\varphi)
\end{array}$$

The notation $\vec{\cdot}$ is used for tuples of objects, should they be variables, constants or complex terms, where the arity is inferred from the context: an example is $\forall\vec{x}{:}p.\ \psi$ in quantification. The logic just defined is denoted by LTL$^\forall$, while standard first-order logic is denoted by FO.

Definition 4 (Free variables, sentences)**.** *For a given formula φ, an occurrence of a variable x is bound by a predicate p if there is a subformula of φ of the form either $\forall x{:}p.\ \psi$ or $\exists x{:}p.\ \psi$ such that the occurrence after the quantifier is exactly the occurrence we are defining. A unbound occurrence is a free occurrence. Note that a variable name can occur more than once in a formula, sometimes both bound and free. Nevertheless variables can be renamed in a way no variable appears twice without affecting the semantics: it is therefore appropriate to call a variable bound or free rather than its occurrence. A formula without free variables is a closed formula or sentence. The notation $t(\vec{x})$, respectively $\varphi(\vec{x})$ indicates a term t whose free variables are among $\vec{x}$, respectively a formula φ whose free variables are among $\vec{x}$.*

Definition 5 (Ground atoms)**.** *A term without variables is a ground term, an atom without (free) variables is a ground atom. For a given signature Γ, the set of ground atoms from P is $\{p(\vec{t}) : p \in P, p(\vec{t})$ is ground$\}$ and it is denoted by Π. Informally a ground atom $p(t)$ with $p \in P$ is called a P-atom.*

2.2 Semantics

A formula is just a formal language statement about an entity and a statement, in its purest nature, has a meaning. Being a formal framework, entities under considerations are rather simple, sets and their elements, and *meaning* actually boils down to truth, a dichotomy between true and false statements. A formula, therefore, expresses either a true or false assertion about some element of a set, or the set itself.

In our framework time is also considered: properties are true or false not only depending on the picked element and underlying domain, but also depending on the time when they are considered. Formulae can express complex temporal relations such as properties lasting for ever or just until some other one becomes true.

The link between bare entities and formulae is given by a (temporal) structure interpreting syntactical symbols to elements of a set called domain. The structure takes care of the temporal part of the formula through a changing part called trace, while static elements such as functions and constants are under the scope of an evaluation, concept we are going to define next.

Definition 6 (Evaluation). *Let Γ be a signature (C, F, Q). An evaluation $\mathcal{E}$ of the signature Γ is a triple $(A, \{c^{\mathcal{E}}\}_{c \in C}, \{f^{\mathcal{E}}\}_{f \in F})$ where:*

- *A is a non-empty countable set called domain*
- *$\mathcal{E}$ assigns each c in C to an element $c^{\mathcal{E}}$ of A*
- *$\mathcal{E}$ assigns each f in F to a function $f^{\mathcal{E}} \colon A^n \to A$, where n is the arity of f*
- *for each element $d \in A$ there is at least a term evaluated to it, i.e. $\mathcal{E}^{-1}(d) \neq \emptyset$ (surjectivity of evaluation)*

The set of all possible tuples from A named from R is $\{r(\vec{d}) : r \in R, \vec{d}$ is a tuple from A with the same arity as r's$\}$, and it is denoted by Ψ_A or just Ψ where the domain is clear from the context. Informally, an object $r(\vec{d})$, where $\vec{d}$ is a tuple from the domain and $r \in R$ is called an R-atom. An evaluation of a particular signature is called just an evaluation when the signature is clear from the context.

Definition 7 (Evaluation of terms). *A term $t(\vec{x})$ can be recursively evaluated under an evaluation $\mathcal{E}$ at a tuple $\vec{a}$ from A as follows:*

- *if $t(\vec{x})$ is a constant symbol c, then $t^{\mathcal{E}}(\vec{a}) = c^{\mathcal{E}}$*
- *if $t(x_1, \ldots, x_n)$ is a variable symbol x_i, then $t^{\mathcal{E}}(\vec{a}) = a_i$*
- *if $t(\vec{x})$ is of the form $f(t_1, \ldots, t_n)$, then $t^{\mathcal{E}}(\vec{a}) = f^{\mathcal{E}}(t_1^{\mathcal{E}}(\vec{a}), \ldots, t_n^{\mathcal{E}}(\vec{a}))$*

Definition 8 (Temporal structure). *Let Γ be a signature (C, F, Q). A temporal Γ-structure $\mathfrak{A}$ is a pair $(\mathcal{E}, w)$ where:*

- *$\mathcal{E}$ is an evaluation*

- *w is an infinite word $w_0 w_1 w_2 \dots$ over subsets of $\Pi \cup \Psi$ called trace such that:*
  - *$w_i \cap \Pi$ is finite, for all $i \geq 0$ (finitely many p's)*
  - *$p^{\mathfrak{A}_i} = \{t^{\mathcal{E}} : p(t) \in w_i\}$, for all $i \geq 0$ (rigidity of interpretation on p's)*

With a slight abuse of notation, the elements $c^{\mathcal{E}}, f^{\mathcal{E}}$ will be denoted respectively by $c^{\mathfrak{A}}, f^{\mathfrak{A}}$, and the notation $r^{\mathfrak{A}_i}$ will be used for $\{r(\vec{d}) \in w_i\}$. A first-order Γ-structure $\mathcal{A}$ is a pair $(\mathcal{E}, w_0)$. A structure is over an evaluation $\mathcal{E}$ when its evaluation is precisely $\mathcal{E}$.

Note the difference here: P-atoms in the trace are just syntactically ground atoms whereas R-atoms come in already interpreted; nevertheless under the symbol $\mathcal{A}_i$, both P- and R-atoms depict domain subsets/relations.

Definition 9 (Semantics of a formula)**.** *Let $\mathfrak{A}$ be a temporal Γ-structure, $\varphi(\vec{x})$ be a LTL$^{\forall}$ formula, and $\vec{a}$ a tuple from A. The formula $\varphi(\vec{x})$ is true in $\mathfrak{A}$ at the point $\vec{a}$ and instant i with $i \geq 0$, denoted by $\mathfrak{A}, i \models \varphi(\vec{a})$, when the following recursive definition is satisfied:*

$$\begin{array}{ll}
\mathfrak{A}, i \models t_1(\vec{a}) = t_2(\vec{a}) & \textit{if} \;\; t_1^{\mathfrak{A}}(\vec{a}) = t_2^{\mathfrak{A}}(\vec{a}) \\
\mathfrak{A}, i \models q(t(\vec{a})) & \textit{if} \;\; t^{\mathfrak{A}}(\vec{a}) \in q^{\mathfrak{A}_i} \\
\mathfrak{A}, i \models \neg\varphi(\vec{a}) & \textit{if} \;\; \mathfrak{A}, i \not\models \varphi(\vec{a}) \\
\mathfrak{A}, i \models \varphi_1(\vec{a}) \wedge \varphi_2(\vec{a}) & \textit{if} \;\; \mathfrak{A}, i \models \varphi_1(\vec{a}) \textit{ and } \mathfrak{A}, i \models \varphi_2(\vec{a}) \\
\mathfrak{A}, i \models \forall \vec{x}{:}p.\ \psi(\vec{x}, \vec{a}) & \textit{if} \;\; \textit{for all } \vec{d} \in p^{\mathfrak{A}_i} \textit{ it holds } \mathfrak{A}, i \models \psi(\vec{d}, \vec{a}) \\
\mathfrak{A}, i \models \bigcirc\varphi(\vec{a}) & \textit{if} \;\; \mathfrak{A}, i+1 \models \varphi(\vec{a}) \\
\mathfrak{A}, i \models \varphi_1(\vec{a})\mathbf{U}\varphi_2(\vec{a}) & \textit{if} \;\; \textit{there is a } k \textit{ with } k \geq i \textit{ such that } \mathfrak{A}, k \models \varphi_2(\vec{a}) \\
 & \quad\;\; \textit{and for } i \leq j < k \textit{ it holds } \mathfrak{A}, j \models \varphi_1(\vec{a})
\end{array}$$

When $i = 0$ in the above definition the index i is omitted, $\mathfrak{A} \models \varphi(\vec{a})$ is simply written, and $\varphi(\vec{x})$ is said to be satisfiable at $\vec{a}$ over the domain A. The structure $\mathfrak{A}$ is called also a model of φ at $\vec{a}$. For a fixed evaluation $\mathcal{E}$ and a sentence φ, a trace w such that $(\mathcal{E}, w) \models \varphi$ is called a satisfying trace and the set of all satisfying traces is denoted by $\mathcal{L}(\varphi)_{\mathcal{E}}$.

Note the semantics of a sentence does not depend upon the choice of $\vec{a}$.

CHAPTER 3

Properties of LTL$^\forall$

This chapter builds tools to show the undecidability of the theoretical monitoring problem in Chapter 5. The idea is to reduce the satisfying trace existence problem in LTL$^\forall$ to finite satisfiability in first-order logic. For definitions and a discussion of first-order concepts the reader can check [18].

Lemma 10 (Reverse skolemization)**.** *For a given first-order logic formula φ there is a constant-symbol-free and function-symbol-free formula φ' that is finitely equisatisfiable to φ.*

Proof. Let us construct φ' by the following algorithm:

1. Put φ in prenex normal form, call the resulting formula φ_1
2. If $c_1, \ldots, c_n$ are all the constant symbols occurring in φ_1, then call φ_2 the formula $\exists x_1, \ldots, x_n.\varphi_1$, where $x_1, \ldots, x_n$ are new variable names
3. Bottom-up remove function symbols from φ_2: if φ_2 is of the form $Q\vec{x}.Q\vec{z}.\psi(\vec{t_1}(f(\vec{x}), \vec{z}), \vec{t_2}(\vec{x}, \vec{z}))$ then set φ_2 to be $Q\vec{x}.\exists y.Q\vec{z}.\psi(\vec{t}(y, \vec{z}), \vec{t_2}(\vec{x}, \vec{z})) \wedge (f(\vec{x}) = y)$, where y is a new variable symbol. After all changes, call φ_3 the resulting formula.
4. Substitute all equalities of the form $f(\vec{x}) = y$ by $p_f(\vec{x}, y)$, where p_f is a new predicate symbol; conjoin to the resulting formula: $\forall\vec{x}.\exists y.p_f(\vec{x}, y) \wedge \forall\vec{x}.\forall y_1.\forall y_2.((p_f(\vec{x}, y_1) \wedge p_f(\vec{x}, y_2)) \Rightarrow y_1 = y_2)$. Call φ' the resulting formula. (This steps makes predicates p_f behave as functions)

It is clear that steps 1,2,4 do not alter finite equisatisfiability, we show that neither does 3. Step 3 substitutes a function term $f(\vec{x})$ by a new existentially quantified variable y, variable representing the result of the interpreted f evaluated on some tuple of the domain deriving from the quantifiers binding $\vec{x}$. The term $f(\vec{x})$ might possibly be nested down into another term $\vec{t_1}$, hence the notation to maintain full generality. Now, let $\mathcal{A}$

be a finite first-order logic structure, so that $f^{\mathcal{A}}\colon A \times \ldots \times A \to A$ is a function; then $Q\vec{x}.Q\vec{z}.\psi(\vec{t_1}(f(\vec{x}), \vec{z}), \vec{t_2}(\vec{x}, \vec{z}))$ is therein true if and only if:

$$\begin{aligned}
&\textit{for all/there exists } d_1 \in A, \ldots, \textit{ for all/there exists } d_n \in A,\\
&\qquad\qquad \mathcal{A} \models Q\vec{z}.\psi(\vec{t_1}(f(\vec{d}), \vec{z}), \vec{t_2}(\vec{d}, \vec{z}))\\
\iff &\textit{for all/there exists } d_1 \in A, \ldots, \textit{ for all/there exists } d_n \in A, \text{ there exists } d' \in A,\\
&\qquad\qquad \mathcal{A} \models Q\vec{z}.\psi(\vec{t_1}(d', \vec{z}), \vec{t_2}(\vec{d}, \vec{z})) \wedge (f(\vec{d}) = d')\\
\iff &\textit{for all/there exists } d_1 \in A, \ldots, \textit{ for all/there exists } d_n \in A,\\
&\qquad\qquad \mathcal{A} \models \exists y.Q\vec{z}.\psi(\vec{t_1}(y, \vec{z}), \vec{t_2}(\vec{d}, \vec{z})) \wedge (f(\vec{d}) = y)
\end{aligned}$$

and the last line is true if and only if $\mathcal{A} \models Q\vec{x}.\exists y.Q\vec{z}.\psi(\vec{t_1}(y, \vec{z}), \vec{t_2}(\vec{d}, \vec{z})) \wedge (f(\vec{x}) = y)$. In every finite structure the two formulae are equivalent and hence finite equisatisfiable. The thesis follows. $\square$

Theorem 11 (FO finite sat is reducible to trace existence)**.** *Let Γ be a signature $(C, F, P \cup Q)$, $\mathcal{E}$ an evaluation of Γ with an infinite domain, and φ a first-order logic formula (from another first-order signature). Then whenever $|P| \gg |\varphi|$ there is a (constructible) formula $\psi \in LTL^{\forall}$ such that φ is finitely satisfiable in first-order logic if and only if ψ has a satisfying trace.*

Proof. Let us construct the formula ψ by the following algorithm:

1. Reverse skolemization: apply to φ the algorithm of Theorem 10 and call the resulting formula φ_1

2. Substitution of quantifiers: bottom-up replace each subformula of the form $\forall x.\phi$ by $\forall x\!:\!d.\ \phi$ and each subformula of the form $\exists x.\phi$ by $\exists x\!:\!d.\ \phi$. Call φ_2 the resulting formula

3. Restriction of interpretation:

   a) for each predicate p of arity n appearing in the so-far-obtained formula, conjoin $\forall(x_1, \ldots, x_n)\!:\!p_i.\ d(x_1) \wedge \ldots \wedge d(x_n)$

   b) conjoin $\exists x\!:\!d.\ d(x)$, to guarantee non-emptiness of domain

   Call the resulting formula ψ.

It is sufficient to show that φ_1 is finitely satisfiable if and only if ψ has a satisfying trace. Suppose that there is a finite model (D, I) for φ_1. Set the domain for $\mathcal{E}$ to be D and note there are no constant symbols nor function symbols to evaluate. Define the one-world trace σ to be

$$\left(\bigcup_{\vec{t}^{\mathcal{E}} \in p^I, p \text{ in } \varphi_1} p(\vec{t})\right) \cup \left(\bigcup_{t^{\mathcal{E}} \in D} d(t)\right) \cup \left(\bigcup_{\vec{t}^{\mathcal{E}} \in p^I, p \text{ in } \varphi_1} \{d(t_1), \ldots, d(t_n)\}\right)$$

It is easily verified that $(\mathcal{E}, \sigma) \models \psi$: the trace basically contains a copy of the interpretation I, a copy of the domain D disguised as the predicate d, and forces all predicates from φ_1 in the predicated d.

Vice versa, suppose there is a one-world trace σ over $\mathcal{E}$ satisfying ψ and call $\mathcal{A}$ the structure $(\mathcal{E}, \sigma)$. The model (D, I) for φ_1 is defined as follows:

- the domain D is set to $d^{\mathcal{A}}$
- the interpretation of predicates is the same: $p^I = p^{\mathcal{A}}$

Such a construction is easily seen to be a model. $\square$

CHAPTER 4

Spawning automata

As we are going to see in the next chapter, the results in Chapter 3 will soon forbid the construction of general early-as-possible monitors, yet a correct algorithm is still achievable. The construction begins in this chapter with a new kind of automata paralleling [15], an automata based algorithm devising a finite state machine from two Büchi automata, one from the formula itself and one from its negation. Since the logic is now more sophisticated, the original Büchi automata model [35, 34] is no longer adequate and care need be taken to handle domains, term evaluations, infinite relations. Most importantly the different quantifier semantics makes automata dependant on the runtime trace.

These extensions lead to the idea of spawning automata, new non-deterministic automata that, instantiating new subautomata at runtime, manage to expand quantifiers and handle parametrised temporal subformulae on-the-fly. Spawning automata, similarly to Büchi automata, deal with formulae semantics but still cannot dissect, or look into, first-order subformulae: instead, they delegate the task to new runtime-parametrised subautomata, that in turn repeat the process until the lowest layer is reached. The construction is thus stratified according to the so-called "level" of a automaton, basically the highest number of nested first-order quantifiers in the relative subformula, the lowest corresponding to a regular propositional Büchi automaton. The structure of (sub-) automata depends only on the (sub-) formulae, therefore a blueprint of all subautomata can be built in advance even though the real value of the parameter will be known only during execution.

The goal of this section is defining spawning automata and proving that they correctly act as acceptors for LTL$^{\forall}$. The formal theory will be developed step by step, starting from the abstract definition of these new machines and ending with the concrete construction of an automaton from a LTL$^{\forall}$ formula. The first definition is preliminary and will be used to define the accepting condition of spawning automata: it is a set of boolean negation-free formulae whose atoms are generic elements from some set. Although it is unusual to assign a truth value to bizarre set elements like automata, the construction is well-known in the automata community, see for instance [29].

Definition 12 (Positive boolean formulae)**.** *The set of positive boolean formulae over a set X, denoted by $\mathcal{B}^+(X)$, is the set of formulae generated by the following grammar:*

$$\varphi ::= a \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \top \mid \bot$$

where $a \in X$ and φ_1 and φ_2 are positive boolean formulae over X. A formula $\beta \in \mathcal{B}^+(X)$ is satisfied by a set $Y \subseteq X$, denoted by $Y \models \beta$, if the truth assignment setting all elements of Y to true and all elements of $X - Y$ to false satisfies β.

Next we see the main definition of the section, the spawning automaton's. The definition follows familiar lines with a supplementary function $\delta_\downarrow$, needed for the spawning.

Definition 13 (Büchi spawning automaton)**.** *A non-deterministic spawning automaton $\mathcal{A}$ is a 7-uple $(\Sigma, B, B_0, \delta_\rightarrow, \delta_\downarrow, \mathcal{F}, l)$ where Σ is an infinite set of symbols called the alphabet, B is a finite set of states*[1]*, B_0 is a subset of B called the set of initial states, $\delta_\rightarrow \colon B \times \Sigma \rightarrow 2^B$ is the transition relation, $\delta_\downarrow \colon B \times \Sigma \rightarrow \mathcal{B}^+(\mathcal{A}^{<l})$ is the spawning function, where $\mathcal{A}^{<l}$ denotes the set of all spawning automata whose levels are less than l, $\mathcal{F}$ is a possibly empty finite set of sets, each called acceptance set, and l is an integer number greater than or equal to zero called level.*

The core of the model is the new recursive accepting condition. The function $\delta_\downarrow$ links together the top-level automaton with his lower-level subautomata, using a boolean condition that forces particular subautomata to accept or reject. The quantifier semantics can thus be modelled, being nothing more than a short cut for a runtime conjunction or disjunction. The automaton is non-deterministic, it runs in parallel multiple threads of computation, accepting when one run only is accepted, but rejecting solely when all runs are rejected.

Definition 14 (Accepting run)**.** *A run ρ of a spawning automaton $(\Sigma, B, B_0, \delta_\rightarrow, \delta_\downarrow, \mathcal{F}, l)$ reading $w_0 w_1 \ldots \in \Sigma^\omega$ is an infinite sequence of states $b_0 b_1 \ldots$, starting with an initial state $b_0 \in B_0$, and such that $b_{i+1} \in \delta_\rightarrow(b_i, w_i)$, for all $i \geq 0$. The run ρ is locally accepting if for all $F \in \mathcal{F}$ there are infinitely many indices i such that $b_i \in F$. The definition of accepting run is recursive: when the level is 0, the run ρ is accepting if it is just locally accepting, when the level is greater than 0, the ρ must be locally accepting and for all $i \geq 0$ there must be a set $Y \subseteq \mathcal{A}^{<l}$ such that $Y \models \delta_\downarrow(b_i, w_i)$ and all automata $\mathcal{A}' \in Y$ must have an accepting run reading w^i. When there is an accepting run reading a word w the word itself is called accepted word (by the automaton).*

Now that all tools are ready it is time to build a particular spawning automaton, one that acts as an acceptor for our new logic. Notation and terminology have been taken from [15]. Note that the definitions of the subformulae and closure sets differ from the standard: quantified formulae are not split further in the subformulae set, e.g. $\forall x : p.\ \psi$ is treated here as [15] would treat an atomic proposition. The quantified formulae will be split later in subautomata responsible for the first-order parts of formulae.

[1]usually indicated by Q but here such a letter is used for relation symbols

Definition 15 (Closure set)**.** *Let φ be a sentence opportunely rewritten with only $\neg$, $\wedge$, $\forall$, $\bigcirc$, $\mathbf{U}$; such a formula is called elementarily rewritten. The purely temporal subformula set of φ, or simply subformula set, denoted by $subf(\varphi)$, is the smallest set of formulae X satisfying the following properties:*

- $\varphi \in X$
- $\star\psi \in X$ *implies* $\psi \in X$*, for* $\star \in \{\neg, \bigcirc\}$
- $\psi_1 \circ \psi_2 \in X$ *implies* $\psi_1, \psi_2 \in X$*, for* $\circ \in \{\wedge, \mathbf{U}\}$*.*

The closure set of φ, denoted by $cl(\varphi)$, is the closure of $subf(\varphi)$ under negation, i.e. $cl(\varphi) = subf(\varphi) \cup \{\neg\psi \mid \psi \in subf(\varphi)\}$.

Definition 16 (Elementary set)**.** *Let φ be an elementarily rewritten formula. A set $b \subseteq cl(\varphi)$ is an elementary subset if for all $\psi, \psi_1 \wedge \psi_2, \psi_1\mathbf{U}\psi_2 \in cl(\varphi)$ it holds:*

- $\psi \in b$ *if and only if* $\neg\psi \notin b$
- $\psi_1 \wedge \psi_2 \in b$ *if and only if* $\psi_1, \psi_2 \in b$
- *if* $\top \in cl(\varphi)$*, then* $\top \in b$
- *if* $\psi_2 \in b$*, then* $\psi_1\mathbf{U}\psi_2 \in b$
- *if* $\psi_1\mathbf{U}\psi_2 \in b$ *but* $\psi_2 \notin b$*, then* $\psi_1 \in b$

To handle the first-order part, a fundamental concept to bear in mind is the level of a formula. This number is simply the number of first-order quantifiers in the deepest nesting and will be the maximum depth of our automaton. Each $\delta_\downarrow$ spawns an automaton one level lower than its parent, down to 0 where spawning automata become regular propositional Büchi automata.

Definition 17 (Level of a formula)**.** *Let φ be an elementarily rewritten formula. The level of φ, denoted by $level(\varphi)$, is the number of first-order quantifiers in the deepest nesting, or more formally:*

$$level(\varphi) = \begin{cases} 0 & \text{if } \varphi \text{ is quantifier-free} \\ 1 + level(\forall x{:}p.\ \psi) & \text{if } \varphi \text{ is of the form } \forall x{:}p.\ \psi \\ max(level(\psi_1), level(\psi_2)) & \text{if } \varphi \text{ is of the form } \psi_1 \wedge \psi_2 \text{ or } \psi_1\mathbf{U}\psi_2 \\ level(\psi) & \text{if } \varphi \text{ is of the form } \bigcirc\psi \text{ or } \neg\psi \end{cases}$$

Finally the construction is presented in the form of a constructive definition, easily turnable into an algorithm. While most of the definition is similar to [15], note in particular the definition of $\delta_\downarrow$ and how it mimics the definition of the quantifier semantics.

Definition 18 ($\mathcal{A}_\varphi$)**.** *Let Γ be a signature, $\mathcal{E}$ an evaluation of Γ and φ an elementarily rewritten formula. The symbol $\mathcal{A}_\varphi$ inductively indicates the 7-uple $(\Sigma, B, B_0, \delta_\rightarrow, \delta_\downarrow, \mathcal{F}, l)$ where:*

- $\Sigma = 2^{\Pi \cup \Psi}$
- *B is the set of elementary sets of φ*
- $B_0 = \{b \in B \mid \varphi \in b\}$
- $l = level(\varphi)$
- *$\delta_{\to}$: if the conditions*
  - *for all $q(t) \in cl(\varphi)$, $q(t) \in b$ if and only if $(\mathcal{E}, \sigma) \models q(t)$*
  - *for all $t_1 = t_2 \in cl(\varphi)$, $t_1^{\mathcal{E}} = t_2^{\mathcal{E}}$*

  does not *hold both at the same time, then $\delta_{\to}(b, \sigma) = \emptyset$; otherwise $\delta_{\to}(b, \sigma)$ is the set of sets $b' \in B$ that satisfy the following two conditions:*
  - *for all $\bigcirc\psi \in cl(\varphi)$, it holds $\bigcirc\psi \in b$ if and only if $\psi \in b'$*
  - *for all $\psi_1 \mathbf{U} \psi_2 \in cl(\varphi)$, it holds $\psi_1 \mathbf{U} \psi_2 \in b$ if and only if $\psi_2 \in b$, or $\psi_1 \in b$ and $\psi_1 \mathbf{U} \psi_2 \in b'$*
- *$\delta_{\downarrow}$: if $l = 0$, then $\delta_{\downarrow}(b, \sigma) = \top$, otherwise $\delta_{\downarrow}(b, \sigma)$ is recursively:*

$$\left(\bigwedge_{\forall x:p.\psi \in b} \left(\bigwedge_{p(t) \in \sigma} \mathcal{A}_{\psi(t^{\mathcal{E}})} \right) \right) \wedge \left(\bigwedge_{\neg\forall x:p.\psi \in b} \left(\bigvee_{p(t) \in \sigma} \mathcal{A}_{\neg\psi(t^{\mathcal{E}})} \right) \right)$$

  where the first inner conjunction is meant to be $\top$ if there is no $p(t) \in \sigma$, whereas the disjunction is meant to be $\bot$ under the same condition.

- $\mathcal{F} = \{F_{\psi_1 \mathbf{U} \psi_2} \mid \psi_1 \mathbf{U} \psi_2 \in cl(\varphi)\}$ *where* $F_{\psi_1 \mathbf{U} \psi_2} = \{b \in B \mid \psi_1 \mathbf{U} \psi_2 \in b \Rightarrow \psi_2 \in b\}$

The language accepted by $\mathcal{A}_\varphi$, denoted by $\mathcal{L}(\mathcal{A}_\varphi)_{\mathcal{E}}$ or simply by $\mathcal{L}(\mathcal{A}_\varphi)$, is the subset of Σ^ω consisting of words accepted by $\mathcal{A}_\varphi$.

The following theorem crowns our work with an expected though technical theorem. The proof is a double induction argument following [15], with adjustment on indices. The claims and the structure have been highlighted typographically and linguistically, making it easier to read albeit adding some redundancy and length.

Theorem 19 (Spawning automata are acceptors)**.** *Let Γ be a signature, φ be an elementarily rewritten formula, and $\mathcal{E}$ an evaluation of Γ. Then $\mathcal{L}(\mathcal{A}_\varphi)_{\mathcal{E}} = \mathcal{L}(\varphi)_{\mathcal{E}}$.*

Proof. "$\subseteq$": We need to prove that for all formulae φ, whenever $\mathcal{A}_\varphi$ accepts a word w, then this word itself satisfies φ over $\mathcal{E}$. The proof is by induction on $level(\varphi)$, our claim is:

$$\textit{for all } \varphi \in \mathrm{LTL}^{\forall}.\ \textit{for all } b_0 b_1 \ldots \textit{ accepting run of } \mathcal{A}_\varphi \textit{ reading } w.$$
$$\textit{for all } \psi \in cl(\varphi).\ \textit{for all } i \geq 0.\ \psi \in b_i \iff (\mathcal{E}, w^i) \models \psi$$

Note that the thesis follows because $\varphi \in cl(\varphi)$, and since $w^0 = w$, it holds $(\mathcal{E}, w) \models \varphi$.

$level(\varphi) = 0$: let $b_0 b_1 \ldots$ be an accepting run of $\mathcal{A}_\varphi$ reading w, we need to prove a statement on all formulae in $cl(\varphi)$. We will proceed by structural induction on those formulae, observing that all formulae in $cl(\varphi)$ are quantifier-free and thus it is not necessary to prove the quantifier's case. The base case runs similarly as in [10], with some adjustments on indexes.

• Equality: by definition of $\delta_\rightarrow$.

• Atoms: for an arbitrary $i \geq 0$ and $q(t) \in cl(\varphi)$:

$$\begin{aligned} q(t) \in b_i &\iff (\mathcal{E}, w_i) \models q(t) \qquad \text{by definition of } \delta_\rightarrow \\ &\iff (\mathcal{E}, w^i) \models q(t) \end{aligned}$$

• Negation: for an arbitrary index $i \geq 0$, it holds:

$$\begin{aligned} \neg\psi' \in b_i &\iff \psi' \notin b_i \qquad \text{by definition of elementary set} \\ &\iff (\mathcal{E}, w^i) \not\models \psi' \qquad \text{by (inner) induction hypotheses} \\ &\iff (\mathcal{E}, w^i) \models \neg\psi' \end{aligned}$$

• Conjunction: for an arbitrary index $i \geq 0$, it holds:

$$\begin{aligned} \psi_1 \wedge \psi_2 \in b_i &\iff \psi_1, \psi_2 \in b_i \qquad \text{by definition of elementary set} \\ &\iff (\mathcal{E}, w^i) \models \psi_1 \text{ and } (\mathcal{E}, w^i) \models \psi_2 \qquad \text{by (inner) induction hypothesis} \\ &\iff (\mathcal{E}, w^i) \models \psi_1 \wedge \psi_2 \end{aligned}$$

• Next: for an arbitrary index $i \geq 0$, it holds:

$$\begin{aligned} \bigcirc\psi' \in b_i &\iff \psi' \in b_{i+1} \qquad \text{by definition of } \delta_\rightarrow \\ &\iff (\mathcal{E}, w^{i+1}) \models \psi' \qquad \text{by (inner) induction hypothesis on index } i+1 \\ &\iff (\mathcal{E}, w^i) \models \bigcirc\psi' \end{aligned}$$

Note in this case the induction hypothesis is applied on a different index from the beginning: this and similarly until operator's case are the reasons for the strengthened induction claim on an arbitrary index (wrt. [10]).

• Until: for an arbitrary index $i \geq 0$, the proof is split into an "only-if" and an "if" part.

"$\Rightarrow$": suppose $\psi_1 \mathbf{U} \psi_2 \in b_i$, we will show $(\mathcal{E}, w^i) \models \psi_1 \mathbf{U} \psi_2$. First we show that there is an index $j \geq i$ such that $(\mathcal{E}, w^j) \models \psi_2$. In fact, suppose the contrary, for all $j \geq i$ it holds $(\mathcal{E}, w^j) \not\models \psi_2$; then by (inner) induction hypothesis it follows that for all $j \geq i$ it holds $\psi_2 \notin b_j$. Because $\psi_1 \mathbf{U} \psi_2 \in b_i$ and ψ_2 appears in no state after b_i, it must be that $\psi_1 \mathbf{U} \psi_2 \in b_j$ (and $\psi_1 \in b_j$) for all $j \geq i$, by $\delta_\rightarrow$ definition. On the other hand, $b_0 b_1 \ldots$ is an accepting run, therefore there exist infinitely many $j \geq i$ such that $b_j \in F_{\phi_1 \mathbf{U} \psi_2}$ or equivalently $\phi_1 \mathbf{U} \psi_2 \notin b_j \vee \psi_2 \in b_j$. This is a contradiction, therefore there is a $j \geq i$ such that $(\mathcal{E}, w^j) \models \psi_2$: let $\bar{\jmath}$ denote the minimal of such j's.

We still need to show that for all $k : i \leq k < \bar{\jmath}$ it holds $(\mathcal{E}, w^k) \models \psi_1$. From the minimality of $\bar{\jmath}$ and induction hypothesis follows $\psi_2 \notin b_i$. As $\psi_1 \mathbf{U} \psi_2 \in b_i$ it is deducted that $\psi_1 \in b_i$ and $\psi_1 \mathbf{U} \psi_2 \in b_{i+1}$. Repeating the same reasoning, we obtain $\psi_1 \in b_k$ (and $\psi_1 \mathbf{U} \psi_2 \in b_k$) for all $k : i \leq k < \bar{\jmath}$. The thesis follows from (inner) induction hypothesis

"$\Leftarrow$": suppose $(\mathcal{E}, w^i) \models \psi_1 \mathbf{U} \psi_2$, we will show $\psi_1 \mathbf{U} \psi_2 \in b_i$. From the hypotheses there is a $j \geq i$ such that $(\mathcal{E}, w^j) \models \psi_2$ and for all $k : i \leq k < j$ (if any) it holds $(\mathcal{E}, w^k) \models \psi_1$. From induction hypothesis we immediately derive $\psi_2 \in b_j$ and $\psi_1 \in b_k$. From definition of elementary set it follows $\psi_1 \mathbf{U} \psi_2 \in b_j$ and if $j = i$ we are done. Otherwise, with an inductive argument on $j-1, j-2, \ldots, i$, by definition of $\delta_{\rightarrow}$ we infer $\psi_1 \mathbf{U} \psi_2 \in b_{j-1}$, $\psi_1 \mathbf{U} \psi_2 \in b_{j-2}$, $\ldots$, $\psi_1 \mathbf{U} \psi_2 \in b_i$.

The proof is now at the (outer) inductive case.

$\mathit{level}(\varphi) = l > 0$: we suppose our claim holds for all formulae whose levels are less than l. Fixed an arbitrary accepting run $b_0 b_1 \ldots$ of $\mathcal{A}_\varphi$, we prove the claim on all $\psi \in \mathit{cl}(\varphi)$ by structural induction of those formulae. Note how in proofs above the level of such formulae has never been used: in fact the level is needed only when removing quantifiers and there is none at level 0. The level is going to be used in the inner induction quantifier case. The cases for equalities, atoms, negation, conjunction, next and until are exactly the same and are not repeated here.

• Equality, atoms, negation, conjunction, next, until: for an arbitrary $i \geq 0$, same as above.

• Universal quantifier: our claims becomes

$$\textit{for all } i \geq 0.\ \forall x{:}p.\ \psi' \in b_i \iff (\mathcal{E}, w^i) \models \forall x{:}p.\ \psi'$$

Note we are in the inner level of the induction proof. Let $i \geq 0$ be an arbitrary index, the proof is split in an "only-if" and an "if" part.

"$\Rightarrow$": suppose $\forall x{:}p.\ \psi' \in b_i$, we must show $(\mathcal{E}, w^i) \models \forall x{:}p.\ \psi'$, equivalent to for all $t : p(t) \in w_i$, it holds $(\mathcal{E}, w^i) \models \psi'(t^{\mathcal{E}})$. If there is no such t the claim holds vacuously, otherwise there are $t_1, \ldots, t_k$ such t's and the spawning function $\delta_{\downarrow}(b_i, w_i)$ contains the conjunctive clause $\mathcal{A}_{\psi'(t_1^{\mathcal{E}})} \wedge \ldots \wedge \mathcal{A}_{\psi'(t_k^{\mathcal{E}})}$. As $b_0 b_1 \ldots$ is an accepting run of $\mathcal{A}_\varphi$, there exists a Y_i satisfying $\delta_{\downarrow}(b_i, w_i)$ such that all $\mathcal{A} \in Y_i$ have an accepting run $b'_i b'_{i+1} \ldots$ reading w^i. Therefore, as $\mathit{level}(\psi'(t_j^{\mathcal{E}})) < l$, applying the (outer) induction hypothesis to $\mathcal{A}_{\psi'(t_j^{\mathcal{E}})}$ and $b'_i b'_{i+1} \ldots$, we obtain:

$$\textit{for all } \phi \in \mathit{cl}(\psi'(t_j^{\mathcal{E}})).\ \textit{for all } l \geq 0.\ \phi \in b'_{i+l} \iff (\mathcal{E}, w^{i+l}) \models \phi$$

For $\phi = \psi'(t_j^{\mathcal{E}})$ and $l = 0$ we derive $(\mathcal{E}, w^i) \models \psi'(t_j^{\mathcal{E}})$. The same argument can be applied on all $j : 1 \leq j \leq k$ and therefore the thesis.

"$\Leftarrow$": suppose $(\mathcal{E}, w^i) \models \forall x{:}p.\ \psi'$, we will show $\forall x{:}p.\ \psi' \in b_i$. To reach a contradiction, suppose $\forall x{:}p.\ \psi' \notin b_i$, then $\neg \forall x{:}p.\ \psi' \in b_i$ by definition of elementary set. If there is no $t : p(t) \in w_i$ then the disjunction clause in $\delta_{\downarrow}(b_i, w_i)$ is $\bot$ and $b_0 b_1 \ldots$ cannot be an accepting run of $\mathcal{A}_\varphi$ (at index i). Therefore there are $t_1, \ldots, t_k$ such that $p(t_j) \in w_i$ for $j : 1 \leq j \leq k$, and the disjunctive clause of $\delta_{\downarrow}(b_i, w_i)$ is $\mathcal{A}_{\neg\psi'(t_1^{\mathcal{E}})} \vee \ldots \vee \mathcal{A}_{\neg\psi'(t_k^{\mathcal{E}})}$. Because $b_0 b_1 \ldots$ is an accepting run of $\mathcal{A}_\varphi$, there exists a set Y_i that satisfies $\delta_{\downarrow}(b_i, w_i)$ and all the automata in Y_i accept w^i. Therefore there is a $\bar{\jmath} : 1 \leq \bar{\jmath} \leq k$ such that $\mathcal{A}_{\neg\psi'(t^{\mathcal{E}}_{\bar{\jmath}})}$ has accepting run $b'_i b'_{i+1} \ldots$ reading w_i. Because $level(\neg\psi'(t^{\mathcal{E}}_{\bar{\jmath}})) < l$, applying our (outer) induction hypothesis to $\mathcal{A}_{\neg\psi'(t^{\mathcal{E}}_{\bar{\jmath}})}$ and b'_i, we obtain:

$$\textit{for all } \phi \in cl(\neg\psi'(t^{\mathcal{E}}_{\bar{\jmath}})).\ \textit{for all } l \geq 0.\ \phi \in b'_{i+l} \iff w^{i+l} \models \phi$$

For $\phi = \neg\psi'(t^{\mathcal{E}}_{\bar{\jmath}})$ and $l = 0$, we derive $(\mathcal{E}, w^i) \models \neg\psi'(t^{\mathcal{E}}_{\bar{\jmath}})$, a contradiction with our initial hypotheses.

"$\supseteq$": we need to prove that every model $(\mathcal{E}, w)$ of a formula φ is accepted by its constructed automaton $\mathcal{A}_\varphi$. The proof is by induction on $level(\varphi)$, the induction claim being:

$$\begin{aligned}&\textit{for all } \varphi \in \mathrm{LTL}^{\forall}.\ \textit{for all } w \in \Sigma^\omega : (\mathcal{E}, w) \models \varphi.\\ &\qquad (\textit{for all } i \geq 0.\ b_i := \{\psi \in cl(\varphi) : (\mathcal{E}, w^i) \models \psi\})\\ &\qquad \Rightarrow b_0 b_1 \ldots \textit{ is an accepting run of } \mathcal{A}_\varphi \textit{ reading } w\end{aligned}$$

Note the induction claim implies the thesis.

$level(\varphi) = 0$: let w be a word, note as the level is 0 the accepting conditions are only the local acceptance condition. We show $b_0 b_1 \ldots$ is an accepting run reading w.

• $b_0 b_1 \ldots$ is a well defined run reading w: From $\mathrm{LTL}^{\forall}$ semantics first all b_i are elementary sets of φ and hence belong to $\mathcal{A}_\varphi$, second $\varphi \in b_0$, and third $b_{i+1} \in \delta_{\rightarrow}(b_i, w_i)$.

• $b_0 b_1 \ldots$ is accepting: we show for all $\psi_1 \mathbf{U} \psi_2 \in cl(\varphi)$ there exist infinitely many $i \geq 0$ such that $b_i \in F_{\psi_1 \mathbf{U} \psi_2}$. For an arbitrary $\psi_1 \mathbf{U} \psi_2 \in cl(\varphi)$, suppose to reach a contradiction that only finitely many $i \geq 0$ are such that $b_i \in F_{\psi_1 \mathbf{U} \psi_2}$. Then there is a $k \geq 0$ such that for all $j \geq k$ it holds $b_j \notin F_{\psi_1 \mathbf{U} \psi_2}$ and therefore $\psi_1 \mathbf{U} \psi_2 \in b_j$ and $\psi_2 \notin b_j$, by definition of $F_{\psi_1 \mathbf{U} \psi_2}$. In particular, from $\psi_1 \mathbf{U} \psi_2 \in b_k$ and by construction of ρ_k, there must be some $l \geq k$ such that $(\mathcal{E}, w^l) \models \psi_2$ and thus $\psi_2 \in b_l$ with $l \geq k$, contradiction. Therefore $b_0 b_1 \ldots$ is accepting.

$level(\varphi) = l > 0$: we suppose our claim holds for all formulae with level strictly less than l. Fixed a word w and constructed $b_0 b_1 \ldots$, we need to show $b_0 b_1 \ldots$ is an accepting run reading w. In this case the accepting conditions comprehend an ulterior condition on the spawning automata on which the induction hypothesis is going to be used. Note as before the sets b_i are elementary, they are connected by $\delta_{\rightarrow}$, and $\varphi \in b_0$. Also, the locally acceptance condition is proved in the exact same way as before thus only the new spawning acceptance condition is proven.

We show for all $i \geq 0$ there is a Y_i satisfying $\delta_{\downarrow}(b_i, w_i)$ such that all $\mathcal{A} \in \delta_{\downarrow}(b_i, w_i)$ are accepting w^i. Define the following sets:

$$Y_i^{\forall} = \begin{cases} \{\ \mathcal{A}_{\psi(t^{\mathcal{E}})} \mid \forall x{:}p.\,\psi \in b_i, p(t) \in w_i\ \} & \textit{if there are } \forall x{:}p.\ \psi \in b_i \textit{ and } p(t) \in w_i \\ \emptyset & \textit{otherwise} \end{cases}$$

$$Y_i^{\exists} = \begin{cases} \emptyset & \textit{if there is no } \neg\forall x{:}p.\ \psi \in b_i \\ \{\ \mathcal{A}_{\neg\psi(t^{\mathcal{E}})} \mid \neg\forall x{:}p.\,\psi \in b_i, p(t) \in w_i, (\mathcal{E}, w^i) \not\models \psi(t^{\mathcal{E}})\} & \textit{otherwise} \end{cases}$$

These sets are well defined, in particular regarding $Y_i^{\exists}$ the conditions embrace all the possible cases (completeness): indeed if there are $\neg\forall x : p.\ \psi \in b_i$ then by construction $(\mathcal{E}, w^i) \models \neg\forall x : p.\ \psi$ thus there are $p(t) \in w_i$ with $(\mathcal{E}, w^i) \not\models \psi(t^{\mathcal{E}})$. Let Y_i be $Y_i^{\forall} \cup Y_i^{\exists}$: by construction it satisfies $\delta_{\downarrow}(b_i, w_i)$, we still need to show every automaton therein included is accepting w^i.[2] Now, for $\mathcal{A}_\phi \in Y_i$ we have either $\phi = \psi(t^{\mathcal{E}})$ for some $\forall x : p.\ \psi \in b_i$ and $p(t) \in w_i$, or $\phi = \neg\psi(t^{\mathcal{E}})$ for some $\neg\forall x : p.\ \psi \in b_i$ and $p(t) \in w_i$ with $(\mathcal{E}, w^i) \not\models \psi(t^{\mathcal{E}})$. Either way, it follows $(\mathcal{E}, w^i) \models \phi$. Since $level(\mathcal{A}_\phi) < l$ we can apply our induction hypothesis and construct an accepting run $b_0 b_1 \dots$ reading w^i: the thesis follows. $\square$

[2]Note that in $Y_i^{\exists}$ all the possible t's are taken in the second case of the definition: one would have sufficed to respect the semantics of the existential quantifier.

CHAPTER 5

Monitoring in depth

Monitoring is both a theoretical and practical problem. In the first section we are going to derive results on the theoretical side, proving that the so-called prefix problem is generally undecidable. The impossibility to solve the prefix problem denies the construction of a complete monotonic monitor, albeit a correct yet incomplete implementation is still possible: this is the aim of the second section.

5.1 Theoretical results

In Chapter 1 we set our intent to build an online, monotonic, as-early-as-possible, first-order monitor that further fits in previous projects. This section begins with the formal description of the problem and finishes with a negative result stating such a monitor is impossible.

First our monitor has to be online, or processing events while the system is still running. In other words it cannot store the trace and then use some finite trace semantics to evaluate a formula and it cannot be too slow to drag down the whole system. Translated in complexity theory terms, our monitoring algorithm must have a low complexity when reading and processing an event, but it might have a higher complexity during the generation of the monitor.

Another argument against finite trace semantics is the second requirement, the monotonicity: if finite trace semantics were to be used, the output value could change once set: for the formula $\Diamond p(1)$ is evaluated to $\bot$ as long as $p(1)$ has not been seen, but magically turns to be $\top$ after a $p(1)$ event. In such a case the meaning of $\top$ and $\bot$ is lost: reporting $\bot$, say, does not mean that a violation has occurred any more, but it might be that just not enough time has elapsed. Monotonicity leads to the so-called prefix semantics, based on definitions of good and bad prefixes:

Definition 20 (Good and bad prefix). *Let Γ be a signature, let $\mathcal{E}$ be an evaluation of Γ, let Σ be $2^{\Pi\cup\Psi}$, and let φ be a formula. The set of good prefixes of φ, denoted by $good_{\mathcal{E}}(\varphi)$*

is the set $\{u \in \Sigma^* \mid \textit{for all } w \in \Sigma^\omega, (\mathcal{E}, uw) \models \varphi\}$. *Similarly the set of bad prefixes of* φ, *denoted by* $bad_{\mathcal{E}}(\varphi)$ *is the set* $\{u \in \Sigma^* \mid \textit{for all } w \in \Sigma^\omega, (\mathcal{E}, uw) \not\models \varphi\}$. *In the rest of this chapter the symbol* Σ *will be used to denote* $2^{\Pi \cup \Psi}$.

Definition 21 (Prefix semantics)**.** *Let* Γ *be a signature, let* $\mathcal{E}$ *be an evaluation of* Γ, *and let* Σ *be* $2^{\Pi \cup \Psi}$. *The prefix semantics is a function* $[\cdot \models \cdot]_3^{\mathcal{E}} : \Sigma^* \times LTL^\forall \to \{\bot, \top, ?\}$ *defined as follows (cf. [15],* $[\cdot \models \cdot]_3$ *is therein called LTL*$_3$ *semantics):*

$$[u \models \varphi]_3^{\mathcal{E}} = \begin{cases} \top & \textit{if for all } w \in \Sigma^\omega, (\mathcal{E}, uw) \models \varphi \\ \bot & \textit{if for all } w \in \Sigma^\omega, (\mathcal{E}, uw) \not\models \varphi \\ ? & \textit{otherwise} \end{cases}$$

The prefix semantics function is a cautious 3-value semantics offering as much information as possible about the truth of a formula after a finite prefix yet without any possibility of future change. It results into a monotonic logic.

The third requirement asks for an as-early-as-possible monitor, meaning it must detect minimal prefixes: since each bad (resp. good) prefix can be extended only to bad (resp. good) prefixes it follows there is always a minimal subprefix. The requirement, apparently innocent, will be the cause of undecidability of the prefix problem, reducible to trace existence, reducible in turn to finite satisfiability in first-order logic, undecidable by Trakhtenbrot theorem.

The other two requirements are implicitly fulfilled. We now state formally two equivalent variants of the prefix problem:

Prefix problem 1
Given: $\varphi \in \mathrm{LTL}^\forall$, $u \in \Sigma^*$,
Question: $u \in good_{\mathcal{E}}(\varphi)$? (resp. $u \in bad_{\mathcal{E}}(\varphi)$?)

Prefix problem 2
Given: $\varphi \in \mathrm{LTL}^\forall$, $u \in \Sigma^*$,
Question: compute $[u \models \varphi]_3^{\mathcal{E}}$

The attentive reader might already have noticed that not every formula has a good or bad prefix: for consider $\Box\Diamond p(1)$, asking for $p(1)$ to occur infinitely often. No finite prefix will ever fulfil this condition, fate decided only by infinite tails: any monitor then has to resign and output '?' endlessly. We want to avoid these formulae indeed and only start a monitor in case of a monitorable formula.

Historically [32], monitorable formulae were those whose set of models is a safety or co-safety property: informally a subset of infinite words P is a safety property if each word not in the subset has a bad prefix. We report here, adapted, the formal definition given in Section 3.3.2 of [10], which the reader should refer to for a brief discussion of safety properties in the model checking scenario:

Definition 22 (Safety property)**.** *Given an atomic propositions set AP, a subset* P *of infinite words over* 2^{AP} *is called a safety property if for all words* $w \in (2^{AP})^\omega \setminus P$ *there exists a finite prefix* u *such that*

$$P \cap \{w' \mid u \textit{ is a finite prefix of } w'\} = \emptyset$$

Safety and co-safety properties are glamour properties in the formal methods community although our framework, wanting also to be as general as it could be, consider the strictly broader class defined in [15, 31]:

Definition 23 (Ugly prefix, monitorable formula)**.** *Let L be a language of infinite words over Σ. A finite word u is called an ugly prefix for L if it cannot be extended to a finite good or bad prefix. A formula φ is non-monitorable if $\mathcal{L}(\varphi)$ has an ugly prefix, otherwise φ is called monitorable.*

A (counter-)example for the strict containment is shown in Lemma 3.8 of [15]; the paper also discusses monitorable properties in general and the problem of determining whether a formula is monitorable, although the complexity is not known to date. All the work has been done in a propositional context and our is an attempt to extend it to first-order. A first difference is that first-order monitors depend on the underlying evaluation $\mathcal{E}$: for consider $\exists x:p.\ \exists y:p.\ x \neq y$. An evaluation whose the domain is any singleton set will always lead to $\bot$, whereas in case of more-than-2-elements domains the formula could be evaluated to $\top$.

Finally we link the prefix problem, or theoretical monitoring, to the trace existence problem studied in the previous section.

Theorem 24 (Equivalence of monitoring and trace existence)**.** *Let Γ be a signature $(C, F, P \cup Q)$, let $\mathcal{E}$ be an evaluation of Γ. Then the satisfying trace problem is reducible to the monitoring problem. Vice versa, if $|P| < \infty$ then the monitoring problem is reducible to the satisfying trace problem. All the problems are meant to be over the same fixed evaluation $\mathcal{E}$.*

Proof. Let us first reduce satisfying trace existence to monitoring. Let φ be a formula and consider any $\sigma \in \Sigma^*$ of length 1. Then φ has a satisfying trace over $\mathcal{E}$ if and only if $[\sigma \models \bigcirc\varphi]_3^{\mathcal{E}} \neq \bot$.

Vice versa, we reduce monitoring to satisfying trace existence. Let u be a finite word $w_0, \ldots, w_n \in \Sigma^*$ and $\varphi \in \mathrm{LTL}^{\forall}$ a formula, the goal is to compute $[u \models \varphi]_3^{\mathcal{E}}$ by satisfying trace existence. First, we define a formula ψ_u whose models exactly coincide with u on their first part. Let

$$\psi_{w_0} = \bigwedge_{p \in P, p(t) \in w_0} p(t) \wedge \bigwedge_{p \in P} (\forall x:p.\ \bigvee_{p(t) \in w_0} x = t)$$

where the disjunction is meant to be $\top$ if there is no p-atom, for any $p \in P$.[1]

A model of ψ_{w_0} is precisely a structure whose first world agrees with w_0 on its P-atoms. Similarly define $\psi_{w_1}, \ldots, \psi_{w_n}$ and let

$$\psi = \psi_{w_0} \wedge \bigcirc\psi_{w_1} \wedge \bigcirc\bigcirc\,\psi_{w_2} \wedge \ldots \wedge \bigcirc^n\psi_{w_n}$$

We solve the satisfying trace existence problem for $\varphi^+ = \psi \wedge \varphi$ and $\varphi^- = \psi \wedge \neg\varphi$. From their semantics they cannot have both a satisfying trace at the same time: if both have

[1]Note the necessity of finiteness in the signature: without it the formula would be infinite

one, then there are two traces $w^+, w^- \in \Sigma^\omega$ agreeing with u such that $(\mathcal{E}, w^+) \models \varphi$ and $(\mathcal{E}, w^-) \models \neg\varphi$.[2] In this case $[u \models \varphi]_3^{\mathcal{E}} =?$. Now say φ^+ has a satisfying trace but φ^- does not: then for all the traces $w \in \Sigma^\omega$ agreeing with u it holds $(\mathcal{E}, w) \not\models \neg\varphi$, or equivalently $(\mathcal{E}, w) \models \varphi$. That is, for all $w' \in \Sigma^\omega$ it holds $(\mathcal{E}, uw') \models \varphi$ and $[u \models \varphi]_3^{\mathcal{E}} = \top$. The other case where φ^+ is unsatisfiable and φ^- is satisfiable is similar and leads to $[u \models \varphi]_3^{\mathcal{E}} = \bot$. □

Combining Theorem 11 and 24 the main theoretical result of the thesis is reached, the undecidability of monitoring. The assumption of a large $|P|$ is derived from Theorem 11: having a large number of predicates in the signature is common practice in any logic, where it is usually countable infinite (cf. [18]), and from the practical side it enhances design flexibility and expressivity. A precise count is outside the scope of this thesis, but roughly it is at least the sum of the number of predicate symbols and the number of function symbols in the first-order formula encoding a Turing Machine "deciding" the halting problem.

Corollary 25. *Monitoring in LTL$^\forall$ is undecidable, if the number of predicates in P is large.*

5.2 Practical monitoring

In the previous section we showed that an online, monotonic, as-early-as-possible first order monitor is impossible. The aim of this section is that a correct monitor is still constructible if we relax the as-early-as-possible constraint: our monitor is not forced anymore to report minimal prefixes but once it outputs '$\top$' or '$\bot$', it is the real value indeed. Theoretically speaking, our monitor could output '?' forever and retain correctness, but we will see in practice this is rarely the case. Moreover it outperforms previous techniques on some structured formulae (see progression in [13] and compare with Section 2 of Chapter 7).

The idea takes root in [15], using a Büchi-like automata as in [35, 34]: two nondeterministic Büchi automata, one for φ and one for $\neg\varphi$, are run in parallel. If $\neg\varphi$'s automaton, say, stops because no run could advance, φ's automaton always will be accepting and thus φ be true: in this case a monitor for φ can return '$\top$', and vice versa.

Differently from [15] our automata cannot be turned into DFA's because of their spawned component. In fact, in order to implement a quantifier semantics dependable on the running trace, the automaton cannot be expanded completely beforehand, but needs to spawn appropriate subautomata depending on the considered world: for example, a universal quantifier is expanded in a conjunction of subautomata but the number of conjuncts is the cardinality of the interpreted predicate quantified upon, and the interpretation indeed depends on a particular world in the trace. The monitor then is built on the automata model, recursively spawning submonitors (instead of subautomata):

[2] *agreeing with u* means that the first $n + 1$ worlds of w^+ and w^- are precisely u

here we define the new $\delta_{\downarrow}(b, \sigma)$:

$$\left(\bigwedge_{\forall x:p.\psi \in b} \left(\bigwedge_{p(t)\in\sigma} M_{\mathcal{E},\psi(t^{\mathcal{E}})}\right)\right) \wedge \left(\bigwedge_{\neg\forall x:p.\psi \in b} \left(\bigvee_{p(t)\in\sigma} M_{\mathcal{E},\neg\psi(t^{\mathcal{E}})}\right)\right)$$

Note this is identical to automata's where submonitors $M_{\mathcal{E},\psi}$ replace subautomata $\mathcal{A}_{\psi}$; when the evaluation is clear from the context a (sub)monitor is indicated by M_{φ}. The notation $M_{\varphi}(u)$ indicates the output of the algorithm M_{φ} after having read u, world by world.

The following algorithm in pseudo-code implements the above mentioned idea: each monitor is split in two parts, each one representing two non-deterministic automata, respectively for φ and for $\neg\varphi$. The two run in parallel, exploiting buffers $\texttt{buff}_{\varphi}$ and $\texttt{buff}_{\neg\varphi}$ to remember potentially accepting runs, i.e. runs that in principle could turn into accepting ones in the future. In particular no stopped or refusing run is stored. A run is made of pairs, each indicating a state and the correspondent spawned function $\delta_{\downarrow}$: the submonitors therein present will keep running until the spawned function is true or the entire run is removed from the buffer because it has stopped or become false. During each iteration of the algorithm (lines 2–24), a world is read, all runs in the buffers are advanced and hence the world is passed over to submonitors. After the recursion stack has emptied, the base case being a propositional monitor, the monitor checks buffers for emptiness symbolising definitely refusing automata. In case no automata has already refuted, the monitoring goes on and '?' is returned.

```
Input            : last emitted event $\sigma$
Output           : approximation of $[u\sigma \models \varphi]_3^{\mathcal{E}}$, where $u$ is the word read so far
Static variables : $\texttt{buff}_\varphi, \texttt{buff}_{\neg\varphi}$   (if $level(\varphi) > 0$)
Initialisation   : $\texttt{buff}_\varphi \leftarrow \{ \ [(b, \top)] \mid b \in B_0^\varphi \ \}$   (if $level(\varphi) > 0$)
if $level(\varphi) = 0$ then
    return $[u\sigma \models \varphi]_3^{\mathcal{E}}$ from a propositional monitor
end
$\texttt{buff}_\varphi \leftarrow \{[h_1, \dots, h_n, (b, \delta_\downarrow(b, \sigma)), (b', \top)] \mid [h_1, \dots, h_n, (b, \top)] \in \texttt{buff}_\varphi, b' \in \delta_\rightarrow(b, \sigma)\}$, similar for $\texttt{buff}_{\neg\varphi}$
for $[h_1, \dots, h_n, h_{n+1}] \in \texttt{buff}_\varphi$ do
    for $i \leftarrow n$ downto $1$ do
        $(b, obl) \leftarrow h_i$
        Send $\sigma$ to all submonitors-variables in $obl$, wait for verdicts
        Skip all returned ?'s, replace other values in corresponding variables
        if $obl \equiv \bot$ then
            remove $[h_1, \dots, h_{n+1}]$ from $\texttt{buff}_\varphi$ (resp. $\texttt{buff}_{\neg\varphi}$)
        end
        if $obl \equiv \top$ then
            remove $h_i$
        end
    end
end
if $\texttt{buff}_\varphi = \emptyset$ then
    return $\bot$
end
if $\texttt{buff}_{\neg\varphi} = \emptyset$ then
    return $\top$
end
return ?
```

Algorithm 5.1: Monitoring algorithm $M_{\mathcal{E},\varphi}$

Theorem 26 (Correctness of monitor)**.** *Let Γ be a finite signature, $\mathcal{E}$ an evaluation over Γ, φ a sentence, and u a finite word. If $M_{\mathcal{E},\varphi}(u) = \top$, then $u \in good_{\mathcal{E}}(\varphi)$. Similarly if $M_{\mathcal{E},\varphi}(u) = \bot$, then $u \in bad_{\mathcal{E}}(\varphi)$.*

Proof. The proof is by induction on $level(\varphi)$ using the claim proved in Theorem 19, that here is restated:

$$\textit{for all } \varphi \in \mathrm{LTL}^{\forall}.\ \textit{for all } b_0 b_1 \dots \textit{ accepting run of } \mathcal{A}_\varphi \textit{ reading } w.$$
$$\textit{for all } \psi \in cl(\varphi).\ \textit{for all } i \geq 0.\ \psi \in b_i \iff (\mathcal{E}, w^i) \models \psi$$

$level(\varphi) = 0$: the thesis follows from the correctness of a propositional monitor, see [14].

level$(\varphi) > 0$: let k be $|u|$. To reach a contradiction, suppose $M_{\mathcal{E},\varphi}(u) = \top$ while u is not a good prefix, i.e. there is a word $w \in \Sigma^\omega$ such that $(\mathcal{E}, vw) \models \neg\varphi$. We will show that $[(b_0, \delta_\downarrow(b_0, u_0)), \ldots, (b_{k-1}, \delta_\downarrow(b_{k-1}, u_{k-1}))]$ belongs to $\texttt{buff}_{\neg\varphi}$ after having read u, eventually with some pairs removed: in this way $\texttt{buff}_{\neg\varphi}$ could not have been empty and $\top$ returned.

First, since $(\mathcal{E}, vw) \models \neg\varphi$, the automaton $\mathcal{A}_{\neg\varphi}$ has an accepting run $b_0 b_1 \ldots b_{k-1}\rho$ reading uw, and therefore, step after step, the monitor builds the correct run at line 4, having a structurally equal $\delta_\rightarrow$. Second such run is never removed from the buffer at line 11: suppose the contrary, then in some cycle l there was a $\delta_\downarrow(b_j, u_j)$ evaluated to $\bot$, with $0 \leq j \leq l < k$. If

$$\delta_\downarrow(b_j, u_j) = \left(\bigwedge_{\forall x:p.\psi \in b_j} \left(\bigwedge_{p(t) \in u_j} M_{\mathcal{E},\psi(t^\mathcal{E})} \right) \right) \wedge \left(\bigwedge_{\neg\forall x:p.\psi \in b_j} \left(\bigvee_{p(t) \in u_j} M_{\mathcal{E},\neg\psi(t^\mathcal{E})} \right) \right).$$

then at least one conjunct is evaluated to $\bot$, say it is $M_{\mathcal{E},\psi(t^\mathcal{E})}(u_j, \ldots, u_l)$ from "second $\bigwedge$"; the case where all monitors from the "$\bigvee$" are evaluated to $\bot$ is similar. Because *level*$(\psi(t^\mathcal{E})) <$ *level*(φ), from the induction hypothesis it follows that for all infinite word w' it holds $(\mathcal{E}, u_j \ldots u_l w') \models \neg\psi(t^\mathcal{E})$, and therefore $(\mathcal{E}, u_j \ldots u_l w') \models \neg\forall x : p.\ \psi$. But $b_0 b_1 \ldots b_{k-1}\rho$ is and accepting run in $\mathcal{A}_{\neg\varphi}$ reading uw, therefore from the above mentioned claim and $\forall x : p.\ \psi \in b_j$, it follows $(\mathcal{E}, (uw)^j) \models \forall x : p.\ \psi$, where $(uw)^j = u_j \ldots u_{k-1} w$. The contradiction arises when w' is taken to be $u_{l+1} \ldots u_{k-1} w$.

The other statement is proven similarly. □

CHAPTER 6

Model checking

This chapter deals with model checking, a static verification technique ensuring validity of relevant system properties up to mathematical rigour [10]. The entire system and the properties are first abstracted and formalised, then specific algorithms are run, extensively exploring the whole state space for violations, brute-force that guarantees mathematical correctness in case of success or that finds an explicit counterexample in case of failure.

Model checking in the propositional case is well understood, both in LTL and in CTL, a branching temporal logic similar to LTL but that allows quantifiers over paths. Our work is an attempt to extend the problem of model checking to first-order temporal logic and show its relevance with monitoring. The chosen logic is here LTL$^\forall$ for consistency reason with the monitoring problem.

Definition 27 (First-order Kripke structure)**.** *A (first-order) Kripke structure $\mathcal{K}$ is a 6-uple $(S, s_0, \rightarrow, L, \Gamma, \mathcal{E})$ where:*

- *S is a finite set of states*
- *s_0 is a special initial state in S*
- *Γ is an infinitely countable signature*
- *$\mathcal{E}$ is a computable evaluation $(A, \{c^\mathcal{E}\}, \{f^\mathcal{E}\})$ of Γ, i.e. the domain A has a computable representation, evaluating terms is computable, and evaluated function are computable themselves*
- *$\rightarrow$ is a binary left-total transition relation over S*
- *$L \colon S \rightarrow 2^{\Pi \cup \Psi}$ is the labelling function such that the sets $L(s) \cap \Psi$ are computable for all $s \in S$, i.e. they have some algorithmic finite representation.*

A run in the Kripke structure is an infinite sequence of states $s_0 s_1 s_2 \dots$ in which each s_i belongs to S and s_0 is really the initial state; it generates a trace $L(s_0)L(s_1)L(s_2)\dots$, hence we will use the same notation for both, run and trace, the reader being able to distinguish them from the context. The language of the Kripke structure $\mathcal{L}(\mathcal{K})$ is the set of all possible runs in $\mathcal{K}$. For any given formula φ, the notation $\mathcal{K} \models \varphi$ means each generated trace in $\mathcal{K}$ is a model for φ, or equivalently $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$. The expression "first-order" will be omitted where it is clear from the context.

The requirements on the evaluation and on the labelling function let us have objects expressing infinite information through finite algorithms and thus they can be input of decision problems. Note the difference with the monitoring scenario: in that case the system was thought as an external environment that could be queried for evaluation of terms and predicate semantics, here the system itself is modelled and thus input into a Turing machine. The requirement on Γ is similar, we want some flexibility on names but not too much to result in an machine-unmanageable object. The idea is taken from [11] weakened to just computable representation instead of automatic ones. We define now the model checking problem:

Model checking problem
Given: $\mathcal{K}$ Kripke structure, φ formula,
Question: $\mathcal{K} \models \varphi$?

From our definitions it follows that the model checking problem is well-defined. The next Lemma shows a bit unexpectedly that domain and functions could be trimmed down to finite objects. The point in defining the model checking problem on computable Kripke structures instead of finite ones is the more general nature of the approach and intrinsically a property of our logic, especially of our quantifier semantics.

Lemma 28 (Reduction to finite Kripke structures)**.** *Let $\mathcal{K}$ be a Kripke structure and φ a formula. Then there is a Kripke structure $\mathcal{K}'$ such that*

- *the domain of the evaluation in $\mathcal{K}'$ is finite ($|A'| < \infty$)*
- *the image of the labelling function in $\mathcal{K}'$ is finite ($|L'(s')| < \infty$, for all $s' \in S'$)*
- *$\mathcal{K} \models \varphi$ if and only if $\mathcal{K}' \models \varphi$*

Proof. Suppose, without loss of generality, that all the variables occurring in the formula have different names; the new Kripke structure $\mathcal{K}'$ is the 6-uple $(S, s_0, \rightarrow, L', \Gamma, \mathcal{E}')$ where $\mathcal{E}'$ and L' are a new evaluation and a new labelling function we are about to define. Let $\mathcal{E}'$ be the triple $(A', \{c^{\mathcal{E}'}, \}, \{f^{\mathcal{E}'}\})$, the domain A' is the union of all sets A'_t, constructed recursively, bottom-up, as follows: for each term t in φ,

- if t is a constant c, then $A'_c = \{c^{\mathcal{E}}\}$
- if t is a variable x bound to p, then $A'_x = \{p(t) \mid s \in S, p(t) \in L(s) \cap \Pi\}$

- if t is a term $f(t_1, \ldots, t_n)$ starting with a function symbol, then $A'_{f(t_1,\ldots,t_n)} = f^{\mathcal{E}}(D_{t_1}, \ldots, D_{t_n})$ is the (finite) image of $f^{\mathcal{E}}$ over the appropriate domain.

where $\mathcal{E}$ is the evaluation of the old Kripke structure $\mathcal{K}$. The evaluation of constants is the same while the evaluation of functions is restricted and co-restricted to A'. The new labelling function is defined by the following equations: for each state $s \in S$,

$$L'(s) = (L(s) \cap \Pi) \cup \{r(\vec{d}) \mid r(\vec{d}) \in L(s), \vec{d} \text{ is from } A'\}.$$

The semantics of the formula has not changed: for each run ρ in $\mathcal{K}$ and each corresponding run ρ' in $\mathcal{K}'$, it holds $(\mathcal{E}, \rho) \models \varphi$ if and only if $(\mathcal{E}', \rho') \models \varphi$. Before stating the induction claim, let us set some terminology and notations: the tuple of domain elements $\vec{d}$ agrees with the tuple of variables $\vec{x}$ in the formula $\psi(\vec{x})$ if whenever a component x_i is bound to a predicate p, then the corresponding d_i belongs to A'_{x_i}, for all i. Also $\mathfrak{A}$ indicates $(\mathcal{E}, \rho)$ and $\mathfrak{A}_i$ indicates $(\mathcal{E}, \rho_i)$, and similar notations hold for $\mathfrak{A}'$ and $\mathfrak{A}'_i$. The induction claim is:

$$\begin{aligned}&\textit{for all } \psi(\vec{x}) \in \textit{subf}(\varphi), \textit{for all } i \geq 0, \textit{for all } \vec{d} \text{ agreeing with } \vec{x},\\ &\qquad\qquad (\mathcal{E}, \rho), i \models \psi(\vec{d}) \iff (\mathcal{E}', \rho'), i \models \psi(\vec{d})\end{aligned}$$

- Atoms: let $\psi(\vec{x})$ be an atom $q(t(\vec{x}))$, choose an index $i \geq 0$ and a tuple $\vec{d}$ agreeing with $\vec{x}$:

  $$\begin{aligned}(\mathcal{E}, \rho), i \models q(t(\vec{d})) &\iff t^{\mathcal{E}}(\vec{d}) \in q^{\mathfrak{A}_i}\\ &\iff t^{\mathcal{E}'}(\vec{d}) \in q^{\mathfrak{A}_i} \cap A' = q^{\mathfrak{A}'_i}\end{aligned}$$

  where the left arrow in the last equivalence is trivial whereas the right arrow follows because $\vec{d}$ agrees with $\vec{x}$ and thus $t^{\mathcal{E}'}(\vec{d})$ is a well-defined element[1] of A'. The last statement is equivalent to $(\mathcal{E}', \rho'), i \models q(t(\vec{d}))$.

- Quantified subformulae: if ψ is a quantified formula $\forall x : p.\ \phi$, fixed $i \geq 0$ and a tuple $\vec{d}$ agreeing with $\vec{x}$:

  $$(\mathcal{E}, \rho), i \models \forall x : p.\ \phi(\vec{d}) \iff \textit{for all } p(t) \in w_i,\ (\mathcal{E}, \rho), i \models \phi(\vec{d}, t^{\mathcal{E}})$$

  If $\vec{d}$ agrees with $\vec{x}$, then $(\vec{d}, t^{\mathcal{E}})$ agrees with $(\vec{x}, x)$ and the induction hypothesis can be applied; moreover $t^{\mathcal{E}} = t^{\mathcal{E}'}$ because it is a ground term.

  $$\begin{aligned}\textit{for all } t \in w_i,\ (\mathcal{E}, \rho), i \models \phi(\vec{d}, t^{\mathcal{E}}) &\iff \textit{for all } t \in w_i,\ (\mathcal{E}', \rho'), i \models \phi(\vec{d}, t^{\mathcal{E}'})\\ &\iff (\mathcal{E}', \rho'), i \models \forall x : p.\ \phi(\vec{d})\end{aligned}$$

  All the other cases are just routine, they are reported briefly.

[1]possibly a tuple of elements; the notations $\vec{t}^{\mathcal{E}'}(\vec{d})$ and $r^{\mathfrak{A}_i} \cap (A' \times \ldots \times A')$ are avoided here because very heavy

- Negation:

$$\begin{aligned}(\mathcal{E},\rho), i \models \neg\phi(\vec{d}) &\iff (\mathcal{E},\rho), i \not\models \phi(\vec{d})\\ &\iff (\mathcal{E}',\rho'), i \not\models \phi(\vec{d})\\ &\iff (\mathcal{E}',\rho'), i \models \neg\phi(\vec{d})\end{aligned}$$

- Conjunction:

$$\begin{aligned}(\mathcal{E},\rho), i \models (\phi_1 \wedge \phi_2)(\vec{d}) &\iff (\mathcal{E},\rho), i \models \phi_1(\vec{d}) \textit{ and } (\mathcal{E},\rho) \models \phi_2(\vec{d})\\ &\iff (\mathcal{E}',\rho') \models \phi_1(\vec{d}) \textit{ and } (\mathcal{E}',\rho') \models \phi_2(\vec{d})\\ &\iff (\mathcal{E}',\rho') \models (\phi_1 \wedge \phi_2)(\vec{d})\end{aligned}$$

- Next operator:

$$\begin{aligned}(\mathcal{E},\rho), i \models \bigcirc\phi(\vec{d}) &\iff (\mathcal{E},\rho), i+1 \models \phi(\vec{d})\\ &\iff (\mathcal{E}',\rho'), i+1 \models \phi(\vec{d})\\ &\iff (\mathcal{E}',\rho'), i \models \bigcirc\phi(\vec{d})\end{aligned}$$

- Until operator:

$$\begin{aligned}(\mathcal{E},\rho), i \models (\phi_1\mathbf{U}\phi_2)(\vec{d}) &\iff \textit{there exists } j \geq i,\ (\mathcal{E},\rho), j \models \phi_2(\vec{d})\\ &\qquad \textit{and for all } k : i \leq k < j,\ (\mathcal{E},\rho), k \models \phi_1(\vec{d})\\ &\iff \textit{there exists } j \geq i,\ (\mathcal{E}',\rho'), j \models \phi_2(\vec{d})\\ &\qquad \textit{and for all } k : i \leq k < j,\ (\mathcal{E}',\rho'), k \models \phi_1(\vec{d})\\ &\iff (\mathcal{E}',\rho'), i \models (\phi_1\mathbf{U}\phi_2)(\vec{d})\end{aligned}$$

□

The previous result is even more striking compared to the monitoring problem: in fact, it is reasonable to ask that a similar result also holds for the monitoring scenario, since in LTL the two run in parallel. This is not the case: differently from LTL where the number of events allowed in a trace is finite, both in model checking and monitoring—$2^{|AP|}$, where AP are the atomic propositions in the observed formula—in LTL$^\forall$ infinitely many different events could be observed in a trace during monitoring but not in model checking, the former requiring just an infinite domain, the latter because traces come from a labelling function over finitely many states and are thus to repeat in infinite runs. An example of an infinite trace in monitoring is $\{p(1)\}\{p(2)\}\{p(3)\}\ldots$ where the domain is countably infinite and the signature requires only a symbol p. A reasonable question is whether there is any formula for which that trace is a satisfying trace that cannot be tweaked in a satisfying trace where the number of events therein occurring is finite: this corresponds to ask for the ultimately periodic model property in LTL [33]. If no such formula existed, traces with infinitely many events would be irrelevant and all this discussion void. However such a formula does exist, e.g. $\Box(\exists x : p.\top \wedge \forall y : p. \bigcirc \Box \neg p(y))$, therefore differentiating both LTL$^\forall$ from LTL, and the model checking problem (easier) from the monitoring problem (harder).

After these considerations the next result is less surprising.

Theorem 29 (Reduction to LTL model checking)**.** *The model checking problem in LTL$^\forall$ is reducible to the model checking problem in LTL.*

Proof. We will show that for each first-order Kripke structure $\mathcal{K}'$ and formula φ' there are a LTL Kripke structure $\mathcal{K}$ and a formula φ such that $\mathcal{K}' \models \varphi'$ if and only if $\mathcal{K} \models \varphi$. Without loss of generality, let $\mathcal{K}'$ be a LTL$^\forall$ Kripke structure $(S', s'_0, \rightarrow', L', \Gamma, \mathcal{E})$ whose domain and labelling function's image are finite, and let φ be a formula. The LTL formula φ' is constructed as follows:

```
• rename variables in $\varphi'$ so they have all different names, number states in $\mathcal{K}'$
• $\varphi \leftarrow \varphi'$
• while $\varphi$ contains an expression of the form $\phi' \leftarrow Q\vec{x} : p.\ \psi(\vec{x})$ do
    – define: $\phi \leftarrow \top$
    – for each $s'_j \in S'$ do
        ∗ $T \leftarrow \{\vec{t}^{\mathcal{E}} \mid p(\vec{t}) \in L'(s'_j) \cap \Pi\}$
        ∗ if $(Q = \forall)$ then
          $\phi \leftarrow \phi \wedge (s(j) \Rightarrow \bigwedge_{\vec{d} \in T} \psi(\vec{d}))$
          else
          $\phi \leftarrow \phi \wedge (s(j) \Rightarrow \bigvee_{\vec{d} \in T} \psi(\vec{d}))$
    – substitute $\phi'$ by $\phi$ in $\varphi$
• replace each atom $q(\vec{d})$ by $q_{\vec{d}}$ (including $s(j)$ by $s_j$)
```

The propositional Kripke structure $\mathcal{K} = (S, s_0, R, L, AP)$ is defined as follows:

- $S = S'$, $s_0 = s'_0$, $R = \rightarrow'$
- $AP = \{q_{\vec{d}} \mid q \in Q', \vec{d} \text{ from the domain of } \mathcal{E}\}$
- $L(s_j) = \{s_j\} \cup \{q_{\vec{d}} \mid q(\vec{d}) \in L'(s'_j)\}$

First, the algorithm terminates because φ' has a finite number of quantifiers, and φ is a well-defined LTL formula, it contains only temporal and boolean connectives and propositional atoms: the original φ' is a sentence, thus when we reach the replacement step all the atoms $q(\vec{t})$ are ground, because their variables were bound by some quantifiers. Also, $\mathcal{K}$ is a well-defined LTL Kripke structure.

Second, the semantics is preserved: let $\overline{\mathcal{K}}$ be the first-order Kripke structure obtained from $\mathcal{K}'$ modifying the labelling function to $\overline{L}(s_i) = \{s(i)\} \cup L'(s_i)$, and let $\overline{\varphi}$ be the LTL$^\forall$ formula obtained from φ' after one iteration of the while loop. Runs ρ' in $\mathcal{K}'$ correspond to runs $\overline{\rho}$ in $\overline{\mathcal{K}}$ and vice versa. Now we claim:

$$\textit{for all run } \rho' \in \mathcal{K}', \textit{for all } i \geq 0,\ \rho', i \models \phi' \textit{ if and only if } \overline{\rho}, i \models \phi,$$

where ϕ' is the selected expression in the while loop and ϕ is built from ϕ'. Note that this implies $\mathcal{K}' \models \varphi' \iff \overline{\mathcal{K}} \models \overline{\varphi}$. For the proof, suppose $\phi' = \forall \vec{x}{:}p.\ \psi(\vec{x})$, then:

$$\begin{aligned}
(\mathcal{E}, \rho'), i \models \forall \vec{x}{:}p.\ \psi(\vec{x}) &\iff \textit{for all } \vec{d} \in p^{\mathfrak{A}'_i}, \quad (\mathcal{E}, \rho'), i \models \psi(\vec{d}) && (6.1)\\
&\iff (\mathcal{E}, \rho'), i \models \psi(\vec{d_1}) \wedge \ldots \wedge \psi(\vec{d_n}) && (6.2)\\
&\iff (\mathcal{E}, \overline{\rho}), i \models \psi(\vec{d_1}) \wedge \ldots \wedge \psi(\vec{d_n}) && (6.3)\\
&\iff (\mathcal{E}, \overline{\rho}), i \models s(j) \Rightarrow (\psi(\vec{d_1}) \wedge \ldots \wedge \psi(\vec{d_n}) && (6.4)\\
&\iff (\mathcal{E}, \overline{\rho}), i \models \phi && (6.5)
\end{aligned}$$

because

1. definition, where $\mathfrak{A}_i = (\mathcal{E}, \rho'_i)$
2. $p^{\mathfrak{A}'_i} = \{\vec{d_1}, \ldots, \vec{d_n}\}$
3. $\overline{\mathcal{K}}$ differs from $\mathcal{K}$ by just one predicate symbol s
4. s_j is the i-th state in $\overline{\rho}$, thus $s(j)$ is there true
5. $s(j)$ is the only s-predicate true at $\overline{\rho}_i$

To see why $\overline{\mathcal{K}} \models \overline{\varphi} \iff \mathcal{K} \models \varphi$ just observe that the last for loop substitutes ground atoms by equivalent boolean atoms, both in the formula and in the Kripke structure, thus preserving the semantics.

□

CHAPTER 7

Practical examples

This chapter outlines two concrete examples: the first is a modelling exercise in the PCBRP framework, the second is a detailed monitoring example on the Android platform.

7.1 Modelling a policy in the business framework

As already explained in the introduction, the PCBRP project studies techniques to verify and reason about complex data flowing in business processes and governed by business rules. The technical report [3] investigates the model checking problem in two flavours, when a concrete initial database is provided, and when properties are to be checked for all possible databases (both argued not to be even semi-decidable). The system model does not need to be specified because process fragments are automatically combined according to first-order temporal logic rules. Data is in form of JSON objects and queries (policies) are also CTL*(FO) formulae.

For more detail we invite the reader to have a look at [3], our concern here is to describe how LTL$^\forall$ monitoring could be applied to the purchase order example of Section 2 of such report: some information system handles incoming orders, decides to decline them or to further process them, taking care of the packaging, shipping and issuing of invoices. The system, in [3] originally composed of process fragments, is not shown here as it is a black box from monitor's perspective. Stock and orders are presented in a two part JSON database, the leftmost part being the concrete data, the rightmost containing type definitions:

```
{  "order" : [1],                          DB = { order: List[Integer],
   "gold"  : true,                                gold: Bool,
   "stock" : [ { "ident" : "Mouse",               stock: List[Stock],
                 "price" : 10,                    status: Status  }
                 "available" : 0 },
              { "ident" : "Monitor",     Stock = { ident: String,
                "price" : 200,                     price: Integer,
                "available" : 2 },                 available: Integer }
              { "ident" : "Computer",
                "price" : 1000,          Status = { open: List[Integer],
                "available" : 4 } ],                value: Integer,
"status" : [ "open" : [],                           shipping: Integer,
               "value" : 0,                         paid: Bool,
               "shipping" : 0,                      shipped: Bool,
               "paid" : false,                      final: Bool }
               "shipped" : false,
               "final" : false ] }
```

The database is updated by the underlying business process and likely to change event after event. The best place for its dynamical nature is the trace, of which we provide two alternative formalizations.

The first approach deploys one hidden object, representing the whole database in a single P-predicate DB, the only element to appear in the trace. An example is $\{\mathsf{DB}(db_1)\}\{\mathsf{DB}(db_1)\}\{\mathsf{DB}(db_2)\}$, where different subscripts indicate that changes to the database have taken place. The only way to access database elements is by ad hoc accessors, functions applicable to the hidden objects db_i, that return the desired field values. Since functions interpretation is rigid through time, the necessity to change database object in the trace is now clear, or no update could be possible. In our example the accessors for the DB predicate are *order*, *gold*, *stock*, and *status* and similar ones exist for Status and Stock. Accessors can be nested, for instance the expression $shipped(status(db))$ is valid, and it returns false in the example. The constraint of having a single DB object per world pertains to the design phase, but it could be checked against the policy $\Box(\forall x\,{:}\,\mathsf{DB}.\ \forall y\,{:}\,\mathsf{DB}.\ x = y)$. This methods masks the database to the monitor, that relies heavily on the system evaluation for accessing concrete data values: on some hand the desirable OOP principle of data hiding is respected, but on the other hand the technique does not exploit the expressiveness of $\mathrm{LTL}^{\forall}$.

The second approach, closely related to relational algebra, is in some sense the opposite, it expresses each complex JSON object as a P-predicate with a special id field (primary key) plus a field for each key-value pair. The database objects becomes $\mathsf{DB}(id, order, gold, stock, status)$ and similar predicates exist for the status and the stock. Arrays are modelled as lists constructed by a functional constructor *cons*: for instance the list $[1, 2, 3]$ is just shorthand notation for $cons(1, cons(2, cons(3, \mathsf{emptyList})))$. Aggregation, such as each database object has a status object, here is a design issue, but it could possibly be checked against the policy:

$$\Box(\exists dbId, o, g, s, status\,{:}\,\mathsf{DB}.\ \exists statusId, v, shg, p, shd, f\,{:}\,\mathsf{Status}.\ status = statusId).$$

Everything seen so far occurs in the trace, the following JSON database is the first world:

```
DB(db1, [1], true, [stock1, stock2, stock3], status1)
Status(status1, [], 0, 0, false, false, false)
Stock(stock1, "Mouse", 10, 0)
Stock(stock2, "Monitor", 200, 2)
Stock(stock3, "Computer", 1000, 4)
```

This method bears the advantage to be more transparent but it requires more space.

Let us focus now on the policy to be monitored: as stated in [3], "crucial for [this] (our) example is the list of open items, under status, which has to be empty to be able to ship a purchase order. If it is not, constituents of the order are missing and need to be ordered until the list is empty". This is easily transported into our notations: in the first case we have

$$(\exists db\!:\!\mathsf{DB}.\ \mathit{shipped}(\mathit{status}(db)) = \mathsf{false})\mathbf{U}(\exists db\!:\!\mathsf{DB}.\ \mathit{open}(\mathit{status}(db)) = [])$$

and in the second case we have

$$\begin{aligned}
&(\exists \mathit{dbId}, \mathit{order}, \mathit{gold}, \mathit{stock}, \mathit{statusId}_1 : \mathsf{DB}.\\
&\ \exists \mathit{statusId}_2, \mathit{open}, \mathit{value}, \mathit{shipping}, \mathit{paid}, \mathit{shipped}, \mathit{final}\!:\!\mathsf{Status}.\\
&\quad \mathit{statusId}_1 = \mathit{statusId}_2 \land \mathit{shipped} = \mathsf{false})\mathbf{U}\\
&(\exists \mathit{dbId}, \mathit{order}, \mathit{gold}, \mathit{stock}, \mathit{statusId}_1 : \mathsf{DB}.\\
&\ \exists \mathit{statusId}_2, \mathit{open}, \mathit{value}, \mathit{shipping}, \mathit{paid}, \mathit{shipped}, \mathit{final}\!:\!\mathsf{Status}.\\
&\quad \mathit{statusId}_1 = \mathit{statusId}_2 \land \mathit{open} = []).
\end{aligned}$$

Both notations have pros and cons, but the real choice here is between succinctness and transparency.

7.2 Monitoring on Android: an example

This section takes on from the discussion in [13], showing a practically applicable example of first-order monitoring on the Android platform. Suppose that some application could establish connections to the outside world through telephony, a GPS system and various TCP/IP ports, scenario shared by most of the apps on Google Play [5]. The goal is to monitor such application and forbid it to use all connections but telephony. The policy is a simple safety property and looks like this:

$$\Box(\neg\mathsf{gps}(\mathsf{on}) \land (\forall x\!:\!\mathsf{openPort}.\ \neg\mathsf{isTransmitting}(x))$$

where gps and openPort are unary P-predicates, and isTransmitting is a unary R-predicate. The idea is to report violations whether either GPS is turned on or some transmitting port is, whereas an open inactive port is no harm.

There are two ways to model a lack of GPS connections: following the open world assumption, no gps atoms in the trace means that the status of the positioning system is unknown, therefore a new atom gps(off) must be added to the trace. Depending on the design, this could be enough to guarantee that the two atoms mutually exclude each other, or alternatively an axiom could be appended to the formula:

$$\begin{aligned}&\Box((\exists x\!:\!\mathsf{gps}.\ \top) \wedge (\forall x\!:\!\mathsf{gps}.\ \forall y\!:\!\mathsf{gps}.\ x = y) \wedge (\forall x\!:\!\mathsf{gps}.\ x = \mathsf{on} \vee x = \mathsf{off})\\ &\ \wedge(\forall x\!:\!\mathsf{gps}.\ x = \mathsf{on} \iff x \neq \mathsf{off})) \wedge (\mathsf{on} \neq \mathsf{off})\end{aligned}$$

The second way is the closed world assumption, basically meaning a missing atom is a false atom. Following this view, we will consider $\mathsf{gps}(on)$ to be a propositional predicate gps and rewrite the formula as:

$$\Box(\neg\mathsf{gps} \wedge (\forall x\!:\!\mathsf{openPort}.\ \neg\mathsf{isTransmitting}(x))$$

Monitoring this formula is perfectly possible, but so far our algorithm does not bear any clear advantage over other techniques: we want now to add some tautological property that will simplify monitoring. This is counter-intuitive especially with respect to progression where a longer formula usually results in a higher workload.

Suppose now that, from knowledge of the system, the following property is true:

$$\Box(\mathsf{browser} \Rightarrow \Diamond(\exists x\!:\!\mathsf{openPort}.\ \mathsf{isTransmitting}(x)))$$

The formula asserts that an active browser will open a transmitting port sooner or later, where $\mathsf{browser}$ is a propositional atom like gps, that can be active or inactive. Conjoining the formulae together and merging the globally operators, we obtain a new formula φ:

$$\begin{aligned}&\Box((\neg\mathsf{gps} \wedge (\forall x\!:\!\mathsf{openPort}.\ \neg\mathsf{isTransmitting}(x))\wedge\\ &\quad(\mathsf{browser} \Rightarrow \Diamond(\exists x\!:\!\mathsf{openPort}.\ \mathsf{isTransmitting}(x))))\end{aligned}$$

The keen reader would note that each prefix containing a $\mathsf{browser}$ atom is necessarily a bad prefix: in fact, that would imply that $(\exists x\!:\!\mathsf{openPort}.\ \mathsf{isTransmitting}(x))$ will become true eventually, while at the same time it must be false globally, because its negation is always true. Let us see a comparison of progression and our algorithm on the trace $\emptyset\emptyset\{\mathsf{browser}\}\emptyset$. Call ψ the subformula $\forall x\!:\!\mathsf{openPort}.\ \neg\mathsf{isTransmitting}(x)$, then progression goes:

$$\begin{aligned}prog(\varphi, \emptyset) &= \varphi \wedge prog(\neg\mathsf{gps} \wedge \psi \wedge (\mathsf{browser} \Rightarrow (\Diamond\neg\psi)), \emptyset)\\ &= \varphi \wedge \top \wedge prog(\psi, \emptyset) \wedge (\bot \Rightarrow prog(\Diamond\neg\psi, \emptyset))\\ &= \varphi \wedge \top \wedge \top \wedge \top\\ &\equiv \varphi\end{aligned}$$

$$prog(\varphi, \emptyset) = \ldots = \varphi$$

$$\begin{aligned}
prog(\varphi, \{\mathsf{browser}\}) &= \varphi \wedge \top \wedge prog(\psi, \{\mathsf{browser}\}) \wedge (\top \Rightarrow prog(\Diamond\neg\psi, \{\mathsf{browser}\})) \\
&\equiv \varphi \wedge prog(\psi, \{\mathsf{browser}\}) \wedge prog(\Diamond\neg\psi, \{\mathsf{browser}\}) \\
&= \varphi \wedge prog(\psi, \{\mathsf{browser}\}) \wedge (prog(\neg\psi, \{\mathsf{browser}\}) \vee (\Diamond\neg\psi)) \\
&= \varphi \wedge prog(\psi, \{\mathsf{browser}\}) \wedge (\neg prog(\psi, \{\mathsf{browser}\}) \vee (\Diamond\neg\psi)) \\
&= \varphi \wedge \top \wedge (\neg\top \vee (\Diamond\neg\psi)) \\
&\equiv \varphi \wedge \Diamond\neg\psi
\end{aligned}$$

$$\begin{aligned}
prog(\varphi \wedge \Diamond\neg\psi, \emptyset) &= \varphi \wedge prog(\Diamond\neg\psi, \emptyset) \\
&= \varphi \wedge (\neg prog(\psi, \emptyset) \vee \Diamond\neg\psi) \\
&= \varphi \wedge (\neg\top \vee \Diamond\neg\psi) \\
&\equiv \varphi \wedge \Diamond\neg\psi
\end{aligned}$$

Progression basically evaluates what can be evaluated at the moment and returns what has to be evaluated in the next state. After the event $\{\mathsf{browser}\}$ has been read, progression is unable to detect that a bad prefix has been reached because it cannot combine semantically $\Box(\ldots \wedge \forall x : \mathsf{openPort}.\ \neg\mathsf{isTransmitting}(x))$ and $(\top \Rightarrow \Diamond\neg\forall x : \mathsf{openPort}.\ \neg\mathsf{isTransmitting}(x))$.

Now let us see our automata-based approach: to simplify the exposition, the two Büchi automata are generated by Oddoux's and Gastin's web tool [6], keenly decreasing the overall size by three kinds of simplifications: on-the-fly, a posteriori, and strongly connected components. Since the tool works with propositional formulae, we use $\Box(\neg g \wedge p \wedge (b \Rightarrow \Diamond\neg p))$ and its negation because propositional skeletons are the same of their first-order counterparts[1]. In principle, altering the underlying automata model could breach correctness, yet our example still properly builds potentially accepting runs in the buffers by $\delta_{\rightarrow}$ and maintains semantics of first-order spawned formulae by $\delta_{\downarrow}$, thus retaining correctness as well (cf. proof of Theorem 26). In fact, the advancing function $\delta_{\rightarrow}$ is taken from a correct albeit simplified automaton, and the spawning function $\delta_{\downarrow}$ is always $\top$ as no open ports are present in the given trace. If there were a non-transmitting open port in some world, say 8080, then the spawned submonitor $M_{\neg\mathsf{isTransmitting}(8080)}$ would still return $\top$ at the same world. The reader should not think that our approach always boils down to propositional monitoring: if a transmitting open port were seen, then any correct implementation of our monitoring algorithm would return $\bot$ straight away, whereas a propositional monitor could not even express the relation between an open and a transmitting port.

Init: $\mathtt{buff}_{\varphi} \leftarrow \{[(s_0, \top)]\}$, $\mathtt{buff}_{\neg\varphi} \leftarrow \{[(s_0', \top)]\}$

$\{\emptyset\}$: after line 4, $\mathtt{buff}_{\varphi} \leftarrow \{[(s_0, p), (s_0, \top)]\}$, $\mathtt{buff}_{\neg\varphi} \leftarrow \{[(s_0', p), (s_0', \top)]\}$

after line 24, $\mathtt{buff}_{\varphi} \leftarrow \{[(s_0, \top)]\}$, $\mathtt{buff}_{\neg\varphi} \leftarrow \{[(s_0', \top)]\}$

$\{\emptyset\}$: as above

[1] p stays for ψ

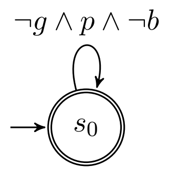

Figure 7.1: automaton for φ

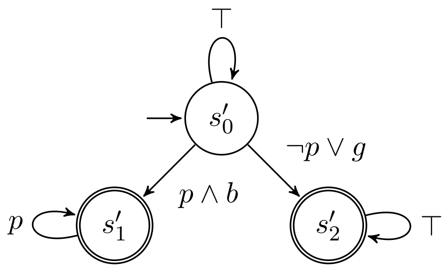

Figure 7.2: automaton for $\neg\varphi$

$\{b\}$: after line 4, $\texttt{buff}_\varphi \leftarrow \emptyset$, $\texttt{buff}_{\neg\varphi} \leftarrow \{[(s'_0, p), (s'_1, \top)], [(s'_0, p), (s'_0, \top)]\}$

after line 24, $\texttt{buff}_\varphi \leftarrow \emptyset$, $\texttt{buff}_{\neg\varphi} \leftarrow \{[(s'_1, \top)], [(s'_0, \top)]\}$

After the third event $\{b\}$, representing the world of an activated browser $\{\mathsf{browser}\}$, the algorithm correctly returns $\perp$ with no need to keep processing further.

CHAPTER 8

Conclusions

8.1 Summary

This thesis discusses the monitoring problem for applications to the PCBRP project and on the Android platform. After a theoretical digression devising a new suitable first-order temporal logic, the prefix problem was discussed and argued to be the *real* monitoring problem, the only one satisfying our reasonable assumptions. Next its theoretical insolubility was shown although a practically interesting algorithm tails in the following chapter. The algorithm outperforms previous techniques on some formulae as shown in the examples section. Finally the related model checking was reduced to the propositional case and its intrinsic difference to the monitoring problem commented.

	Satisfiability/Trace exist.	**Prefix problem**	**Model checking**
LTL	PSpace-complete	PSpace-complete	PSpace-complete
$\text{LTL}^{\forall}$	undecidable	undecidable	ExpSpace-membership, PSpace-hard

Table 8.1: complexity results

8.2 Future work

There are two main directions for further research: first, a decidable fragment of $\text{LTL}^{\forall}$ will lead to a decidable prefix problem for which an as-early-as-possible monitor could be found; second, a thorough study on which class of formulae our monitor performs best will deepen our understanding of the algorithm with possible further optimisations.

Both points have already been undergoing some research. For the decidable fragment, there are some naïve answers, e.g. restricting everything to unary ground predicates makes everything propositional, yet no final word has been put. Andréka, Neméti and Van Benthem's ideas on the guarded fragment [9] looked particularly suitable, but deep differences in the logics blighted our efforts: modal logic formulae translated to first-order logic become formulae with an open variable, meaningfully different from the closed sentences of our approach. The technique was nonetheless interesting per se, filtering the whole set of models down to some smaller candidate set and then brute-forcing it.

Another attempted technique was taken from [11], where automatic structures allowed quantification even over infinite domains. Although promising, our reduction is from finite satisfiability, basically encoding a Turing Machine solving the Halting problem in a single-world trace, therefore the same reduction is applicable, and the problem still undecidable, as long as the domain can provide new elements representing the used portion of the infinite tape, should they be generated by some regular automaton or not.

Any successful attempt would make the one-world-trace fragment decidable first, but without trimming down too much to fall back on propositional LTL. After, the result would be lifted to first-order. To date, the problem in our framework is open, up to author's knowledge.

The second point, a deeper understanding of the automata-based algorithm, has started, although the research is still in a preliminary stage. Concrete implementation code in Scala has been written in NICTA and tested against real formulae. The code is proprietary and still under development so we cannot share the details here, but encouraging results were shown on some formulae (on other ones the speed was similar to progression), particularly formulae for which semantics information may lead to an early stop of the monitor, whereas syntactical means are insufficient; an example was provided in Section 2 of Chapter 7. Particularly interesting and relating to performance speed is also the question of trace dependency: which formulae generate a monitor who has to remember long parts of the trace during computation? Indeed, monitors that are as-independent-as-possible are the most desirable: for instance, consider how a monitor $\Diamond p$ can throw away event after event, as long as it waits for p. The definition of trace dependency has not been formalised yet, although the author sketched some first attempts still lacking independence from the model of computation (Turing machines).

Links

[1] PCBRP official website: `http://ssrg.nicta.com.au/projects/PCBRP/`

[2] Nicta official website: `http://nicta.com.au`

[3] PCBRP technical paper: `http://ssrg.nicta.com.au/projects/PCBRP/papers/PCBRP-technical-paper.pdf`

[4] Maude official website: `http://maude.cs.uiuc.edu/overview.html`

[5] Google Play website, apps section: `http://play.google.com/store/apps`

[6] LTL 2 BA, fast translation from LTL formulae to Büchi automata, by Denis Oddoux and Paul Gastin. Online tool: `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`

[7] Android official website: `http://www.android.com`

[8] Symantec security report on AndroidOS.FakePlayer: `http://www.symantec.com/security_response/writeup.jsp?docid=2010-081100-1646-99`

Bibliography

[9] H. Andréka, I. Németi, and J. van Bentham. Modal Languages and Bounded Fragments of Predicate Logic. *Journal of Philosophical Logic*, 27:217–274, 1998.

[10] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[11] David Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proceedings of the 28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–60. Schloss Dagstuhl - Leibniz Center for Informatics, 2008.

[12] Andreas Bauer, Rajeev Gore, and Alwen Tiu. A first-order policy language for history-based transaction monitoring. In M. Leucker and C. Morgan, editors, *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5684 of *Lecture Notes in Computer Science*, pages 96–111, Berlin, Heidelberg, August 2009. Springer-Verlag.

[13] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. Runtime verification meets Android security. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM)*, volume 7226 of *LNCS*, pages 174–180. Springer, 2012.

[14] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *RV*, pages 126–138, 2007.

[15] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14, 2011.

[16] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *MobiSys*, pages 225–238, 2008.

[17] Raouf Boutaba, Wojciech M. Golab, and Youssef Iraqi. Lightpaths on demand: a web-services-based management system. *Comm. Mag.*, 42(7):101–107, July 2004.

[18] René Cori and Daniel Lascar. *Logique mathématique. Cours et exercices. I: Calcul propositionnel, algèbres de Boole, calcul des prédicats. Préface de J.-L. Krivine.* AXIOMES. Paris: Masson. xv, 385 p. , 1993.

[19] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[20] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, pages 627–638, 2011.

[21] Dov M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, 1987.

[22] Sylvain Hallé and Roger Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72, 2008.

[23] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN*, pages 245–264, 2000.

[24] Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *ASE*, pages 135–143, 2001.

[25] Klaus Havelund and Grigore Rosu. Testing linear temporal logic formulae on finite execution traces. Technical report, 2001.

[26] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS*, pages 342–356, 2002.

[27] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *STTT*, 6(2):158–173, 2004.

[28] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In *PDPTA*, pages 279–287, 1999.

[29] Christof Löding and Wolfgang Thomas. Alternating automata and logics over infinite words. In *IFIP TCS*, pages 521–535, 2000.

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

[31] Amir Pnueli and Aleksandr Zaks. Psl model checking and run-time verification via testers. In *FM*, pages 573–586, 2006.

[32] Grigore Rosu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[33] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[34] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[35] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.