Formal Verification of Embedded Real-Time Systems

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

vorgelegt von

Matthias Fruth

geboren am 30. November 1979 in Dresden

der Fakultät Informatik
der Technischen Universität Dresden

eingereicht am 1. Februar 2005

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. habil. Horst Reichel
Betreuer: Dipl.-Ing. Rocco Deutschmann

Acknowledgements

I would like to express my gratitude to all those people who supported me during the preparation of this thesis.

Professor Horst Reichel always had time for me. During my whole course of studies, I could benefit from his valuable support and guidance, and from many fruitful discussions. He provided the topic.

Rocco Deutschmann patiently supervised my work. He offered me a lot of freedom for my research and a friendly working environment.

Matthias Roch arranged a visit to the BMW Research and Innovation Centre in Munich.

Finally, no part of my studies had been possible without the generous support from my parents.

Thank you all!

Contents

Contents v

1. Introduction 1

2. Temporal logics 3
- 2.1. Linear-time temporal logic (LTL) 3
- 2.2. Finite-trace semantics 4
- 2.3. Real-time LTL (RTLTL) 8
- 2.4. Equivalences 11
- 2.5. Safety and liveness properties 11
- 2.6. The truth checking problem 13

3. Automata theory 15
- 3.1. Finite automata for finite and infinite traces 15
- 3.2. Büchi automata 19
- 3.3. Translating LTL formulae into Büchi automata 20
- 3.4. Translating RTLTL formulae into NFA 22
- 3.5. Checking finite traces 28
- 3.6. Complexity 31

4. Term rewriting 33
- 4.1. Preliminaries 33
- 4.2. Simple rewriting 35
- 4.3. Event-consuming rewriting 37
- 4.4. Complexity 40

5. Experiments and results 41

6. Conclusion 43

A. Software 45
- A.1. System requirements 45
- A.2. User interface 45
- A.3. Trace syntax 46
- A.4. Property syntax 47
- A.5. Usage examples 48

List of Figures 49

Bibliography 51

1. Introduction

Motivation In future, all systems will first and foremost be computer systems. As a consequence of this development, systems in all application domains, regardless whether they are implemented on hardware, software, or hybrid architectures, will essentially be concurrent and reactive. While their correct operation is often critical in several ways, complexity grows rapidly. For these reasons, verification becomes an absolute necessity.

In the last decades, a variety of verification techniques has been proposed. Traditional approaches, based on testing and simulation, are often limited to relatively weak specification languages and partial coverage of a system's behaviour. *Formal methods* [CGP99] cover all possible execution paths of a system by exploring a full system model and thus find all violations of the system specification. Unfortunately, source code of real-world systems is in many cases unavailable to the verifying party, and then models cannot be constructed. Furthermore, due to the state explosion problem, formal verification is often infeasible for large systems.

A solution to these problems is *runtime verification* [HR01b], a semi-formal[1] automatic verification method that checks single execution traces of a running system against temporal-logic specifications. This approach does not require additional modelling and can handle much larger systems than formal methods. In order to achieve acceptable coverage rates, both a sensible selection of input traces as well as the formulation of exhaustive specifications are required.

Related work Runtime verification with temporal-logic specifications is a widely researched area. There are several approaches for checking finite traces, using automata theory [GH01, FS04], term rewriting [HR01b, RH], and others[2].

The automata-theoretic methods translate temporal-logic formulae into finite automata and solve the word problem for given traces. In particular, nondeterministic and alternating finite automata for infinite words with different acceptance conditions, namely *nondeterministic Büchi automata* [GH01] and *alternating Büchi automata* [FS04], are used.

Until recently, the tableau method of Gerth, Peled, Vardi, and Wolper [GPVW95], improved by Daniele, Giunchiglia, and Vardi [DGV99], was undisputed as the state-of-the-art algorithm for translating *linear-time temporal logic (LTL)* [Pnu77] formulae into nondeterministic Büchi automata. This approach uses *labelled generalised Büchi automata* as an intermediate step. It has been adapted by Giannakopoulou and Havelund [GH01] for checking finite traces

[1] Some authors [GH01] talk of runtime verification as a *light-weight* formal method.

[2] For instance, one approach of Roşu and Havelund [RH01] is based on dynamic programming. Another, more recent, approach of them [RH] investigates on so-called *binary transition tree finite state machines* (BTT-FSMs).

with *LTL-X* [CGP99] specifications, in which case the constructed automata are nondeterministic finite automata.

The rewriting-based methods successively apply a set of *rewrite rules* in order to transform the initial problem, consisting of a finite trace and a temporal-logic formula, into simpler problems, until a truth value is obtained. Contrary to automata-theoretic techniques, this approach benefits from very simple formula and trace transformations, while the complexity is moved into the rewriting system.

Topic In this thesis, we explore runtime verification methods for the analysis of real-time systems. The major goal of this thesis is the development of an efficient runtime algorithm that checks given real-time traces against real-time specifications, to be integrated into an industrial validation and verification framework for automotive electronics. We develop a consistent extension of the existing representation formalisms for specification formulae, traces, and verification algorithms to the notion of real-time. In particular, we define a syntactic extension of LTL for the representation of real-time properties.

We present two algorithms that each take as input a finite real-time trace and a real-time specification formula, but solve the verification task in different ways: the first one bases upon the paradigm of automata-theoretic truth checking [Var97, GH01] and the second one upon term rewriting [BN98, DP01]. In order to deal with finite traces, we modify the default, infinite-trace semantics of LTL, ensuing changes in the automata acceptance conditions.

The first algorithm extends the tableau method of [GPVW95] into a procedure that translates real-time LTL formulae into nondeterministic finite automata for finite traces. Further improvements of this algorithm allow the *on-the-fly*[3] evaluation of specifications. We have published preliminary results of this approach in [DFRR04].

The second algorithm uses an extension of the rewriting-based techniques proposed in [HR01a] and [RH] for standard LTL formula. Starting with a finite trace and a real-time LTL formula, a set of rewrite rules is applied until eventually either $true$ or $false$ is returned. We also adapt an *event-consuming* version of this technique which essentially operates *online*[4].

Outline This paper is divided into six chapters and one appendix. In the chapters 2, 3, and 4, we present the necessary fundamental concepts of temporal logics, automata theory, and term rewriting. In chapter 5, we discuss correctness checks and conduct a performance evaluation of all algorithms, based on experimental results. In chapter 6, we conclude and give an outlook to future work. The appendix provides a reference to the software implementation.

[3] In this case, on-the-fly operation means that automaton construction and trace traversal take place simultaneously.

[4] A runtime verification technique operates online if it does not need to store the trace being checked.

2. Temporal logics

In runtime verification, reactive systems are observed by monitoring their execution traces. Both branching-time and linear-time logics provide effective means for expressing system properties [Var01]; however, linear-time logics like linear-time temporal logic (LTL) [Pnu77] are much more appropriate for checking finite traces, since its properties relate to single paths.

In the first section, we define the traditional syntax and semantics of LTL. Subsequently, we discuss modifications of the standard, infinite-trace semantics towards finite traces. In the third section, we develop a syntactic extension of LTL for representing real-time properties. Thereafter, we provide a set of common equivalences that can be used in order to simplify user-provided specifications. In the fifth section, we show how safety and liveness properties can be expressed in our framework and provide an introduction to the specification of real-time properties. Finally, we formalise the practical task of runtime verification as a computation problem and study its complexity.

2.1. Linear-time temporal logic (LTL)

The syntax of LTL contains both propositional formulae, describing static properties of a specific state, as well as temporal formulae, describing safety and liveness properties that apply to every path outgoing of a specific state. In addition to the common propositional operators for negation, disjunction, and conjunction, LTL contains the temporal operators X (strong next), F (future), G (globally), U (strong until), and R (release). In order to guarantee a consistent semantics, we build our formal description of LTL on a minimal complete set of operators.

Definition 2.1 (Syntax of LTL)
Let $\mathcal{P}$ be a nonempty set of atomic propositions. Then, the set of well-formed LTL formulae over $\mathcal{P}$ is inductively defined as follows:

1. *If $p \in \mathcal{P}$, then p is a formula.*
2. *If φ and ψ are formulae, then $\neg\varphi$, $\varphi \vee \psi$, $\mathsf{X}\,\varphi$, and $\varphi \,\mathsf{U}\, \psi$ are formulae.*

The remaining operators are defined by the following abbreviations:

$$\begin{aligned}
\top &\overset{def}{=} p \vee \neg p \text{ for some } p \in \mathcal{P} \\
\bot &\overset{def}{=} \neg\top \\
\varphi \wedge \psi &\overset{def}{=} \neg(\neg\varphi \vee \neg\psi) \\
\varphi \rightarrow \psi &\overset{def}{=} \neg\varphi \vee \psi
\end{aligned}$$

$$
\begin{aligned}
\mathsf{F}\,\varphi &\stackrel{def}{=} \top\,\mathsf{U}\,\varphi \\
\mathsf{G}\,\varphi &\stackrel{def}{=} \neg\,\mathsf{F}\,\neg\varphi \\
\varphi\,\mathsf{R}\,\psi &\stackrel{def}{=} \neg(\neg\varphi\,\mathsf{U}\,\neg\psi)
\end{aligned}
$$

For the evaluation of formulae, we require that unary operators have a higher precedence than binary operators, that is, arguments of unary operators match as small as possible subformulae and arguments of binary operators match as large as possible subformulae.

Traditionally, the semantics of LTL is defined with respect to infinite traces. Informally, a trace is a sequence of events where an event describes the set of variables that are true at a particular time point in the trace. Since we consider real-time traces, we expect that all events are in a consecutive order where the first event is related to the timestamp 0 and directly successive events are seperated by constant distances of time.

Definition 2.2 (Infinite trace)
*An **infinite trace** is an infinite sequence of events $\xi = x_0, x_1, \ldots$ such that $x_i \subseteq \mathcal{P}$ for all $i \geq 0$.*

Definition 2.3 (Suffix)
*Let $\xi = x_0, x_1, \ldots$ be an infinite trace and $i \geq 0$. Then, the trace $\xi^i \stackrel{def}{=} x_i, x_{i+1}, \ldots$ is called a **suffix** of ξ.*

The satisfiability relation for LTL is defined inductively on the syntactic structure of its formulae.

Definition 2.4 (Infinite-trace semantics of LTL)
An infinite trace $\xi = x_0, x_1, \ldots$ satisfies an LTL formula φ, denoted $\xi \models \varphi$, if the following holds:

$$\xi \models p \text{ for } p \in \mathcal{P} \quad \stackrel{def}{\Longleftrightarrow} \quad p \in x_0 \tag{2.1}$$

$$\xi \models \neg\varphi \quad \stackrel{def}{\Longleftrightarrow} \quad \xi \not\models \varphi \tag{2.2}$$

$$\xi \models \varphi \vee \psi \quad \stackrel{def}{\Longleftrightarrow} \quad \xi \models \varphi \text{ or } \xi \models \psi \tag{2.3}$$

$$\xi \models \mathsf{X}\,\varphi \quad \stackrel{def}{\Longleftrightarrow} \quad \xi^1 \models \varphi \tag{2.4}$$

$$\xi \models \varphi\,\mathsf{U}\,\psi \quad \stackrel{def}{\Longleftrightarrow} \quad \xi^i \models \psi \text{ for some } i \geq 0 \text{ and } \xi^j \models \varphi \text{ for all } 0 \leq j < i \tag{2.5}$$

Definition 2.5 (Semantic equivalence)
Two formulae φ and ψ are equivalent, denoted $\varphi \equiv \psi$, if, for all traces ξ, we have $\xi \models \varphi$ if and only if $\xi \models \psi$.

2.2. Finite-trace semantics

Although LTL was originally designed for infinite traces, there are several possible ways of interpreting LTL formulae over finite traces. For the following discussion, we assume that each (truncated) finite trace just reflects a limited

part of the infinite behaviour of a reactive system, and has therefore to be viewed as the prefix of some (maximal) infinite trace.

We adapt the notions of trace and trace suffix to finite sequences of events.

Definition 2.6 (Finite Trace)
*A **finite trace** is a finite sequence of events $\xi = x_0, x_1, \ldots, x_n$ such that $x_i \subseteq \mathcal{P}$ for all $0 \leq i \leq n$.*

Definition 2.7 (Suffix)
*Let $\xi = x_0, \ldots, x_n$ be a finite trace and $i \geq 0$. Then, the trace $\xi^i \stackrel{def}{=} x_i, \ldots, x_n$ is called a **suffix** of ξ.*

Note that $\xi^i = \varepsilon$ if $i \geq |\xi|$.

Choosing an adequate finite-trace semantics is often a problem, particularly when next-state or eventuality formulae must be evaluated at the end of a trace. In some cases, the truth value of a formula cannot be decided by naively applying the traditional, infinite-trace semantics to the given finite sequence of events. For instance, even if p is true for all events of a finite trace, it cannot be determined whether the safety formula $\mathsf{G}\, p$ holds for the untruncated, infinite trace; also, it cannot be concluded that the formula $\mathsf{F}\, \neg p$ holds for some infinite extension of this trace, since p might be true for some future event.

Eisner, Fisman, Havlicek and Lustig [EFH+03] give a good overview of typical problems with and possible semantics for dealing with finite traces. They distinguish three types of finite-trace semantics: *weak*, *neutral*, and *strong semantics*. In the weak view, a formula is true if and only if it is true in some infinite extension of the finite trace. In the strong view, a formula is true if and only if it is true for all infinite extensions of the finite trace, that is, if it evaluates to true within the given, finite trace. The neutral view uses different interpretations for weak and strong operators and is – besides this – analogue to the traditional, infinite-trace semantics.

However, depending on the application context, different semantics may be preferable. Most approaches to checking finite traces, as of Giannakopoulou and Havelund [GH01], and Finkbeiner and Sipma [FS04], use neutral semantics. The authors of [GH01] argue that the X operator in LTL is counterintuitive, since users might misattribute some concept of time to it; accordingly, they use LTL-X, a variant of LTL without the X operator, thus avoiding ambiguous interpretations of X formulae. Havelund and Roşu [HR01a, RH] propose a simple *stationary semantics* that extends each finite trace to an infinite one by repeating its last event infinitely often.

We think that stationary semantics for truncated traces are generally too simple and do not sufficiently reflect the behaviour of the corresponding maximal traces. There are at least three problems: first, states in the truncated trace cannot be distinguished from states in its artificial extension; this can be resolved by adding a new proposition that is true precisely in all artificially added states. Second, repeating a specific state ignores all cases in which at least one different state occurs; this can already be enough to falsify the truth value of a formula. We believe that deciding whether a stationary semantics is

sufficient for specific classes of traces and formulae is hard. Third, the extended trace is infinite and therefore more expensive to check.

Definition 2.8 (Finite-trace semantics of LTL)
A nonempty finite trace $\xi = x_0, \ldots, x_n$ satisfies an LTL formula φ, denoted $\xi \models \varphi$, if the infinite-trace semantics together with the following modification holds:

$$\xi \models \mathsf{X}\,\varphi \quad \overset{def}{\Longleftrightarrow} \quad \xi^1 \neq \varepsilon \text{ and } \xi^1 \models \varphi \tag{2.6}$$

Conversely to infinite traces, $\neg\mathsf{X}\,\varphi$ is not equivalent to $\mathsf{X}\,\neg\varphi$ when evaluated on finite traces. This is because $\mathsf{X}\,\varphi$ and $\mathsf{X}\,\neg\varphi$ both evaluate false for the same trace ξ if the suffix ξ^1 is empty, that is, if ξ consists of only one event. Since we later need a *dual* to each of our operators, we define the *weak next* operator Y by the following abbreviation:

$$\mathsf{Y}\,\varphi \overset{def}{=} \neg\,\mathsf{X}\,\neg\varphi$$

In order to overcome all of the above mentioned problems at once, we use a neutral semantics which respects the finiteness of traces. Thus, contrary to most previous approaches, our semantics is consistent for all nonempty finite traces and all LTL formulae.

Definition 2.9 (Consistency)
A logic is ***consistent*** *if there is no model $\mathcal{M}$ and formula φ such that $\varphi \wedge \neg\varphi$ is true in $\mathcal{M}$.*

Theorem 2.10 (Consistency of LTL for finite traces)
LTL for finite traces is consistent.

Proof Let φ be an LTL formula. We prove this theorem by structural induction on φ. Suppose that there is some finite trace $\xi = x_0, \ldots, x_n$ such that $\xi \models \varphi$ and $\xi \models \neg\varphi$.

1. Be $\varphi \in \mathcal{P}$. Then, either $\varphi \in x_0$ or $\varphi \notin x_0$, thus the theorem holds in this case.

2. Be $\varphi = \mathsf{X}\,\psi$ for some formula ψ. Then, $\xi \models \varphi$ implies $\xi^1 \neq \varepsilon$ and $\xi^1 \models \psi$. Now, $\xi \models \neg\varphi$ implies $\xi^1 \models \neg\psi$. But, by the induction hypothesis, the theorem holds for ψ, and thus this is a contradiction. Hence, the theorem holds in this case.

3. Be $\varphi = \psi\,\mathsf{U}\,\mu$ for some formulae ψ and μ. Then, $\xi \models \varphi$ implies that there is some $i \geq 0$ such that $\xi^i \models \mu$ and $\xi^j \models \psi$ for all $0 \leq j < i$. Now, $\xi \models \neg\varphi$ implies that there is some $0 \leq j < i$ such that $\xi^j \models \neg\mu$. But, by the induction hypothesis, the theorem holds for μ, and thus this is a contradiction. Hence, the theorem holds in this case and with it for all formulae. □

Nevertheless, although finite-trace LTL is consistent, the finite-trace interpretation of LTL formulae is not always intuitive.

Example With respect to nonempty finite traces, $\mathsf{G}\,\top$ is a *tautology*[1] while $\mathsf{G}\,\mathsf{X}\,\top$ is a *contradiction*[2]. Also, $\mathsf{F}\,\bot$ is a contradiction while $\mathsf{F}\,\mathsf{Y}\,\bot$ is a tautology. $\diamond$

Finally, we define the notion of negation normal form. We show that each formula can be transformed into this form, which is required by some of our verification algorithms.

Definition 2.11 (Negation normal form)
*A formula is in **negation normal form** if negation symbols only occur in front of atomic propositions.*

In order to show this, we concretise the notion of duality.

Definition 2.12 (Duality)

1. *Let $\circ$ and $\bullet$ be unary operators of a logic. Then, $\bullet$ is **dual** to $\circ$ if for all formulae φ the following holds:*

$$\neg \circ \varphi \equiv \bullet \neg\varphi$$

2. *Let $\circ$ and $\bullet$ be binary operators of a logic. Then, $\bullet$ is **dual** to $\circ$ if for all formulae φ and ψ the following holds:*

$$\neg(\varphi \circ \psi) \equiv \neg\varphi \bullet \neg\psi$$

Note that if an operator $\bullet$ is dual to another operator $\circ$, then also $\circ$ is dual to $\bullet$. This is a consequence of the fact that $\neg$ is dual to itself.

Lemma 2.13 (Duality of LTL)
Each temporal operator of LTL has a dual.

Proof Directly from the definitions, we obtain the following dualities: $\neg$ and X are dual to itself. $\vee$ and $\wedge$, F and G, and U and R are dual to each other. $\square$

Theorem 2.14 (Negation normal form for LTL)
For each LTL formula, there is an equivalent formula in negation normal form.

Proof Let φ be an LTL formula. We prove this theorem by structural induction on φ.

1. Be φ an atomic proposition. Then, it is already in negation normal form.
2. Be $\varphi = \circ\psi$. By the induction hypothesis, there is a formula $\psi' \equiv \psi$ in negation normal form. Thus, $\circ\psi' \equiv \varphi$ is in negation normal form.
3. Be $\varphi = \psi \circ \mu$. By the induction hypothesis, there are formulae $\psi' \equiv \psi$ and $\mu' \equiv \mu$ in negation normal form. Thus, $\psi' \circ \mu' \equiv \varphi$ is in negation normal form.

[1] A tautology is a formula that is true in all models.
[2] A contradiction is a formula that is false in all models.

4. Be $\varphi = \neg \circ \psi$. Let $\bullet$ be a dual operator of $\circ$. By the induction hypothesis, there is a formula $\psi' \equiv \neg\psi$ in negation normal form. Thus, $\bullet\psi' \equiv \varphi$ is in negation normal form.

5. Be $\varphi = \neg(\psi \circ \mu)$. Let $\bullet$ be a dual operator of $\circ$. By the induction hypothesis, there are formulae $\psi' \equiv \neg\psi$ and $\mu' \equiv \neg\mu$ in negation normal form. Thus, $\psi' \bullet \mu' \equiv \varphi$ is in negation normal form.

Hence, each LTL formula φ has an equivalent formula φ' in negation normal form. □

2.3. Real-time LTL (RTLTL)

In the past, there have been various developments in the field of real-time logics. Real-time CTL [EMSS92], abbreviated RTCTL, extends CTL by annotating strong next and strong until operators with nonnegative integer timepoints. Metric temporal logic [Koy90, AH93], abbreviated MTL, extends LTL in a similar way, but operators can also be annotated by congruence expressions[3].

Our approach extends LTL to real-time by annotating discrete-time intervals to temporal operators. The resulting logic RTLTL can be seen as a linear-time variant of RTCTL, where, conversely to MTL, we assume that time proceeds synchronously, that is, next-time equals next-state. We prefer linear-time to branching-time temporal logic, since we want to express properties of finite traces.

For instance, $\mathsf{G}\,(p \rightarrow \mathsf{F}_{5,10}\, q)$ expresses the property that for each event satisfying p, some future event in the time range from 5 to 10 satisfies q. For real-time systems, the annotated time values directly correspond to system times: provided 1 state represents 1 millisecond, we get "whenever p holds, q must follow within 5 to 10 milliseconds".

Definition 2.15 (Syntax of RTLTL)
Let $\mathcal{P}$ be a nonempty set of atomic propositions. The set of well-formed RTLTL formulae over $\mathcal{P}$ is inductively defined as follows:

1. *If φ is an LTL formula, then it is an RTLTL formula.*

2. *If φ and ψ are RTLTL formulae and a and b are integers with $0 \leq a \leq b$, then $\mathsf{X}_a\, \varphi$ and $\varphi\, \mathsf{U}_{a,b}\, \psi$ are RTLTL formulae.*

The time-bounded analogues of the remaining temporal operators are defined by the following additional abbreviations:

$$\begin{aligned} \mathsf{Y}_a\, \varphi &\overset{def}{=} \neg\, \mathsf{X}_a \neg\varphi \\ \mathsf{F}_{a,b}\, \varphi &\overset{def}{=} \top\, \mathsf{U}_{a,b}\, \varphi \\ \mathsf{G}_{a,b}\, \varphi &\overset{def}{=} \neg\mathsf{F}_{a,b} \neg\varphi \\ \varphi\, \mathsf{R}_{a,b}\, \psi &\overset{def}{=} \neg(\neg\varphi\, \mathsf{U}_{a,b} \neg\psi) \end{aligned}$$

[3] With congruence expressions one can, for instance, describe that a certain property holds precisely in those events of a trace that have an index $i \equiv 0 \pmod 2$.

Definition 2.16 (Finite-trace semantics of RTLTL)
A nonempty finite trace $\xi = x_0, \ldots, x_n$ satisfies an RTLTL formula φ, denoted $\xi \models \varphi$, if the following holds:

$$\xi \models p \text{ for } p \in \mathcal{P} \overset{def}{\iff} p \in x_0 \tag{2.7}$$

$$\xi \models \neg\varphi \overset{def}{\iff} \xi \not\models \varphi \tag{2.8}$$

$$\xi \models \varphi \vee \psi \overset{def}{\iff} \xi \models \varphi \text{ or } \xi \models \psi \tag{2.9}$$

$$\xi \models \mathsf{X}\,\varphi \overset{def}{\iff} \xi^1 \neq \varepsilon \text{ and } \xi^1 \models \varphi \tag{2.10}$$

$$\xi \models \varphi\,\mathsf{U}\,\psi \overset{def}{\iff} \xi^i \models \psi \text{ for some } 0 \leq i \leq n \text{ and } \xi^j \models \varphi \text{ for all } 0 \leq j < i \tag{2.11}$$

$$\xi \models \mathsf{X}_a\,\varphi \overset{def}{\iff} \xi^a \neq \varepsilon \text{ and } \xi^a \models \varphi \tag{2.12}$$

$$\xi \models \varphi\,\mathsf{U}_{a,b}\,\psi \overset{def}{\iff} \xi^i \models \psi \text{ for some } i \leq n \text{ with } a \leq i \leq b \text{ and } \xi^j \models \varphi \text{ for all } j \text{ such that } 0 \leq j < i \tag{2.13}$$

Finally, the finite-trace semantics of the derived operators is as follows.

Corollary 2.17
Let $\xi = x_0, \ldots, x_n$ be a nonempty finite trace, φ and ψ be RTLTL formulae, and a and b be integers with $0 \leq a \leq b$. Then the following holds:

$$\xi \models \top$$

$$\xi \not\models \bot$$

$$\xi \models \varphi \wedge \psi \iff \xi \models \varphi \text{ and } \xi \models \psi$$

$$\xi \models \varphi \rightarrow \psi \iff \xi \not\models \varphi \text{ or } \xi \models \psi$$

$$\xi \models \mathsf{Y}\,\varphi \iff \xi^1 = \varepsilon \text{ or } \xi^1 \models \varphi$$

$$\xi \models \varphi\,\mathsf{R}\,\psi \iff \text{for all } 0 \leq i \leq n, \text{ we have } \xi^i \models \psi \text{ or } \xi^j \models \varphi \text{ for some } 0 \leq j < i$$

$$\xi \models \mathsf{F}\,\varphi \iff \xi^i \models \varphi \text{ for some } 0 \leq i \leq n$$

$$\xi \models \mathsf{G}\,\varphi \iff \xi^i \models \varphi \text{ for all } 0 \leq i \leq n$$

$$\xi \models \mathsf{Y}_a\,\varphi \iff \xi^a = \varepsilon \text{ or } \xi^a \models \varphi$$

$$\xi \models \varphi\,\mathsf{R}_{a,b}\,\psi \iff \xi^a = \varepsilon \text{ or for all } 0 \leq i < n \text{ with } a \leq i \leq b, \text{ we have } \xi^i \models \psi \text{ or } \xi^j \models \varphi \text{ for some } 0 \leq j < i$$

$$\xi \models \mathsf{F}_{a,b}\,\varphi \iff \xi^i \models \varphi \text{ for some } 0 \leq i \leq n,\ a \leq i \leq b$$

$$\xi \models \mathsf{G}_{a,b}\,\varphi \iff \xi^i \models \varphi \text{ for all } 0 \leq i \leq n,\ a \leq i \leq b$$

The following result states that RTLTL is expressively equivalent to LTL.

Theorem 2.18 (Expressiveness of RTLTL)
For each RTLTL formula, there is an equivalent LTL formula, and vice versa.

Proof The direction from LTL to RTLTL is trivial. For the opposite direction, note that the following equivalences hold:

$$\mathsf{X}_0\,\varphi \equiv \varphi \tag{2.14}$$
$$\mathsf{X}_{a+1}\,\varphi \equiv \mathsf{X}\,\mathsf{X}_a\,\varphi \tag{2.15}$$
$$\varphi\,\mathsf{U}_{0,0}\,\psi \equiv \psi \tag{2.16}$$
$$\varphi\,\mathsf{U}_{0,b+1}\,\psi \equiv \psi \lor (\varphi \land \mathsf{X}\,(\varphi\,\mathsf{U}_{0,b}\,\psi)) \tag{2.17}$$
$$\varphi\,\mathsf{U}_{a+1,b+1}\,\psi \equiv \varphi \land \mathsf{X}\,(\varphi\,\mathsf{U}_{a,b}\,\psi) \tag{2.18}$$

Now, each subformula of φ that matches the left-hand side of one of the equivalences is replaced by the corresponding right-hand side, until no matching subformula remains. □

Theorem 2.19 (Complexity of RTLTL to LTL translation)
Given an RTLTL formula φ, there is an equivalent LTL formula of size $2^{\mathcal{O}(|\varphi|)}$.

Proof The algorithm from the proof of theorem 2.18 transforms a given RTLTL formula φ into an equivalent LTL formula φ'. We prove by structural induction on φ that the resulting LTL formula has length $2^{\mathcal{O}(|\varphi|)}$.

1. If φ is already an LTL formula, then the assertion trivially holds.
2. If $\varphi = \mathsf{X}_a\,\psi$, then we have $|\varphi| = 1 + \log a + |\psi|$. By the induction hypothesis, there is an LTL formula $\psi' \equiv \psi$ such that $|\psi'| = 2^{\mathcal{O}(|\psi|)}$. Exhaustive application of the translation rules produces the formula $\varphi' = \mathsf{X} \cdots \mathsf{X}\,\psi'$ (a times X) such that $\varphi' \equiv \varphi$. Hence,
$$|\varphi'| = 1 + \log a + 2^{\mathcal{O}(|\psi|)} \le 2^{\mathcal{O}(\log a + |\psi|)} = 2^{\mathcal{O}(|\varphi|)}.$$
3. If $\varphi = \mu\,\mathsf{U}_{a,b}\,\eta$, then we have $|\varphi| = 1 + \log a + \log b + |\mu| + |\eta|$. By the induction hypothesis, there are formulae $\mu' \equiv \mu$ and $\eta' \equiv \eta$ such that $|\mu'| = 2^{\mathcal{O}(|\mu|)}$ and $|\eta'| = 2^{\mathcal{O}(|\eta|)}$. By a applications of rule 2.18, $b - a$ applications of rule 2.17, one application of rule 2.16, and a number of rule application to μ and η, we obtain an LTL formula $\varphi' \equiv \varphi$. Each application of rule 2.17 adds six, and each application of rule 2.18 adds three new symbols to a formula. Hence,
$$|\varphi'| \le 1 + 6\log b + 2^{\mathcal{O}(|\mu|)} \le 2^{\mathcal{O}(\log a + \log b + |\mu| + |\eta|)} = 2^{\mathcal{O}(|\varphi|)}.$$

□

Theorem 2.18 lets us generalise a number of interesting results from the last section.

Corollary 2.20 (Consistency of RTLTL)
RTLTL for finite traces is consistent.

Corollary 2.21 (Duality of RTLTL)
Each temporal operator of LTL has a dual.

Corollary 2.22 (Negation normal form for RTLTL)
For each LTL formula there is an equivalent formula in negation normal form.

2.4. Equivalences

Before a temporal-logic formula is evaluated by means of automata-theoretic or rewriting-based trace checking, logical equivalences are used in order to eliminate redundancies and to reduce the number of temporal operators. This is done by replacing all subformulae matching the left-hand side of a rule with the corresponding instance of the right-hand side of this rule.

The result are more succint formulae and thus less time and space requirements for automata construction and rewriting. Somenzi and Bloem [SB00], Etessami and Holzmann [EH00], and Tauriainen [Tau03] present compilations of such rules. We introduce our rule set by the following theorem.

Theorem 2.23
Let φ and ψ be RTLTL formulae and a a natural number. Then the following holds:

- $\varphi \vee \neg\varphi \equiv \top$
- $\varphi \wedge \neg\varphi \equiv \bot$
- $\mathsf{X}\,\bot \equiv \bot$
- $\mathsf{Y}\,\top \equiv \top$
- $\mathsf{F}\,\top \equiv \top$
- $\mathsf{F}\,\bot \equiv \bot$
- $\mathsf{G}\,\top \equiv \top$
- $\mathsf{G}\,\bot \equiv \bot$
- $\varphi\,\mathsf{U}\,\top \equiv \top$
- $\varphi\,\mathsf{U}\,\bot \equiv \bot$
- $\varphi\,\mathsf{R}\,\top \equiv \top$
- $\varphi\,\mathsf{R}\,\bot \equiv \bot$
- $\mathsf{F}_{0,a}\,\top \equiv \top$
- $\mathsf{F}_{0,a}\,\bot \equiv \bot$
- $\mathsf{G}_{0,a}\,\top \equiv \top$
- $\mathsf{G}_{0,a}\,\bot \equiv \bot$
- $\varphi\,\mathsf{U}_{0,a}\,\top \equiv \top$
- $\varphi\,\mathsf{U}_{0,a}\,\bot \equiv \bot$
- $\varphi\,\mathsf{R}_{0,a}\,\top \equiv \top$
- $\varphi\,\mathsf{R}_{0,a}\,\bot \equiv \bot$
- $\mathsf{F}\,\mathsf{G}\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\mathsf{F}\,\varphi$
- $\mathsf{G}\,\mathsf{F}\,\mathsf{G}\,\varphi \equiv \mathsf{F}\,\mathsf{G}\,\varphi$
- $\mathsf{X}\,\varphi\,\mathsf{U}\,\mathsf{X}\,\psi \equiv \mathsf{X}\,(\varphi\,\mathsf{U}\,\psi)$
- $\mathsf{Y}\,\varphi\,\mathsf{R}\,\mathsf{Y}\,\psi \equiv \mathsf{R}\,(\varphi\,\mathsf{Y}\,\psi)$

Proof All equivalences follow directly from definition 2.16. □

2.5. Safety and liveness properties

Properties of reactive systems can be classified into two groups: *safety properties* and *liveness properties*. Safety properties express that "nothing bad ever happens" and liveness properties express that "something good eventually happens". A system has a safety property if none of its executions contains the bad feature, and it has a liveness property if all of its executions have the good trait. Following Stirling [Sti01], we formally define generic specification patterns for safety and liveness properties.

Definition 2.24 (Safety and liveness properties)

1. *Let φ be a "bad" feature. Then, a generic* ***safety property*** *is defined as* $\textbf{Safety}(\varphi) \stackrel{def}{=} \mathsf{G}\,\neg\varphi$

2. *Let φ a "good" feature. Then, a generic* ***liveness property*** *is defined as* $\textbf{Liveness}(\varphi) \stackrel{def}{=} \mathsf{F}\,\varphi$

Finally, we consider a number of typical examples for specification formulae in RTLTL. Some of them have been taken from [Fru02].

Example We start with basic examples for the generic properties introduced in definition 2.24. A simple safety property might express that all states of the system are free from errors. This results in the following property:

$$\mathsf{G}\,\neg\mathsf{error}.$$

A simple liveness property might express that from the current state, it is possible to reach a state where the system hast been started but is not ready. Since LTL is limited to single traces, we must also require that this state is included in the current execution trace:

$$\mathsf{F}\,(\mathsf{started} \wedge \neg\mathsf{ready}).$$

Safety properties that contain liveness properties as their prominent feature are called *fairness properties*. They express that "something good happens infinitely often". For instance, the following formula expresses the fact that a process is activated infinitely often during the regarded execution trace:

$$\mathsf{G}\,\mathsf{F}\,\mathsf{activation}.$$

Fairness properties are particularly important in model checking; for an introduction, consult [HR00] or [CGP99].

Now, we want to discuss property specification in three more realistic situations. In the first example, we want to model the requirement that the presence of an *error symptom* for a period of 10 events propagates an error during the next two events. In order to avoid vacous satisfaction of our property, we first check whether the error symptom is actually present in the execution trace to be checked:

$$\mathsf{F}\,\mathsf{error_symptom}.$$

If it is, then we check next whether it occurs for a period of 10 events, which is the precondition of raising an error:

$$\mathsf{F}\,\mathsf{G}_{0,10}\,\mathsf{error_symptom}.$$

Finally, we check the whole property:

$$\mathsf{G}\,(\mathsf{G}_{0,10}\,\mathsf{error_symptom} \rightarrow \mathsf{F}_{0,2}\,\mathsf{error}).$$

In the second example, we want to model the requirement that each request is followed by a grant within the next ten to twenty events. In RTLTL, this reads

$$\mathsf{G}\,(\mathsf{request} \rightarrow \mathsf{F}_{10,20}\,\mathsf{grant}).$$

It is easy to observe that finite traces never satisfy this property. However, the verification engineer in charge can rewrite this specification using the finite-trace semantics of the X operator, such that it gets the desired meaning. In doing so, the original specification is only checked on a trace prefix which is maximal with respect to the inclusion of time-bounded subformulae:

$$\mathsf{G}\,(\mathsf{X}_{20}\,\top \rightarrow (\mathsf{request} \rightarrow \mathsf{F}_{0,20}\,\mathsf{grant})).$$

In the third example, we want to model the requirement that the minimal distance between the first and the second signal is exactly 5 events. For this, we write

$$\mathsf{G}\,(\mathsf{first_signal} \rightarrow (\mathsf{G}_{0,4}\,\neg\mathsf{second_signal} \wedge \mathsf{X}_5\,\mathsf{second_signal})).$$

Hence, we believe that our semantics is truly universal and there is no need to think about stationary semantics. ◇

2.6. The truth checking problem

In this section, we formalise the problem of checking finite traces as a computability problem and state complexity results for LTL and RTLTL. We start with the definition of the truth checking problem for finite traces.

Definition 2.25 (The truth checking problem for finite traces)
The truth checking problem for finite traces, denoted $\mathbf{TCP_{fin}}$, *is defined as follows: Given a finite trace* ξ *and a formula* φ, *does* $\xi \models \varphi$ *hold?*

The best known upper bound for the complexity of truth checking with LTL specifications is that of the labelling algorithm for CTL model checking [CGP99]. Since CTL and LTL coincide for single paths, this method can also be used for checking the truth of an LTL formula.

Proposition 2.26 (Complexity of $\mathbf{TCP_{fin}}$ for LTL [MS03])
The truth checking problem for finite traces ξ *and LTL formulae* φ *can be solved in time* $\mathcal{O}(|\xi| \times |\varphi|)$.

As a direct consequence of this proposition and theorem 2.18, we get the following worst-case complexity result for RTLTL.

Corollary 2.27 (Complexity of $\mathbf{TCP_{fin}}$ for RTLTL)
The truth checking problem for finite traces ξ *and RTLTL formulae* φ *can be solved in time* $\mathcal{O}(|\xi| \times 2^{|\varphi|})$.

We conjecture that the stated upper bounds for both computability problems are essentially optimal.

3. Automata theory

Automata theory has proven to be a useful and important tool in formal verification. In 1982, Sherman [SPH82] showed that the set of execution traces that satisfy a given formula of propositional dynamic logic (PDL) can be recognised by a finite automaton on infinite words. In the following year, Wolper, Vardi, and Sistla [WVS83, VW94] presented the first translation procedure from linear-time temporal logic to finite automata for infinite words. In 1985, model checking was invented by contributions of Lichtenstein and Pnueli [LP85] and Clarke, Emerson, and Sistla [CES86]. Automata-theoretic model checking was first mentioned by Vardi and Wolper [VW86], and it was shown that, for each LTL formula φ, it is possible to construct a corresponding nondeterministic Büchi automaton $\mathcal{A}_\varphi$ that accepts precisely the traces satisfying φ. With the growing interest in light-weight formal methods, truth checking, that is, determining whether a trace satisfies some temporal property, is now becoming a significant alternative to full formal methods [Var97].

In the first section, we introduce finite automata for finite and infinite words and reason about their expressive power. Subsequently, different types of automata for infinite words, so-called Büchi automata are introduced. In the third section, a well-known translation procedure from LTL formulae into Büchi automata is presented. In the fourth section, we improve this method such that RTLTL formula can be translated into finite automata for finite traces. The actual verification algorithm is presented in the fifth section. Finally, we study the complexity of both the translation and the verification algorithms.

3.1. Finite automata for finite and infinite traces

In our approach, temporal specifications are represented by nondeterministic finite automata. First, we recall a couple of basic definitions.

Definition 3.1 (Words, languages, and operations)
*Let Σ be a finite set, usually referred to as **alphabet**.*

1. *A **word** is a finite sequence $a_0 \ldots a_n$ such that $a_i \in \Sigma$ for all $0 \leq i \leq n$.*
2. *An ω-**word** is an infinite sequence $a_0 a_1 \ldots$ such that $a_i \in \Sigma$ for all $i \geq 0$.*
3. *The **empty word** is denoted by ε.*
4. *The **concatenation** operation $\cdot$ on a word u and a word or ω-word v is defined as $u \cdot v \stackrel{def}{=} uv$.*
5. *A **language** is a set of words.*

6. *An ω-**language** is a set of ω-words.*

7. *The **star** operation on a language $\mathcal{L}$ is defined as follows:*

   - $\mathcal{L}^0 \stackrel{def}{=} \{\varepsilon\}$
   - $\mathcal{L}^{i+1} \stackrel{def}{=} \mathcal{L}^i \cdot \mathcal{L}$

Note that, although automata theory is originally concerned with words rather than traces, we will from now on only use the latter notion.

We are now able to define finite automata for finite [HMU01] and infinite [Tho97] traces. Remember that our notation requires traces to be nonempty, corresponding to our finite-trace semantics for LTL and RTLTL. We start with a generic definition of finite automata for infinite traces, which are commonly referred to as ω-automata.

Definition 3.2 (ω-automaton)

1. *An ω-**automaton** is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, A)$ where Q is the finite nonempty **set of states**, Σ is the finite **alphabet**, $\Delta \subseteq Q \times \Sigma \times Q$ is the **transition relation**, $I \subseteq Q$ is the **set of initial states**, and $A : Q^\omega \to \{true, false\}$ is the **acceptance component**.*

2. *An **execution** of $\mathcal{A}$ on an infinite trace $\xi = x_0, x_1, \ldots$ is an infinite sequence of states $\sigma = s_0, s_1, \ldots$ such that $s_0 \in I$ and $(s_i, x_i, s_{i+1}) \in \Delta$ for all $i \geq 0$.*

3. *An execution σ of $\mathcal{A}$ is **accepting** if $A(\sigma) = true$.*

4. *An ω-automaton is called **deterministic** if, for each $s \in Q$ and $a \in \Sigma$, $\{s' \mid (s, a, s') \in \Delta\} \leq 1$. Otherwise it is called **nondeterministic**.*

In fact, ω-automata are a generalisation of finite automata for infinite words with different acceptance components. The acceptance component is that part of an ω-automaton that checks a specific acceptance condition for

$$in(\sigma) = \{q \in Q \mid q \text{ occurs infinitely often in } \sigma\}.$$

Generally, four equivalent acceptance conditions are distinguished: Büchi condition, Muller condition, Rabin condition, and Streett condition [Tho97]. The corresponding ω-automata are named after the respective acceptance condition as Büchi automata, Muller automata, Rabin automata, and Streett automata. In the next section, we will focus on Büchi automata.

Similar to the previous definition, we define finite automata for finite traces. They have the same structure as finite automata for infinite traces, but a simpler acceptance condition.

Definition 3.3 (Nondeterministic finite automaton)

1. *A **nondeterministic finite automaton (NFA)** is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ where Q is the finite nonempty **set of states**, Σ is the finite **alphabet**, $\Delta \subseteq Q \times \Sigma \times Q$ is the **transition relation**, $I \subseteq Q$ is the **set of initial states**, and $F \subseteq Q$ is the **set of accepting states**.*

2. *An* ***execution*** *of $\mathcal{A}$ on a finite trace $\xi = x_0, \ldots, x_n$ is a sequence of states $\sigma = s_0, \ldots, s_{n+1}$ such that $s_0 \in I$ and $(s_i, x_i, s_{i+1}) \in \Delta$ for all $0 \leq i \leq n$.*

3. *An execution σ of $\mathcal{A}$ is* ***accepting*** *if $s_{n+1} \in F$.*

4. *A nondeterministic finite automaton is called* ***deterministic*** *if, for each $s \in Q$ and $a \in \Sigma$, $\{s' \mid (s, a, s') \in \Delta\} \leq 1$.*

In this work, we only consider nondeterministic finite automata, which are expressively equivalent, but more succinct, compared to their deterministic counterparts.

Definition 3.4 (Regular and ω-regular languages)
A language (ω-language) is called ***regular (ω-regular)*** *if it is accepted by a finite automaton. The language accepted by an automaton $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all traces for which $\mathcal{A}$ has an existing execution.*

In order to compare the expressive power of different automata types, we define regular expressions for finite traces and ω-regular expressions for infinite traces.

Definition 3.5 (Regular and ω-regular expressions)

1. *The sets of* ***regular expressions*** *and* ***ω-regular expressions*** *over a finite alphabet Σ are defined as follows:*

   a) *If $a \in \Sigma$, then a is a regular expression.*

   b) *If r and s are regular expressions, then (r), (rs), and $(r)^*$ are regular expressions.*

   c) *If r and s are regular expressions, then (rs^ω) is an ω-regular expression.*

   d) *If r and s are regular expressions, then $(r|s)$ is a regular expression.*

   e) *If r and s are ω-regular expressions, then $(r|s)$ is an ω-regular expression.*

   f) *If r is a regular expression, then it is an ω-regular expression.*

2. *If q and r are regular expressions and s and t are ω-regular expressions over Σ, then the following holds:*

   a) *$\mathcal{L}(a) = \{a\}$ for $a \in \Sigma$.*

   b) *$\mathcal{L}((r)) = \mathcal{L}(r)$.*

   c) *$\mathcal{L}((qr)) = \mathcal{L}(q)\mathcal{L}(r)$.*

   d) *$\mathcal{L}((r)^*) = \mathcal{L}(r)^*$.*

   e) *$\mathcal{L}((s|t)) = \mathcal{L}(s) \cup \mathcal{L}(t)$.*

   f) *$\mathcal{L}((rs^\omega)) = \mathcal{L}(r)\mathcal{L}(s)^\omega$.*

3. *A regular or ω-regular expression is* ***star-free*** *if it does not contain the star operator.*

For convenience, parantheses in regular expressions can be omitted, and the binding priorities of the connectives descend in the order ω, star, concatenation, alternative.

In order to reason about the expressiveness of different logics, it is important to describe which languages and ω-languages are *definable* by formulae of a particular logic. We formalise this notion as follows.

Definition 3.6 (Languages and ω-languages defined by a formula)
The language $\mathcal{L}(\varphi)$ and the ω-language $\mathcal{L}_\omega(\varphi)$ defined by an RTLTL formula φ are:

- $\mathcal{L}(\varphi) = \{\xi \in \Sigma^* \mid \xi \models \varphi\}$
- $\mathcal{L}_\omega(\varphi) = \{\xi \in \Sigma^\omega \mid \xi \models \varphi\}$

Finally, we now state important facts that describe the expressive power of LTL and finite automata for finite and infinite traces. We do not provide "real" proofs for them, but instead give directions to the appropriate references.

Proposition 3.7 (Expressiveness of LTL for infinite traces)
Let $\mathcal{L}$ be an ω-language. Then, the following properties are equivalent:

1. *$\mathcal{L}$ is definable in LTL.*
2. *$\mathcal{L}$ is definable by a star-free ω-regular expression.*
3. *$\mathcal{L}$ is recognisable by an LWAA[1].*

We sketch the proof: let $\mathcal{L}$ be an ω-language. Then, by Kamp [Kam68] and Thomas [Tho81], $\mathcal{L}$ is LTL-definable if and only if it is definable by a star-free ω-regular expression. By Löding and Thomas [LT00] this is if and only if $\mathcal{L}$ is LWAA-recognisable.

Proposition 3.8 (Expressiveness of finite automata for infinite traces)
Let $\mathcal{L}$ be an ω-language. Then, the following properties are equivalent:

1. *$\mathcal{L}$ is recognisable by a nondeterministic Büchi automaton (NBA).*
2. *$\mathcal{L}$ is definable by an ω-regular expression.*

We sketch the proof: let $\mathcal{L}$ be an ω-language. Then, by Büchi [Büc62], $\mathcal{L}$ is NBA-recognisable if and only if it is definable by an ω-regular expression.

Proposition 3.9 (Expressiveness of LTL for finite traces)
Let $\mathcal{L}$ be a language. Then, the following properties are equivalent:

1. *$\mathcal{L}$ is definable in LTL.*
2. *$\mathcal{L}$ is recognisable by an NFA.*

[1] A *linear weak alternating automaton (LWAA)* is an alternating finite automaton for infinite words with a linear transition relation. LWAA are weaker than ω-automata.

3. *$\mathcal{L}$ is definable by a regular expression.*

We sketch the proof: let $\mathcal{L}$ be a language. Then, by Büchi [Büc60], $\mathcal{L}$ is LTL-definable if and only if it is NFA-recognisable. By Kleene's theorem [HMU01], this is if and only if $\mathcal{L}$ is definable by a regular expression.

Hence, we can be convinced that NFA are the suitable automata-theoretic representation for LTL formulae. Nevertheless, we will continue to make use of standard algorithms and intermediate represenations that were originally designed for ω-automata.

3.2. Büchi automata

The first studied and now most popular automata-theoretic representation for temporal-logic properties are Büchi automata [Büc62]. In this section, we present Büchi automata with different but equivalent acceptance conditions.

Definition 3.10 (Nondeterministic Büchi automaton (NBA))
*A **nondeterministic Büchi automaton (NBA)** is an ω-automaton where $A(\sigma) = true$ if and only if $in(\sigma) \cup F \neq \emptyset$ for a designated set $F \subseteq Q$ of **final states**.*

Contrary to finite automata for finite words, nondeterministic Büchi automata have a greater expressive power than their deterministic counterparts.

When finite automata are generated from temporal-logic formulae, it is often more convenient to construct generalised Büchi automata first, which use a set of sets of final states for determining acceptance. The literature distinguishes state-based [GPVW95] and transition-based [GO01] generalised Büchi automata. Here, only state-based generalised Büchi automata are used, that is, those that have an acceptance condition in terms of states rather than transitions.

Definition 3.11 (Generalised Büchi automata (GBA))

1. *A **generalised Büchi automaton (GBA)** is a tuple $\mathcal{A} = (Q, \Delta, I, \mathcal{F})$ where Q is the finite nonempty **set of states**, $\Delta \subseteq Q \times Q$ is the **transition relation**, $I \subseteq Q$ is the **set of initial states**, and $\mathcal{F} \in 2^Q$ is the **set of sets of accepting states**.*

2. *An **execution** of $\mathcal{A}$ on an infinite trace $\xi = x_0, x_1, \ldots$ is an infinite sequence of states $\sigma = s_0, s_1, \ldots$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in \Delta$ for all $i \geq 0$.*

3. *An execution σ of $\mathcal{A}$ is **accepting** if, for each $F \in \mathcal{F}$, there is a state $s \in F$ that occurs infinitely often in σ.*

In order to recognise languages over a given alphabet Σ, we need to add labels to either states or transitions. In our case, we use a labelling function for states. We obtain labelled generalised Büchi automata.

Definition 3.12 (Labelled generalised Büchi automata (LGBA))

1. *A* ***labelled generalised Büchi automaton (LGBA)*** *is a tuple* $\mathcal{A} = (Q, \Sigma, \Delta, I, \mathcal{F}, \mathcal{L})$ *where* Q *is the finite nonempty* ***set of states****,* Σ *is the finite* ***alphabet****,* $\Delta \subseteq Q \times \Sigma \times Q$ *is the* ***transition relation****,* $I \subseteq Q$ *is the* ***set of initial states****,* $\mathcal{F} \in 2^Q$ *is the* ***set of sets of accepting states****, and* $\mathcal{L} : Q \to 2^\Sigma$ *is the* ***labelling function****.*

2. *An* ***execution*** *of* $\mathcal{A}$ *on an infinite trace* $\xi = x_0, x_1, \ldots$ *is an infinite sequence of states* $\sigma = s_0, s_1, \ldots$ *such that* $s_0 \in I$ *and* $(s_i, x_i, s_{i+1}) \in \Delta$ *for all* $i \geq 0$*.*

3. *An execution* $\sigma = s_0, s_1, \ldots$ *of* $\mathcal{A}$ *on an infinite trace* $\xi = x_0, x_1, \ldots$ *is* ***accepting*** *if, for each* $i \geq 0$*,* $x_i \in \mathcal{L}(s_i)$*, and, for each* $F \in \mathcal{F}$*, there is a state* $s \in F$ *that occurs infinitely often in* σ*.*

3.3. Translating LTL formulae into Büchi automata

The first approach to the translation of LTL formulae into Büchi automata that was not worst-case was developed in 1995 by Gerth, Peled, Vardi, and Wolper [GPVW95] and later improved by Daniele, Giunchiglia, and Vardi [DGV99]. Their algorithm translates a given LTL formula into a GBA state graph, then into an LGBA, and finally into an NBA. An important feature of this method is that it can be used *on-the-fly*, that is, the automaton construction is guided by the trace traversal such that only those parts of the automaton currently needed are constructed and stored in memory.

The main concept of the algorithm is a tableau procedure that constructs a GBA state graph by successively expanding a single node that just contains the LTL formula to be translated. This formula must be in negation normal form and must not contain $\rightarrow$, F, or G operators (note that all LTL formulae can be translated into equivalent formulae of this form).

Nodes of the GBA state graph consist of five fields:

1. *index* - a unique identifier.

2. *incoming* - the set of nodes that lead to this node.

3. *new* - the set of formulae that must hold in this node and have not yet been processed.

4. *old* - the set of formulae that must hold in this node and have already been processed.

5. *next* - the set of formulae that must hold in all immediate successors of this node.

A node is expanded by heuristically taking an unprocessed formula out of its *new* field and then, depending on the type of this formula, deterministically applying a tableau rule to the node. By this, the formula is decomposed into new but simpler formulae. The tableau rules can be distinguished into such

Formula	**Changes to the node**
$\top \in \mathcal{P}$	no changes
$\bot \in \mathcal{P}$	discard node
$p \in \mathcal{P}$	$old := old \cup \{p\}$, if $\neg p \notin old$ discard node, otherwise
$\neg p$, $p \in \mathcal{P}$	$old := old \cup \{\neg p\}$, if $p \notin old$ discard node, otherwise
$\varphi \wedge \psi$	$old := old \cup \{\varphi \wedge \psi\}$ $new := new \cup (\{\varphi, \psi\} - old)$
$\mathsf{X}\,\varphi$	$old := old \cup \{\mathsf{X}\,\varphi\}$ $next := next \cup \{\varphi\}$

Figure 3.1.: Non-splitting tableau rules for LTL to GBA translation.

Formula	**Changes to first node**	**Changes to second node**
$\varphi \vee \psi$	$old := old \cup \{\varphi \vee \psi\}$ $new := new \cup (\{\varphi\} - old)$	$old := old \cup \{\varphi \vee \psi\}$ $new := new \cup (\{\psi\} - old)$
$\varphi\,\mathsf{U}\,\psi$	$old := old \cup \{\varphi\,\mathsf{U}\,\psi\}$ $new := new \cup (\{\varphi\} - old)$ $next := next \cup \{\varphi\,\mathsf{U}\,\psi\}$	$old := old \cup \{\varphi\,\mathsf{U}\,\psi\}$ $new := new \cup (\{\psi\} - old)$
$\varphi\,\mathsf{R}\,\psi$	$old := old \cup \{\varphi\,\mathsf{R}\,\psi\}$ $new := new \cup (\{\psi\} - old)$ $next := next \cup \{\varphi\,\mathsf{R}\,\psi\}$	$old := old \cup \{\varphi\,\mathsf{R}\,\psi\}$ $new := new \cup (\{\varphi, \psi\} - old)$

Figure 3.2.: Splitting tableau rules for LTL to GBA translation.

that modify or completely discard a node (see figure 3.1) and such that split a node into exactly two modified replacing nodes (see figure 3.2).

A node is completely expanded if its *new* field is empty. The algorithm maintains completely and incompletely expanded nodes as two seperate sets, whereof the set of completely expanded nodes defines the state graph. Once a node is completely expanded, is is checked whether there is already an equivalent node in the state graph (two nodes are equivalent if they have equal *next* and *old* fields). If that is the case, both nodes are merged, by merging their *incoming* fields. If it is not, the node is added to the state graph and a new node is created which has this node in *incoming*, this node's *new* field as *next* field and all other fields empty.

The algorithm starts with exactly one node that contains precisely the formula to be translated in *new* and "init" in *incoming*. The algorithm terminates when all nodes are completely expanded and returns the state graph.

Now, a GBA $\mathcal{A} = (Q, \Delta, I, \mathcal{F})$ can be obtained from the complete state graph. The acceptance condition ensures that all U formulae are eventually fulfilled.

- The set Q of states is the set of all graph nodes.
- The transition relation is $\Delta := \{(s,t) \in Q \times Q \mid s \in \mathit{incoming}(t)\}$.

- The set of initial states is $I := \{s_0\}$.
- The set $\mathcal{F}$ of sets of final states contains, for each subformula of type $\varphi \mathsf{U} \psi$, one set F of accepting states such that $s \in F$ if and only if $\psi \in old(q)$ or $\varphi \mathsf{U} \psi \notin old(q)$.

In the following step, this GBA is extended to an LGBA $\mathcal{A}' = (Q, \Sigma, \Delta, I, \mathcal{F}, \mathcal{L})$ by adding a labelling function. Thereby, each state s is labelled with all sets in $2^{\mathcal{P}}$ that are compatible with $old(s)$.

- The alphabet is $\Sigma := 2^{\mathcal{P}}$.
- The labelling of a state $s \in Q$ is

$$\mathcal{L}(s) := \{x \mid x \subseteq \mathcal{P} \wedge x \supseteq pos(s) \wedge x \cap neg(s) = \emptyset\}$$

  where $pos(s) \stackrel{def}{=} old(s) \cap \mathcal{P}$ and $neg(s) \stackrel{def}{=} \{\mu \mid \neg\mu \in old(s) \wedge \mu \in \mathcal{P}\}$.

For the sake of model checking, there exist efficient methods for translating such LGBA into NBA [GO01, Tau03]. In general, there is the following upper bound for the complexity of LTL to NBA translations.

Proposition 3.13 (LTL to NBA translation [MSS88, VW94])
Given an LTL formula φ, one can build (in time $2^{\mathcal{O}(|\varphi|)}$) a nondeterministic Büchi automaton $\mathcal{A}_\varphi = (Q, \Sigma, \Delta, I, F)$ where $|Q| = 2^{\mathcal{O}(|\varphi|)}$ and $\Sigma = 2^{\mathcal{P}}$ such that $\mathcal{L}_\omega(\mathcal{A}_\varphi)$ is exactly the set of traces in which the formula φ is true.

3.4. Translating RTLTL formulae into NFA

We extend the algorithm of Gerth, Peled, Vardi and Wolper [GPVW95] such that it can use arbitrary RTLTL formulae as input and such that the resulting automata are just NFA. Consequently, the used specification formulae are more succinct and thus can be parsed more efficiently. Also, the obtained automata have a simpler acceptance condition, what makes them more intuitive and leads to more efficient trace traversal. Simultaneously with these simplifications, the designation of final states in the state graph becomes slightly more complicated.

In order to process RTLTL formulae, we impose additional requirements on the formula to be translated. This formula must be in negation normal form and must not contain $\rightarrow$, F, $\mathsf{F}_{a,b}$, G, or $\mathsf{G}_{a,b}$ operators (note that all RTLTL formulae can be translated into equivalent formulae of this form).

We keep the structure of nodes and the operational principles of the original algorithm but modify the used tableau rules, such that all RTLTL operators can be recognised. We also distinguish tableau rules that update or discard nodes (see figure 3.3) from such that split nodes (see figure 3.4).

After termination of our algorithm, we obtain an NFA $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ from the complete state graph by labelling states with sets of atomic propositions and designating a set of final states as follows:

- The set Q of states is the set of all graph nodes together with a new node s_0 representing *init*.

Formula	**Changes to the node**
$\top \in \mathcal{P}$	no changes
$\bot \in \mathcal{P}$	discard node
$p \in \mathcal{P}$	$old := old \cup \{p\}$, if $\neg p \notin old$ discard node, otherwise
$\neg p$ $(p \in \mathcal{P})$	$old := old \cup \{\neg p\}$, if $p \notin old$ discard node, otherwise
$\varphi \wedge \psi$	$old := old \cup \{\varphi \wedge \psi\}$ $new := new \cup (\{\varphi, \psi\} - old)$
$\mathsf{X}\,\varphi$	$old := old \cup \{\mathsf{X}\,\varphi\}$ $next := next \cup \{\varphi\}$
$\mathsf{X}_0\,\varphi$	$new := new \cup (\{\varphi\} - old)$
$\mathsf{X}_{a+1}\,\varphi$	$old := old \cup \{\mathsf{X}_{a+1}\,\varphi\}$ $next := next \cup \{\mathsf{X}_a\,\varphi\}$
$\mathsf{Y}\,\varphi$	$old := old \cup \{\mathsf{Y}\,\varphi\}$ $next := next \cup \{\varphi\}$
$\mathsf{Y}_0\,\varphi$	$new := new \cup (\{\varphi\} - old)$
$\mathsf{Y}_{a+1}\,\varphi$	$old := old \cup \{\mathsf{Y}_{a+1}\,\varphi\}$ $next := next \cup \{\mathsf{Y}_a\,\varphi\}$
$\varphi\,\mathsf{U}_{0,0}\,\psi$	$new := new \cup (\{\psi\} - old)$
$\varphi\,\mathsf{U}_{a+1,b+1}\,\psi$	$old := old \cup \{\varphi\,\mathsf{U}_{a+1,b+1}\,\psi\}$ $new := new \cup (\{\varphi\} - old)$ $next := next \cup \{\varphi\,\mathsf{U}_{a,b}\,\psi\}$
$\varphi\,\mathsf{R}_{0,0}\,\psi$	$new := new \cup (\{\psi\} - old)$

Figure 3.3.: Non-splitting tableau rules for RTLTL to NFA translation.

- The labelling of a state $s \in Q$ is

$$\mathcal{L}(s) := \{x \mid x \subseteq \mathcal{P} \wedge x \supseteq pos(s) \wedge x \cap neg(s) = \emptyset\}$$

  where $pos(s) \stackrel{def}{=} old(s) \cap \mathcal{P}$ and $neg(s) \stackrel{def}{=} \{\mu \mid \neg\mu \in old(s) \wedge \mu \in \mathcal{P}\}$.

- The alphabet is $\Sigma := 2^{\mathcal{P}}$.
- The transition relation is

$$\Delta := \{(s, a, t) \in Q \times \Sigma \times Q \mid s \in incoming(t) \wedge a \in \mathcal{L}(t)\}.$$

- The set of initial states is $I := \{q \in Q \mid init \in incoming(q)\}$.
- The set F of final states is the set of those states $s \in Q$ such that the following holds:
  - $old(q)$ does not contain any formula of type $\mathsf{X}\,\varphi$, $\mathsf{X}_{a+1}\,\varphi$, or $\varphi\,\mathsf{U}_{a+1,b+1}\,\psi$.
  - If $old(q)$ contains a formula of type $\varphi\,\mathsf{U}\,\psi$ or $\varphi\,\mathsf{U}_{0,b}\,\psi$, then it also contains the formula ψ.

Formula	Changes to first node	Changes to second node
$\varphi \vee \psi$	$old := old \cup \{\varphi \vee \psi\}$ $new := new \cup (\{\varphi\} - old)$	$old := old \cup \{\varphi \vee \psi\}$ $new := new \cup (\{\psi\} - old)$
$\varphi \mathsf{U} \psi$	$old := old \cup \{\varphi \mathsf{U} \psi\}$ $new := new \cup (\{\varphi\} - old)$ $next := next \cup \{\varphi \mathsf{U} \psi\}$	$old := old \cup \{\varphi \mathsf{U} \psi\}$ $new := new \cup (\{\psi\} - old)$
$\varphi \mathsf{U}_{0,b+1} \psi$	$old := old \cup \{\varphi \mathsf{U}_{0,b+1} \psi\}$ $new := new \cup (\{\varphi\} - old)$ $next := next \cup \{\varphi \mathsf{U}_{0,b} \psi\}$	$old := old \cup \{\varphi \mathsf{U}_{0,b+1} \psi\}$ $new := new \cup (\{\psi\} - old)$
$\varphi \mathsf{R} \psi$	$old := old \cup \{\varphi \mathsf{R} \psi\}$ $new := new \cup (\{\psi\} - old)$ $next := next \cup \{\varphi \mathsf{R} \psi\}$	$old := old \cup \{\varphi \mathsf{R} \psi\}$ $new := new \cup (\{\varphi, \psi\} - old)$
$\varphi \mathsf{R}_{0,b+1} \psi$	$old := old \cup \{\varphi \mathsf{R}_{0,b+1} \psi\}$ $new := new \cup (\{\psi\} - old)$ $next := next \cup \{\varphi \mathsf{R}_{0,b} \psi\}$	$old := old \cup \{\varphi \mathsf{R}_{0,b+1} \psi\}$ $new := new \cup (\{\varphi, \psi\} - old)$
$\varphi \mathsf{R}_{a+1,b+1} \psi$	$old := old \cup \{\varphi \mathsf{R}_{a+1,b+1} \psi\}$ $next := next \cup \{\varphi \mathsf{R}_{a,b} \psi\}$	$old := old \cup \{\varphi \mathsf{R}_{a+1,b+1} \psi\}$ $new := new \cup (\{\varphi\} - old)$

Figure 3.4.: Splitting tableau rules for RTLTL to NFA translation.

For our algorithm, we have extended the original GBA acceptance condition (for infinite traces), used by Gerth, Peled, Vardi, and Wolper [GPVW95], into an NFA acceptance condition (for finite traces). The original acceptance condition adds, for each subformula of type $\varphi \mathsf{U} \psi$, one set F to the set $\mathcal{F}$ of sets of accepting states such that $s \in F$ if and only if $\psi \in old(q)$ or $\varphi \mathsf{U} \psi \notin old(q)$. Since our logic also contains the X operator, we designate all states that do not contain X or X_{a+1} formulae or unfulfilled U or $\mathsf{U}_{a,b}$ formulae in their *old* field as final. Since in our approach the next field is also used for next-state conditions resulting from weak operators like Y, R, and their time-bounded variants, we cannot use the simplified acceptance condition proposed by Giannakopoulou and Havelund [GH01], who, for a translation of LTL-X formulae into NFA, designate all states that do not contain U formulae in their *next* field as final.

We now present proofs for termination and correctness of our algorithm. For this, we follow [GPVW95]. Let $Old(s)$ and $Next(s)$ denote the values of $old(s)$ and $next(s)$ at the point where the node s is completely expanded.

Theorem 3.14 (Termination)
Given an RTLTL formula in negation normal form, the algorithm eventually returns an NFA.

Proof The algorithm starts with a single one node. Newly generated nodes only contain formulae generated from other nodes' *new* and *next* fields, and only finitely many of them. Let τ be a ranking function that measures the complexity of a formula (see figure 3.5). We observe the following: the application of a tableau rule to a formula of rank 0 immediately removes the formula from the *new* field without adding new formulae to the node's *new* and *next* fields. Moreover, each application of a tableau rule removes a formula from the *new* field, while adding only finitely many formulae of lower rank. Also,

Formula φ	**Rank** $\tau(\varphi)$
$\top, \bot, p, \neg p \ (p \in \mathcal{P})$	0
$\psi \vee \mu, \psi \wedge \mu, \psi \,\mathsf{U}\, \mu, \psi \,\mathsf{R}\, \mu$	$1 + max\{\tau(\psi), \tau(\mu)\}$
$\mathsf{X}\,\psi, \mathsf{Y}\,\psi$	$1 + \tau(\psi)$
$\mathsf{X}_a\,\psi, \mathsf{Y}_a\,\psi$	$1 + a + \tau(\psi)$
$\psi \,\mathsf{U}_{a,b}\, \mu, \psi \,\mathsf{R}_{a,b}\, \mu$	$1 + a + b + max\{\tau(\psi), \tau(\mu)\}$

Figure 3.5.: A ranking function for RTLTL formulae in negation normal form.

by generating new nodes from completely processed nodes' *next* fields, no new formulae are introduced (all formulae come from the old node's *next* field). Therefore, by the principle of well-founded induction, the algorithm eventually terminates. □

Lemma 3.15

1. *When a node s is updated to become a new node s', the following holds:*

$$\left(\bigwedge old(s) \wedge \bigwedge new(s) \wedge \mathsf{X} \bigwedge next(s)\right) \longleftrightarrow$$
$$\left(\bigwedge old(s') \wedge \bigwedge new(s') \wedge \mathsf{X} \bigwedge next(s')\right).$$

2. *When a node s is split into nodes s_1, s_2 the following holds:*

$$\left(\bigwedge old(s) \wedge \bigwedge new(s) \wedge \mathsf{X} \bigwedge next(s)\right) \longleftrightarrow$$
$$\left(\left(\left(\bigwedge old(s_1) \wedge \bigwedge new(s_1) \wedge \mathsf{X} \bigwedge next(s_1)\right) \vee\right.\right.$$
$$\left.\left.\left(\bigwedge old(s_2) \wedge \bigwedge new(s_2) \wedge \mathsf{X} \bigwedge next(s_2)\right)\right)\right).$$

Proof This follows directly from the tableau rules and the definition of LTL. □

Definition 3.16 (rooted, same-time descendant)

1. *A node s is* ***rooted*** *if it is the initial node with which the tableau construction started or if it has been created from a completed node t by setting $new(s) := next(t)$.*

2. *A node r is a* ***same-time descendant*** *of a node s if it has been created from s by successive application of tableau rules.*

Lemma 3.17

Let s be a rooted node and $t_1, \ldots, t_n$ be all its same-time descendant nodes. Let Ξ be the set of formulae that are in $new(s)$ when it is created.

1. *Then,*

$$\bigwedge \Xi \longleftrightarrow \bigvee_{1 \le i \le n} \left(\bigwedge Old(t_i) \wedge \mathsf{X} \bigwedge Next(t_i)\right).$$

2. *Let ξ be a finite trace. If $\xi \models \bigvee_{1 \leq i \leq n}(\bigwedge Old(t_i) \wedge \mathsf{X} \bigwedge Next(t_i))$, then there exists some $1 \leq i \leq n$ such that the following holds:*

   a) *$\xi \models \bigwedge Old(t_i) \wedge \mathsf{X} \bigwedge Next(t_i)$.*

   b) *For each $\mu \,\mathsf{U}\, \eta \in Old(t_i)$ and for each $\mu \,\mathsf{U}_{a,b}\, \eta \in Old(t_i)$, if $\xi \models \eta$, then $\eta \in Old(t_i)$.*

   c) *For each $\mathsf{X}\, \psi \in Old(t_i)$, we have $\xi^1 \neq \varepsilon$, and, for each $\mathsf{X}_a\, \psi \in Old(t_i)$, we have $\xi^a \neq \varepsilon$.*

Proof

1. This is an immediate consequence of Lemma 3.15.

2. Suppose that $\xi \models \bigvee_{1 \leq i \leq n}(\bigwedge Old(t_i) \wedge \mathsf{X} \bigwedge Next(t_i))$.

   a) This is trivial.

   b) Suppose that $\mu \,\mathsf{U}\, \eta \in Old(t_i)$. By the U-rule, a node s such that $\mu \,\mathsf{U}\, \eta \in Old(s)$ has two same-time descendant nodes s_1 and s_2 such that $\mu \in Old(s_1)$ and $\eta \in Old(s_2)$. Moreover, by Lemma 3.15, every node containing a U formula in old is equivalent to some same-time descendant node that has either μ or η in *Old*. By (a), we have

   $$\begin{aligned} & \xi \models \bigwedge (Old(t_i) \cup \{\mu\}) \wedge \mathsf{X} \left(\bigwedge (Next(t_i) \cup \{\mu \,\mathsf{U}\, \eta\})\right) \\ \text{or} \quad & \xi \models \bigwedge (Old(t_i) \cup \{\eta\}) \wedge \mathsf{X} \bigwedge Next(t_i). \end{aligned}$$

   Hence, if $\xi \models \eta$, then $\eta \in Old(t_i)$. The case $\mathsf{U}_{a,b}\, \psi$ is completely analogue.

   c) Suppose that $\mathsf{X}\, \psi \in Old(t_i)$. By (a), we have $\xi \models \mathsf{X}\, \psi$. From the semantics of X it follows that $\xi^1 \neq \varepsilon$. The case $\mathsf{X}_a\, \psi$ is completely analogue. □

Lemma 3.18
Let ξ be a finite trace such that $\xi \models \bigwedge Old(s) \wedge \mathsf{X} \bigwedge Next(s)$.

1. *Then, there exists a transition from s to s' in $\mathcal{A}$ such that $\xi^1 \models \bigwedge Old(s') \wedge \mathsf{X} \bigwedge Next(s')$.*

2. *Let $\Gamma = \{\eta \mid \mu \,\mathsf{U}\, \eta \in Old(s) \wedge \eta \notin Old(s) \wedge \xi^1 \models \eta\}$. Then, there exists a transition from s to s' such that $\Gamma \subseteq Old(s')$.*

3. *For each $\mathsf{X}\, \psi \in Old(t_i)$, we have $\xi^1 \neq \varepsilon$, and, for each $\mathsf{X}_a\, \psi \in Old(t_i)$, we have $\xi^a \neq \varepsilon$.*

Proof When a node s has been expanded completely, a node s' with $new(s') = Next(s)$ is generated. Then, Lemma 3.17 guarantees that a successor as required exists. □

Lemma 3.19
Let $\mathcal{A}$ be the NFA constructed from the RTLTL formula φ. For every direct successor s of the initial state, we have $\varphi' \in Old(s)$ for some $\varphi' \equiv \varphi$.

Proof Since $\mathcal{A}$ exists, the starting node has not been discarded, and thus $\varphi \neq \bot$. All tableau rules add φ or an equivalent formula to *old*. $\square$

Lemma 3.20
Let $\mathcal{A}$ be the NFA constructed from the RTLTL formula φ. Then,

$$\varphi \leftrightarrow \bigvee_{s \in I, (s,s') \in \Delta} (\bigwedge Old(s') \wedge \mathsf{X} \bigwedge Next(s')).$$

Proof Set $\Xi := \{\varphi\}$ and apply Lemma 3.17. $\square$

Lemma 3.21
Let $\mathcal{A}$ be the NFA constructed from the RTLTL formula φ and $\xi = x_0, \ldots, x_n$ a finite trace. If there exists an accepting execution $\sigma = s_0, \ldots, s_{n+1}$ of $\mathcal{A}$ on ξ such that s_0 is taken to be an initial state of $\mathcal{A}$, then $\xi \vDash \bigwedge Old(s_1)$.

Proof Let $\varphi \in Old(s_1)$. We prove this lemma by structural induction on φ. We have $\varphi \neq \bot$, since there is not tableau rule that adds $\bot$ to *old*. Also, $\xi \vDash \top$.

1. Be $\varphi = p$ for $p \in \mathcal{P}$. Since σ is an execution on ξ, we have $(s_0, x_0, s_1) \in \Delta$, and, by the definition of Δ, $x_0 \in \mathcal{L}(s_1)$. Since $p \in Old(s_1)$, the tableau rules do not allow $\neg p \in Old(s_1)$, and therefore $p \in pos(s_1)$. By the definition of $\mathcal{L}$, we have $x_0 \supseteq pos(s_1)$, thus $p \in x_0$. Hence, $\xi \vDash p$.

2. Be $\varphi = \neg p$ for $p \in \mathcal{P}$. Since σ is an execution on ξ, we have $(s_0, x_0, s_1) \in \Delta$, and, by the definition of Δ, $x_0 \in \mathcal{L}(s_1)$. Since $\neg p \in Old(s_1)$, the tableau rules do not allow $p \in Old(s_1)$, and therefore $p \in neg(s_1)$. By the definition of $\mathcal{L}$, we have $x_0 \cap neg(s_1) = \emptyset$, that is $p \notin x_0$. Hence, $\xi \nvDash p$, that is $\xi \vDash \neg p$.

3. Be $\varphi = \mathsf{X}\,\psi$. Then, $\mathsf{X}\,\psi \in Old(s_1)$, and, since σ is accepting, $\sigma^2 \neq \varepsilon$ and thus $\xi^1 \neq \varepsilon$. Since σ is an accepting execution on ξ, also σ^1 is an accepting execution on ξ^1. Since $\mathsf{X}\,\psi \in Old(s_1)$, by the X-rule, we have $\psi \in Next(s_1)$ and thus $\psi \in Old(s_2)$. Then, by the induction hypothesis, $\xi^1 \vDash \psi$. Hence, $\xi \vDash \mathsf{X}\,\psi$.

4. Be $\varphi = \mathsf{Y}\,\psi$. If $\xi^1 = \varepsilon$, then $\xi \vDash \mathsf{Y}\,\psi$ and we are done. So be $\xi^1 \neq \varepsilon$. Since σ is an accepting execution on ξ, also σ^1 is an accepting execution on ξ^1. Since $\mathsf{Y}\,\psi \in Old(s_1)$, by the Y-rule, we have $\psi \in Next(s_1)$ and thus $\psi \in Old(s_2)$. Then, by the induction hypothesis, $\xi^1 \vDash \psi$. Hence, $\xi \vDash \mathsf{Y}\,\psi$.

5. Be $\varphi = \mu \,\mathsf{R}\, \eta$. Since $\mu \,\mathsf{R}\, \eta \in Old(s_1)$, by the R-rule, we have, for all $1 \leq i \leq n+1$, $\{\mu \,\mathsf{R}\, \eta, \eta\} \subseteq Old(s_i)$ or $\{\mu, \eta\} \subseteq Old(s_j)$ for some $1 \leq j < i$. Then, by the induction hypothesis, we have, for all $0 \leq i \leq n$, $\xi^i \vDash \eta$ or $\xi^j \vDash \mu$ for some $0 \leq j < i$. Hence, $\xi \vDash \mu \,\mathsf{R}\, \eta$.

6. Be $\varphi = \mu \,\mathsf{U}\, \eta$. Since $\mu \,\mathsf{U}\, \eta \in Old(s_1)$, by the U-rule, we have, for all $1 \leq i \leq n+1$, $\{\mu \,\mathsf{U}\, \eta, \mu\} \subseteq Old(s_i)$, or $\eta \in Old(s_i)$ for some $1 \leq i \leq n+1$ and $\mu \in Old(s_j)$ for all $1 \leq j < i$. Then, by the induction hypothesis, we

have, for all $0 \leq i \leq n$, $\xi^i \models \mu$, or $\xi^i \models \eta$ for some $0 \leq i \leq n$ and $\xi^j \models \mu$ for all $0 \leq j < i$. Since σ is accepting, we have the latter case. Hence, $\xi \models \mu \, \mathsf{U} \, \eta$.

7. The cases $\varphi = \mathsf{X}_a \, \psi$, $\varphi = \mathsf{Y}_a \, \psi$, $\varphi = \mu \, \mathsf{U}_{a,b} \, \eta$, and $\varphi = \mu \, \mathsf{R}_{a,b} \, \eta$ are completely analogue. □

Theorem 3.22 (Soundness)
Let $\mathcal{A}$ be the NFA constructed from the RTLTL formula φ and ξ be a finite trace. If $\mathcal{A}$ has an accepting execution $\sigma = s_0, \ldots, s_n$ on ξ, then $\xi \models \varphi$.

Proof From Lemma 3.21 it follows that $\xi \models \bigwedge Old(s_0)$. By Lemma 3.19, we have $\varphi \in Old(s_0)$. Hence, $\xi \models \varphi$. □

Theorem 3.23 (Completeness)
Let $\mathcal{A}$ be the NFA constructed from the RTLTL formula φ and $\xi = x_0, \ldots, x_n$ be a finite trace. If $\xi \models \varphi$, then $\mathcal{A}$ has an accepting execution $\sigma = s_0, \ldots, s_{n+1}$ on ξ.

Proof By Lemma 3.20, there exists a node $s_1 \in I$ such that $\xi \models \bigwedge Old(s_1) \land \mathsf{X} \bigwedge Next(s_1)$. Now, one can construct the execution σ by repeatedly using Lemma 3.20. Namely, if $\xi^i \models \bigwedge Old(s_{i+1}) \land \mathsf{X} \bigwedge Next(s_{i+1})$, then choose s_{i+2} to be a successor of s_{i+1} with $\xi^{i+1} \models \bigwedge Old(s_{i+2}) \land \mathsf{X} \bigwedge Next(s_{i+2})$. Hence, Lemma 3.18 guarantees that we can choose each s_{i+2} such that σ is accepting. □

Finally, we can establish the following total correctness property, which is a direct consequence of the theorems 3.14, 3.22, and 3.23.

Corollary 3.24 (Total Correctness)
Given an RTLTL formula φ in negation normal form, the above algorithm returns an NFA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\varphi)$.

3.5. Checking finite traces

With our knowledge about finite traces, temporal logics for real time, and formula to automata translations, we now compose the desired runtime verification algorithm.

3.5.1. The basic algorithm

The basic algorithm takes an RTLTL formula φ and a finite trace ξ as its arguments and then proceeds in three stages. First, φ is transformed into an equivalent RTLTL formula φ' in negation normal form. Then, an NFA $\mathcal{A}$ that accepts precisely the traces satisfying φ' is computed. Finally, the truth checking problem for ξ and φ is being solved using classic search algorithms.

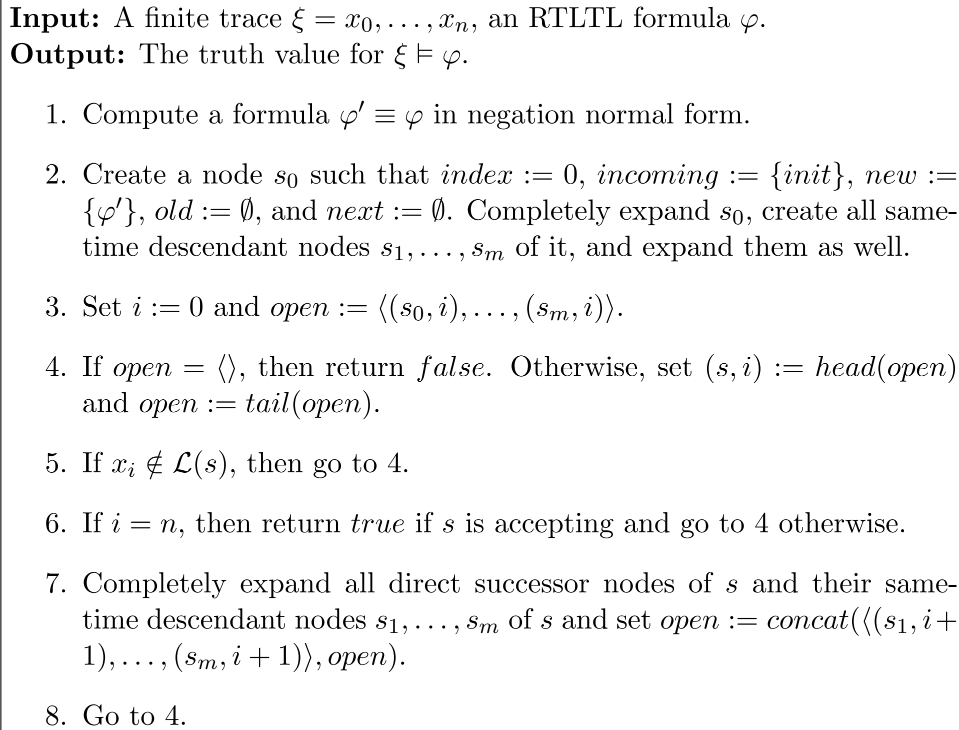
However, depending on the application context, different search strategies are suitable. Both forward and backward depth-first search are favourable as

long as the trace has been pre-computed and stored, while forward depth-first search is the only advisable method for online trace traversal. Also, as we will see later, only forward depth-first search can be used on-the-fly. A comparison of search strategies for checking finite traces with automata can be found in Finkbeiner and Sipma's paper [FS04].

In our case, traces can be recorded in the original system or in a test bench: in other words, they are either generated from a test car run or from measuring data that is obtained by simulation in a hardware-in-the-loop framework. Traces can either be computed once and stored for later processing, or computed and traversed on-the-fly.

3.5.2. On-the-fly operation

The basic algorithm can easily be executed on-the-fly when depth-first search is used. We follow the general approach presented by Courcoubetis, Wolper, Vardi, and Yannakakis [CVWY92] for on-the-fly emptiness checking for Büchi automata. Similar to their method, which relies on a nested depth-first search in two levels, we integrate the node expansion procedure within the search (see figure 3.3).



Input: A finite trace $\xi = x_0, \ldots, x_n$, an RTLTL formula φ.
Output: The truth value for $\xi \models \varphi$.

1. Compute a formula $\varphi' \equiv \varphi$ in negation normal form.
2. Create a node s_0 such that $\mathit{index} := 0$, $\mathit{incoming} := \{\mathit{init}\}$, $\mathit{new} := \{\varphi'\}$, $\mathit{old} := \emptyset$, and $\mathit{next} := \emptyset$. Completely expand s_0, create all same-time descendant nodes $s_1, \ldots, s_m$ of it, and expand them as well.
3. Set $i := 0$ and $\mathit{open} := \langle (s_0, i), \ldots, (s_m, i) \rangle$.
4. If $\mathit{open} = \langle\rangle$, then return $false$. Otherwise, set $(s, i) := \mathit{head}(\mathit{open})$ and $\mathit{open} := \mathit{tail}(\mathit{open})$.
5. If $x_i \notin \mathcal{L}(s)$, then go to 4.
6. If $i = n$, then return true if s is accepting and go to 4 otherwise.
7. Completely expand all direct successor nodes of s and their same-time descendant nodes $s_1, \ldots, s_m$ of s and set $\mathit{open} := \mathit{concat}(\langle (s_1, i+1), \ldots, (s_m, i+1) \rangle, \mathit{open})$.
8. Go to 4.

Figure 3.6.: On-the-fly algorithm for RTLTL to NFA translation.

Our method works as follows: first, the input formula is translated into negation normal form (step 1). Then, the starting node for the tableau and all its same-time descendant nodes are being created and expanded (step 2). After that, the trace index i is being initialised and, for each of the created nodes, a

pair consisting of the node and the index of the first event is added to the list of open pairs (step 3). If there are no open pairs, the algorithm terminates, returning $false$, since no accepting execution of the trace has been found; otherwise, the next open node and its corresponding trace index are taken from the list (step 4). If the respective trace event is a valid label for the current state, the transition from the previous state (or from s_0 for states created in step 2) to the current state is enabled for this event; otherwise the current state is discarded and a new one is chosen in step 4 (step 5). If the trace index equals n, that is, if the regarded event was the last one of the trace, the algorithm terminates returning true if s satisfies the acceptance condition and backtracking at step 4 otherwise (step 6). If the trace index is less than n, the transition to this node is enabled and there are unprocessed events until the end of the trace; then, all direct successor nodes of the current node and all their same-time descendant nodes are being created and, together with the index of the next event, added to the list of open pairs (step 7). Finally, the loop is started again in step 4 (step 8).

We end this section with the mandatory termination and correctness proofs.

Theorem 3.25 (Termination)
Given a finite trace and an RTLTL formula in negation normal form, the algorithm eventually returns true or $false$.

Proof Since the on-the-fly algorithm uses the same tableau rules as the classic one, each node has only finitely many same-time descendants. Once the algorithm has been initialised by steps 1 to 3, *open* can only be extended by new elements in step 7. When this happens, each new element contains the index of the successor event. Thus, eventually, the algorithm terminates or all elements of *open* contain the event index n. In the latter case, the algorithm terminates in step 6. □

Theorem 3.26 (Soundness)
Let ξ be a finite trace and φ be an RTLTL formula. If the algorithm returns true, then $\xi \models \varphi$.

Proof If the algorithm returns *true*, it has terminated in step 6. Then, ξ has fulfilled the condition in step 5 and the automaton must therefore have an accepting execution on it. Hence, theorem 3.22 implies $\xi \models \varphi$. □

Theorem 3.27 (Completeness)
Let ξ be a finite trace and φ be an RTLTL formula in negation normal form. If $\xi \models \varphi$, then the algorithm returns true.

Proof It is easy to see that the on-the-fly algorithms starts with all same-time descendant nodes of the initial node of the classic algorithm. Also, for each node, all successor nodes and their same-time descendants are created. Thus, the on-the-fly algorithm constructs an automaton with the same states and transitions as the classic algorithm does. By theorem 3.23, $\xi \models \varphi$ implies that

the automaton has an accepting execution on ξ. The algorithm only returns $false$ if all possible executions have been exploited. Since there is an accepting execution for ξ, it is eventually found, and *true* is returned in step 6. □

Corollary 3.28 (Total Correctness)
Given a finite trace ξ and an RTLTL formula φ in negation normal form, the above algorithm returns true if and only if $\xi \vDash \varphi$.

3.5.3. Further efficiency improvements

Several authors have proposed additional technical improvements of the basic algorithm, which can be used in order to achieve further reductions in running time and memory consumption.

Gerth, Vardi, Peled, and Wolper [GPVW95] propose that early splitting of nodes shall be avoided. Instead, splitting tableau rules can be applied such that the target node is only updated with additional information that is later, once the first node is completely expanded, used to create the second node. Daniele, Giunchiglia, and Vardi [DGV99] propose significant modifications that result in fewer formulae to be stored and earlier detection of redundancies and inconsistencies; this results in smaller automata and thus finding matching states faster. In his extensive analysis of translations from LTL into nondeterministic and alternating Büchi automata, Tauriainen [Tau03] uses powerful tools as language containment checking and boolean optimisation techniques in order to simplify resulting automata. All these ideas are compatible to our on-the-fly algorithm.

Giannakopoulou and Havelund [GH01] argue that nodes with different *old* fields can be regarded as equivalent as long as they have equal *next* fields; unfortunately, this potential state-space reduction is not possible in our approach, since we use *old* in the acceptance condition. Several authors [SB00, EWS01, Ete02, FW02, Fri03, GBS02] apply simulation relations in order to minimise Büchi automata obtained from LTL formulae; again, this technique is only applicable to completely constructed automata and can in particular not be used on-the-fly.

3.6. Complexity

We finish this chapter with an overview about the computational complexity of the presented translation from RTLTL to NFA, the complexity of the word problem for NFA, and the complexity of the resulting truth checking problem for NFA.

Theorem 3.29 (Complexity of RTLTL to NFA translation)
Given an RTLTL formula φ in negation normal form, there is an equivalent NFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$ with $2^{\mathcal{O}(|\varphi|)}$ states.

Proof The algorithm from section 3.4 translates a given RTLTL formula φ into an equivalent NFA. We prove by structural induction on φ that the resulting NFA has $2^{\mathcal{O}(|\varphi|)}$ states. Without loss of generality, we assume that φ is already in negation normal form.

1. If φ does not contain any temporal operator, then $\mathcal{A}$ just contains one state, and we are done.
2. Let $\varphi = \mathsf{X}_{a+1}\,\psi$. It is easy to see that φ is only propagated through the *next* fields, and the constructed automaton $\mathcal{A}$ has $a{+}2 = 2^{\mathcal{O}(\log a)} = 2^{\mathcal{O}(|\varphi|)}$ states.
3. Let $\varphi = \mu \,\mathsf{R}\, \eta$. By the induction hypothesis, the automata for μ and η have size $2^{\mathcal{O}(|\mu|)}$ and $2^{\mathcal{O}(|\mu|)}$. Hence, one application of the R-rule then constructs the automaton for $\mathcal{A}$ such that it has

   $$1 + 2^{\mathcal{O}(|\mu|)} + 2^{\mathcal{O}(|\eta|)} \leq 2^{\mathcal{O}(|\mu+\eta|)} \leq 2^{\mathcal{O}(|\varphi|)}$$

   states.
4. Let $\varphi = \mu \,\mathsf{R}_{a+1,b+1}\, \eta$. By the induction hypothesis, the automata for μ, η, and $\mu \,\mathsf{R}_{i,j}\, \eta$ $(i \leq a, j \leq b)$ have size $2^{\mathcal{O}(|\mu|)}$, $2^{\mathcal{O}(|\mu|)}$, and $2^{\mathcal{O}(|\varphi|)}$. Hence, at most $b+1$ applications of the $\mathsf{R}_{a+1,b+1}$-rule then construct the automaton for $\mathcal{A}$ such that it has at most

   $$\mathcal{O}(b) \cdot (2^{\mathcal{O}(|\mu|)} + 2^{\mathcal{O}(|\eta|)}) \leq 2^{\mathcal{O}(\log b + |\mu+\eta|)} \leq 2^{\mathcal{O}(|\varphi|)}$$

   states.
5. The other cases are completely analogue. □

Note that our restriction on input formulae without $\rightarrow$, F, $\mathsf{F}_{a,b}$, G, and $\mathsf{G}_{a,b}$ operators can cause an exponential blowup when formulae containing such operators are translated into the required notation. Nevertheless, this does not affect the validity of this theorem, since this blowup can be avoided by introducing additional rules for these operators. Also, in the average case, this problem does not occur.

Theorem 3.30 (Complexity of the word problem for NFA)
The word problem for finite traces ξ and NFA with n states can be solved in time $\mathcal{O}(n^{|\xi|})$ and space $\mathcal{O}(\log n \times |\xi|)$.

Proof Each NFA state can have at most n outgoing transitions, that is, at most n direct successor states. Checking a given finite trace ξ for acceptance involves checking all possible sequences of states of length $|\xi|$. Assuming that transitions are ordered by the index of their starting state, only one state index for each event of ξ must be stored for backtracking, requiring $\log n$ space per event. □

The last result is a direct consequence of the previous two.

Corollary 3.31 (Complexity of $\mathbf{TCP_{fin}}$ via NFA)
Let ξ be a finite trace and φ be an RTLTL formula which is given by an NFA. Then, the truth checking problem can be solved in time $2^{\mathcal{O}(|\varphi| \times |\xi|)}$ and space $2^{\mathcal{O}(|\varphi|)} \times |\xi|$.

4. Term rewriting

Term rewriting provides a powerful and established theoretical framework for reasoning in all kinds of logics. The main advantage of rewriting over automata theory as a foundation for runtime verification is that logics and combinatorics can be decoupled, That means that the generic inference rules of the underlying logic can be executed independently from the specific rewrite rules of the specification logic. Consequently, the semantics of the "embedded" logic can be represented very simply and efficiently. Also, there does not exist any distinction such as that between automata generation and trace traversal in the automata-theoretic approach anymore; instead, traces and formulae are processed simultaneously.

The most promising results for the application of term rewriting in runtime verification have been reported by Havelund and Roşu [HR01a, RH], who present a rewriting-based algorithm that checks finite traces against LTL formulae. Their algorithm uses MAUDE, a highly efficient rewriting system, which exhibits the power feature of *memoisation*, that is, rewrite patterns can be cached for faster execution in future. Since our framework is not compatible to MAUDE, we have optimised our rewrite rules such that they can be efficiently executed in a deterministic manner, without complicated heuristics.

In the first section, we give a brief introduction to the theory of term rewriting. Subsequently, we present two rewriting-based runtime verification algorithms, each based on a different set of rewrite rules, which both extend the techniques of [HR01a, RH] to RTLTL and use a neutral instead of a stationary semantics. Finally, we study the complexity of both approaches.

4.1. Preliminaries

We now present the theoretical basics of term rewriting, following the textbook of Baader and Nipkow [BN98] and the survey of Dershowitz and Plaisted [DP01].

Definition 4.1 (Signature)
*A **signature** Σ is a set of **function symbols**, where each $f \in \Sigma$ is associated with a non-negative integer n, the **arity** of f. For $n \geq 0$, we denote the set of all n-ary elements of Σ by $\Sigma^{(n)}$. The elements of $\Sigma^{(0)}$ are also called **constant symbols**.*

Definition 4.2 (Term)
*Let Σ be a signature and V be a set of **variables** such that $\Sigma \cap V = \emptyset$. Then, the set $T(\Sigma, V)$ of all **Σ-terms** over V is defined as follows:*

1. $V \subseteq T(\Sigma, V)$.

2. *For all* $n \geq 0$, *all* $f \in \Sigma^{(n)}$, *and all* $t_1, \ldots, t_n \in T(\Sigma, V)$, *we have* $f(t_1, \ldots, t_n) \in T(\Sigma, V)$.

Definition 4.3 (Subterm)
A term t *is a* ***subterm*** *of the term* u *if one of the following holds:*

1. $t = u$.

2. $u = f(u_1, \ldots, u_n)$ *and* t *is a subterm of* u_i *for some* $1 \leq i \leq n$.

Definition 4.4 (Context)
Let $\diamond$ *be a new symbol which does not occur in* $\Sigma \cup V$. *Then, a* Σ***-context*** *over* V *is a term* $t \in T(\Sigma, V \cup \{\diamond\})$ *where* $\diamond$ *represents a* ***hole***.

That is, a context is a term where one of the variables has a prominent role.

Definition 4.5 (Substitution)
Let Σ *be a signature and* V *be a set of variables. Then, a* $T(\Sigma, V)$***-substitution*** *is a function* $\sigma : V \rightarrow T(\Sigma, V)$.

The application of a substitution σ to a term simultaneously replaces all occurences of a variable v by $\sigma(v)$.

Definition 4.6 (Rewrite rule, rewrite system, rewrite relation)
Let Σ *be a signature and* V *be a set of variables.*

1. *A* ***rewrite system*** $\mathcal{R}$ *is a set of* ***rewrite rules*** $l \rightarrow r$ *such that* $l, r \in T(\Sigma, V)$.

2. *The* ***rewrite relation*** $\rightarrow_{\mathcal{R}}$ *is the smallest relation such that for all contexts* $t \in T(\Sigma, V \cup \{\diamond\})$, *rules* $l \rightarrow r \in \mathcal{R}$, *and substitutions* σ, *we have* $t[l\sigma] \rightarrow_{\mathcal{R}} t[r\sigma]$.

3. *A* ***derivation*** *for* $\rightarrow_{\mathcal{R}}$ *is a sequence* $t_0 \rightarrow_{\mathcal{R}}\rightarrow t_1 \rightarrow_{\mathcal{R}}\rightarrow \ldots$ *where* $t_i \in T(\Sigma, V)$ *for all* $i \geq 0$.

That is, every set of rewrite rules induces a rewrite relation on the set of terms. If a rewrite rule is applied to a term $t[l\sigma]$ where $l\sigma$ is an instance of the left-hand side of a rule $l \rightarrow r$, then each occurence of $l\sigma$ is replaced by the corresponding instance $r\sigma$ of the right-hand side of the rule, thereby rewriting $t[l\sigma]$ to $t[r\sigma]$.

We would like to remark that this introduction is not aimed on designing or using “real” rewriting system. Contrariwise, our approach rather benefits from two important design constraints: first, our system does not provide any of the advanced features that customary rewriting engines usually have, like memoisation and heuristic rule application. Second, proving termination of our rewrite relation is simple. Hence, an exposition of *equational logic*, which may be relevant for systems like MAUDE, it is out of scope here.

4.2. Simple rewriting

In this section, we describe our first rewriting-based algorithm for checking finite traces against RTLTL formulae. This method extends Havelund and Roşu's approach [HR01a, RH] to real-time formulae with a neutral semantics. Note that although the rules are defined recursively, they can easily be implemented in a way such that each temporal operator is evaluated iteratively rather than recursively.

The algorithm starts with a nonempty finite trace $\xi = x_0, \ldots, x_n$ and an RTLTL formula φ that does not contain the $\rightarrow$ operator (note that all RTLTL formulae can be translated into equivalent formulae of this form). We suppose that $p \in P$ is an atomic proposition and that simple expressions of the form $p \in x$, $\xi = \varepsilon$, and $\xi \neq \varepsilon$ are evaluated instantly whereever they occur in a derivation.

In both of the next two algorithms, we use the following set of rewrite rules for termination an evaluation when *true* or *false* is reached, and for rewriting trivial RTLTL formulae and boolean expressions:

$$
\begin{aligned}
\xi \vdash \mathsf{t}rue &\rightarrow true \\
\xi \vdash \mathsf{f}alse &\rightarrow false \\
\xi \vdash \mathsf{F}_{0,0}\,\varphi &\rightarrow \xi \vdash \varphi \\
\xi \vdash \mathsf{G}_{0,0}\,\varphi &\rightarrow \xi \vdash \varphi \\
\xi \vdash \varphi\,\mathsf{U}_{0,0}\,\psi &\rightarrow \xi \vdash \psi \\
\xi \vdash \varphi\,\mathsf{R}_{0,0}\,\psi &\rightarrow \xi \vdash \psi \\
true \vee \varphi &\rightarrow true \\
true \wedge \varphi &\rightarrow \varphi \\
false \vee \varphi &\rightarrow \varphi \\
false \wedge \varphi &\rightarrow false
\end{aligned}
$$

These rules shall be used in order to simplify formulae before each application of a rewrite rule from without this set. In addition, this algorithm uses the following set of rewrite rules:

$$
\begin{aligned}
\xi \vdash \top &\rightarrow true \\
\xi \vdash \bot &\rightarrow false \\
\xi \vdash p &\rightarrow p \in x_0 \\
\xi \vdash \neg\varphi &\rightarrow \xi \nvdash \varphi \\
\xi \vdash \varphi \vee \psi &\rightarrow \xi \vdash \varphi \ \vee\ \xi \vdash \psi \\
\xi \vdash \varphi \wedge \psi &\rightarrow \xi \vdash \varphi \ \wedge\ \xi \vdash \psi \\
\xi \vdash \mathsf{X}\,\varphi &\rightarrow \xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \varphi \\
\xi \vdash \mathsf{Y}\,\varphi &\rightarrow \xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \varphi \\
\xi \vdash \mathsf{F}\,\varphi &\rightarrow \xi \vdash \varphi \ \vee\ (\xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \mathsf{F}\,\varphi) \\
\xi \vdash \mathsf{G}\,\varphi &\rightarrow \xi \vdash \varphi \ \wedge\ (\xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \mathsf{G}\,\varphi) \\
\xi \vdash \varphi\,\mathsf{U}\,\psi &\rightarrow \xi \vdash \psi \ \vee\ (\xi \vdash \varphi \ \wedge\ \xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \varphi\,\mathsf{U}\,\psi)
\end{aligned}
$$

$$
\begin{array}{rcl}
\xi \vdash \varphi \,\mathsf{R}\, \psi & \rightarrow & \xi \vdash \psi \ \wedge\ (\xi \vdash \varphi \ \vee\ \xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \varphi \,\mathsf{R}\, \psi) \\
\xi \vdash \mathsf{X}_a\, \varphi & \rightarrow & \xi^a \neq \varepsilon \ \wedge\ \xi^a \vdash \varphi \\
\xi \vdash \mathsf{Y}_a\, \varphi & \rightarrow & \xi^a = \varepsilon \ \vee\ \xi^a \vdash \varphi \\
\xi \vdash \mathsf{F}_{0,b+1}\, \varphi & \rightarrow & \xi \vdash \varphi \ \vee\ (\xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \mathsf{F}_{0,b}\, \varphi) \\
\xi \vdash \mathsf{F}_{a+1,b+1}\, \varphi & \rightarrow & \xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \mathsf{F}_{a,b}\, \varphi \\
\xi \vdash \mathsf{G}_{0,b+1}\, \varphi & \rightarrow & \xi \vdash \varphi \ \wedge\ (\xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \mathsf{G}_{0,b}\, \varphi) \\
\xi \vdash \mathsf{G}_{a+1,b+1}\, \varphi & \rightarrow & \xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \mathsf{G}_{a,b}\, \varphi \\
\xi \vdash \varphi \,\mathsf{U}_{0,b+1}\, \psi & \rightarrow & \xi \vdash \psi \ \vee\ (\xi \vdash \varphi \ \wedge\ \xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \varphi \,\mathsf{U}_{0,b}\, \psi) \\
\xi \vdash \varphi \,\mathsf{U}_{a+1,b+1}\, \psi & \rightarrow & \xi \vdash \varphi \ \wedge\ \xi^1 \neq \varepsilon \ \wedge\ \xi^1 \vdash \varphi \,\mathsf{U}_{a,b}\, \psi \\
\xi \vdash \varphi \,\mathsf{R}_{0,b+1}\, \psi & \rightarrow & \xi \vdash \psi \ \wedge\ (\xi \vdash \varphi \ \vee\ \xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \varphi \,\mathsf{R}_{0,b}\, \psi) \\
\xi \vdash \varphi \,\mathsf{R}_{a+1,b+1}\, \psi & \rightarrow & \xi \vdash \varphi \ \vee\ \xi^1 = \varepsilon \ \vee\ \xi^1 \vdash \varphi \,\mathsf{R}_{a,b}\, \psi
\end{array}
$$

We now provide proofs for termination and correctness of this algorithm.

Theorem 4.7 (Termination)
Given a finite trace ξ and an RTLTL formula φ, the algorithm eventually returns $true$ or $false$.

Proof Let τ be a ranking function that measures the complexity of a formula (see figure 4.1). The truth values $true$ and $false$ and any simple expression

Formula φ	**Rank** $\tau(\varphi)$
$\top, \bot, p$	0
$\neg\varphi, \mathsf{X}\,\psi, \mathsf{Y}\,\psi, \mathsf{F}\,\psi, \mathsf{G}\,\psi$	$1 + \tau(\varphi)$
$\psi \vee \mu, \psi \wedge \mu, \psi \,\mathsf{U}\, \mu, \psi \,\mathsf{R}\, \mu$	$1 + max\{\tau(\psi), \tau(\mu)\}$
$\mathsf{X}_a\, \psi, \mathsf{Y}_a\, \psi, \mathsf{X}_a\, \psi, \mathsf{G}_{a,b}\, \psi$	$1 + a + \tau(\psi)$
$\psi \,\mathsf{U}_{a,b}\, \mu, \psi \,\mathsf{R}_{a,b}\, \mu$	$1 + a + b + max\{\tau(\psi), \tau(\mu)\}$

Figure 4.1.: A ranking function for RTLTL formulae.

that can instantly be evaluated to $true$ or $false$ have rank 0. We observe that for each rewrite rule, the right-hand side only contains finitely many formulae and each formula on the right-hand side has lower rank than each formula on the left-hand side. Therefore, each derivation of a formula consists of finitely many derivations of formulae of lower rank. Hence, each derivation is finite and derives $true$ or $false$. □

Theorem 4.8 (Soundness and Completeness)
Let ξ be a finite trace and φ be an RTLTL formula. Then, $\xi \vdash \varphi$ if and only if $\xi \models \varphi$.

Proof All rewrite rules $l \rightarrow r$ can be regarded as equivalences $l \equiv r$ which follow directly from definition 2.16 and corollary 2.17. □

Finally, we can establish the following total correctness property, which is a direct consequence of theorem 4.7 and theorem 4.8.

Corollary 4.9 (Total Correctness)
Given a finite trace ξ *and an RTLTL formula* φ*, the algorithm eventually returns* $\xi \vDash \varphi$*.*

4.3. Event-consuming rewriting

The major drawback of the simple algorithm is that, in general, it takes multiple traversals to check a trace. Havelund and Roşu [HR01a, RH] have proposed a so-called *event-consuming* variant of the simple algorithm that can be used online, that is, it needs at most one traversal and therefore only needs to store one event of the trace being checked.

The event-consuming algorithm starts with a nonempty finite trace $\xi = x_0, \ldots, x_n$ and an RTLTL formula φ in negation normal form that does not contain the $\rightarrow$ operator (note that all RTLTL formulae can be translated into equivalent formulae of this form). As in the simple algorithm, we suppose that $p \in \mathcal{P}$ is an atomic proposition and that simple expressions of the form $p \in x$, $\xi = \varepsilon$, and $\xi \neq \varepsilon$ are evaluated instantly whereever they occur in a derivation.

In the event-consuming approach, the formula to be evaluated on a trace is transformed into another formula that has to be evaluated on the suffix obtained by omitting the first event. We say that the formula "consumes" the event. Consuming an event is represented by the function

$$\{\,.\,\} : Formula \times Event \rightarrow Formula.$$

The basic two rewrite rules distinguish between final and non-final events, that is, such events that are located at the end of a trace and such that are not.

$$\xi \vdash \varphi \quad \rightarrow \quad \varphi\{x_0\} \text{ if } \xi^1 = \varepsilon$$
$$\xi \vdash \varphi \quad \rightarrow \quad \xi^1 \vdash \varphi\{x_0\} \text{ if } \xi^1 \neq \varepsilon$$

The event consumption function is given in the remainder of this section. When a definition is executed by the rewriting system, it can be treated as a rule. The consumption of final events x is defined as follows:

$$\begin{aligned}
\top\{x\} &\stackrel{def}{=} true \\
\bot\{x\} &\stackrel{def}{=} false \\
p\{x\} &\stackrel{def}{=} p \in x \\
\neg p\{x\} &\stackrel{def}{=} p \notin x \\
(\varphi \vee \psi)\{x\} &\stackrel{def}{=} \varphi\{x\} \vee \psi\{x\} \\
(\varphi \wedge \psi)\{x\} &\stackrel{def}{=} \varphi\{x\} \wedge \psi\{x\} \\
(\mathsf{X}\,\varphi)\{x\} &\stackrel{def}{=} false \\
(\mathsf{Y}\,\varphi)\{x\} &\stackrel{def}{=} true \\
(\mathsf{F}\,\varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \\
(\mathsf{G}\,\varphi)\{x\} &\stackrel{def}{=} \varphi\{x\}
\end{aligned}$$

$$
\begin{aligned}
(\varphi \,\mathsf{U}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \\
(\varphi \,\mathsf{R}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \\
(\mathsf{X}_{a+1}\, \varphi)\{x\} &\stackrel{def}{=} false \\
(\mathsf{Y}_{a+1}\, \varphi)\{x\} &\stackrel{def}{=} true \\
(\mathsf{F}_{0,b+1}\, \varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \\
(\mathsf{F}_{a+1,b+1}\, \varphi)\{x\} &\stackrel{def}{=} false \\
(\mathsf{G}_{0,b+1}\, \varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \\
(\mathsf{G}_{a+1,b+1}\, \varphi)\{x\} &\stackrel{def}{=} true \\
(\varphi \,\mathsf{U}_{0,b+1}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \\
(\varphi \,\mathsf{U}_{a+1,b+1}\, \psi)\{x\} &\stackrel{def}{=} false \\
(\varphi \,\mathsf{R}_{0,b+1}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \\
(\varphi \,\mathsf{R}_{a+1,b+1}\, \psi)\{x\} &\stackrel{def}{=} true
\end{aligned}
$$

The consumption of non-final events x is defined as follows:

$$
\begin{aligned}
\top\{x\} &\stackrel{def}{=} true \\
\bot\{x\} &\stackrel{def}{=} false \\
p\{x\} &\stackrel{def}{=} p \in x \\
\neg p\{x\} &\stackrel{def}{=} p \notin x \\
(\varphi \vee \psi)\{x\} &\stackrel{def}{=} \varphi\{x\} \vee \psi\{x\} \\
(\varphi \wedge \psi)\{x\} &\stackrel{def}{=} \varphi\{x\} \wedge \psi\{x\} \\
(\mathsf{X}\, \varphi)\{x\} &\stackrel{def}{=} \varphi \\
(\mathsf{Y}\, \varphi)\{x\} &\stackrel{def}{=} \varphi \\
(\mathsf{F}\, \varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \vee \mathsf{F}\, \varphi \\
(\mathsf{G}\, \varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \wedge \mathsf{G}\, \varphi \\
(\varphi \,\mathsf{U}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \vee (\varphi\{x\} \wedge \varphi \,\mathsf{U}\, \psi) \\
(\varphi \,\mathsf{R}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \wedge (\varphi\{x\} \vee \varphi \,\mathsf{R}\, \psi) \\
(\mathsf{X}_{a+1}\, \varphi)\{x\} &\stackrel{def}{=} \mathsf{X}_{a}\, \varphi \\
(\mathsf{Y}_{a+1}\, \varphi)\{x\} &\stackrel{def}{=} \mathsf{Y}_{a}\, \varphi \\
(\mathsf{F}_{0,b+1}\, \varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \vee \mathsf{F}_{0,b}\, \varphi \\
(\mathsf{F}_{a+1,b+1}\, \varphi)\{x\} &\stackrel{def}{=} \mathsf{F}_{a,b}\, \varphi \\
(\mathsf{G}_{0,b+1}\, \varphi)\{x\} &\stackrel{def}{=} \varphi\{x\} \wedge \mathsf{G}_{0,b}\, \varphi \\
(\mathsf{G}_{a+1,b+1}\, \varphi)\{x\} &\stackrel{def}{=} \mathsf{G}_{a,b}\, \varphi \\
(\varphi \,\mathsf{U}_{0,b+1}\, \psi)\{x\} &\stackrel{def}{=} \psi\{x\} \vee (\varphi\{x\} \wedge \varphi \,\mathsf{U}_{0,b}\, \psi)
\end{aligned}
$$

$$\begin{aligned}
(\varphi\,\mathsf{U}_{a+1,b+1}\,\psi)\{x\} &\stackrel{def}{=} \varphi\{x\} \wedge \varphi\,\mathsf{U}_{a,b}\,\psi \\
(\varphi\,\mathsf{R}_{0,b+1}\,\psi)\{x\} &\stackrel{def}{=} \psi\{x\} \wedge (\varphi\{x\} \vee \varphi\,\mathsf{R}_{0,b}\,\psi) \\
(\varphi\,\mathsf{R}_{a+1,b+1}\,\psi)\{x\} &\stackrel{def}{=} \varphi\{x\} \vee \varphi\,\mathsf{R}_{a,b}\,\psi
\end{aligned}$$

Remember that this algorithm also uses the set of simplification rules introduced at the beginning of the previous section.

We now provide proofs for termination and correctness of this algorithm.

Theorem 4.10 (Termination)
Given a finite trace ξ and an RTLTL formula φ in negation normal form, the algorithm eventually returns $true$ or $false$.

Proof It is easy to observe that the event consumption function $\{\,.\,\}$ is only applied finitely often for each event in the finite trace. Also, the basic rewrite rules are applied at most once for each event. Showing termination of the remaining rewrite rules is analogue to the proof of theorem 4.7. □

Theorem 4.11 (Soundness and Completeness)
Let ξ be a finite trace and φ be an RTLTL formula in negation normal form. Then, $\xi \vdash \varphi$ if and only if $\xi \vDash \varphi$.

Proof From definition 2.16 and corollary 2.17 we can easily derive that, for each RTLTL formula φ and for each event $x_0 \in \mathcal{P}$, the following holds:

1. If $\xi^1 = \varepsilon$, then $(\xi \vdash \varphi) \equiv (\varphi\{x_0\})$.
2. If $\xi^1 \neq \varepsilon$, then $(\xi \vdash \varphi) \equiv (\xi^1 \vdash \varphi\{x_0\})$.

Consequently, these rewrite rules are sound. Also from definition 2.16 and corollary 2.17, the simplification rules from the previous section are sound. □

Finally, we can establish the following total correctness property, which is a direct consequence of theorem 4.10 and theorem 4.11.

Corollary 4.12 (Total Correctness)
Given a finite trace ξ and an RTLTL formula φ in negation normal form, the algorithm eventually returns $\xi \vDash \varphi$.

We conclude this section with an example.

Example Let $\mathcal{P} = \{p, q, r\}$ be a set of atomic propositions, $\xi = \{q\}, \{q, r\}$ be a trace, and $\varphi = p\,\mathsf{R}\,(q \vee r)$ be a formula. Then, one possible derivation of $\xi \vdash \varphi$ is as follows:

$$\begin{aligned}
\{q\}, \{q, r\} &\vdash p\,\mathsf{R}\,(q \vee r) \\
\{q, r\} &\vdash (p\,\mathsf{R}\,(q \vee r))\{q\} \\
\{q, r\} &\vdash (q \vee r)\{q\} \wedge (p\{q\} \vee (p\,\mathsf{R}\,(q \vee r))) \\
\{q, r\} &\vdash (q\{q\} \vee r\{q\}) \wedge (p\{q\} \vee (p\,\mathsf{R}\,(q \vee r)))
\end{aligned}$$

$$\begin{array}{rcl} \{q,r\} & \vdash & (true \vee false) \wedge (false \vee (p\,\mathsf{R}\,(q \vee r))) \\ \{q,r\} & \vdash & p\,\mathsf{R}\,(q \vee r)) \\ & & (p\,\mathsf{R}\,(q \vee r))) \{q,r\} \\ & & (q \vee r)\{q,r\} \\ & & q\{q,r\} \vee r\{q,r\} \\ & & true \vee true \\ & & true \end{array}$$

◇

4.4. Complexity

We close this chapter with an overview about the computational complexity of the presented rewriting-based runtime verification algorithms. The first two facts have already been proven by Roşu and Havelund [RH].

Proposition 4.13 ($\mathbf{TCP_{fin}}$ for LTL via simple rewriting [RH])
Let ξ be a finite trace and φ be an LTL formula. Then, solving $\xi \vdash \varphi$ by simple rewriting needs $\mathcal{O}(|\xi|^{|\varphi|})$ time.

Proposition 4.14 ($\mathbf{TCP_{fin}}$ for LTL via event-consuming rewriting [RH])
Let ξ be a finite trace and φ be an LTL formula in negation normal form. Then, solving $\xi \vdash \varphi$ by event-consuming rewriting needs $\mathcal{O}(2^{|\varphi|})$ space.

Theorem 4.15 ($\mathbf{TCP_{fin}}$ for RTLTL via simple rewriting)
Let ξ be a finite trace and φ be an RTLTL formula. Then, solving $\xi \vdash \varphi$ by simple rewriting needs $\mathcal{O}(|\xi|^{|\varphi|})$ time and $\mathcal{O}(|\xi| \times |\varphi|^2)$ space.

Proof We can observe that, for each subformula of φ and for each suffix of ξ, the trace is traversed at most once. In particular, that holds for formulae containing real-time operators. Thus, the time upper bound from proposition 4.13 holds.

Each RTLTL formula φ has $\mathcal{O}(|\varphi|)$ subformulae. When a formula is evaluated on a trace, at most one backtracking point may be set for each formula and each event. At each backtracking point, the formula of size $\mathcal{O}(|\varphi|)$ must be stored. □

Theorem 4.16 ($\mathbf{TCP_{fin}}$ for RTLTL via event-consuming rewriting)
Let ξ be a finite trace and φ be an RTLTL formula in negation normal form. Then, solving $\xi \vdash \varphi$ by event-consuming rewriting needs $\mathcal{O}(|\xi| \times 2^{|\varphi|})$ time and $\mathcal{O}(2^{|\varphi|})$ space.

Proof We can observe that the trace is being traversed only once. Since processing RTLTL formulae does not involve more complex rules than processing standard LTL formulae, the space upper bound from proposition 4.13 holds.

Each expression derived from the initial truth checking problem can be rewritten to a truth value in linear time. Since this is done at most once per event, we have the time upper bound. □

5. Experiments and results

In the last sections we have seen that translating LTL formulae to NFA, solving the word problem for NFA, and solving the truth checking problem for finite traces by means of online techniques are all computationally hard problems. However, theoretical upper bounds and practical feasibility of algorithms differ. When we applied our algorithms to artificially generated problems for the purpose of benchmarking and to industrial systems in the field of automotive electronics, verification was almost always accomplishable in runtime – as required.

We test the software implementation of our algorithms with respect to two requirements: correctness and performance. In order to check the correctness of temporal-logic to automata translations, Tauriainen and Heljanko [TH02] suggest constructing the automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ for the input formula φ and then performing the soundness test

$$\mathcal{A}_\varphi \cap \mathcal{A}_{\neg\varphi} = \emptyset$$

and the completeness test

$$\mathcal{A}_\varphi \cup \mathcal{A}_{\neg\varphi} = \Sigma^*.$$

For this purpose, they randomly generate LTL formulae and then automatically inspect the computed Büchi automata.

Given that such a procedure tends to be rather complicated and expensive, we instead test the correct translation for a broad range of manually selected examples. This is also our first choice for testing the implementation of the rewriting-based algorithms, since they are much simpler. In addition to examining the results for sample truth checking problems, we also ensure the correct functionality of each method and each class of the software implementation through *unit tests*.

In the sequel, we focus on performance issues. Several authors propose methods for conducting benchmarks of LTL to Büchi automata translations: For example, Daniele, Giunchiglia, and Vardi [DGV99] use a random distribution in order to generate test formulae. Gastin and Oddoux [GO01] construct parameterised test formulae of the forms

$$\theta_n := \neg((\mathsf{G}\,\mathsf{F}\,p_1 \wedge \cdots \wedge \mathsf{G}\,\mathsf{F}\,p_n) \rightarrow \mathsf{G}\,(q \rightarrow \mathsf{F}\,r))$$

and

$$\varphi_n := \neg(p_1\,\mathsf{U}\,(p_2\,\mathsf{U}\,(\cdots\,\mathsf{U}\,p_n)\cdots)).$$

For our first performance tests, we used parameterised formulae similar to those of [GO01] in order to find possible bottlenecks in algorithms and implementation. It turned out that evaluating intricately nested formulae was particularly hard for the automata-theoretic algorithms when checking very long

traces, while usually no problems emerged from the presence of real-time operators. The implementation of the rewriting-based algorithms was subsequently optimised by replacing recursively defined rewrite rules with their equivalent iterative variants, which can be taken from definition 2.16 and corollary 2.17.

In order to compare the performance of our algorithm with that of other runtime verification algorithms, we used the traffic-light example of Roşu and Havelund [RH]. In this example, a sample trace is generated by repeating the sequence

$$green, yellow, red, green, yellow, red, green, yellow, red, red$$

as often as desired and then checking this trace against the specification formula

$$green \rightarrow (\neg red \, \mathsf{U} \, yellow).$$

On a PC equipped with an AMD Athlon XP 2500+ processor with 1833 MHz and 512 MB RAM, we benchmarked execution time and memory usage using the profiling tools `profile` and `hotshot` of Python and the Windows XP Task Manager. The results are listed in figure 5.1.

Algorithm	**Trace length**	**CPU time**	**Memory**
NFA, forward	100,000	5 s	6.7 MB
NFA, on-the-fly	100,000	4 s	6.7 MB
Rewriting, simple	100,000	3 s	6.7 MB
Rewriting, event-consuming	100,000	8 s	6.7 MB
NFA, forward	1,000,000	58 s	67.6 MB
NFA, on-the-fly	1,000,000	43 s	67.6 MB
Rewriting, simple	1,000,000	31 s	67.6 MB
Rewriting, event-consuming	1,000,000	82 s	67.6 MB

Figure 5.1.: Performance evaluation on the traffic-light example.

Overall, these results are at least as good as those of Roşu and Havelund [RH], who have used the same example to evaluate simple and event-consuming rewriting algorithms for LTL. Their implementation of the simple approach was not able to finish checking a trace of 10,000 events within 10 hours. With their implementation of the event-consuming approach, they reported to need 3 seconds for a trace of 100,000 events and 1,500 seconds for a trace of 10,000,000 events. When the memoisation feature of Maude was enabled, checking the last trace with the event-consuming algorithm took them only 185 seconds.

Hence, though our simple rewriting algorithm usually needs to perform multiple traversals of the trace, its iterative implementation is considerably faster than that of our event-consuming algorithm. Therefore, the simple algorithm is always favourable for traces that are available offline and can be stored in memory completely. Nevertheless, subject to a full rewriting engine becoming available, the event-consuming algorithm could put its clearly more advanced design into practice and outperform the simple approach for both online and offline operation, then taking advantage of automatic rewriting of intermediate terms and caching of frequently used rewrite patterns.

6. Conclusion

In this thesis, we have presented runtime verification methods for the analysis of real-time systems.

For this purpose, we have developed a syntactic real-time extension of LTL, called RTLTL. We have modified the standard semantics of LTL, which is defined for infinite traces, and defined a new finite-trace semantics for RTLTL, which, as far as we know, is different from all existing approaches in runtime verification. Our semantics is consistent and enjoys the existence of duals for each operator. This permits specifications that are very succint and more efficient to check than such in standard LTL. Also, the specification of properties in RTLTL is much more intuitive, since the X operator has a definite meaning here.

We have comprehensively exposed the fundamentals of automata theory and term rewriting. Based on the theory of both fields, we have presented different methods for checking finite traces against real-time specifications. All algorithms have been implemented in an industrial validation and verification framework for automotive electronics.

The automata-theoretic methods extend existing algorithms into translation procedures from RTLTL formulae into NFA. This is at the same time an improvement in terms of the used logic (which now allows more intuitive and more succinct specification formulae) as well as in terms of the used automata (which are now simpler and suitable for finite traces). A given formula can be translated into a finite automaton on-the-fly while the corresponding trace is being traversed.

The rewriting-based methods extend existing algorithms for LTL towards RTLTL and towards our semantics for finite traces. Both use very simple and intuitive rewrite rules, which are easily adaptable to different logics. The simple rewriting algorithm cannot be used online, but it is our fastest method for all considered real-world applications. In order to optimise it for offline trace checking, we suggest using backward depth-first search in a way similar to that used in the classic CTL labelling algorithm for model checking. The event-consuming approach would profit greatly from an implementation in the rewriting system Maude. However, Maude is not yet compatible to the enclosing framework of our tool.

Our runtime verification algorithms all consist of a fully formal specification component and a semi-formal verification component. All presented algorithms are fully automatic and do not require any additional modelling. The event-consuming rewriting algorithm can be used online, ergo in a genuine runtime-verification manner.

Our complexity analysis has shown that truth checking for finite traces and RTLTL specifications is in general a hard task. Under practical considerations,

however, the results of the performed experiments show that our algorithms are efficient, scalable, and competitive to the best available reference algorithms. In the intended application domain, these algorithms are in many respects more powerful than traditional testing.

Future work shall be concerned with the study of different logics, property specification patterns, optimisations of the verification algorithms, and coverage control.

In order to support more succinct and more intuitive specifications, we suggest to enrich the logic RTLTL with past operators and to consider MTL as an alternative specification language.

For the specification of real-world systems, defining common analytical patterns can improve the verification process and ensure correct specification of nontrivial behaviour. This includes defining patterns for rising and falling edges of signals as well as a introducing a more extensive system of specification patterns, as proposed by Dwyer, Avrunin, and Corbett [DAC99].

For the initial rewriting step that takes place on each user-provided specification formula, an improvement of our set of equivalences may be beneficial.

The algorithm for RTLTL to NFA translation can be enhanced to an online method if it is modified such that deterministic finite automata are produced. Simplest, the classic powerset algorithm for NFA determinisation can be used. If this transformation is not executed on-the-fly, according to our experiments, only a slight degradation of efficiency must be expected.

Last but not least, since runtime verification never covers full system behaviour, coverage statistics shall complement the validation of single traces. Good starting points for a study of those are the papers of Chockler, Kupferman, and Vardi [CKV03] and of Finkbeiner, Sankaranarayanan, and Sipma [FSS02].

A. Software

All algorithms described in this thesis have been implemented in the software tool RTTC. This software is provided as a source-code module written in the programming language PYTHON[1].

Documentation is available in three forms: first, this chapter serves as a complete reference manual to the programming interface, that is, to all visible methods of the module. Second, an HTML documentation generated by DOXYGEN[2] and PYTHFILTER[3] describes all methods and classes of the module in a more technical way. Third, the source code itself contains further comments on the implementation details.

A.1. System requirements

The software requires version 2.2 or above of the PYTHON interpreter.

A.2. User interface

A.2.1. Settings

Time resolution of single events in seconds

possible values	all fractions greater than 0
default value	`0.01`
get method	`getRealTimeResolution()`
set method	`setRealTimeResolution(`*float*`)`

Search algorithm

possible values	0 for forward DFS 1 for backward DFS 2 for on-the-fly forward DFS
default value	`2`
get method	`getSearchAlgorithm()`
set method	`setSearchAlgorithm(`*int*`)`

Note that this setting is ignored unless the verification method is NFA acceptance.

[1] See `http://www.python.org/`.

[2] Doxygen is a documentation system for various programming languages. More information on Doxygen is available at `http://www.doxygen.org/`.

[3] Since Doxygen is not directly applicable to Python source code, we have used Matthias Baas' pythfilter as a preprocessing step. More information on using Doxygen with Python source code is available from his homepage at `http://i31www.ira.uka.de/~baas/pydoxy/`.

Verification method

possible values	0 for simple rewriting 1 for NFA acceptance 2 for event-consuming rewriting
default value	`0`
get method	`getVerificationMethod()`
set method	`setVerificationMethod(`*`int`*`)`

A.2.2. Initialisation

At first, in order to access the software, the Python module `rttc` must be imported by calling `import rttc`. Then, an instance of the trace checker can be created by initialising a `TraceChecker` object `t`, that is, by calling `t = rttc.TraceChecker()`.

A.2.3. Verification

The main method of the software is `TraceChecker.check`. It is executed on a `TraceChecker` object `t` by calling `t.check(`*`string`*`, `*`string`*`)` with the first argument containing the path to a stored trace and the second argument containing a valid property string. This method returns *`True`* if the trace satisfies the given property and *`False`* otherwise.

A.3. Trace syntax

In order for the algorithms to work reliably and efficiently, valid traces have to comply with a number of assumptions which are not checked within this software:

- The trace is a list of events, seperated by linebreaks.
- The first trace line contains only variable names, seperated by tabulators. Variables names that are referenced in the property may only occur once in the list of variables names. All variable names of the trace are translated into lower case, and valid properties may only contain lower-case variable names (as defined below).
- All following trace lines contain only *float* or *int* values, seperated by tabulators. The first value of each trace line is the timestamp of the corresponding event. All trace lines are ordered by their timestamps, in ascending order. The first line must correspond to timestamp 0.
- For each timestamp, there must be exactly one complete trace line, that is, one line containing values for every variable of the property. Note that there may be arbitrarily many incomplete lines before and after complete lines.

A.4. Property syntax

In this section, we describe the input syntax of valid properties by giving EBNF[4] expressions for them. Note that, although they are usually allowed, optional spaces have been omitted here for better reading; they are only mandatory where indicated by the symbol ⊔. Except in the middle of ⟨*value*⟩ expressions, parantheses can always be used to avoid ambiguity, but they are never mandatory.

A.4.1. Atomic formulae

$$
\begin{aligned}
letter &::= a \mid \cdots \mid z \\
digit &::= 0 \mid \cdots \mid 9 \\
chraracter &::= \langle letter \rangle \mid \langle digit \rangle \mid _ \\
variableName &::= \langle character \rangle \; \{\langle character \rangle\} \\
\\
value &::= [-] \; \langle digit \rangle \; \texttt{.} \; \langle digit \rangle \; \{\langle digit \rangle\} \\
arithmeticComparison &::= \langle variableName \rangle \; \langle comparisonRelation \rangle \; \langle value \rangle \\
\\
truthValue &::= true \mid false \\
atomicFormula &::= \langle arithmeticComparison \rangle \mid \langle truthValue \rangle
\end{aligned}
$$

A.4.2. Propositional operators

$$
\begin{aligned}
unaryPropositionalOperator &::= \texttt{!} \\
binaryPropositionalOperator &::= \texttt{\&\&} \mid \texttt{||} \mid \texttt{->} \mid \texttt{<->}
\end{aligned}
$$

A.4.3. LTL operators

$$
\begin{aligned}
unaryLTLOperator &::= \texttt{F} \mid \texttt{G} \\
binaryLTLOperator &::= \texttt{U} \mid \texttt{R}
\end{aligned}
$$

A.4.4. RTLTL operators

$$
\begin{aligned}
unaryRTLTLOperator &::= (\texttt{F} \mid \texttt{G}) \; \texttt{[}\langle value \rangle\texttt{,}\langle value \rangle\texttt{]} \mid (\texttt{X} \mid \texttt{Y}) \; \texttt{[}\langle value \rangle\texttt{]} \\
binaryRTLTLOperator &::= (\texttt{U} \mid \texttt{R}) \; \texttt{[}\langle value \rangle\texttt{,}\langle value \rangle\texttt{]}
\end{aligned}
$$

[4]EBNF stands for *Extended Backus-Naur Form*, which is a common notation for the description of formal languages [HMU01].

A.4.5. RTLTL formulae

$$
\begin{aligned}
unaryOperator \;&::=\; \langle unaryPropositionalOperator\rangle \mid \\
&\quad \langle unaryLTLOperator\rangle \mid \\
&\quad \langle unaryRTLTLOperator\rangle
\end{aligned}
$$

$$
\begin{aligned}
binaryOperator \;&::=\; \langle binaryPropositionalOperator\rangle \mid \\
&\quad \langle binaryLTLOperator\rangle \mid \\
&\quad \langle binaryRTLTLOperator\rangle
\end{aligned}
$$

$$
\begin{aligned}
formula \;&::=\; \langle atomicFormula\rangle \mid \\
&\quad \langle unaryOperator\rangle \sqcup \langle formula\rangle \mid \\
&\quad \langle formula\rangle \sqcup \langle binaryOperator\rangle \sqcup \langle formula\rangle
\end{aligned}
$$

A.5. Usage examples

```
import rttc
t = rttc.TraceChecker()

t.check('trace.txt', 'G error = 0')
t.check('trace.txt', 'G error_count < 10')
t.check('trace.txt', 'blocked = 1 U start = 1')

t.check('trace.txt', 'F G[0,0.1] error_symptom = 1')
t.check('trace.txt',
        'G (G[0,0.1] error_symptom = 1 -> F[0,0.02] error = 1)')

t.check('trace.txt', 'G (request = 1 -> F[0.1, 0.2] grant = 1)')
t.check('trace.txt',
        'G (X[0.2] true -> request = 1 -> F[0.1,0.2] grant = 1)')
```

List of Figures

3.1. Non-splitting tableau rules for LTL to GBA translation. 21
3.2. Splitting tableau rules for LTL to GBA translation. 21
3.3. Non-splitting tableau rules for RTLTL to NFA translation. . . . 23
3.4. Splitting tableau rules for RTLTL to NFA translation. 24
3.5. A ranking function for RTLTL formulae in negation normal form. 25
3.6. On-the-fly algorithm for RTLTL to NFA translation. 29

4.1. A ranking function for RTLTL formulae. 36

5.1. Performance evaluation on the traffic-light example. 42

Bibliography

[AH93] Rajeev Alur and Thomas A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Büc60] J. Richard Büchi. Weak second order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*. Stanford University Press, 1962.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[CKV03] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage Metrics for Formal Verification. In *Proceedings of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[CVWY92] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICS 1999)*. ACM Press, 1999.

[DFRR04] Rocco Deutschmann, Matthias Fruth, Horst Reichel, and Hans-Christian Reuss. Trace Checking with Real-Time Specifications. In *Proceedings of the 5th Symposium on Formal Methods*

for Automation and Safety in Railway and Automotive Systems (FORMS 2004), 2004.

[DGV99] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved Automata Generation for Linear Temporal Logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV 1999)*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[DP01] Nachum Dershowitz and David Plaisted. Rewriting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1. The MIT Press, 2001.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *Proceedings of the 15nd International Workshop on Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[EH00] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi Automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[EMSS92] E. Allen Emerson, Aloysius K. Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative Temporal Reasoning. *Real-Time Systems*, 4(4):331–352, 1992.

[Ete02] Kousha Etessami. A Hierarchy of Polynomial-Time Computable Simulations for Automata. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*, volume 2421 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[EWS01] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001)*, volume 2076 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Fri03] Carsten Fritz. Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA 2003)*, volume 2759 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Fru02] Matthias Fruth. Überwachung von Java-Programmen mittels Java PathFinder. Bachelor thesis, Dresden University of Technology, Dresden, Germany, 2002.

[FS04] Bernd Finkbeiner and Henny Sipma. Checking Finite Traces Using Alternating Automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

[FSS02] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of the 2nd Workshop on Runtime Verification (RV 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[FW02] Carsten Fritz and Thomas Wilke. Simulation Relations for Alternating Büchi Automata. Extended Technical Report, Christian-Albrechts-Universität Kiel, Kiel, Germany, 2002.

[GBS02] Sankar Gurumurthy, Roderick Bloem, and Fabio Somenzi. Fair Simulation Minimization. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[GH01] Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*. IEEE Computer Society Press, 2001.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[GPVW95] Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th IFIP WG 6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV 1995)*, volume 38 of *IFIP Conference Proceedings*. Chapman and Hall, 1995.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[HR01a] Klaus Havelund and Grigore Roşu. Monitoring Programs using Rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*. IEEE Computer Society Press, 2001.

[HR01b] Klaus Havelund and Grigore Roşu, editors. *Proceedings of the 1st Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[Kam68] Johann A. W. Kamp. *Tense Logic and the theory of linear order*. PhD thesis, University of California, Los Angeles, California, USA, 1968.

[Koy90] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL 1985)*, 1985.

[LT00] Christof Löding and Wolfgang Thomas. Alternating Automata and Logics over Infinite Words. In *Proceedings of the IFIP International Conference on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics (IFIP TCS 2000)*, volume 1872 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[MS03] Nicolas Markey and Philippe Schnoebelen. Model Checking a Path (Preliminary Report). In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR 2003)*, volume 2761 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science (LICS 1988)*, 1988.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS 1977)*. IEEE Computer Society Press, 1977.

[RH] Grigore Roşu and Klaus Havelund. Rewriting-based Techniques for Runtime Verification. *Automated Software Engineering (to appear)*.

[RH01] Grigore Roşu and Klaus Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. Technical Report 01.15, Research Institute for Advanced Computer Science, Moffett Field, California, USA, 2001.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi Automata from LTL Formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[SPH82] Rivi Sherman, Amir Pnueli, and David Harel. Is the Interesting Part of Process Logic Uninteresting?: A Translation from PL to PDL. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL 1982)*, 1982.

[Sti01] Colin Stirling. *Modal and Temporal Properties of Processes*. Springer-Verlag, 2001.

[Tau03] Heikki Tauriainen. On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata. Research report A83, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, 2003.

[TH02] Heikki Tauriainen and Keijo Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):57–70, 2002.

[Tho81] Wolfgang Thomas. A combinatorial approach to the theory of ω-automata. *Information and Computation*, 48:261–283, 1981.

[Tho97] Wolfgang Thomas. Languages, Automata, and Logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Beyond Words*, volume 3 of *Handbook of Formal Languages*. Springer-Verlag, 1997.

[Var97] Moshe Y. Vardi. Alternating Automata: Unifying Truth and Validity Checking for Temporal Logics. In *Proceedings of the 14th International Conference on Automated Deduction (CADE 1997)*, volume 1249 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Var01] Moshe Y. Vardi. Branching vs. Linear Time: Final Showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[VW86] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the 1st Symposium on Logic in Computer Science (LICS 1986)*. IEEE Computer Society Press, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about Infinite Computation Paths (Extended Abstract). In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS 1983)*. IEEE Computer Society Press, 1983.