Runtime Verification for Linear-Time Temporal Logic

Martin Leucker(✉)

Institut für Softwaretechnik und Programmiersprachen, Universtität zu Lübeck, Lübeck, Germany
leucker@isp.uni-luebeck.de

Abstract. In this paper and its accompanying tutorial, we discuss the topic of runtime verification for linear-time temporal logic specifications. We recall the idea of runtime verification, give ideas about specification languages for runtime verification and develop a solid theory for linear-time temporal logic. Concepts like monitors, impartiality, and anticipation are explained based on this logic.

1 Introduction

Software and software systems are increasingly ubiquitous in everyday life. Besides traditional applications such as word processors or spreadsheets running on workstations, software is an important part of consumer devices such as mobile phones or digital cameras, and functions as embedded control devices in cars or in power plants. Especially in such embedded application domains, it is essential to guarantee that the deployed software works in a correct, secure, and reliable manner, as life may depend on it.

For example, the software within a car's anti-skid system must speed with exactly the right velocity to stabilize the car. Moreover, for a power plant it is important that no intruder gets control over the plant and that it works also in case of a partial break-down of some of its parts.

Software engineering has been driven as a field by the struggle for guaranteed quality properties ever since, but nowadays and especially in the embedded domain, legislation and certification authorities are requiring proof of the most critical software properties in terms of a documented verification process.

Traditionally, one considers three main verification techniques: *theorem proving* [BC04], *model checking* [CGP01], and *testing* [Mye04,BJK+05]. Theorem proving, which is mostly applied manually, allows to show correctness of programs similarly as a proof in mathematics shows correctness of a theorem. Model checking, which is an automatic verification technique, is mainly applicable to finite-state systems. Testing covers a wide field of diverse, often ad-hoc, and incomplete methods for showing correctness, or, more precisely, for finding bugs.

These techniques are subject to a number of forces imposed by the software to build and the development process followed, and provide different trade-offs between them. For example, some require a formal model, like model checking,

J.P. Bowen et al. (Eds.): SETSS 2016, LNCS 10215, pp. 151–194, 2017.
DOI: 10.1007/978-3-319-56841-6_5

give stronger or weaker confidence, like theorem proving over testing, or are graceful in case of error handling.

Runtime verification is being pursued as a *lightweight* verification technique complementing verification techniques such as model checking and testing and establishes another trade-off point between these forces. One of the main distinguishing features of runtime verification is due to its nature of being performed at runtime, which opens up the possibility to *act* whenever incorrect behavior of a software system is detected.

The aim of this course is to give a comprehensive introduction into runtime verification based on linear-time temporal logic. Rather than completeness, we aim for a solid formal underpinning of the concepts.

The paper is organized as follows: In the next section, we provide an informal introduction to the field of runtime verification. We sketch main ideas intuitively and describe several application areas. At the heart of runtime verification, we identify the specification of correctness properties and the synthesis of corresponding monitors, which may then be used for verification but also for steering a system. In Sects. 3–5, we develop formal semantics for one specification language viz. linear-time temporal logic together with corresponding monitoring procedures. In Sect. 3, we assume the execution be terminated, while in Sects. 4 and 5, we consider online monitoring with continuously expanding executions. In Sect. 4, we discuss the concept of impartiality in detail while Sect. 5 focusses on anticipation.

2 Fundamental Ideas of Runtime Verification

Let us start with recalling the fundamental concepts of verification and let us describe, from an abstract point of view, the concept of runtime verification.

First, let us clarify the terms *verification* and, to contrast its idea, also *validation*. Validation and verification can be distinguished by checking which of the following two questions gets answered: [Boe81]

Validation. Are we building the right product? – (Does the system meet the client's expectations?)

Verification. "Are we building the product right?" – (Does the system meet its specification?)

Definition 1 (Verification). *Verification is comparing code with its specification.*

A verification technique should therefore always be of a formal nature, while validation, necessarily, cannot fully rely on formal concepts as the customer's expectations are not formalized. As we will see, *runtime verification is a verification technique*, despite it only verifies partially the system under scrutiny.

We follow [DGR04] and define a *software failure* as a deviation between the *observed* behavior and the *required* behavior of the software system. A *fault* is defined as the deviation between the current behavior and the expected behavior,

which is typically identified by a deviation of the current and the expected state of the system. A fault might lead to a failure, but not necessarily. An error, on the other hand, is a mistake made by a human that results in a fault and possibly in a failure.

As we have just learned, verification comprises all techniques suitable for showing that a system satisfies its specification.

Traditional verification techniques comprise theorem proving [BC04], model checking [CGP01], and testing [Mye04, BJK+05]. *Runtime verification*,[1] is a relatively new verification technique, which manifested itself within the previous years as a *lightweight* verification technique:

Definition 2 (Runtime Verification). Runtime verification *(RV) is the discipline of computer science that deals with the study, development and application of those verification techniques that allow for checking whether a* run *of a system under scrutiny* satisfies or violates *a given correctness property.*

Definition 3 (Run). *A run of a system is a possibly infinite sequence of the system's states. Formally, a run may be considered as a possibly infinite* word *or* trace*.*

Runs are formed by current variable assignments, or as the sequence of actions a system is emitting or performing.

Definition 4 (Execution). *An* execution *of a system is a* finite prefix *of a run and, formally, it is a finite trace. When running a program, we can only observe executions, which, however, restrict the corresponding evolving run as being their prefix.*

In runtime verification, we check whether a run of a system adhere to given correctness properties. RV is primarily used on executions. A monitor checks whether an execution meets a correctness property.

Definition 5 (Monitor). *A* monitor *is a device that reads a finite trace and yields a certain* verdict*.*

Figure 1 shows a monitor M which tests an execution of the system consisting of the components C_i against a formal correctness property. These components can be hardware components, procedures or any other structuring element of the system. The lines can be the wiring of hardware components, the call stack of procedures or any other connection of the components.

A monitor may use more than one input stream as opposed to what is shown in Fig. 1. Likewise, a monitor can check the relations of multiple values. In *distributed runtime verification*, a set of monitors operating at different locations may combine their monitoring power to deduce the suitable verdict [SVAR04, MB15, SS14, BLS06a].

Here, a verdict is typically a truth value from some truth domain. A truth domain is a lattice with a unique top element *true* and a unique bottom

[1] http://www.runtime-verification.org.

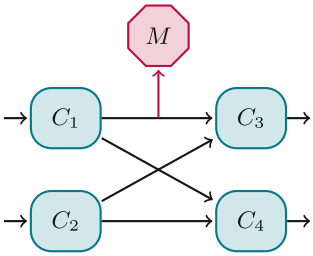


Fig. 1. Monitor M checks correctness of components C_i.

element *false*. This definition covers the standard two-valued truth domain $\mathbb{B} = \{true, false\}$ but also fits for monitors yielding a probability in $[0, 1]$ with which a given correctness property is satisfied. Sometimes, one might be even more liberal and consider also verdicts that are not elements of a truth domain, though we do not follow this view in this paper.

A monitor may on one hand be used to check the *current* execution of a system. In this setting, which is termed *online monitoring*, the monitor should be designed to consider executions in an *incremental fashion* and in an *efficient manner*. On the other hand, a monitor may work on a (finite set of) *recorded* execution(s), in which case we speak of *offline monitoring*.

RV and the Word Problem. In its simplest form, a monitor decides whether the current execution satisfies a given correctness property by outputting either *yes/true* or *no/false*. Formally, when $[\![\varphi]\!]$ denotes the set of valid executions given by property φ, runtime verification boils down to checking whether the execution w is an element of $[\![\varphi]\!]$. Thus, in its mathematical essence, runtime verification answers the *word problem*, i. e. the problem whether a given word is included in some language. Note that often, the word problem can be decided with lower complexity compared to, for example, the subset problem: Language containment for non-deterministic finite-automata is PSPACE-complete [SC85], while deciding whether a given word is accepted by a non-deterministic automaton is NLOGSPACE-complete [HU79].

2.1 Some Requirements on Monitors

Definition 6 (Impartiality). Impartiality *requires that a finite trace is not evaluated to true or, respectively false, if there still exists a (possibly infinite) continuation leading to another verdict.*

Definition 7 (Anticipation). Anticipation *requires that once every (possibly infinite) continuation of a finite trace leads to the same verdict, then the finite trace evaluates to this very same verdict.*

Intuitively, the first maxim postulates that a monitor only decides for *false*—meaning that a misbehavior has been observed—or *true*—meaning that the

current behavior fulfills the correctness property, regardless of how it continues—only if this is indeed the case. Clearly, this maxim requires to have at least three different truth values: *true*, *false*, and *inconclusive*, but of course more than three truth values might give a more precise assessment of correctness. The second maxim requires a monitor to indeed report *true* or *false*, if the correctness property is indeed violated or satisfied. In simple words, *impartiality* and *anticipation,* guarantee that the semantics is neither premature nor overcautious in its evaluations.

See the forthcoming sections for a more elaborate discussion of these issues in the context of linear temporal logic.

In runtime verification, monitors are typically generated automatically from some high-level specification. As runtime verification has its roots in model checking, often some variant of linear temporal logic, such as LTL [Pnu77], is employed. But also formalisms inspired by the linear μ-calculus have been introduced, for example in [DSS+05], which explains an accompanying monitoring framework.

Actually, one of the key problems addressed in runtime verification is the generation of monitors from high-level specifications, and we discuss this issue in much more detail in this course.

2.2 Runtime Verification in the Plethora of Verification Techniques

RV Versus Testing. As runtime verification does not consider each possible execution of a system, but just a single or a finite subset, it shares similarities with *testing*, which terms a variety of usually incomplete verification techniques.

Typically, in testing one considers a finite set of finite input-output sequences forming a *test suite* [PL04]. Test-case execution is then checking whether the output of a system agrees with the predicted one, when giving the input sequence to the system under test.

A different form of testing, however, is closer to runtime verification, which is sometimes termed *oracle-based testing*. Here, a test-suite is only formed by input-sequences. To make sure that the output of the system is as anticipated, a so-called *test oracle* has to be designed and "attached" to the system under test. Thus, in essence, runtime verification can be understood as this form of testing. There are, however, differences in the foci of runtime verification and oracle-based testing:

- In testing, an oracle is typically defined directly, rather than generated from some high-level specification.
- On the other hand, providing a suitable set of input sequences to "exhaustively" test a system, is rarely considered in the domain of runtime verification.

Thus, runtime verification can also considered as a form of *passive testing*.

When monitors are equipped in the final software system, one may also understand runtime verification as "testing forever", which makes it, in a certain sense, complete.

RV Versus Model Checking. In essence, model checking describes the problem of determining whether, given a model $\mathcal{M}$ and a correctness property φ, all computations of $\mathcal{M}$ satisfy φ. Model checking [CGP01], which is an automatic verification technique, is mainly applicable to finite-state systems, for which all computations can exhaustively be enumerated, though model checking techniques for certain pushdown systems or counter machines exist as well.

In the automata theoretic approach to model checking [VW86], a correctness property φ is transformed to an automaton $\mathcal{M}_{\neg\varphi}$ accepting all runs violating φ. This automaton is put in parallel to a model $\mathcal{M}$ to check whether $\mathcal{M}$ has a run violating φ.

Runtime verification has its origins in model checking, and, to a certain extend, the key problem of generating monitors is similar to the generation of automata in model checking. However, there are also important differences to model checking:

- While in model checking, *all executions* of a given system are examined to answer whether they satisfy a given correctness property φ, which corresponds to the language inclusion problem, runtime verification deals with the word problem.
- While model checking typically considers *infinite* traces, runtime verification deals with *finite* executions—as executions have necessarily to be finite.
- While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an *incremental fashion.*

These differences make it necessary to adapt the concepts developed in model checking to be applicable in runtime verification. For example, while checking a property in model checking using a kind of backwards search in the model is sometimes a good choice, it should be avoided in online monitoring as this would require, in the worst case, the whole execution trace to be stored for evaluation.

From an application point of view, there are also important differences between model checking and runtime verification.

Runtime verification deals only with observed executions as they are generated by the real system. Thus runtime verification is applicable to *black box systems* for which no system model is at hand. In model checking, however, a suitable model of the system to be checked must be constructed as—before actually running the system—all possible executions must be checked.

If such a precise model of the underlying system is given, and, if moreover a bound on the size of its state space is known, powerful, so-called *bounded model-checking techniques* can be applied [BCC+03] for analyzing the system. The crucial idea, which is equally used in conformance testing [Vas73, Cho78], is that for every finite-state system, an infinite trace must reach at least one state twice. Thus, if a finite trace reaches a state a second time, the trace can be extended to an infinite trace by taking the corresponding loop infinitely often. Likewise, considering all finite traces of length up-to the state-place plus one, one

has information on all possible loops of the underlying system, without actually working on the system's state space directly.

Clearly, similar correspondences would be helpful in runtime verification as well. However, in runtime verification, an upper bound on the system's state space is typically not known. More importantly, the states of an observed execution usually do not reflect the system's state completely but do only contain the value of certain variables of interest. Thus, seeing a state twice in an observed execution does not allow to infer that the observed loop can be taken ad infinitum.

That said, current research also focusses on the combination of runtime verification and model checking respectively formal verification techniques. See [Leu12, CAPS15] for details.

Furthermore, model checking suffers from the so-called *state explosion problem*, which terms the fact that analyzing all executions of a system is typically been carried out by generating the whole state space of the underlying system, which is often huge. Considering a single run, on the other hand, does usually not yield any memory problems, provided that when monitoring online only a finite *history* of the execution has to be stored.

Last but not least, in online monitoring, the complexity for *generating* the monitor is typically negligible, as the monitor is often only generated once. However, the *complexity of the monitor*, i. e. its memory and computation time requirements for checking an execution are of important interest, as the monitor is part of the running system and should influence the system as less as possible.

2.3 Applications

Runtime Reflection. Runtime verification itself deals (only) with the *detection* of violations (or satisfactions) of correctness properties. Thus, whenever a violation has been observed, it typically does not influence or change the program's execution, say for trying to repair the observed violation. However, runtime verification is the basis for concepts also dealing with observed problems, as we discuss in this section.

The idea of monitoring a system and reacting are to a certain extent covered by the popular notion of *FDIR*, which stands for *Fault Detection, Identification, and Recovery* or sometimes for *Fault Diagnosis, Isolation, and Recovery* or various combinations thereof [CR94]. The general idea of FDIR is that a failure within a system shows up by a fault. A fault, however, does typically not *identify* the failure: for example, there might be different *reasons* why a monitored client does not follow a certain protocol, one of them, e.g., that it uses an old version of a protocol. If this is identified as the failure, reconfiguration may switch the server to work with the old version of the protocol.

Crow and Rushby instantiated the scheme FDIR using Reiter's theory of diagnosis from first principles in [CR94]. Especially, the detection of errors is carried out using diagnosis techniques. In *runtime reflection* [BLS06b], runtime verification is proposed as a tool for fault detection, while a simplified version of Reiter's diagnosis is suggested for identification.

Runtime reflection (RR) is an architecture pattern for the development of reliable systems.

– A *monitoring layer* is enriched with
– a *diagnosis layer* and a subsequent
– *mitigation layer*.

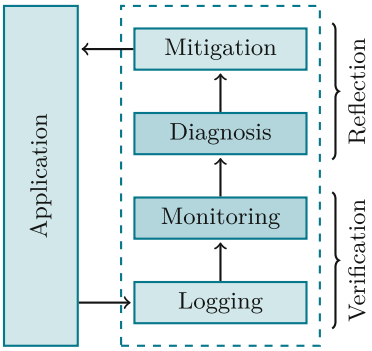


Fig. 2. An application and the layers of the runtime reflection framework.

The architecture consists of four layers as shown in Fig. 2, whose role will be sketched in the subsequent paragraphs.

The role of the *logging layer* is to observe system events and to provide them in a suitable format for the monitoring layer. Typically, the logging layer is realized by adding code annotations within the system to build. However, separated stand-alone loggers, logging for example network traffic, can realize this layer as well. While the goal of a logger is to provide information on the current run to a monitor, it may not assume (much) on the properties to be monitored.

The *monitoring layer* consists of a number of monitors (complying to the logger interface of the logging layer) which observe the stream of system events provided by the logging layer. Its task is to detect the presence of faults in the system without actually affecting its behavior. In runtime reflection, it is assumed to be implemented using runtime verification techniques. If a violation of a correctness property is detected in some part of the system, the generated monitors will respond with an alarm signal for subsequent diagnosis.

Following FDIR, we separate the detections of faults from the identification of failures. The *diagnosis layer* collects the verdicts of the distributed monitors and deduces an explanation for the current system state. For this purpose, the diagnosis layer may infer a (minimal) set of system components, which must be assumed faulty in order to explain the currently observed system state. The procedure is solely based upon the results of the monitors and general information on the system. Thus, the diagnostic layer is not directly communicating with the application.

The results of the system's diagnosis are then used in order to *reconfigure* the system to mitigate the failure, if possible. However, depending on the diagnosis and the occurred failure, it may not always be possible to re-establish a determined system behavior. Hence, in some situations, e. g., occurrence of fatal errors, a recovery system may merely be able to store detailed diagnosis information for off-line treatment.

Monitor-Oriented Programming. Monitoring-Oriented Programming (MOP) [CR07], proposed by Feng and Rosu, is a software development methodology, in which the developer specifies desired properties using a variety of (freely definable) specification formalisms, along with code to execute when properties are violated or validated. The MOP framework automatically generates monitors from the specified properties and then integrates them together with the user-defined code into the original system. Thus, it extends ideas from runtime verification by means for *reacting* on detected violations (or validations) of properties to check. This allows the development of *reflective* software systems: A software system can monitor its own execution such that the subsequent execution is influenced by the code a monitor is executing in reaction to its observations– again influencing the observed behavior and consequently the behavior of the monitor itself.

RR differs from monitor-oriented programming in two dimensions. First, MOP aims at a programming methodology, while RR should be understood as an architecture pattern. This implies that MOP support has to be tight to a programming language, for example Java resulting in jMOP, while in RR, a program's structure should highlight that it follows the RR pattern. The second difference of RR in comparison to MOP is that RR introduces a diagnosis layer not found in MOP.[2]

When to Use RV?. Let us conclude the description of runtime verification by listing certain application domains, highlighting the distinguishing features of runtime verification:

– The verification verdict, as obtained by model checking or theorem proving, is often referring to a *model of the real system* under analysis, since applying these techniques directly to the real implementation would be intractable. The model typically reflects most important aspects of the corresponding implementation, and checking the model for correctness gives useful insights to the implementation. Nevertheless, the implementation might behave slightly different than predicted by the model. Runtime verification may then be used to easily *check the actual execution* of the system, to make sure that the implementation really meets its correctness properties. Thus, runtime verification may act as a *partner to theorem proving and model checking.*

[2] Clearly, in the MOP framework, a diagnosis can be carried out in the code triggered by a monitor. This yields a program using the MOP methodology and following the RR pattern.

- Often, some information is available *only at runtime* or is conveniently checked at runtime. For example, whenever library code with no accompanying source code is part of the system to build, only a vague description of the behavior of the code might be available. In such cases, runtime verification is an *alternative to theorem proving and model checking.*
- The behavior of an application may *depend heavily on the environment* of the target system, but a precise description of this environment might not exist. Then it is not possible to obtain the information necessary to test the system in an adequate manner. Moreover, formal correctness proofs by model checking or theorem proving may only be achievable by taking certain assumptions on the behavior of the environment—which should be checked at runtime. In this scenario, runtime verification outperforms classical testing and *adds on formal correctness proofs* by model checking and theorem proving.
- In the case of systems where *security is important* or in the case of safety-critical systems, it is useful also to monitor behavior or properties that have been statically proved or tested, mainly to have a double check that everything goes well: Here, runtime verification acts as a partner of theorem proving, model checking, and testing.

The above mentioned items can be found in a combined manner especially in highly dynamic systems such as *adaptive*, *self-organizing*, or *self-healing* systems (see [HS06] for an overview on such approaches towards self-management).

The behavior of such systems depends heavily on the environment and changes over time, which makes their behavior hard to predict—and hard to analyze prior to execution. To assure certain correctness properties of especially such systems, we expect runtime verification to become a major verification technique.

Let us conclude this subsection with a general taxonomy of runtime verification aspects shown in Fig. 3.

2.4 Gathering Information About Executions

In this course, we mainly focus on specification means for correctness properties and corresponding monitor synthesis procedures. However, one of the fundamental question in runtime verification is also how to obtain the underlying atomic system events or observations that build the basis for correctness specifications. We leave details on this to other works but only list fundamental concepts.

A typical approach is instrument the code to provide logging information, say be means of code manipulations. These may be applied directly on the source code, the byte code or binary code level. The programmer of the underlying software may have also used dedicated logging frameworks and these are the sequence of log events is to be analyzed. Operating system typically also provide tools for providing trace information of running processes. A relatively new direction is to use debug capabilities of the processors of underlying execution platform and to monitor the system using dedicated hardware. Especially the latter approach caters for analyzing timing properties as the system does not get influenced itself by the monitoring process. Figure 4 summarizes the options.

Fig. 3. Taxonomy for RV

2.5 RV Frameworks

The popularity of runtime verification can also be witnessed by the large number of corresponding runtime verification frameworks. An (incomplete) list of current runtime verification frameworks as shown in Fig. 5.

We will now study several high-level specification languages and discuss adaptions of their semantics suitable for runtime verification. Mostly, we consider the linear-time temporal logic LTL, first considered for specifications of computations by Amir Pnueli [Pnu77].

2.6 A Primer on Linear-Time Temporal Logic

Runs are Words. The system to monitor is typically driven by some program that consists of several commands and hereby interacts with its environment.

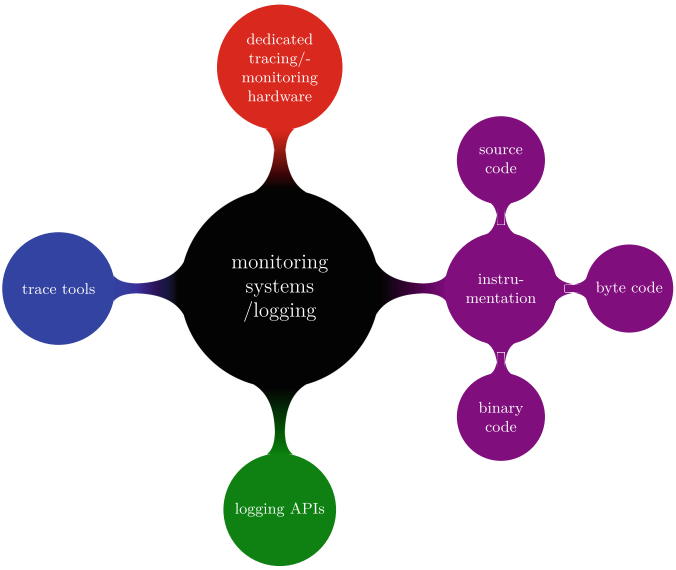


Fig. 4. Observing the System

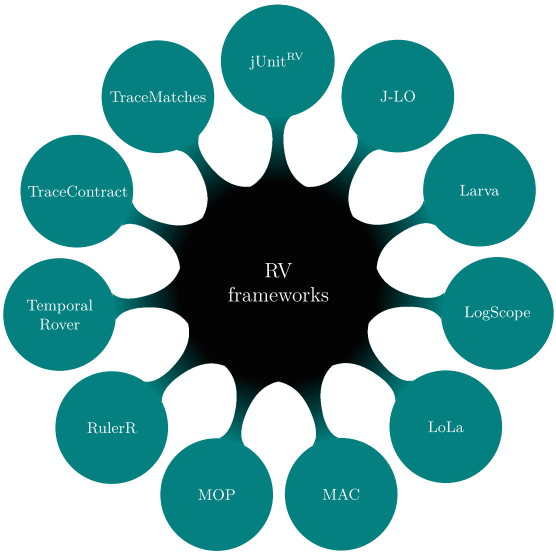


Fig. 5. An (imcomplete) list of runtime verification framworks

The idea of monitoring is now to observe and analyze partially such an execution which is built-up by artefacts. These artifacts may vary depending on the application of runtime verification. For example, we may observe *events* that occur within the system's execution, or, we may have access to the system's

variables in each execution step, giving a comprehensive picture of the system's state. Often, we can monitor the system's input-/output behavior. These settings have in common that an execution can easily be described by a linear sequence of the artifacts to be observed.

In runtime verification, we therefore aim at specifying the shape of linear sequences. In a second step, the goal is to synthesize monitors that check whether a single or a set of linear sequences adheres to the specification.

Formally, executions and runs can be understood as (finite or infinite, respectively) *words* over the corresponding alphabet, i.e., the power set of the atomic propositions.

Let Σ be an alphabet and $n \in \mathbb{N}$.

We then use the following notation shown in Table 1.

Table 1. Notation for words.

Notation	Meaning
Σ^*	Set of all *finite* words over Σ
Σ^n	All words in Σ^* of length n
$\Sigma^{\leq n}$	All words in Σ^* of length at most n
$\Sigma^{\geq n}$	All words in Σ^* of length at least n
Σ^+	$= \Sigma^{\geq 1}$
Σ^ω	Set of all *infinite* words over Σ
Σ^∞	$= \Sigma^* \cup \Sigma^\omega$

A state can be seen as an element $a \in \Sigma$. Now a run is an infinite word $w \in \Sigma^\omega$ and an execution a finite prefix $w \in \Sigma^*$. Runtime verification is about checking if an execution is correct, so we need to specify the set of correct executions as a language $L \subseteq \Sigma^*$. Therefore a correctness property is a language L.

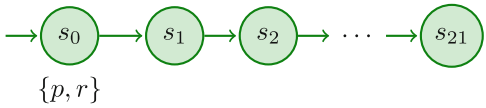
In general, logical calculi are a versatile tool and basis for deriving specification formalisms. Linear-time temporal logic (LTL) is especially useful for specifying properties of linear sequences. In the following we study different versions of LTL with a semantics adapted towards its application in runtime verification.

Linear temporal logic builds on propositional logic. As such, it allows the definition of *atomic propositions*, typically denoted by letters such as p or q, as well as the combination of formulas by *conjunction*, denoted by $\wedge$, *disjunction*, denoted by $\vee$, and *negation*, denoted by $\neg$. Clearly, only one of conjunction or disjunction is necessary in the presence of negation. However, for convenience, we typically use both operators in our logics.

Propositional logic can only talk about the current situation, let it be a certain time step, an event, a current memory assignment etc. It becomes a *temporal logic* by adding temporal quantifiers (sometimes also called operators).

Using propositional logic without temporal operators we describe only the first state.

Example 1. Consider AP $= \{p, q, r, s\}$ and an initial state s_0 of an execution w in which p and r holds. We then have



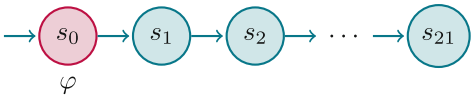
$w \models \text{true}$ $\quad$ $w \not\models \text{false}$

$w \models p$ $\quad$ $w \models p \wedge r \vee q$

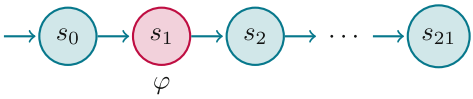
$w \models \neg q \wedge \neg s$ $\quad$ $w \not\models q$.

In LTL, we usually have two operators *next*, denoted by X, and, *until*, denoted by U. *Next* is a unary operator and the meaning of a formula $X\varphi$ is that φ has to hold in the *next* situation, formally the next position of a linear sequence, or *word*. *Until* is a binary operator and the meaning of $\varphi\ U\ \psi$ is that ψ has to hold at some point and φ has to hold up-to this moment. With these two temporal operators, it is now possible to specify not only propositions of the current situation but also on future situations. However, for convenience, we work with further operators, as visualized in the following.

Formula: φ The formula φ holds for an execution if φ holds in the first state s_0 of that execution.

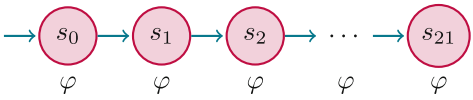


Next: $\mathrm{X}\varphi$ The formula $\mathrm{X}\varphi$ holds in state s_i if φ holds in state s_{i+1}. If there is no state s_{i+1} then $\mathrm{X}\varphi$ *never* holds.



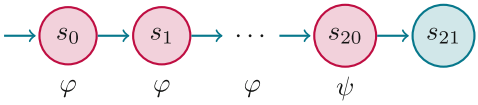
Weak Next: $\overline{\mathrm{X}}\varphi$ The formula $\overline{\mathrm{X}}\varphi$ holds in state s_i if φ holds in state s_{i+1}. If there is no state s_{i+1} then $\overline{\mathrm{X}}\varphi$ *always* holds.

Globally: $\mathrm{G}\varphi$ The formula $\mathrm{G}\varphi$ holds in state s_i if φ holds in all states s_j for $j \geq i$.



Finally: $\mathrm{F}\varphi$ The formula $\mathrm{F}\varphi$ holds in state s_i if there is a state s_j for $j \geq i$ in which φ holds.

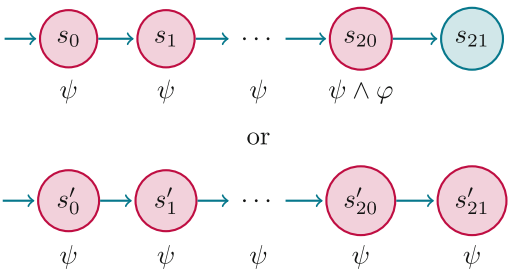
Until: $\varphi \,\mathrm{U}\, \psi$ The formula $\varphi \mathrm{U} \psi$ holds in state s_i if there is a state s_j for $j \geq i$ in which ψ holds and φ holds in all states s_k for $i \leq k < j$.



Notice that a state in which φ holds is not required in all cases!

Release: $\varphi \,\mathrm{R}\, \psi$ The formula $\varphi \mathrm{R} \psi$ holds in state s_i if there is a state s_j for $j \geq i$ in which φ holds and ψ holds in all states s_k for $i \leq k \leq j$.

If there is no such state s_j then the $\varphi \mathrm{R} \psi$ holds if ψ holds in all states s_k for $k \geq i$.



LTL Syntax. Let us give a precisedefinition of LTL's syntax.

Definition 8 (Syntax of LTL Formulae). *Let $p \in \mathrm{AP}$ be an atomic proposition from a finite set of atomic propositions* AP*. The set of LTL formulae is inductively defined by the following grammar:*

$$\begin{aligned} \varphi ::= \mathrm{true} \mid p \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi \mid F\varphi \mid \\ \mathrm{false} \mid \neg p \mid \varphi \wedge \varphi \mid \overline{X}\varphi \mid \varphi R \varphi \mid G\varphi \mid \\ \neg\varphi \end{aligned}$$

Thus, in total LTL's syntax as considered here consists of the logical contants true and false, conjunction and disjunction, negation, and the temporal operators *next* X, *weak next* $\overline{\mathrm{X}}$, *until* U, and *release* R and its special casses *finally* F and *globally* G.

The previous definition of LTL's syntax is not minimal in the sense that, as we will see when considering the semantics, some operators can be expressed by others. For example, negation is only added for convenience as for every operator true, $\vee$, X, U and F also its dual operator false, $\wedge$, $\overline{\mathrm{X}}$, R and G, respectively is added, and each atomic proposition p may be used *positively* as p or *negatively* as $\neg p$.

The *operator precedence* is needed to determine an unambiguous derivation of an LTL formula if braces are left out in nested expressions. The higher the rank of an operator is the later it is derivated.

Braces only need to be added if an operator of lower or same rank should be derivated later than the current one.

Definition 9 (operator precedence of LTL).

1. *negation operator:* $\neg$
2. *unary temporal operators:* $\mathrm{X}, \bar{\mathrm{X}}, \mathrm{G}, \mathrm{F}$
3. *binary temporal logic operators:* U, R
4. *conjunction operator:* $\wedge$
5. *disjunction operator:* $\vee$

Example 2.

$$\begin{aligned} & \mathrm{G}\ \neg x\ \vee\ \ \neg x\ \mathrm{U}\ \mathrm{G}y\ \ \wedge z \\ \equiv\ & \mathrm{G}\,(\neg x) \vee \Big(\big((\neg x)\,\mathrm{U}\,(\mathrm{G}y) \big) \wedge z \Big) \end{aligned}$$

3 FLTL Semantics

3.1 Semantics

We will now give a formal sematics matching the informal ideas from the last section on how LTL works. We first assume that the execution under scrutiny has terminated. Thus, we are given a final complete word over the alphabet $\Sigma = 2^{\mathrm{AP}}$ and we are after a two-valued semantics for Finite Linear Temporal Logic (FLTL), which tells us wether the execution/run fulfills the correctness property or not.

Given a word $w \in \Sigma^+$, we define when the word satisfies a given property φ. If so, we write $w \models \varphi$, and, if not, we sometimes write $w \not\models \varphi$. In other words, the semantics of LTL formula with respect to words is typically given as a relation $\models$ but we write, to simplify readability, $w \models \varphi$ rather than $(w, \varphi) \in \models$.

In the formal definition of LTL semantics we denote parts of a word as follows: Let $w = a_1 a_2 \dots a_n \in \Sigma^n$ be a finite word over the alphabet $\Sigma = 2^{\mathrm{AP}}$ and let $i \in \mathbb{N}$ with $1 \leq i \leq n$ be a position in this word. Then

- $|w| := n$ is the length of the word,
- $w_i = a_i$ is the i-th letter of the word and
- $w^i = a_i a_{i+1} \dots a_n$ is the subword starting with letter i.

We now give the formal two-valued semantics for LTL on finite completed non-empty words. It uses the standard ideas of LTL derived in the previous section. Recall that X and $\overline{\mathrm{X}}$ represent that idea that property at the end of the trace *is not* or respectively, *is* satisfied.

Definition 10 (FLTL Semantics). *Let φ, ψ be LTL formulae and let $w \in \Sigma^+$ be a finite word. Then the semantics of φ with respect to w is inductively defined as in Fig. 6.*

Let us consider several examples. More specifically, let us consider several *patterns*. The patterns are taken from [DAC99]. On the website http://patterns.projects.cis.ksu.edu/ many more real world pattern are described.

In the following examples we consider a property φ whose validity should be specified with respect to a *scope* expressed by a property ψ. We consider these scopes:

$$
\begin{array}{ll}
w \models \text{true} & \\
w \models p & \text{iff } p \in w_1 \\
w \models \neg p & \text{iff } p \notin w_1 \\
w \models \neg \varphi & \text{iff } w \not\models \varphi \\
w \models \varphi \vee \psi & \text{iff } w \models \varphi \text{ or } w \models \psi \\
w \models \varphi \wedge \psi & \text{iff } w \models \varphi \text{ and } w \models \psi \\
w \models \mathrm{X}\,\varphi & \text{iff } |w| > 1 \text{ and, for } |w| > 1, w^2 \models \varphi \\
w \models \overline{\mathrm{X}}\,\varphi & \text{iff } |w| = 1 \text{ or, for } |w| > 1, w^2 \models \varphi \\
w \models \varphi\,\mathrm{U}\,\psi & \text{iff } \exists i, 1 \leq i \leq |w| : (w^i \models \psi \text{ and } \forall k, 1 \leq k < i : w^k \models \varphi) \\
w \models \varphi\,\mathrm{R}\,\psi & \text{iff } \exists i, 1 \leq i \leq |w| : (w^i \models \varphi \text{and } \forall k, 1 \leq k \leq i : w^k \models \psi) \\
 & \quad \text{or } \forall i, 1 \leq i \leq |w| : w^i \models \psi \\
w \models \mathrm{F}\,\varphi & \text{iff } \exists i, 1 \leq i \leq |w| : w^i \models \varphi \\
w \models \mathrm{G}\,\varphi & \text{iff } \forall i, 1 \leq i \leq |w| : w^i \models \varphi
\end{array}
$$

Fig. 6. Semantics of FLTL

everytime: all states
before ψ: all states before the first state in which ψ holds (if there is such a state)
after ψ: all states after and including the first state in which ψ holds (if there is such a state)

And we consider the patterns *Absence, Existence*, and *Universality.*

Example 3 (Absence). The formula φ does not hold

everytime: $\mathrm{G}\neg\varphi$
before ψ: $(\mathrm{F}\psi) \rightarrow (\neg\varphi \mathrm{U} \psi)$
after ψ: $\mathrm{G}(\psi \rightarrow (\mathrm{G}\neg\varphi))$

Example 4 (Existence). The formula φ holds in the future

everytime: $\mathrm{F}\varphi$
before ψ: $\mathrm{G}\neg\psi \vee \neg\psi \mathrm{U}(\varphi \wedge \neg\psi)$
after ψ: $\mathrm{G}\neg\psi \vee \mathrm{F}(\psi \wedge \mathrm{F}\varphi)$

Example 5 (Universality). The formula φ holds

everytime: $\mathrm{G}\varphi$
before ψ: $(\mathrm{F}\psi) \rightarrow (\varphi \mathrm{U} \psi)$
after ψ: $\mathrm{G}(\psi \rightarrow \mathrm{G}\varphi)$

Let us now recall the idea of equivalent formulas

Definition 11 (Equivalence of Formulae). *Let* $\Sigma = 2^{\mathrm{AP}}$ *and* φ *and* ψ *be LTL formulae over* AP. φ *and* ψ *are* equivalent, *denoted by* $\varphi \equiv \psi$, *iff*

$$\forall w \in \Sigma^+ : w \models \varphi \Leftrightarrow w \models \psi.$$

Globally and finally can easily be expressed using until and release:

$$\mathrm{F}\varphi \equiv \mathrm{true}\,\mathrm{U}\varphi \qquad \mathrm{G}\varphi \equiv \mathrm{false}\,\mathrm{R}\varphi$$

The negation can always be moved in front of the atomic propositions using the dual operators:

De Morgan Rules of Propositional Logic

$$\begin{aligned}\neg(\varphi \vee \psi) &\equiv \neg\varphi \wedge \neg\psi\\ \neg(\varphi \wedge \psi) &\equiv \neg\varphi \vee \neg\psi\end{aligned}$$

De Morgan Rules of Temporal Logic

$$\begin{aligned}\neg(\varphi \mathrm{U} \psi) &\equiv \neg\varphi \mathrm{R} \neg\psi\\ \neg(\varphi \mathrm{R} \psi) &\equiv \neg\varphi \mathrm{U} \neg\psi\\ \neg(\mathrm{G}\varphi) &\equiv \mathrm{F}\neg\varphi\\ \neg(\mathrm{F}\varphi) &\equiv \mathrm{G}\neg\varphi\\ \neg(\mathrm{X}\varphi) &\equiv \overline{\mathrm{X}}\neg\varphi\\ \neg(\overline{\mathrm{X}}\varphi) &\equiv \mathrm{X}\neg\varphi\end{aligned}$$

Fixed Point Equations. The following *fixed point equations* can be used to stepwise unwind until and release:

$$\begin{aligned}\varphi \mathrm{U} \psi &\equiv \psi \vee (\varphi \wedge \mathrm{X}(\varphi \mathrm{U} \psi))\\ \varphi \mathrm{R} \psi &\equiv \psi \wedge (\varphi \vee \overline{\mathrm{X}}(\varphi \mathrm{R} \psi))\end{aligned}$$

Consequently such fix point equations for globally and finally are special cases of the above ones:

$$\begin{aligned}\mathrm{G}\varphi &\equiv \varphi \wedge \overline{\mathrm{X}}(\mathrm{G}\varphi)\\ \mathrm{F}\varphi &\equiv \varphi \vee \mathrm{X}(\mathrm{F}\varphi)\end{aligned}$$

Using all these equivalences one may notice that only a small set of LTL operators is needed in a minimal syntax to provide the full expressiveness of LTL.

Definition 12 (Negation Normal Form (NNF)). *An LTL formula* φ *is in* Negation Normal Form (NNF) *iff* $\neg$ *only occurs in front of atomic propositions* $p \in \mathrm{AP}$.

Lemma 1. *For every LTL formula there exists an equivalent formula in NNF.*

Proof. Recursively apply De Morgan rules of propositional logic and De Morgan rules of temporal logic.

Given the semantics of FLTL, it is easy to create a monitoring device that reads a finite string and an LTL formula and outputs the semantics of the string as a monitoring verdict. We leave this as an excercise to the reader. In the next section, we will elaborate a semantics that is suitable for finite but continuesly expanding words and we provide a more sophisticated monitoring procedure that may be slightly adapted to serve also for FLTL.

4 Impartial Runtime Verification

For now we are able to decide if a finite terminated execution/word models an LTL formula. This approach is reasonable whenever a terminated run of system is analyzed. Often, especially in online monitoring, the execution is still running so that the word to analyze is continuously expanding. Thus, instead of considering a terminated word we will start thinking about what happens if the word gets extended with more letters step by step. Regarding the maxims impartiality and anticipation, we will address the impartiality in this section. To this end, our monitor will be able to answer the question if a word models an LTL formula with one of the following statements:

- Yes it does and it will always do.
- Yes it does, but this may change.
- No it does not, but this may change.
- No it does not and will never do.

We will introduce the concept of truth domains providing multiple logical values that can be used as results if an LTL formula gets evaluated. Here, we build on the theory of lattices. After presenting a first monitoring approach for such a four-valued LTL semantics we will introduce automata-based monitoring. This answers the question on how the recursive evaluation function can be implemented in an efficient way. It gets translated into a Mealy machine—one of the most basic and easy to implement machine models.

4.1 Truth Domains

Definition 13 (Lattice). *A* lattice *is a partially ordered set* $(\mathcal{L}, \sqsubseteq)$ *where for each* $x, y \in \mathcal{L}$*, there exists*

1. *a unique* greatest lower bound *(glb), which is called the* meet *of* x *and* y*, and is denoted with* $x \sqcap y$*, and*
2. *a unique* least upper bound *(lub), which is called the* join *of* x *and* y*, and is denoted with* $x \sqcup y$*.*

If the ordering relation $\sqsubseteq$ is obvious we denote the lattice with the set $\mathcal{L}$.

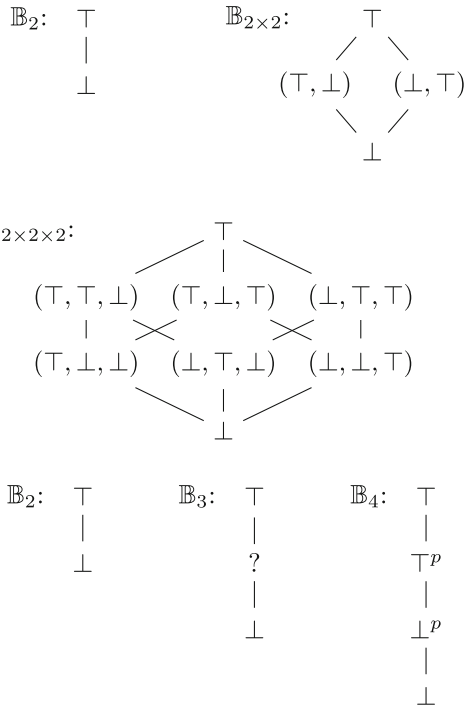
Definition 14 (Finite Lattice). *A lattice* $(\mathcal{L}, \sqsubseteq)$ *is called* finite *iff* $\mathcal{L}$ *is finite.*

Every non-empty finite lattice has two well-defined unique elements: A least element, called *bottom*, denoted with $\bot$ and a greatest element, called *top*, denoted with $\top$.

Hasse diagrams are used to represent a finite partially ordered set. Each element of the set is represented as a vertex in the plane. For all $x, y \in \mathcal{L}$ where $x \sqsubseteq y$ but no $z \in \mathcal{L}$ exists where $x \sqsubseteq z \sqsubseteq y$ a line that goes *upward* from x to y is drawn.

Example 6 (Hasse Diagram). Hasse diagram for $\mathbb{B}_2 = \{\bot, \top\}$ with $\bot \sqsubseteq \top$:

Example 7 (Lattices).



Definition 15 (Distributive Lattices). *A lattice* $(\mathcal{L}, \sqsubseteq)$ *is called a* distributive lattice *iff we have for all elements* $x, y, z \in \mathcal{L}$

$$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z) \text{ and}$$
$$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z).$$

Definition 16 (De Morgan Lattice). *A distributive lattice* $(\mathcal{L}, \sqsubseteq)$ *is called a* De Morgan lattice *iff every element* $x \in \mathcal{L}$ *has a unique* dual *element* $\overline{x}$*, such that*

$$\overline{\overline{x}} = x \text{ and } x \sqsubseteq y \text{ implies } \overline{y} \sqsubseteq \overline{x}.$$

Definition 17 (Boolean Lattice). *A De Morgan lattice is called* Boolean lattice *iff for every element* x *and its dual element* $\overline{x}$ *we have*

$$x \sqcup \overline{x} = \top \text{ and } x \sqcap \overline{x} = \bot.$$

Every Boolean lattice has 2^n elements for some $n \in \mathbb{N}$.

Definition 18 (Truth Domain). *A* Truth Domain *is a finite De Morgan Lattice.*

Example 8 (Truth Domains). The following lattices are all Truth Domains:

- $\mathbb{B}_2 = \{\top, \bot\}$ with $\bot \sqsubseteq \top$ and $\overline{\top} = \bot$ and $\overline{\bot} = \top$.
- $\mathbb{B}_3 = \{\top, ?, \bot\}$ with $\bot \sqsubseteq ? \sqsubseteq \top$ and $\overline{\top} = \bot$, $\overline{?} = ?$ and $\overline{\bot} = \top$.
- $\mathbb{B}_4 = \{\top, \top^p, \bot^p, \bot\}$ with $\bot \sqsubseteq \bot^p \sqsubseteq \top^p \sqsubseteq \top$ and $\overline{\top} = \bot$, $\overline{\top^p} = \bot^p$, $\overline{\bot^p} = \top^p$ and $\overline{\bot} = \top$.

where we call ? also *inconclusive/don't know* and $\top^p$ and $\bot^p$ *presumably true* and *presumably false*, respectively.

4.2 Four-Valued Impartial LTL Semantics: FLTL$_4$

Let us now continue with the development of an impartial semantics of FLTL. Recall that the idea of impartiality is to for a go for a final verdict ($\top$ or $\bot$) only if you really know. In other words, *impartiality* requires that a finite trace is not evaluated to *true* or, respectively *false*, if there still exists an (possibly infinite) continuation leading to another verdict. Impartiality requires more than two truth values so that in general the semantics of a formula with respect to a trace is no longer a relation but a semantic function yielding a suitable truth value.

Definition 19 (Semantic Function). *The* semantic function

$$\mathrm{sem}_k : \Sigma^+ \times \mathrm{LTL} \to \mathbb{B}_k$$

maps a word $w \in \Sigma^+$ *and a an LTL formula* φ *to a logic value* $b \in \mathbb{B}_k$.

We use $[\![w \models \varphi]\!]_k = b$ *instead of* $\mathrm{sem}_k(w, \varphi) = b$.

Clearly, this is a conservative extension and the semantics for (two-valued) FLTL can easily be given as a function:

We defined the FLTL semantics as *relation* $w \models \varphi$ between a word $w \in \Sigma^+$ and an LTL formula φ. This can be *interpreted as semantic function*

$$\mathrm{sem}_2 : \Sigma^+ \times \mathrm{LTL} \to \mathbb{B}_2,$$
$$\mathrm{sem}_2(w, \varphi) = [\![w \models \varphi]\!]_2 := \begin{cases} \top & \text{if } w \models \varphi \\ \bot & \text{else.} \end{cases}$$

We can now define the notion of an impartial semantics formally:

Definition 20 (Impartial Semantics). *Let $\Sigma = 2^{\mathrm{AP}}$ be an alphabet, $w \in \Sigma^+$ a word and φ an LTL formula. A semantic function is called* impartial *iff for all $u \in \Sigma^*$*

$$[\![w \models \varphi]\!] = \top \textit{ implies } [\![wu \models \varphi]\!] = \top$$
$$[\![w \models \varphi]\!] = \bot \textit{ implies } [\![wu \models \varphi]\!] = \bot.$$

We want to create impartial four-valued semantics for LTL on finite, non-completed words using the truth domain $(\mathbb{B}_4, \sqsubseteq)$. Let us look at examples our semantics should adhere to:

Example 9 ((FLTL vs. FLTL$_4$).

The indices 2 and 4 denote FLTL resp. FLTL$_4$.

$$[\![\emptyset \models \mathrm{X}a]\!]_2 = \bot \qquad [\![\emptyset \models \mathrm{X}a]\!]_4 = \bot^p$$
$$[\![\emptyset\emptyset \models \mathrm{X}a]\!]_2 = \bot \qquad [\![\emptyset\emptyset \models \mathrm{X}a]\!]_4 = \bot$$
$$[\![\emptyset\{a\} \models \mathrm{X}a]\!]_2 = \top \qquad [\![\emptyset\{a\} \models \mathrm{X}a]\!]_4 = \top$$
$$[\![\emptyset \models \overline{\mathrm{X}}a]\!]_2 = \top \qquad [\![\emptyset \models \overline{\mathrm{X}}a]\!]_4 = \top^p$$
$$[\![\emptyset\emptyset \models \overline{\mathrm{X}}a]\!]_2 = \bot \qquad [\![\emptyset\emptyset \models \overline{\mathrm{X}}a]\!]_4 = \bot$$
$$[\![\emptyset\{a\} \models \overline{\mathrm{X}}a]\!]_2 = \top \qquad [\![\emptyset\{a\} \models \overline{\mathrm{X}}a]\!]_4 = \top$$

At the end of the word X evaluates to $\bot^p$ instead of $\bot$ and $\overline{\mathrm{X}}$ evaluates to $\top^p$ instead of $\top$. The idea of $\bullet^p$ is that it identifies the (two-valued) semantics if the word ends here but may change depending on the future. Fulfilling the introduced equivalences and fix point equations we get at the end of the word: U evaluates to $\bot^p$ instead of $\bot$, R evaluates to $\top^p$ instead of $\top$.

We now give the formal four-valued semantics for LTL on finite, non-completed and non-empty words:

Definition 21 (FLTL$_4$ Semantics). *Let φ, ψ be LTL formulae and let $w \in \Sigma^+$ be a finite word. Then the semantics of φ with respect to w is inductively defined as follows:*

$$
\begin{aligned}
[\![w \models \mathit{true}]\!]_4 &= \top \\
[\![w \models \mathit{false}]\!]_4 &= \bot \\
[\![w \models p]\!]_4 &= \begin{cases} \top & \text{if } p \in w_1 \\ \bot & \text{if } p \notin w_1 \end{cases} \\
[\![w \models \neg p]\!]_4 &= \begin{cases} \top & \text{if } p \notin w_1 \\ \bot & \text{if } p \in w_1 \end{cases} \\
[\![w \models \neg\varphi]\!]_4 &= \overline{[\![w \models \varphi]\!]_4} \\
[\![w \models \varphi \vee \psi]\!]_4 &= [\![w \models \varphi]\!]_4 \sqcup [\![w \models \psi]\!]_4 \\
[\![w \models \varphi \wedge \psi]\!]_4 &= [\![w \models \varphi]\!]_4 \sqcap [\![w \models \psi]\!]_4 \\
[\![w \models X\varphi]\!]_4 &= \begin{cases} [\![w^2 \models \varphi]\!]_4 & \text{if } |w| > 1 \\ \bot^p & \text{else} \end{cases} \\
[\![w \models \overline{X}\varphi]\!]_4 &= \begin{cases} [\![w^2 \models \varphi]\!]_4 & \text{if } |w| > 1 \\ \top^p & \text{else} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![w \models \varphi U \psi]\!]_4 \\
&\quad = \left(\bigsqcup_{1 \le i \le |w|} \left([\![w^i \models \psi]\!]_4 \sqcap \bigsqcap_{1 \le j < i} [\![w^j \models \varphi]\!]_4 \right) \right) \\
&\qquad \sqcup \left(\bot^p \sqcap \bigsqcap_{1 \le i \le |w|} [\![w^i \models \varphi]\!]_4 \right) \\
&[\![w \models \varphi R \psi]\!]_4 \\
&\quad = \left(\bigsqcup_{1 \le i \le |w|} \left([\![w^i \models \varphi]\!]_4 \sqcap \bigsqcap_{1 \le j \le i} [\![w^j \models \psi]\!]_4 \right) \right) \\
&\qquad \sqcup \left(\top^p \sqcap \bigsqcap_{1 \le i \le |w|} [\![w^i \models \psi]\!]_4 \right) \\
&\quad [\![w \models F\varphi]\!]_4 = \bot^p \sqcup \bigsqcup_{1 \le i \le |w|} [\![w^i \models \varphi]\!]_4 \\
&\quad [\![w \models G\varphi]\!]_4 = \top^p \sqcap \bigsqcap_{1 \le i \le |w|} [\![w^i \models \varphi]\!]_4
\end{aligned}
$$

Definition 22 (Equivalence of Formulae). *Let* $\Sigma = 2^{\mathrm{AP}}$ *and* φ *and* ψ *be LTL formulae over* AP. φ *and* ψ *are* equivalent*, denoted by* $\varphi \equiv \psi$*, iff*

$$\forall w \in \Sigma^+ : [\![w \models \varphi]\!] = [\![w \models \psi]\!].$$

Monitor Function. The idea of the monitoring function is to process a word while it is read, from-left-right. In other words, our goal is to build up a monitor function for evaluating each subsequent letter of non-completed words. Such a function takes an LTL formula φ in NNF and a letter $a \in \Sigma$, performs (not recursive) formula rewriting (progression) and returns $[\![a \models \varphi]\!]_4$ and a new LTL formula φ' that the next letter has to fulfill. To this end, we rewrite the LTL formula to keep track of what is done and what still needs to be checked.

For example, let $w \in \Sigma^+$ be a word and $p \in \mathrm{AP}$ a letter. We can compute $[\![w \models \mathrm{X}p]\!]_4$ by doing nothing and letting someone else check $[\![w^2 \models p]\!]_4$. We can compute $[\![w \models a]\!]_4$ by checking $p \in w_1$.

Then the LTL formula is over. This is denoted by true or false as new formula.

It is straight forward to evaluate atomic propositions, positive operators of propositional logic $(\wedge, \vee)$ and next-formulas. Thanks to De Morgan rules of propositional and temporal logic for negation $(\neg)$ and fixed point equations for U and R those formulas do not have be treated explicitly.

Let $\Sigma = 2^{\mathrm{AP}}$ be the finite alphabet, $p \in \mathrm{AP}$ an atomic proposition, $a \in \Sigma$ a letter, and φ and ψ LTL formulae.

We then define the function $\mathrm{evlFLTL}_4 : \Sigma \times \mathrm{LTL} \to \mathbb{B}_4 \times \mathrm{LTL}$ inductively as shown in Fig. 7.

Example 10 (Impartial Evaluation of Globally).
Consider $w = \{a\}\{a\}\emptyset$. First letter:

$$\begin{aligned}\mathrm{evlFLTL}_4(\{a\}, \mathrm{G}a) &= \mathrm{evlFLTL}_4(\{a\}, a \wedge \overline{\mathrm{X}}\mathrm{G}a)\\ &= (v_1 \sqcap v_2, \varphi_1 \wedge \varphi_2)\\ &= (\top \sqcap \top^p, \mathrm{true} \wedge \mathrm{G}a)\\ &= (\top^p, \mathrm{G}a)\end{aligned}$$

$$\begin{aligned}\text{where } (v_1, \varphi_1) &= \mathrm{evlFLTL}_4(\{a\}, a) = (\top, \mathrm{true})\\ (v_2, \varphi_2) &= \mathrm{evlFLTL}_4(\{a\}, \overline{\mathrm{X}}\mathrm{G}a) = (\top^p, \mathrm{G}a).\end{aligned}$$

Next letters:

- $\mathrm{evlFLTL}_4(\{a\}, \mathrm{G}a) = (\top^p, \mathrm{G}a)$
- $\mathrm{evlFLTL}_4(\emptyset, \mathrm{G}a) = (\bot, \mathrm{false})$

4.3 Automata-Based Monitoring for FLTL_4

Within the automata-theoretic approach to monitoring, one creates an automaton, for example a deterministic one with output. Whenever a new observation on the underlying system is made, it is send to the automaton as input and the output yields the verdict for the trace observed so far.

The automaton synthesized for a property to check can typically be understood as a pre-computation of the respective monitoring function developed in the previous section. If, for example, the monitor is deterministic and realized as

$$
\begin{aligned}
\mathrm{evlFLTL}_4(a, \mathrm{true}) &= (\top, \mathrm{true}) \\
\mathrm{evlFLTL}_4(a, \mathrm{false}) &= (\bot, \mathrm{false}) \\
\mathrm{evlFLTL}_4(a, p) &= \begin{cases} (\top, \mathrm{true}) & \text{if } p \in a \\ (\bot, \mathrm{false}) & \text{else} \end{cases} \\
\mathrm{evlFLTL}_4(a, \neg p) &= \begin{cases} (\bot, \mathrm{false}) & \text{if } p \in a \\ (\top, \mathrm{true}) & \text{else} \end{cases} \\
\mathrm{evlFLTL}_4(a, \varphi \vee \psi) &= (v_\varphi \sqcup v_\psi, \varphi' \vee \psi'), \text{ where} \\
&\quad (v_\varphi, \varphi') = \mathrm{evlFLTL}_4(a, \varphi) \text{ and} \\
&\quad (v_\psi, \psi') = \mathrm{evlFLTL}_4(a, \psi) \\
\mathrm{evlFLTL}_4(a, \varphi \wedge \psi) &= (v_\varphi \sqcap v_\psi, \varphi' \wedge \psi'), \text{ where} \\
&\quad (v_\varphi, \varphi') = \mathrm{evlFLTL}_4(a, \varphi) \text{ and} \\
&\quad (v_\psi, \psi') = \mathrm{evlFLTL}_4(a, \psi) \\
\mathrm{evlFLTL}_4(a, \mathrm{X}\,\varphi) &= (\bot^p, \varphi) \\
\mathrm{evlFLTL}_4(a, \overline{\mathrm{X}}\,\varphi) &= (\top^p, \varphi) \\
\mathrm{evlFLTL}_4(a, \varphi\,\mathrm{U}\,\psi) &= \mathrm{evlFLTL}_4(a, \psi \vee (\varphi \wedge \mathrm{X}(\varphi\,\mathrm{U}\,\psi))) \\
\mathrm{evlFLTL}_4(a, \varphi\,\mathrm{R}\,\psi) &= \mathrm{evlFLTL}_4(a, \psi \wedge (\varphi \vee \overline{\mathrm{X}}(\varphi\,\mathrm{R}\,\psi))) \\
\mathrm{evlFLTL}_4(a, \mathrm{F}\,\varphi) &= \mathrm{evlFLTL}_4(a, \varphi \vee \mathrm{X}\,\mathrm{F}\,\varphi) \\
\mathrm{evlFLTL}_4(a, \mathrm{G}\,\varphi) &= \mathrm{evlFLTL}_4(a, \varphi \wedge \overline{\mathrm{X}}\,\mathrm{G}\,\varphi)
\end{aligned}
$$

Fig. 7. Evaluation of LTL formulas with an impartial, four-valued semantics

a transition table, only a simple look-up in the table is necessary for processing the observation. Thus, the automata-theoretic approach to monitoring is considered to be efficient at runtime but of course, the precomputed transition table may be huge.

The goal of this subsection is to provide a synthesis procedure for FLTL_4, based on the evaluation function given in the previous subsection.

The translation is guided by the following observation: $\mathrm{evlFLTL}_4$ gets a *letter* and a *formula* and outputs a *logic value* and a *new formula*. We use *formula* as *state* of the Mealy machine and *letter* as *input* and *logic value* as *output*. The *next state* (new formula) depends on the *state* (formula) and *input* (letter), while the*Output* depends on *state* (formula) and *input* (letter).

Definition 23 (Deterministic Mealy Machine). *A (deterministic)* Mealy machine *is a tupel* $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta)$ *where*

- Σ *is the* input alphabet*,*
- Q *is a finite set of* states*,*
- $q_0 \in Q$ *is the* initial state*,*
- Γ *is the* output alphabet *and*
- $\delta : Q \times \Sigma \to \Gamma \times Q$ *is the* transition function

Definition 24 (Run of a Deterministic Mealy Machine). *A* run *of a (deterministic) Mealy machine* $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta)$ *on a finite word* $w \in \Sigma^n$ *with outputs* $o_i \in \Gamma$ *is a sequence*

$$t_0 \overset{(w_1,o_1)}{\rightarrow} t_1 \overset{(w_2,o_2)}{\rightarrow} \dots \overset{(w_{n-1},o_{n-1})}{\rightarrow} t_{n-1} \overset{(w_n,o_n)}{\rightarrow} t_n$$

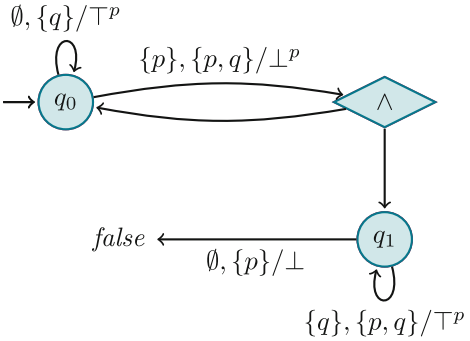
such that

- $t_0 = q_0$ *and*
- $(t_i, o_i) = \delta(t_{i-1}, w_i)$

The output *of the run is* o_n.

Our synthesis procedure will first generate *alternating* machines, which then may be translated into non-deterministic or deterministic machines. Intuitively, an alternating machine may proceed from a boolean combination of states to a subsequent boolean combination of states, following the transition function. Non-deterministic and deterministic machines are, respectively, intuitively in a set or a single state.

Alternating Mealy Machine



Input	$\emptyset$	$\{p\}$	$\{q\}$	$\emptyset$	$\{q\}$
Output	$\top^p$	$\bot^p$	$\top^p$	$\bot$	$\bot$

Definition 25 (Positive Boolean Combination (PBC)). *Given a set* Q *we define the set of all* positive Boolean combinations *(PBC) over* Q*, denoted by* $B^+(Q)$*, inductively as follows:*

- $\{\text{true}, \text{false}\} \subseteq B^+(Q)$,
- $Q \subseteq B^+(Q)$ *and*
- $\forall \alpha, \beta \in B^+(Q) : \alpha \vee \beta, \alpha \wedge \beta \in B^+(Q)$.

Example 11. Consider $\text{AP} = \{a, b, c\}$

- $a \in B^+(\text{AP}), \{a\} \notin B^+(\text{AP})$,
- $a \wedge b \vee a \wedge c \in B^+(\text{AP})$,

– $\text{true} \in B^+(\text{AP})$ and $\text{false} \in B^+(\text{AP})$.

Now, we are ready to define alternating Mealy machines, their extended transition function as well as their runs:

Definition 26 (Alternating Mealy Machine (AMM)). *A* alternating Mealy machine *(AMM) is a tupel* $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta)$ *where*

– Σ *is the* input alphabet*,*
– Q *is a finite set of* states*,*
– $q_0 \in Q$ *is the* initial state *and*
– Γ *is a finite, distributive lattice, the* output lattice*,*
– $\delta : Q \times \Sigma \to B^+(\Gamma \times Q)$ *is the* transition function

Sometimes, we understand $\delta : Q \times \Sigma \to B^+(\Gamma \times Q)$ as a function $\delta : Q \times \Sigma \to \Gamma \times B^+(Q)$, yielding a tuple with the first component having the value of the respective meets and joins of individual outputs and second component having the positive Boolean combination of the respective second components.

Definition 27 (Extended Transition Function). *Let* $\delta : Q \times \Sigma \to \Gamma \times B^+(Q)$ *be the transition function of an alternating mealy machine. Then the extended transition function* $\hat{\delta} : B^+(Q) \times \Sigma \to \Gamma \times B^+(Q)$ *is inductively defined as follows*

– $\hat{\delta}(q, a) = \delta(q, a)$*,*
– $\hat{\delta}(\text{true}, a) = (\top, \text{true})$*,* $\hat{\delta}(\text{false}, a) = (\bot, \text{false})$*,*
– $\hat{\delta}(q_1 \vee q_2, a) = (o_1 \sqcup o_2, q_1' \vee q_2')$ *and*
– $\hat{\delta}(q_1 \wedge q_2, a) = (o_1 \sqcap o_2, q_1' \wedge q_2')$*,*
where $(o_1, q_1') = \hat{\delta}(q_1, a)$ *and* $(o_2, q_2') = \hat{\delta}(q_2, a)$*.*

Definition 28 (Run of an Alternating Mealy Machine). *A* run *of an alternating Mealy machine* $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta)$ *on a finite word* $w \in \Sigma^n$ *with outputs* $o_i \in \Gamma$ *is a sequence*

$$t_0 \overset{(w_1,o_1)}{\to} t_1 \overset{(w_2,o_2)}{\to} \dots \overset{(w_{n-1},o_{n-1})}{\to} t_{n-1} \overset{(w_n,o_n)}{\to} t_n$$

such that

$$t_0 = q_0 \text{ and } (t_i, o_i) = \hat{\delta}(t_{i-1}, w_i),$$

where $\hat{\delta}$ *is the extended transition function of* $\mathcal{M}$*.*

The output *of the run is* o_n*.*

Let us now derive the necessary machinery for translation alternating machines to deterministic ones.

Definition 29 (Model Relation for PBCs). *Let* Q *be a set. A subset* $S \subseteq Q$ *is a* model *of a positive Boolean combination* $\alpha \in B^+(Q)$*, denoted by* $S \models \alpha$*, iff* α *evaluates to true in propositional logic interpreting all* $p \in S$ *as true and all* $p \in Q \backslash S$ *as false.*

Definition 30 (Equivalence of PBCs). *Let Q be a set and $\alpha \in B^+(Q)$ and $\beta \in B^+(Q)$ be positive Boolean combinations over Q. α and β are* equivalent*, denoted by $\alpha \equiv \beta$, iff*

$$\forall S \subseteq Q : S \models \alpha \Leftrightarrow S \models \beta.$$

Definition 31 (Equivalence Classes of PBCs). *Let Q be a set. The* equivalence class $[\alpha]$ *of a positive Boolean combination $\alpha \in B^+(Q)$ over Q is defined as follows*

$$[\alpha] = \{\beta \in B^+(Q) \mid \alpha \equiv \beta\}.$$

The set of all equivalence classes of positive Boolean combinations over Q is denoted by the following quotient set

$$B^+(Q)/\equiv = \{[\alpha] \mid \alpha \in B^+(Q)\}.$$

Alternating Mealy machines can easily be translated into (deterministic) Mealy machines. The idea is to use $B^+(Q)/\equiv$ instead of $B^+(Q)$ as a state space and to extend the transition function correspondingly. This is well defined:

Lemma 2. *Let $\hat{\delta}$ be the extended transition function of an AMM $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta)$, $a \in \Sigma$, $o, p \in \Gamma$ and $\alpha, \beta, \alpha', \beta' \in B^+(Q)$ such that*

$$\begin{aligned} \alpha &\equiv \beta, \\ (o, \alpha') &= \hat{\delta}(\alpha, a) \text{ and} \\ (p, \beta') &= \hat{\delta}(\beta, a). \end{aligned}$$

Then

$$\begin{aligned} o = p \text{ and} \qquad (*) \\ \alpha' \equiv \beta'. \end{aligned}$$

The proof of $(*)$ requires the output lattice to be distributive.

In other words, equivalent combinations of states yield the same output and an equivalent combination of successor states. Thus, whenever we perform a transition, we can normalize the resulting Boolean combination of states without changing the output for an input word. As for any fixed set of states, there are only finitely many non-equivalent formulae, we have the general result that we can transform an alternating machine to a deterministic one. However, let us be more specific.

We can use $B^+(Q)/\equiv$ instead of Q as states. We still need a well defined representative for $[\alpha]$ for $\alpha \in B^+(Q)$. In other words: Given $\alpha \in Q$, how to find $[\alpha]$? We use disjunctive normal form of α.

Definition 32 (Disjunctive Normal Form (DNF)). *A positive Boolean combination $\alpha \in B^+(Q)$ over a set Q is in* disjunctive normal form *(DNF) iff*

$$\alpha = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} q_{i,j}$$

for $q_{i,j} \in Q$. A disjunctive normal form $\alpha \in B^+(Q)$ is called minimal *if there is no disjunctive normal form $\beta \in B^+(Q)$ s. t. $\alpha \equiv \beta$ and β contains less operators.*

Lemma 3. *For every positive Boolean combination $\alpha \in B^+(Q)$ there exists a positive Boolean combination β such that $\alpha \equiv \beta$ and β is in minimal DNF.*

Proof uses distributivity of propositional logic.

$B^+(Q)/\equiv$ is finite. Let Q be the set of states of an AMM. Then Q is finite.
Then there are at most $2^{|Q|}$ many different α for

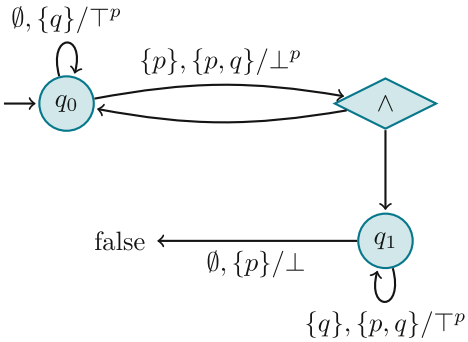
$$\alpha = \bigvee_{i=1}^{n} q_i \text{ and } n \text{ different } q_i \in Q.$$

Then there are at most $2^{2^{|Q|}}$ many different β for

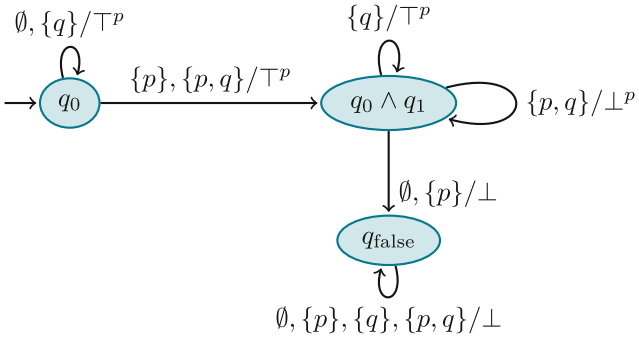
$$\beta = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} q_{i,j} \text{ minimal and } q_{i,j} \in Q.$$

Then there are at most $2^{2^{|Q|}}$ many different $[\beta]$ for $\beta \in B^+(Q)$.

Example 12 (Alternating Mealy Machine).



Example 13 (Translated Mealy Machine).



Using similar ideas, we can translate an AMM to Non-Deterministic or Universal MM.

Automata Based RV. We monitor an LTL formula φ by evaluating its current subformula ψ w.r.t. the current letter a.

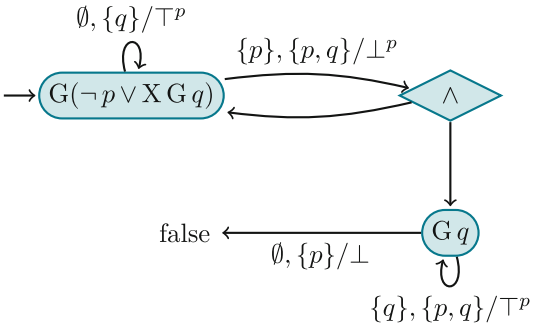
The monitor function evlFLTL$_4$, which takes an LTL formula ψ in NNF and a letter $a \in \Sigma$ and returns $[\![a \models \psi]\!]_4$ and a new LTL formula ψ', can be interpreted as transition function of an AMM where the states are subformulae of φ, the initial state is φ, the current state is ψ, we read the letter a, we output $[\![a \models \psi]\!]_4$ and the next state is the new formula ψ'.

Let $\Sigma = 2^{\mathrm{AP}}$ be the finite alphabet, $p \in \mathrm{AP}$ an atomic proposition, $a \in \Sigma$ a letter, φ, ψ_1, ψ_2 LTL formulae in NNF and Q the set of all subformulae of φ.

We then define the transition function $\delta_4^a : Q \times \Sigma \to B^+(\mathbb{B}_4 \times Q)$ of the monitor AMM $\mathcal{M}_\varphi = (\Sigma, Q, \varphi, \mathbb{B}_4, \delta_4^a)$ inductively as follows:

$$
\begin{aligned}
\delta_4^a(\mathrm{true}, a) &= (\top, \mathrm{true}) \\
\delta_4^a(\mathrm{false}, a) &= (\bot, \mathrm{false}) \\
\delta_4^a(p, a) &= \begin{cases} (\top, \mathrm{true}) & \text{if } p \in a \\ (\bot, \mathrm{false}) & \text{else} \end{cases} \\
\delta_4^a(\neg p, a) &= \begin{cases} (\top, \mathrm{true}) & \text{if } p \notin a \\ (\bot, \mathrm{false}) & \text{else} \end{cases} \\
\delta_4^a(\psi_1 \vee \psi_2, a) &= \delta_4^a(\psi_1, a) \vee \delta_4^a(\psi_2, a) \\
\delta_4^a(\psi_1 \wedge \psi_2, a) &= \delta_4^a(\psi_1, a) \wedge \delta_4^a(\psi_2, a) \\
\delta_4^a(\mathrm{X}\psi_1, a) &= (\bot^p, \psi_1) \\
\delta_4^a(\overline{\mathrm{X}}\psi_1, a) &= (\top^p, \psi_1) \\
\delta_4^a(\psi_1 \mathrm{U} \psi_2, a) &= \delta_4^a(\psi_2 \vee (\psi_1 \wedge \mathrm{X}(\psi_1 \mathrm{U} \psi_2)), a) \\
\delta_4^a(\psi_1 \mathrm{R} \psi_2, a) &= \delta_4^a(\psi_2 \wedge (\psi_1 \vee \overline{\mathrm{X}}(\psi_1 \mathrm{R} \psi_2)), a) \\
\delta_4^a(\mathrm{F}\psi_1, a) &= \delta_4^a(\psi_1 \vee (\mathrm{X}(\mathrm{F}\psi_1)), a) \\
\delta_4^a(\mathrm{G}\psi_1, a) &= \delta_4^a(\psi_1 \wedge (\overline{\mathrm{X}}(\mathrm{G}\psi_1)), a)
\end{aligned}
$$

Example 14. Graph of the monitor $\mathcal{M}_\varphi$ of the formula $\varphi = \mathrm{G}(p \to \mathrm{XG}q)$:



In practical implementations one may omit the AMM and generate the MM directly out of the LTL formula. Define a function smplfy : LTL → LTL that

transforms LTL formulae into a unique normal form. Use all simplified positive Boolean combinations of subformulae of φ as states for $\mathcal{M}_\varphi$. Define $\delta_4 : Q \times \Sigma \to \mathbb{B}_4 \times Q$ inductively as follows:

$$\begin{aligned}
\delta_4(\psi_1 \vee \psi_2, a) &= (v_{\psi_1} \sqcup v_{\psi_2}, \text{smplfy}(\psi_1' \vee \psi_2')), \text{ where} \\
&\quad (v_{\psi_1}, \psi_1') = \delta_4(\psi_1, a) \text{ and} \\
&\quad (v_{\psi_2}, \psi_2') = \delta_4(\psi_2, a) \\
\delta_4(\psi_1 \wedge \psi_2, a) &= (v_{\psi_1} \sqcap v_{\psi_2}, \text{smplfy}(\psi_1' \wedge \psi_2')), \text{ where} \\
&\quad (v_{\psi_1}, \psi_1') = \delta_4(\psi_1, a) \text{ and} \\
&\quad (v_{\psi_2}, \psi_2') = \delta_4(\psi_2, a) \\
\delta_4(\psi_1, a) &= \delta_4^a(\psi_1, a) \text{ for any other formula } \psi_1.
\end{aligned}$$

5 Anticipatory LTL Semantics

Using the monitor construction for FLTL_4 we are able to build an automata based impartial LTL monitor. Our monitor can tell us if a property is fulfilled or violated by a run and if this may change or will last forever. In this section we will go for anticipation. Recall that, in simple words, *impartiality* means to say $\top$ or $\bot$ only if you are sure while *anticipation* means to say $\top$ or $\bot$ once you can be sure.

In the next sections we will follow the same steps as for creating the impartial monitor:

- Define an anticipatory LTL semantics.
- Recall a suitable automaton type and its translation towards a deterministic one.
- Define a monitor construction based on the new semantics.

In the next subsections we will introduce the necessary machinery to finally be able to present a monitor construction for LTL_3.

LTL on Infinite Words. The idea of the anticipatory semantics is to say $\top$ once every infinite continuation evaluates to $\top$, to say $\bot$ once every infinite continuation evaluates to $\bot$, and to otherwise say ?, as the verdict may depend on the future of the underlying execution.

As our anticipatory semantics depends on infinite continuations, we recall the LTL semantics on infinite words.

An infinite word w is an infinite sequence over the alphabet $\Sigma = 2^{\text{AP}}$. w can be interpreted as function $w : \mathbb{N}\backslash\{0\} \to \Sigma$. w can be interpreted as concatenation of many finite and one infinite words.

Example 15 (Infinite Words).
Consider the alphabet $\Sigma = 2^{\text{AP}}$ with $\text{AP} = \{p, q\}$.

- $\{p\}^\omega$ denotes the infinite word where every letter is $\{p\}$ and can be interpreted as $w(i) = \{p\}$ for all $i \geq 1$.

– $\emptyset(\{q\}\{p\})^\omega$ can be interpreted as

$$w(i) = \begin{cases} \emptyset & \text{if } i = 1 \\ \{q\} & \text{if } i \equiv 0 \mod 2 \\ \{p\} & \text{else} \end{cases}$$

Semantics. We have introduced LTL first for finite words, as this was needed for runtime verification on finite, terminated executions. However LTL on infnite words is currently the traditional way to define LTL semantics due to its typical use in model checking, where most often infinite runs are considered. Moreover, the semantics is slightly simpler to define for the next-operators. Actually the semantics of $\mathrm{X}\varphi$ and $\overline{\mathrm{X}}\varphi$ do not differ—on infinite traces. This is due to the fact that for an infinite trace, there always exists a subsequent positon in which φ can be evaluated.

In the formal definition of LTL semantics we denote parts of a word as follows: Let $w = a_1a_2a_3 \ldots \in \Sigma^\omega$ be an infinite word over the alphabet $\Sigma = 2^{\mathrm{AP}}$ and let $i \in \mathbb{N}$ with $i \geq 1$ be a position in this word. Then

– $w_i = a_i$ is the i-th letter of the word,
– $w^{(i)} = a_1a_2 \ldots a_i$ is the prefix of w of length i and
– w^i is the subword of w s. t. $w = w^{(i-1)}w^i$.

We now give the formal two-valued semantics for LTL on infinite words:

Definition 33 (LTL Semantics on Infinite Words). *Let φ, ψ be LTL formulae and let $w \in \Sigma^\omega$ be an infinite word. Then the semantics of φ with respect to w is inductively defined as follows:*

$$\begin{aligned}
&w \models \text{true} && \\
&w \models p && \textit{iff } p \in w_1 \\
&w \models \neg p && \textit{iff } p \notin w_1 \\
&w \models \neg\varphi && \textit{iff } w \not\models \varphi \\
&w \models \varphi \vee \psi && \textit{iff } w \models \varphi \textit{ or } w \models \psi \\
&w \models \varphi \wedge \psi && \textit{iff } w \models \varphi \textit{ and } w \models \psi \\
&w \models X\varphi && \textit{iff } w^2 \models \varphi \\
&w \models \overline{X}\varphi && \textit{iff } w^2 \models \varphi \\
&w \models \varphi \mathrm{U} \psi && \textit{iff } \exists i \geq 1 : (w^i \models \psi \textit{ and } \forall k, 1 \leq k < i : w^k \models \varphi) \\
&w \models \varphi \mathrm{R} \psi && \textit{iff } \exists i \geq 1 : (w^i \models \varphi \textit{ and } \forall k, 1 \leq k \leq i : w^k \models \psi) \\
& && \quad \textit{or } \forall i \geq 1 : w^i \models \psi \\
&w \models \mathrm{F}\varphi && \textit{iff } \exists i \geq 1 : w^i \models \varphi \\
&w \models \mathrm{G}\varphi && \textit{iff } \forall i \geq 1 : w^i \models \varphi
\end{aligned}$$

We defined the LTL semantics on infinite words as *relation* $w \models \varphi$ between a word $w \in \Sigma^\omega$ and a LTL formula φ. This can be *interpreted as semantic function*

$$\begin{aligned} \mathrm{sem}_\omega &: \Sigma^\omega \times \mathrm{LTL} \to \mathbb{B}_2, \\ \mathrm{sem}_\omega(w, \varphi) = [\![w \models \varphi]\!]_\omega &:= \begin{cases} \top & \text{if } w \models \varphi \\ \bot & \text{else.} \end{cases} \end{aligned}$$

The set of models of an LTL formula φ defines a language $\mathcal{L}(\varphi) \subseteq \Sigma^\omega$ of infinite words over $\Sigma = 2^{\mathrm{AP}}$ as follows:

$$\mathcal{L}(\varphi) = \{w \in \Sigma^\omega \mid [\![w \models \varphi]\!]_\omega = \top\}.$$

Note that the De Morgan rules, equivalences for G and F and the fixed point equations for U and R are still valid.

5.1 Anticipatory LTL Semantics: LTL_3

Recall that *anticipation* requires that once every (possibly infinite) continuation of a finite trace leads to the same verdict, then the finite trace evaluates to this very same verdict.

FLTL_4 is not anticipatory: $[\![\{p\} \models \mathrm{XXfalse}]\!]_4 = \bot^p$ but it should yield $\bot$ as any finite extension will eventually reveal that the formula is falsified.

Definition 34 LTL_3 (Semantics). *Let φ be an LTL formula and let $u \in \Sigma^*$ be a finite word. Then the semantics of φ with respect to u is defined as follows:*

$$[\![u \models \varphi]\!]_3 = \begin{cases} \top & \text{if } \forall w \in \Sigma^\omega : [\![uw \models \varphi]\!]_\omega = \top \\ \bot & \text{if } \forall w \in \Sigma^\omega : [\![uw \models \varphi]\!]_\omega = \bot \\ ? & \text{else.} \end{cases}$$

Example 16. Consider $\varphi = \mathrm{G}(p \to \mathrm{F}\mathit{false})$ and $\emptyset\{q\}\{p\}\emptyset \in \Sigma^*$ for $\Sigma = 2^{\mathrm{AP}}$ and $\mathrm{AP} = \{p, q\}$. We then have

- $[\![\emptyset \models \varphi]\!]_3 = ?$
- $[\![\emptyset\{q\} \models \varphi]\!]_3 = ?$
- $[\![\emptyset\{q\}\{p\} \models \varphi]\!]_3 = \bot$
- $[\![\emptyset\{q\}\{p\}\emptyset \models \varphi]\!]_3 = \bot$
- $[\![\emptyset\{q\}\{p\}u \models \varphi]\!]_3 = \bot$ for all $u \in \Sigma^*$

Possible Verdicts of LTL Formulae. Consider a word $w \in \Sigma^*$ for $\Sigma = 2^{\mathrm{AP}}$ and propositions $p, q \in \mathrm{AP}$. We then have

- $[\![w \models p\mathrm{U}q]\!]_3 \in \{\top, ?, \bot\}$
- $[\![w \models p\mathrm{R}q]\!]_3 \in \{\top, ?, \bot\}$
- $[\![w \models \mathrm{F}p]\!]_3 \in \{\top, ?\}$

– $[\![w \models \mathrm{G}p]\!]_3 \in \{?, \bot\}$
– $[\![w \models \mathrm{GF}p]\!]_3 = ?$
– $[\![w \models \mathrm{FG}p]\!]_3 = ?$

5.2 Monitorable Properties

In this subsection, we study the notion of monitorable properties. In essence, a property is called *monitorable* if there is no finite point in time from we may conclude that we stay with the verdict? for ever. For comparison, we also recall the notion of safety and co-safety properties, which we do first.

Definition 35 (Good, Bad and Ugly Prefixes). *Given a language $L \subseteq \Sigma^\omega$ of infinite words over Σ we call a finite word $u \in \Sigma^*$*

– *a* good prefix *for L if $\forall w \in \Sigma^\omega : uw \in L$,*
– *a* bad prefix *for L if $\forall w \in \Sigma^\omega : uw \notin L$ and*
– *an* ugly prefix *for L if $\forall v \in \Sigma^* : uv$ is neither a good prefix nor a bad prefix.*

Example 17 (The Good, The Bad and The Ugly).

– $\{p\}\{q\}$ is a good prefix for $\mathcal{L}(\mathrm{F}q)$.
– $\{p\}\{q\}\{p\}$ is a good prefix for $\mathcal{L}(\mathrm{F}q)$.
– $\{p\}\{q\}$ is a bad prefix for $\mathcal{L}(\mathrm{G}p)$.
– every $w \in \Sigma^*$ is an ugly prefix for $\mathcal{L}(\mathrm{GF}p)$.
– $\{p\}$ is an ugly prefix for $\mathcal{L}(p \rightarrow \mathrm{GF}p)$.

LTL_3 indentifies good/bad prefixes Given an LTL formula φ and a finite word $u \in \Sigma^*$, then

$$[\![u \models \varphi]\!]_3 = \begin{cases} \top & \text{if } u \text{ is a good prefix for } \mathcal{L}(\varphi) \\ \bot & \text{if } u \text{ is a bad prefix for } \mathcal{L}(\varphi) \\ ? & \text{otherwise} \end{cases}$$

Safety Properties assert that nothing bad happens. Such a property is *violated* iff something *bad* happens after *finitely many steps*. ($\rightarrow$ A bad prefix exists.)

Co-Safety Properties assert that something good happens. Such a property is *fulfilled* iff something *good* happens after *finitely many steps*. ($\rightarrow$ A good prefix exists.)

Definition 36 ((Co-)Safety Languages). *A language $L \subseteq \Sigma^\omega$ is called*

– *a* safety language *if for all $w \notin L$ there is a prefix $u \in \Sigma^*$ of w which is a bad prefix for L.*
– *a* co-safety language *if for all $w \in L$ there is a prefix $u \in \Sigma^*$ of w which is a good prefix for L.*

Definition 37 ((Co-)Safety Properties). *An LTL formula φ is called*

- *a* safety property *if its set of models $\mathcal{L}(\varphi)$ is a safety language.*
- *a* co-safety property *if its set of models $\mathcal{L}(\varphi)$ is a co-safety language.*

Example 18 Consider propositions $p, q \in \mathrm{AP}$.

Formula	Safety	Co-Safety
$\mathrm{G}\, p$	✔	✘
$\mathrm{F}\, p$	✘	✔
$\mathrm{X}\, p$	✔	✔
$\mathrm{G}\,\mathrm{F}\, p$	✘	✘
$\mathrm{F}\,\mathrm{G}\, p$	✘	✘
$\mathrm{X}\, p \vee \mathrm{G}\,\mathrm{F}\, p$	✘	✘
$p\,\mathrm{U}\, q$	✘	✔
$p\,\mathrm{R}\, q$	✔	✘

- $p\mathrm{U}q$ is not a safety property, because $\{p\}^\omega \not\models p\mathrm{U}q$, but there is no bad prefix.
- $p\mathrm{U}q$ is a co-safety property, because every infinite word $w \in \Sigma^\omega$ with $w \models p\mathrm{U}q$ must contain the releasing q in a finite prefix.
- $p\mathrm{R}q$ is not a co-safety property, because $\{q\}^\omega \models p\mathrm{R}q$, but there is no good prefix.
- $p\mathrm{R}q$ is a safety property, because every infinite word $w \in \Sigma^\omega$ with $w \not\models p\mathrm{R}q$ must contain the violating absence of q in a finite prefix.

Let us now turn our attention to the notion of monitorabilty, which intuitively characterizes a property as monitorable, if eventually, we might get a definite verdict when monitoring it.

Definition 38 (Monitorable Languages). *A language $L \subseteq \Sigma^\omega$ is called* monitorable *iff L has no ugly prefix.*

Definition 39 (Monitorable Properties). *An LTL formula φ is called* monitorable *iff its set of models $\mathcal{L}(\varphi)$ is monitorable.*

Safety Properties

Co-Safety Properties

Remark 1. Safety and Co-Safety Properties are monitorable.

Theorem 1. *The class of* monitorable properties

- *comprises safety- and co-safety properties, but*

– *is strictly larger than their union.*

Proof. Consider AP $= \{p, q, r\}$ and $\varphi = ((p \vee q)\mathrm{U}r) \vee \mathrm{G}p.\{p\}^\omega \models \varphi$ without good prefix, therefore φ is not a co-safety property. $\{q\}^\omega \not\models \varphi$ without bad prefix, therefore φ is not a safety property. Every finite word $u \in \Sigma^*$ that is not a bad prefix can become a good prefix by appending $\{r\}$. Every finite word $u \in \Sigma^*$ that is not a good prefix can become a bad prefix by appending $\emptyset$. No ugly prefix exists as every prefix is either good, bad or can become good or bad by appending $\{r\}$ or $\emptyset$.

5.3 Monitor Construction for Anticipatory Runtime Verification

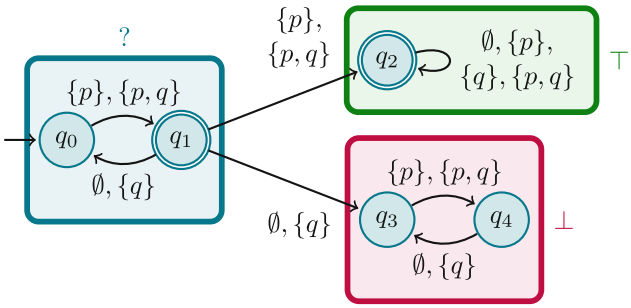
In this section we will recall the idea of translating LTL (with its two-valued semantics on infinite words) into Büchi automata and then use this to present a monitor construction for anticipatory and impartial runtime verification.

Our goal is to construct a Moore machine $\mathcal{M}^\varphi$ for an LTL formula φ that reads a word letter by letter and outputs in every state the value of $[\![w \models \varphi]\!]_3$ where w is the word read so far.

A first idea would be to reuse the evlFLTL$_4$ function and to perform an additional check on the resulting formula. More precisely, one could return $\top$ or $\bot$ if the formula is a tautology or unsatisfiable, respectively, and to return? in any other case. However, a satisfiability check is a complex task so that we are after a different approach.

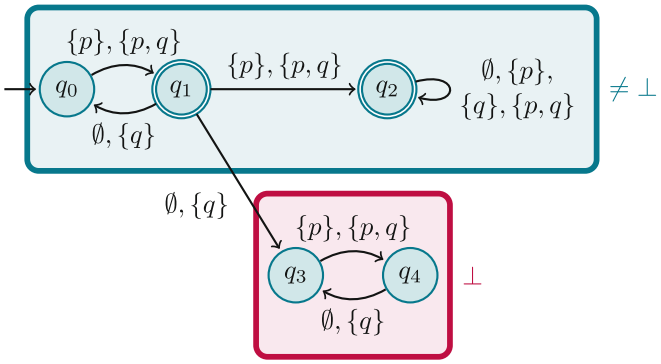
Instead, we follow the idea to construct a Büchi automaton (BA) accepting precisely the models of the LTL formula and analyse it. More precisely, we identify good states $\top$ in which the BA will accept on every continuation, bad states $\bot$, in which the BA will reject on every continuation, yielding other states ?, in which no conclusion is possible yet.

For example consider AP $= \{p, q\}$ and $\Sigma = 2^{\mathrm{AP}}$ and the following automaton:



Our goal is to create a Moore machine using these labels as outputs.

While it is algorithmically easy to identify the $\bot$-states in a non-deterministic machine, the $\top$-states are difficult to estimate as they require universality check. Therefore, we take a slighltly different approach. We only identify bad states ($\bot$) and label everything else as not bad ($\neq \bot$). Perform this for the LTL formulae φ and $\neg\varphi$. Good states for φ are bad states for $\neg\varphi$.



Note that an LTL formula can be complemented by adding ¬. And that complementing a BA potentially needs *exponential time*.

Büchi Automata (BA).

Definition 40 (ω-regular Languages). *A language $L \subseteq \Sigma^\omega$ over an alphabet Σ is called* ω-regular *iff there are regular languages $U_i, V_i \subseteq \Sigma^*$ for $i \in \{1, \ldots, m\}$ such that*

$$L = \bigcup_{i=1}^{m} U_i \circ V_i^\omega.$$

Example 19 (ω-regular Languages). Consider an alphabet $\Sigma = 2^{\mathrm{AP}}$ for $\mathrm{AP} = \{p, q\}$.

$$\mathcal{L}(\mathrm{G}p) = \{\{p\}, \{p, q\}\}^\omega$$
$$\mathcal{L}(\mathrm{F}p) = \Sigma^* \circ \{\{p\}, \{p, q\}\} \circ \Sigma^\omega$$

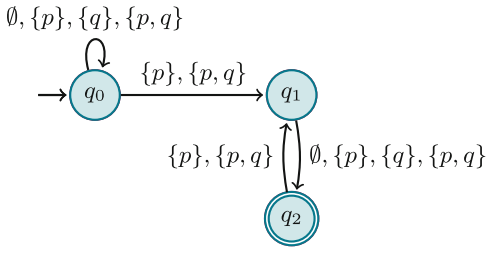


Fig. 8. A examplifying Büchi automaton

Büchi automata were first introduced by Büchi in [Buc62] for obtaining a decision procedure for the monadic second-order theory of structures with one successor. Let us establish the key concepts of this kind of automata to the extent needed in our thesis. For a thorough introduction to Büchi automata we refer to [Tho90]. We start directly with their definition:

Definition 41 (Non-deterministic Büchi Automata (BA)). *A (non-deterministic)* Büchi automaton *is a tuple* $\mathcal{A} = (\Sigma, Q, Q_0, \Delta, F)$ *such that*

- Σ *is the* input alphabet*,*
- Q *is the finite non-empty set of* states*,*
- $Q_0 \subseteq Q$ *is the set of* initial states*,*
- $\Delta \subseteq Q \times \Sigma \times Q$ *is the* transition relation *and*
- $F \subseteq Q$ *is the set of* accepting states*.*

A Büchi automaton may be represented as an edge-labeled directed graph. Its nodes are the states and an edge labeled by $a \in \Sigma$ leads from a node (state) $q \in Q$ to a node (state) $q' \in Q$ iff $(q, a, q') \in \Delta$. The initial state is marked with an incoming arrow. A final state, on the other hand, is identified by a second circle around the node. Figure 8 shows an exemplifying Büchi automaton over the alphabet $\Sigma = 2^{\mathrm{AP}}$ for $\mathrm{AP} = \{p, q\}$.

The automaton operates on infinite input words. The idea of its behavior is that it chooses (non-deterministically) a possible successor state q' such that $(q, a, q') \in \Delta$, provided it is in the state q and reads an action a. Of course, the automaton starts in its initial state.

Definition 42 (Run of a BA). *A run of a BA* $\mathcal{A} = (\Sigma, Q, Q_0, \Delta, F)$ *on an infinite word* $w \in \Sigma^\omega$ *is a function* $\rho : \mathbb{N} \to Q$ *such that*

- $\rho(0) \in Q_0$ *and*
- $\forall i \in \mathbb{N} \backslash \{0\} : (\rho(i-1), w_i, \rho(i)) \in \Delta$.

Sometimes, we represent a run ρ only by its sequence of images. For example, a run of the automaton shown in Fig. 8 on the word $\{p\}\{p,q\}(\{q\}\{p\})^\omega$ is given by the sequence $q_0 q_0 q_1 (q_2 q_1)^\omega$.

Definition 43 (Accepting Runs of a BA). *A run* ρ *of a BA* $\mathcal{A} = (\Sigma, Q, Q_0, \Delta, F)$ *is called* accepting *iff*

$$\mathrm{Inf}(\rho) \cap F \neq \emptyset,$$

where $\mathrm{Inf}(\rho)$ *denotes the* set of states visited infinitely often *given by*

$$\mathrm{Inf}(\rho) = \Big\{ q \in Q \;\Big|\; |\{k \in \mathbb{N} \mid \rho(k) = q\}| = \infty \Big\}.$$

$\mathcal{A}$ *accepts* w *if there is an accepting run* ρ *of* $\mathcal{A}$ *on* w*.*

Again the language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ of an automata $\mathcal{A}$ with the alphabet Σ is defined as follows:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}.$$

We make use of the fundamental result by Vardi and Wolper:

Theorem 2 (From LTL to BA [VW86]). *For a given LTL formula* φ *over an alphabet* Σ *we can construct a BA* $\mathcal{A}^\varphi$ *that accepts precisely the models of* φ*. Moreover, the size of the automaton is exponential in the lenght of the formula.*

5.4 Emptiness per State

For a given LTL formula φ over an alphabet Σ we will construct a Moore machine $\mathcal{M}^\varphi$ that reads finite words $w \in \Sigma^*$ and outputs $[\![w \models \varphi]\!]_3 \in \mathcal{B}_3$.

For the next steps let

$$\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$$

denote the BA accepting all models of φ and

$$\mathcal{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$$

denote the BA accepting all words falsifying φ.

Definition 44 (BA With Adjusted Initial State). *For an BA $\mathcal{A}$, we denote by $\mathcal{A}(q)$ the BA that coincides with $\mathcal{A}$ except for the set of initial states Q_0, which is redefined in $\mathcal{A}(q)$ as $Q_0 = \{q\}$.*

Definition 45 (Emptiness per State). *We then define a function $\mathcal{F}^\varphi : Q^\varphi \to \mathbb{B}_2$ (with $\mathbb{B}_2 = \{\top, \bot\}$) where we set $\mathcal{F}^\varphi(q) = \top$ iff $\mathcal{L}(\mathcal{A}^\varphi(q)) \neq \emptyset$.*

Using $\mathcal{F}^\varphi$, we define the NFA $\hat{\mathcal{A}}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$ with $\hat{F}^\varphi = \{q \in Q^\varphi \mid \mathcal{F}^\varphi(q) = \top\}$. Analogously, we set $\hat{\mathcal{A}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ with $\hat{F}^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid \mathcal{F}^{\neg\varphi}(q) = \top\}$.

To determine $F^\varphi(q)$, we identify in linear time the strongly connected components in A^φ, which can be done using Tarjan's algorithm [Tar72] or nested depth-first algorithms as examined in [SE05].

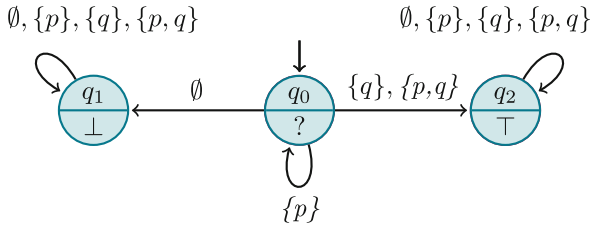
Lemma 4 (LTL_3 Evaluation). *With the notation as before, we have*

$$[\![w \models \varphi]\!]_3 = \begin{cases} \top & \text{if } w \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \\ \bot & \text{if } w \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \\ ? & \text{if } w \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi} \cap \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})) \end{cases}$$

Proof. $[\![w \models \varphi]\!]_3 = \top$ if $w \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$

- Feeding a finite prefix $w \in \Sigma^*$ to the BA $\mathcal{A}^{\neg\varphi}$,
  we reach the set $\delta^{\neg\varphi}(Q_0^{\neg\varphi}, w) \subseteq Q^{\neg\varphi}$ of states.
- If $\exists q \in \delta^{\neg\varphi}(Q_0^{\neg\varphi}, w) : \mathcal{L}(\mathcal{A}^{\neg\varphi}(q)) \neq \emptyset$ then
  we can choose $\sigma \in \mathcal{L}(\mathcal{A}^{\neg\varphi}(q))$ such that $w\sigma \in \mathcal{L}(\mathcal{A}^{\neg\varphi})$.
- Such a state q exists by definition iff $w \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$.
- If $w \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$ then every possible continuation $w\sigma$ of w will be rejected by $\mathcal{A}^{\neg\varphi}$, i.e. $[\![w\sigma \models \varphi]\!]_\omega = \top$ for all $\sigma \in \Sigma^\omega$. Therefore we have $[\![w \models \varphi]\!]_3 = \top$.
  $[\![w \models \varphi]\!]_3 = \bot$ if $w \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$
- can be seen by substituting φ for $\neg\varphi$.
  $[\![w \models \varphi]\!]_3 =?$ if $w \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \cap \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$
- If $\exists q \in \delta^{\neg\varphi}(Q_0^{\neg\varphi}, w) : \mathcal{L}(\mathcal{A}^{\neg\varphi}(q)) \neq \emptyset$ and
  $\exists q' \in \delta^{\varphi}(Q_0^{\varphi}, w) : \mathcal{L}(\mathcal{A}^{\varphi}(q')) \neq \emptyset$ then
  we can choose $\sigma \in \mathcal{L}(\mathcal{A}^{\neg\varphi}(q))$ and $\sigma' \in \mathcal{L}(\mathcal{A}^{\varphi}(q'))$ such that $[\![w\sigma \models \varphi]\!]_2 = \bot$ and $[\![w\sigma' \models \varphi]\!]_2 = \top$.
- Hence we have $[\![w \models \varphi]\!]_3 = ?$.

Deterministic Moore Machine (FSM). Our goal is now to derive a deterministic Moore machine. As an example, let us consider the Moore machine that we will obtain for $p\mathrm{U}q$



Definition 46 (Deterministic Moore Machine (FSM)). *A (deterministic)* Moore machine *is a tupel* $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta, \lambda)$ *where*

- Σ *is the* input alphabet,
- Q *is a finite set of* states,
- $q_0 \in Q$ *is the* initial state,
- Γ *is the* output alphabet,
- $\delta : Q \times \Sigma \to Q$ *is the* transition function *and*
- $\lambda : Q \to \Gamma$ *is the* output function.

Definition 47 (Run of a Deterministic Moore Machine). *A* run *of a (deterministic) Moore machine* $\mathcal{M} = (\Sigma, Q, q_0, \Gamma, \delta, \lambda)$ *on a finite word* $w \in \Sigma^n$ *with outputs* $o_i \in \Gamma$ *is a sequence*

$$t_0 \overset{w_1}{\to} t_1 \overset{w_2}{\to} \dots \overset{w_{n-1}}{\to} t_{n-1} \overset{w_n}{\to} t_n$$

such that

- $t_0 = q_0$,
- $t_i = \delta(t_{i-1}, w_i)$ *and*
- $o_i = \lambda(t_i)$.

The output *of the run is* $o_n = \lambda(t_n)$.

Let $\tilde{\mathcal{A}}^{\varphi} = (\Sigma, \tilde{Q}^{\varphi}, q_0^{\varphi}, \tilde{\delta}^{\varphi}, \tilde{F}^{\varphi})$ and $\tilde{\mathcal{A}}^{\neg\varphi} = (\Sigma, \tilde{Q}^{\neg\varphi}, q_0^{\neg\varphi}, \tilde{\delta}^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be the equivalent DFAs of the NFAs $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$.

Definition 48 (Monitor $\mathcal{M}^{\varphi}$ for an LTL formula φ). *We define the product automaton* $\overline{\mathcal{A}}^{\varphi} = \tilde{\mathcal{A}}^{\varphi} \times \tilde{\mathcal{A}}^{\neg\varphi}$ *as the Moore machine* $(\Sigma, \overline{Q}, \overline{q}_0, \mathbb{B}_3, \overline{\delta}, \overline{\lambda})$, *where*

- $\overline{Q} = \tilde{Q}^{\varphi} \times \tilde{Q}^{\neg\varphi}$,
- $\overline{q}_0 = (\tilde{q}_0^{\varphi}, \tilde{q}_0^{\neg\varphi})$,
- $\overline{\delta}((q, q'), a) = (\tilde{\delta}^{\varphi}(q, a), \tilde{\delta}^{\neg\varphi}(q', a))$ *and*
- $\overline{\lambda} : \overline{Q} \to \mathbb{B}_3$ *with*

$$\overline{\lambda}((q, q')) = \begin{cases} \top & \text{if } q' \notin \tilde{F}^{\neg\varphi} \\ \bot & \text{if } q \notin \tilde{F}^{\varphi} \\ ? & \text{if } q \in \tilde{F}^{\varphi} \text{ and } q' \in \tilde{F}^{\neg\varphi}. \end{cases}$$

The monitor $\mathcal{M}^{\varphi}$ *of* φ *is obtained by minimizing* $\overline{\mathcal{A}}^{\varphi}$.

The overall construction is summarized in Fig. 9.

Figure 10 shows an example construction for the formula $p\mathrm{U}q$.

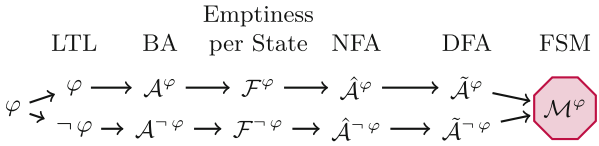


Fig. 9. The overall construction for LTL3 monitors.

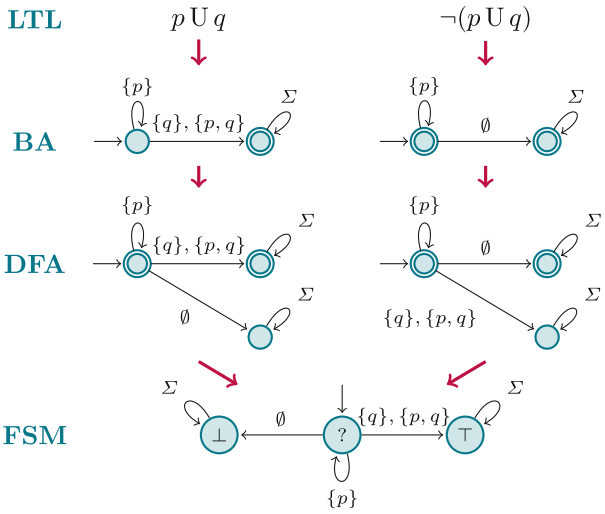


Fig. 10. The monitor construction for $p\mathrm{U}q$ in LTL_3

5.5 Analysis

Let us first look at the complexity of the monitor construction. To this, recall the construction as depicted in Fig. 11.

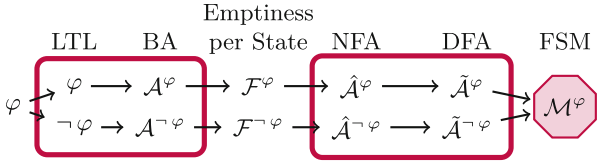


Fig. 11. The complexity of the LTL_3 monitor construction

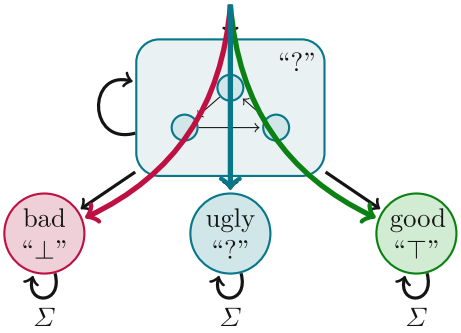
Thus, $|M| \in 2^{2^{O(|\varphi|)}}$ and the complexity is dominated by the translation of the underlying formulas into the Büchi automaton and the determinisation of the respective NFA. As the resulting monitor is unique, the whole construction is optimal. That said, the latter steps of the translation can also by done "on-the-fly" allowing to trade space for runtime.

Practical examples show that resulting monitors for typical properties are of small sizes. See [BLS11] for details.

Let us briefly look at the general shape of a monitor. Due to minimization, there are at most two sinks outputting $\bot$ or $\top$. In all other states, the output is ?. However, there might be (single) sink outputting ?, while other states outputting ? may reach the $\top$ or $\bot$-states. A simple analysis reveals that a sink labelled ? is there if and only if the underlying property is monitorable. The previous constructions yield a 2EXPTIME algorithm for deciding monitorability. However, the exact complexity of deciding monitorability is so-far unknown.

Let us close this section by sketching the general shape of an LTL_3 monitor and the corresponding types of prefixes leading to respectively bad, ugly, and good states.

Structure of Monitors



Classification of Prefixes of Words

Bad prefixes Ugly prefixes Good prefixes

6 Conclusion

In this paper, we provided an introduction to the field of runtime verification and a detailed overview of the approach based on monitoring linear-time temporal logic specifications. On this road, we learned about the difference of monitoring completed runs and monitoring ongoing executions. In the latter case, we realized that multiple verdicts rather than two are more appropriate. We provided two approaches for the monitoring expanding executions incorporating the ideas of impartiality and anticipation. Impartiality means to stay with the verdicts $\top$ and $\bot$ once decided for them while anticipation requires to go for $\top$ and $\bot$ (and stay there) as soon as possible (in the sense made precise in the previous sections). Moreover, we discussed that automata-based approaches can be understood as pre-computations of rewriting-based approaches and hereby, we were able to bridge these two worlds.

We have explored only a very limited area of the rapidly expanding field of runtime verification. We have not covered applications where further entities

come into play such as data or concurrency. Moreover, we only touched monitoring but did not discuss steering of a system or enforcing properties at runtime. Nevertheless, we have given a first detailed introduction into one main part of the field.

Acknowledgement. Many thanks goes to the team at ISP for fruitful discussions about the content of this chapter, especially to Malte Schmitz.

References

[BC04] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)

[BCC+03] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)

[BJK+05] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems, Advanced Lectures. LNCS, vol. 3472. Springer, Heidelberg (2005). The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004

[BLS06a] Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: Proceedings of the Australian Software Engineering Conference (ASWEC 2006), pp. 243–252. IEEE (2006)

[BLS06b] Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: ASWEC, pp. 243–252 (2006)

[BLS11] Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. (TOSEM), **20**(4) (2011, in press)

[Boe81] Boehm, B.W.: Software Engineering Economics. Prentice Hall, Englewood Cliffs (1981)

[Buc62] Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Method, and Philosophy of Science, pp. 1–12. Stanford University Press, Stanford (1962)

[CAPS15] Chimento, J.M., Ahrendt, W., Pace, G.J., Schneider, G.: StARVOOrS: a tool for combined static and runtime verification of Java. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 297–305. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23820-3_21

[CGP01] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2001)

[Cho78] Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)

[CR94] Crow, J., Rushby, J.: Model-based reconfiguration: diagnosis and recovery. NASA Contractor report 4596, NASA Langley Research Center (1994)

[CR07] Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: OOPSLA, pp. 569–588 (2007)

[DAC99] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420 (1999)

[DGR04] Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. **30**(12), 859–872 (2004)

[DSS+05] D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: TIME (2005)

[HS06] Hinchey, M.G., Sterritt, R.: Self-managing software. IEEE. Comput. **39**(2), 107–109 (2006)

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Boston (1979)

[Leu12] Leucker, M.: Sliding between model checking and runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 82–87. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35632-2_10

[MB15] Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, 25–29 May 2015, pp. 494–503. IEEE Computer Society (2015)

[Mye04] Myers, G.J.: The Art of Software Testing, 2nd edn. Wiley, Hoboken (2004)

[PL04] Pretschner, A., Leucker, M.: Model-based testing – a glossary. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 607–609. Springer, Heidelberg (2005). doi:10.1007/11498490_27

[Pnu77] Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)

[SC85] Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (1985)

[SE05] Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31980-1_12

[SS14] Scheffel, T., Schmitz, M.: Three-valued asynchronous distributed runtime verification. In: Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, 19–21 October 2014, pp. 52–61. IEEE (2014)

[SVAR04] Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) 26th International Conference on Software Engineering (ICSE 2004), 23–28 May 2004, Edinburgh, UK, pp. 418–427. IEEE Computer Society (2004)

[Tar72] Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)

[Tho90] Thomas, W.: Automata on infinite objects (Chap. 4). In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B, pp. 133–191. Elsevier Science Publishers B. V., Amsterdam (1990)

[Vas73] Vasilevski, M.P.: Failure diagnosis of automata. Cybernetic **9**(4), 653–665 (1973)

[VW86] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS, pp. 332–344 (1986)