Teaching Runtime Verification

Martin Leucker

Universtität zu Lübeck
Institut für Softwaretechnik und Programmiersprachen

Abstract. In this paper and its accompanying tutorial, we discuss the topic of teaching runtime verification. The aim of the tutorial is twofold. On the one hand, a condensed version of a course currently given by the author will be given within the available tutorial time, giving an idea about the topics of the course. On the other hand, the experience gained by giving the course should also be presented and discussed with the audience. The overall goal is to simplify the work of colleagues developing standard and well accepted courses in the field of runtime verification.

1 Introduction

Runtime Verification (RV) has become a mature field within the last decades. It aims at checking correctness properties based on the actual execution of a software or hardware system.

Research on runtime verification is traditionally presented at formal methods conferences like CAV (computer aided verification) or TACAS (tools and algorithms for the analysis of systems), or, software engineering conferences like ICSE or ASE. Starting in 2001, the RV community has formed its own scientific event, the runtime verification workshop, which has in the meantime been upgraded to the runtime verification conference. There is a community forming webpage that is available under the address `www.runtime-verification.org` and first definitions and explanations entered their way into the online dictionary `wikipedia`. Last but not least, several courses on runtime verification are given to PhD, master, or even bachelor students at several universities.

So far, however, no dedicated text book on the topic of runtime verification is available and actual courses on runtime verification are still to be considered preliminary as the field of runtime verification is still undergoing rapid changes and no kernel material of the field has been identified, or, at least has not been fixed by the community.

In this paper and its accompanying tutorial, we discuss the topic of *teaching runtime verification.* It is based on the author's course given at the University of Lübeck. The course took place once a week, each time 1.5 hours, and in total about 14 times. The aim of this paper and its accompanying tutorial is twofold. On the one hand, a condensed version of the course should be shown, giving an idea about the outline and topics of the course. On the other hand, the experience gained by giving the course are also presented and discussed with the audience. The overall goal of the tutorial is to simplify the work of colleagues developing standard and well accepted courses in the field of runtime verification.

Content of the RV course

1. The tutorial/course starts with a short discussion on typical areas that are preferably addressed at runtime. It is motivated why static verification techniques must often be encompassed by runtime verification techniques.
2. Runtime verification is defined and a taxonomy for runtime verification is developed. The taxonomy will be the basis for getting a systematic picture on the field of runtime verification and may also be used to organize the different contributions by the runtime verification community. Runtime verification is identified as a research discipline aiming at synthesizing monitors from high level specifications, integrating them into existing execution frameworks, and using the results of monitors for steering or guiding a program. It may work on finite, finite but continuously expanding, or on prefixes of infinite traces.
3. In the subsequent part of the tutorial/course synthesis techniques for Linear Temporal Logic (LTL) will be presented. Both, approaches based on rewriting the formula to check and approaches based on translating the formula at hand into an automaton will be briefly described. Moreover the conceptual difference between these two fundamental approaches will be explained.
4. The second part of the tutorial deals with integrating monitors into running systems and with techniques for steering the executing system based on the results of monitors.
5. In the third part we will list existing runtime verification frame works, which will eventually be classified with respect to the initially developed taxonomy.

Intended Audience. The tutorial is especially intended for current or future lecturers in the field of runtime verification. At the same time, as the main ideas of the underlying course are taught, it is of interest to advanced master students and PhD students for getting an introduction to the field of runtime verification. Finally, researchers active in formal methods who want to get comprehensive picture on the field of runtime verification may benefit from the tutorial as well.

2 The Virtue of Runtime Verification

The course starts with a short discussion on typical areas that are preferably addressed at runtime. It is motivated why static verification must often be encompassed by runtime verification techniques. We do so 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.
- 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 only 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.

3 Runtime Verification—Definition and Taxonomy

3.1 Towards a Definition

A *software failure* is the 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 [1].

According to IEEE [2], *verification* comprises all techniques suitable for showing that a system satisfies its specification. Traditional verification techniques comprise theorem proving [3], model checking [4], and testing [5,6]. A relatively new direction of verification is *runtime verification*,[1] which manifested itself within the previous years as a *lightweight* verification technique:

Definition 1 (Runtime Verification). Runtime verification *is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a* run *of a system under scrutiny (SUS) satisfies or violates a given correctness property. Its distinguishing research effort lies in* synthesizing monitors from high level specifications.

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

Monitors. A run of a system is understood as a possibly infinite sequence of the system's states, which are formed by current variable assignments, or as the sequence of (input/output) actions a system is emitting or performing. Formally, a run may be considered as a possibly infinite *word* or *trace*. 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. While, in verification, we are interested in the question whether a run, and more generally, all runs of a system adhere to given correctness properties, executions are the primary object analyzed in the setting of RV.

Checking whether an execution meets a correctness property is typically performed using a *monitor*. 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. However, to cover richer approaches to RV, we define the notion of monitors in a slightly more general form:

Definition 2 (Monitor). *A* monitor *is a device that reads a finite trace and yields a certain* verdict*.*

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 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 (see Section 4.1 for a precise definition). Sometimes, one might be even more liberal and consider also verdicts that are not elements of a truth domain.

3.2 Taxonomy

A taxonomy may be used to get a systematic account to the field of runtime verification and to organize the different contributions by the RV community into a global picture. Figure 1 shows a taxonomy that is briefly described in the following.

First, runtime verification may work on (i) finite (terminated), (ii) finite but continuously expanding, or (iii) on prefixes of infinite traces. For the two latter cases, a monitor should adhere to the two maxims *impartiality* and *anticipation*. *Impartiality* requires that a finite trace is not evaluated to *true* or, respectively *false*, if there still exists an (infinite) continuation leading to another verdict. *Anticipation* requires that once every (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

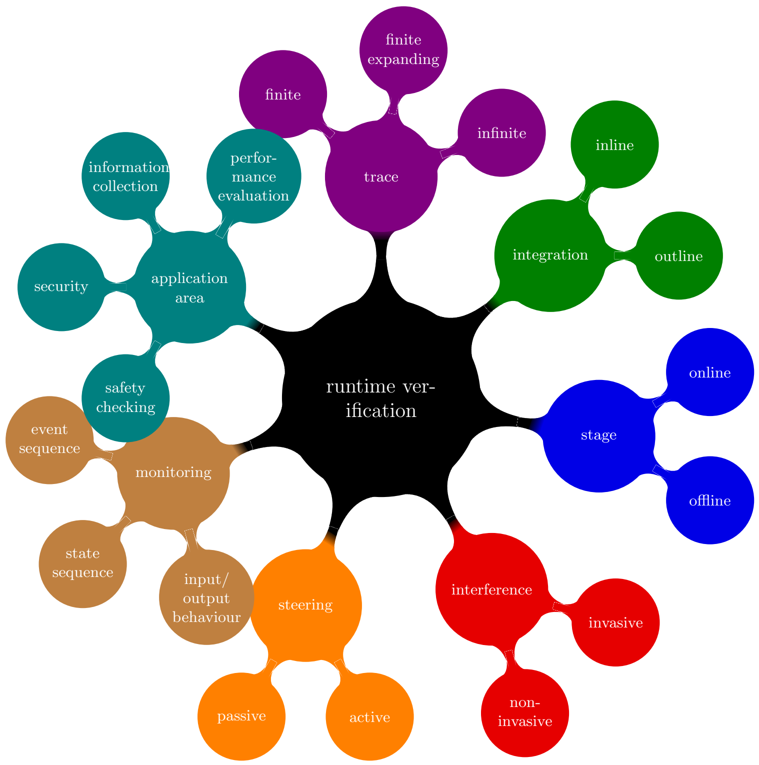

Fig. 1. Taxonomy of runtime verification

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.

RV approaches may differ in what part of a run is actually monitored. For example, a system may be analyzed with respect to its *input/output behavior*, one of its *state sequences*, or wrt. a sequence of *events* related to the system's execution.

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*. On the other hand, a monitor may work on a (finite set of) *recorded* execution(s), in which case we speak of *offline monitoring*.

Synthesized monitoring code may be interweaved with the program to check, i.e. it may be *inlined*, or, it may be used to externally synthesize a monitoring device, i.e., it may be *outlined*. Clearly, inlined monitors act online.

The monitor typically interferes with the system to observe, as it runs, for example, on the same CPU as the SUS. However, using additional computation resources, monitoring might not change the behavior of the SUS. We distinguish these two cases using the terms *invasive* and *non-invasive* monitoring.

While, to our understanding, runtime verification is mainly concerned with the synthesis of efficiently operating monitors, RV frameworks may be distinguished by whether the resulting monitor is just observing the program's execution and reporting failures, i.e., it is *passive*, or, whether the monitor's verdict may be used to actually steer or heal the system's execution, i.e., it is *active*.

Runtime verification may be used for different applications. Most often, it is used to check *safety conditions*. Similarly, it may be used to ensure *security conditions*. However, it is equally suited to simply *collect information* of the system's execution, or, for *performance evaluation* purposes.

4 Runtime Verification for LTL

As considered the heart of runtime verification, the main focus of an RV course lies on synthesis procedures yielding monitors from high-level specifications. We outline several monitor synthesis procedures for Linear-time Temporal Logic (LTL, [7]). In general, two main approaches can be found for synthesizing monitoring code: Monitors may either be given in terms of an automaton, which is precomputed from a given correctness specification. Alternatively, the correctness specification may be taken directly and *rewritten* in a tableau-like fashion when monitoring the SUS. We give examples for both approaches.

4.1 Truth Domains

We consider the traditional two-valued semantics with truth values *true*, denoted with $\top$, and *false*, denoted with $\bot$, next to truth values giving more information to which degree a formula is satisfied or not. Since truth values should be combinable in terms of Boolean operations expressed by the connectives of the underlying logic, these truth values should form a certain lattice.

A *lattice* is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where for each $x, y \in \mathcal{L}$, there exists (i) a unique *greatest lower bound* (glb), which is called the *meet* of x and y, and is denoted with $x \sqcap y$, and (ii) a unique *least upper bound* (lub), which is called the *join* of x and y, and is denoted with $x \sqcup y$. A lattice is called *finite* iff $\mathcal{L}$ is finite. Every finite lattice has a well-defined unique least element, called *bottom*, denoted with $\bot$, and analogously a greatest element, called *top*, denoted with $\top$. A lattice is *distributive*, iff $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$, and, dually, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$. In a *de Morgan* lattice, every element x has a unique *dual* element $\overline{x}$, such that $\overline{\overline{x}} = x$ and $x \sqsubseteq y$ implies $\overline{y} \sqsubseteq \overline{x}$. As the common denominator of the semantics for the subsequently defined logics is a finite de Morgan lattice, we define:

Definition 3 (Truth domain). *We call $\mathcal{L}$ a* truth domain*, if it is a finite de Morgan lattice.*

4.2 LTL—Syntax and Common Semantics

As a starting point for all subsequently defined logics, we first recall linear temporal logic (LTL).

For the remainder of this paper, let AP be a finite and non-empty set of *atomic propositions* and $\Sigma = 2^{\mathrm{AP}}$ a finite *alphabet*. We write a_i for any single element of Σ, i.e., a_i is a possibly empty subset of propositions taken from AP.

Finite traces (which we call interchangeably words) over Σ are elements of Σ^*, usually denoted with $u, u', u_1, u_2, \ldots$ The empty trace is denoted with ϵ. Infinite traces are elements of Σ^ω, usually denoted with $w, w', w_1, w_2, \ldots$ For some infinite trace $w = a_0 a_1 \ldots$, we denote with w^i the suffix $a_i a_{i+1} \ldots$ In case of a finite trace $u = a_0 a_1 \ldots a_{n-1}$, u^i denotes the suffix $a_i a_{i+1} \ldots a_{n-1}$ for $0 \leq i < n$ and the empty string ϵ for $n \leq i$.

The set of LTL formulae is defined using *true*, the atomic propositions $p \in \mathrm{AP}$, *disjunction*, *next* X, and *until* U, as positive operators, together with *negation* $\neg$. We moreover add dual operators, namely *false*, $\neg p$, *weak next* $\bar{X}$, and *release* R, respectively:

Definition 4 (Syntax of LTL formulae). *Let p be an atomic proposition from a finite set of atomic propositions* AP*. The set of LTL formulae, denoted with LTL, is inductively defined by the following grammar:*

$$\begin{aligned}
\varphi &::= \mathit{true} \mid p \mid \varphi \vee \varphi \mid \varphi\, U\, \varphi \mid X\varphi \\
\varphi &::= \mathit{false} \mid \neg p \mid \varphi \wedge \varphi \mid \varphi\, R\, \varphi \mid \bar{X}\varphi \\
\varphi &::= \neg\varphi
\end{aligned}$$

In a corresponding course, typically further operators are introduced as abbreviations like *finally* F and *globally* G etc.

In the sequel, we introduce several semantic functions, both classical versions for finite and infinite traces and versions adapted to suit the needs in runtime verification. To this end, we consider linear temporal logics $\mathfrak{L}$ with a syntax as in Definition 4, together with a semantic function $[\_ \models \_]_{\mathfrak{L}} : \Sigma^{\omega/*} \times \mathrm{LTL} \to \mathbb{B}_{\mathcal{L}}$ that yields an element of the truth domain $\mathbb{B}_{\mathcal{L}}$, given an infinite or finite trace and an LTL formula. The logics considered in the following have a common part, but differ in certain aspects. The common part of the semantics is shown in Figure 2.

For two formulae $\varphi, \psi \in \mathrm{LTL}$, we say that φ is equivalent to ψ, denoted with $\varphi \equiv_{\mathfrak{L}} \psi$, iff for all $w \in \Sigma^{\omega/*}$, we have $[w \models \varphi]_{\mathfrak{L}} = [w \models \psi]_{\mathfrak{L}}$.

4.3 LTL on Finite Traces

Let us first turn our attention to linear temporal logics over finite traces. We start by recalling a finite version of LTL on finite traces described by Manna and Pnueli [8], here called FLTL.

When interpreting LTL formulae over finite traces, the question arises, how to understand $X\varphi$ when a word consists of a single letter, since then, no next position exists on which one is supposed to consider φ. The classical way to deal with this situation, as apparent for example in Kamp's work [9] is to understand

Boolean constants

$$\begin{aligned}[w \models \mathit{true}]_{\mathfrak{L}} &= \top \\ [w \models \mathit{false}]_{\mathfrak{L}} &= \bot\end{aligned}$$

Boolean combinations

$$\begin{aligned}[w \models \neg\varphi]_{\mathfrak{L}} &= \overline{[w \models \varphi]_{\mathfrak{L}}} \\ [w \models \varphi \vee \psi]_{\mathfrak{L}} &= [w \models \varphi]_{\mathfrak{L}} \sqcup [w \models \psi]_{\mathfrak{L}} \\ [w \models \varphi \wedge \psi]_{\mathfrak{L}} &= [w \models \varphi]_{\mathfrak{L}} \sqcap [w \models \psi]_{\mathfrak{L}}\end{aligned}$$

atomic propositions

$$[w \models p]_{\omega} = \begin{cases} \top & \text{if } p \in a_0 \\ \bot & \text{if } p \notin a_0 \end{cases} \qquad [w \models \neg p]_{\omega} = \begin{cases} \top & \text{if } p \notin a_0 \\ \bot & \text{if } p \in a_0 \end{cases}$$

until/release

$$[w \models \varphi\ U\ \psi]_{\mathfrak{L}} = \begin{cases} \top & \text{there is a } k, 0 \leq k < |w| : [w^k \models \psi]_{\mathfrak{L}} = \top \text{ and} \\ & \text{for all } l \text{ with } 0 \leq l < k : [w^l \models \varphi] = \top \\ \bot & \text{else} \end{cases}$$

$$[w \models \varphi\ R\ \psi]_{\mathfrak{L}} = \begin{cases} \top & \text{for all } k, 0 \leq k < |w| : [w^k \models \psi]_{\mathfrak{L}} = \top \text{ or} \\ & \text{there is a } k, 0 \leq k < |w| : [w^k \models \varphi]_{\mathfrak{L}} = \top \text{ and} \\ & \text{for all } l \text{ with } 0 \leq l \leq k : [w^l \models \psi] = \top \\ \bot & \text{else} \end{cases}$$

Fig. 2. Semantics of LTL formulae over a finite or infinite trace $w = a_0a_1 \ldots \in \Sigma^{*/\omega}$

X as a *strong* next operator, which is false if no further position exists. Manna and Pnueli suggest in [8] to enrich the standard framework by adding a dual operator, the weak next $\bar{X}$, which allows to smoothly translate formulae into negation normal form. In other words, the *strong* X operator is used to express with $X\varphi$ that a next state must exist and that this next state has to satisfy property φ. In contrast, the *weak* $\bar{X}$ operator in $\bar{X}\varphi$ says that if there is a next state, then this next state has to satisfy the property φ. We call the resulting logic FLTL defined over the set of LTL formulae (Definition 4) FLTL.

Definition 5 (Semantics of FLTL [8]). *Let $u = a_0 \ldots a_{n-1} \in \Sigma^*$ denote a finite trace of length n, with $u \neq \epsilon$. The* truth value *of an FLTL formula φ wrt. u, denoted with $[u \models \varphi]_F$, is an element of $\mathbb{B}_2$ and is inductively defined as follows: Boolean constants, Boolean combinations, and atomic propositions are defined as for LTL (see Figure 2, taking u instead of w). (Weak) next are defined as shown in Figure 3.*

Let us first record that the semantics of FLTL is *not* given for the empty word. Moreover, note that a single letter does satisfy *true* but does not satisfy $X\mathit{true}$.Also, $[u \models \neg X\varphi]_F = [u \models \bar{X}\neg\varphi]_F$ follows from LTL whenever $|u| > 1$

(weak) next

$$[u \models X\varphi]_F = \begin{cases} [u^1 \models \varphi]_F & \text{if } u^1 \neq \epsilon \\ \bot & \text{otherwise} \end{cases} \qquad [u \models \bar{X}\varphi]_F = \begin{cases} [u^1 \models \varphi]_F & \text{if } u^1 \neq \epsilon \\ \top & \text{otherwise} \end{cases}$$

Fig. 3. Semantics of FLTL formulae over a trace $u = a_0 \dots a_{n-1} \in \Sigma^*$

and from inspecting the semantics in Figure 3 when $|u| = 1$. Thus, every FLTL formula can be transformed into an equivalent formula in negation normal form.

Monitors for LTL on finite traces The simple answer here is to say that for a finite word, the semantics of an LTL formula can immediately be computed from the semantics definition. However, a slightly more clever way is presented in the next subsection.

4.4 LTL on Finite But Expanding Traces

Let us now consider an LTL semantics adapted towards monitoring finite but expanding traces. Especially when monitoring online, a run of SUS may be given letter-by-letter, say, state-by-state, event-by-event etc. A corresponding monitoring procedure should ideally be able to process such an input string letter-by-letter and should be impartial wrt. the forthcoming letters to receive.

The idea, which is already used in [10], is to use a four-valued semantics, consisting of the truth values *true* ($\top$), *false* ($\bot$), *possibly true* ($\top^p$), and *possibly false* ($\bot^p$). The latter two values are used to signal the truth value of the input word wrt. the two valued semantics *provided the word will terminate now.* More specifically, the four-valued semantics differs from the two-valued semantics shown in the previous subsection only be yielding *possibly false* rather than *false* at the end of a word for the strong next operator and *possibly true* rather than *true* for the weak next operator. We sometimes call the resulting logic FLTL$_4$.

Definition 6 (Semantics of FLTL$_4$). *Let $u = a_0 \dots a_{n-1} \in \Sigma^*$ denote a finite trace of length n, with $u \neq \epsilon$. The* truth value *of an FLTL$_4$ formula φ wrt. u, denoted with $[u \models \varphi]_4$, is an element of $\mathbb{B}_4$ and is inductively defined as follows: Boolean constants, Boolean combinations, and atomic propositions are defined as for LTL (see Figure 2, taking u instead of w). (Weak) next are defined as shown in Figure 4.*

Monitoring expanding traces. While for a given finite trace, the semantics of an LTL formula could be computed according to the semantics definition, it is important for practical applications, especially in online verification, to compute the semantics in an incremental, more precisely, in a left-to-right fashion for the given trace.

(weak) next

$$[u \models X\varphi]_4 = \begin{cases} [u^1 \models \varphi]_4 & \text{if } u^1 \neq \epsilon \\ \bot^p & \text{otherwise} \end{cases} \qquad [u \models \bar{X}\varphi]_4 = \begin{cases} [u^1 \models \varphi]_4 & \text{if } u^1 \neq \epsilon \\ \top^p & \text{otherwise} \end{cases}$$

Fig. 4. Semantics of FLTL$_4$ formulae over a trace $u = a_0 \ldots a_{n-1} \in \Sigma^*$

To do so, we provide a *rewriting* based approach (see also [11]). Thanks to the equivalences $\varphi\ U\ \psi \equiv \psi \vee (\varphi \wedge X(\varphi\ U\ \psi))$ and $\varphi\ R\ \psi \equiv \psi \wedge (\varphi \vee \bar{X}(\varphi\ R\ \psi))$ for until and release, we may always assume that the given formula is a boolean combination of atomic propositions and next-state formulas. Now, given a single, presumably final letter of a trace, the atomic propositions may be evaluated as to whether the letter satisfies the proposition. Each (strong) next-formula, i.e., a formula starting with a strong next, evaluates to *possibly false*, while each weak-next formula evaluates to *possibly true*. The truth value of the formula is then the boolean combination of the respective truth values, reading $\wedge$ as $\sqcap$, $\vee$ as $\sqcup$, and $\neg$ as $\bar{\ }$. Likewise, the formula to check may be rewritten towards a formula to be checked when the next letter is available. An atomic proposition is evaluated as before yielding the formulas *true* or *false*. A formula of the form $X\varphi$ or $\bar{X}\varphi$ is rewritten to φ. In Algorithm 1, a corresponding function is described in pseudo code, yielding for the formula to check and a single letter a tuple consisting of the current truth value in the first component and the formula to check with the next letter in the second component.

The same algorithm may also be used for evaluating the (two-valued) semantics of an FLTL formula in a left-to-right fashion, by mapping *possibly true* to *true* and *possibly false* to *false*, when reaching the end of the word.

4.5 LTL on Infinitive Traces

LTL formulae over infinite traces are interpreted as usual over the two valued truth domain $\mathbb{B}_2$.

Definition 7 (Semantics of LTL [7]). *The semantics of LTL formulae over infinite traces $w = a_0a_1 \ldots \in \Sigma^\omega$ is given by the function $[\_ \models \_]_\omega : \Sigma^\omega \times LTL \to \mathbb{B}_2$, which is defined inductively as shown in Figures 2,5.*

Inspecting the semantics, we observe that there is no difference of X and $\bar{X}$ in LTL over infinite traces. Recall that $\bar{X}$ acts differently when finite words are considered.

We call $w \in \Sigma^\omega$ a *model* of φ iff $[w \models \varphi] = \top$. For every LTL formula φ, its set of models, denoted with $\mathcal{L}(\varphi)$, is a regular set of infinite traces which is accepted by a corresponding Büchi automaton [12,13].

Algorithm 1. Evaluating FLTL4 for each subsequent letter

```
evalFLTL4 $true$    a = ($\top$,$\top$)
evalFLTL4 $false$   a = ($\bot$,$\bot$)
evalFLTL4 p         a = ((p in a),(p in a))
evalFLTL4 $\neg\varphi$        a = let (valPhi,phiRew) = evalFLTL4 $\varphi$ a
                        in ($\overline{\text{valPhi}}$,$\neg$phiRew)
evalFLTL4 $\varphi \vee \psi$    a = let
                          (valPhi,phiRew) = evalFLTL4 $\varphi$ a
                          (valPsi,psiRew) = evalFLTL4 $\psi$ a
                        in (valPhi $\sqcup$ valPsi,phiRew $\vee$ psiRew)
evalFLTL4 $\varphi \wedge \psi$    a = let
                          (valPhi,phiRew) = evalFLTL4 $\varphi$ a
                          (valPsi,psiRew) = evalFLTL4 $\psi$ a
                        in (valPhi $\sqcap$ valPsi,phiRew $\wedge$ psiRew)
evalFLTL4 $\varphi\ U\ \psi$   a = evalFLTL4 $\psi \vee (\varphi \wedge X(\varphi\ U\ \psi))$ a
evalFLTL4 $\varphi\ R\ \psi$   a = evalFLTL4 $\psi \wedge (\varphi \vee \bar{X}(\varphi\ R\ \psi))$ a
evalFLTL4 $X\varphi$       a = ($\bot^p$,$\varphi$)
evalFLTL4 $\bar{X}\varphi$       a = ($\top^p$,$\varphi$)
```

(weak) next

$$[w \models X\varphi]_\omega = [w^1 \models \varphi]_\omega$$
$$[w \models \bar{X}\varphi]_\omega = [w^1 \models \varphi]_\omega$$

Fig. 5. Semantics of LTL formulae over an infinite traces $w = a_0 a_1 \ldots \in \Sigma^\omega$

LTL$_3$ In[14], we proposed LTL$_3$ as an LTL logic with a semantics for finite traces, which follows the idea that a finite trace is a prefix of a so-far unknown infinite trace. More specifically, LTL$_3$ uses the standard syntax of LTL as defined in Definition 4 but employs a semantics function $[u \models \varphi]_3$ which evaluates each formula φ and each finite trace u of length n to one of the truth values in $\mathbb{B}_3 = \{\top, \bot, ?\}$. $\mathbb{B}_3 = \{\top, \bot, ?\}$ is defined as a de Morgan lattice with $\bot \sqsubset\ ?\ \sqsubset \top$, and with $\bot$ and $\top$ being complementary to each other while ? being complementary to itself.

The idea of the semantics for LTL$_3$ is as follows: If every infinite trace with prefix u evaluates to the same truth value $\top$ or $\bot$, then $[u \models \varphi]_3$ also evaluates to this truth value. Otherwise $[u \models \varphi]_3$ evaluates to ?, i. e., we have $[u \models \varphi]_3 =?$ if different continuations of u yield different truth values. This leads to the following definition:

Definition 8 (Semantics of LTL$_3$). *Let $u = a_0 \ldots a_{n-1} \in \Sigma^*$ denote a finite trace of length n. The* truth value *of a LTL$_3$ formula φ wrt. u, denoted with $[u \models \varphi]_3$, is an element of $\mathbb{B}_3$ and defined as follows:*

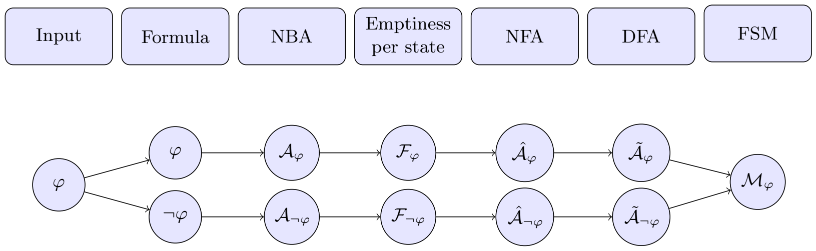
Input
Formula
NBA
Emptiness per state
NFA
DFA
FSM
φ
$\neg\varphi$
$\mathcal{A}_\varphi$
$\mathcal{A}_{\neg\varphi}$
$\mathcal{F}_\varphi$
$\mathcal{F}_{\neg\varphi}$
$\hat{\mathcal{A}}_\varphi$
$\hat{\mathcal{A}}_{\neg\varphi}$
$\tilde{\mathcal{A}}_\varphi$
$\tilde{\mathcal{A}}_{\neg\varphi}$
$\mathcal{M}_\varphi$

Fig. 6. The procedure for getting $[u \models \varphi]$ for a given φ

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

Monitoring of LTL$_3$. We briefly sketch the monitor synthesis procedure developed in [15]. The synthesis procedure follows the automata-based approach in synthesizing a Moore machine as a monitor for a given correctness property. While, in general, also a Moore machine could have been generated for FLTL as well, we refrained to do so for two reasons: First, the presented procedure for FLTL works *on-the-fly* and thus might be more efficient in practice. Second, both a rewriting approach and an automaton-based approach should be presented in the underlying course.

For LTL$_3$, an automaton approach is more adequate due to the fact that LTL$_3$ is *anticipatory*. Anticipation typically requires rewrite steps followed by a further analysis easily done using automata theory. See [16] for a more elaborate discussion of these issues in the context of linear temporal logic.

The synthesis procedure for LTL$_3$ first translates a given formula into the Büchi automaton accepting all its models. Reading a finite prefix of a run, using the corresponding automaton, *false* can be derived for the given formula, whenever there is no accepting continuation in the respective Büchi automaton. Likewise, *true* can be derived, when, for a given finite word, the automaton accepting all counter examples reaches only states the have no accepting continuation anymore. Using this idea, the corresponding Büchi automata can be translated into NFA, then DFA, and, finally into a (minimal) FSM (Moore machine) as *the* (unique) monitor for a given LTL$_3$ formula (see Figure 6).

While the sketched procedure should be improved in practical implementations, the chosen approach manifests itself beneficial for teaching, as a simple, clear, roadmap is followed.

4.6 Extensions

The studied versions of LTL were chosen to show certain aspects of monitoring executions. For practical applications, several extensions such as real-time aspects or monitoring computations, which requires a meaningful treatment of data values, is essential. Due to time constraints, these topics have not been adressed in the underlying course, though research results and corresponding RV frameworks are available (see also Section 6).

5 Monitors and the Behavior to Check

This part of the tutorial deals with the problem of integrating monitors into SUS and with techniques to steer the executing system by means of the results of monitors. This aspect of runtime verification was discussed only briefly in the corresponding runtime verification course. The main goal was to give a general overview of approaches for connecting monitors to existing systems.

Monitoring systems. Generally, we distinguish using *instrumentation*, using *logging APIs*, using *trace tools*, or dedicated *tracing hardware*. Popular in runtime verification is the use of code instrumentation, for which either the *source code*, the (virtual) *byte code*, or the *binary code* of an application is enriched by the synthesized monitoring code. Code instrumentation allows a tight integration with the running system and is especially useful when monitoring online. However, code instrumentation affects the runtime of the original system. It is thus not recommend whenever the underlying systems has been verified to meet certain safety critical timing behavior. Using standard logging frameworks, like *log4j*, allows to decompose the issue of logging information of the running system from the issue of analyzing the logged information with respect to failures. In principal, the logged information may be stored and analyzed later, or, using additional computing ressources online, thus not affecting system's execution. Logging APIs, however, require the source code of the SUS. Tracing tools like Unix' *strace* run the system under scrutiny in a dedicated fashion an provide logging information. Again, the timing behavior of the system may be influenced. The advantage of such tracing tools lies in their general applicability, the disadvantage in their restricted logging information. Finally, dedicated tracing hardware may be used to monitor a system *non-invasively* [17].

Steering systems. Whenever a monitor reports a failure, one might be interested in responding to the failure, perhaps even healing the failure. Clearly, this goal is only meaningful in an online monitoring approach. We distinguish the following forms of responding to failures: *informing*, where only further information is presented to a tester or user of a system, *throwing exceptions* for systems that can deal with exceptions, or *executing code*, which may be user provided or synthesized automatically. The latter approach is well supported by frameworks using code instrumentation as healing code may easily be provided together with the monitoring property.

Runtime verification frameworks may differ in their understanding of failures. Most frameworks identify failure and fault. Then, whenever a monitor reports a failure and thus a fault, healing code may be executed to deal with the fault.

When distinguishing between failures and faults, it may be beneficial to start a *diagnosis* identifying a fault, whenever a monitor reports a failure (see [18]).

6 Existing Frameworks

In the third part we will visit existing runtime verification frame works and map the approaches to the initially developed taxonomy. Due to the limited space of the proceedings, we only list the considered frameworks in an alphabetical order: (i) Eagle [19] (ii) J-LO [20] (iii) Larva [21] (iv) LogScope [22] (v) LoLa [23] (vi) MAC [24] (vii) MOP [25] (viii) RulerR [26] (ix) Temporal Rover [10] (x) TraceContract [27] (xi) Tracesmatches [28] .

Acknowledgement. We thank Klaus Havelund for many fruitful discussions on the topic of this paper.

References

1. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on Software Engineering (TSE) 30(12), 859–872 (2004)
2. IEEE Std 1012 - 2004 IEEE standard for software verificiation and validation. IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998), 1–110 (2005)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. An EATCS Series. Springer, Heidelberg (2004)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
5. Myers, G.J., Badgett, T., Thomas, T.M., Sandler, C.: The Art of Software Testing, 2nd edn. John Wiley and Sons (2004)
6. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 391–438. Springer, Heidelberg (2005)
7. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77), October 31-November 2, pp. 46–57. IEEE Computer Society Press, Providence (1977)
8. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
9. Kamp, H.W.: Tense Logic and the Theory of Linear Order. PhD thesis, University of California, Los Angeles (1968)
10. Drusinsky, D.: The Temporal Rover and the Atg Rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000)
11. Geilen, M.: On the construction of monitors for temporal logic properties. Electronic Notes on Theoretical Computer Science (ENTCS) 55(2) (2001)
12. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Symposium on Logic in Computer Science (LICS 1986), pp. 332–345. IEEE Computer Society Press, Washington, D.C (1986)

13. Vardi, M.Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
14. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of Real-Time Properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006)
15. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (TOSEM) 20(4) (July 2011) (in press)
16. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation 20(3), 651–674 (2010)
17. Hochberger, C., Leucker, M., Weiss, A., Backasch, R.: A generic hardware architecture for runtime verification. In: ECSI (ed.) Proceedings of the 2010 Conference on System, Software, SoC and Silicon Debug, pp. 79–84 (2010)
18. 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)
19. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
20. Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: Fifth Workshop on Runtime Verification (RV 2005). ENTCS, Elsevier (2005)
21. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs (tool paper). In: Hung, D.V., Krishnan, P. (eds.) SEFM, pp. 33–37. IEEE Computer Society (2009)
22. Barringer, H., Groce, A., Havelund, K., Smith, M.H.: An entry point for formal methods: Specification and analysis of event logs. In: Bujorianu, M.L., Fisher, M. (eds.) FMA. EPTCS, vol. 20, pp. 16–21 (2009)
23. 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: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), pp. 166–174 (2005)
24. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design 24(2), 129–155 (2004)
25. Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 2007 (2007)
26. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruler. J. Log. Comput. 20(3), 675–706 (2010)
27. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for Trace Analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
28. Avgustinov, P., Tibble, J., Bodden, E., Hendren, L.J., Lhoták, O., de Moor, O., Ongkingco, N., Sittampalam, G.: Efficient trace monitoring. In: Tarr, P.L., Cook, W.R. (eds.) OOPSLA Companion, pp. 685–686. ACM (2006)