Runtime verification on data-carrying traces

Jan-Christoph Küster

A thesis submitted for the degree of Doctor of Philosophy at The Australian National University

October 2016
Except where otherwise indicated, this thesis is my own original work.

Jan-Christoph Küster
17th October 2016
to my aunt, Susanne.
Acknowledgements

First and foremost, I would like to sincerely thank my main advisor Andreas Bauer, who convinced me not only to move to Australia and conduct research with him, but to whom I also could always reach out for deep and guiding discussions—even after he had left the country to work at TU Munich and later in a busy industry job. My thanks further go to Peter Baumgartner and Steve Blackburn, my two other advisors. I look back to helpful discussions with Peter and Steve, as well as Steve’s students and research group, who all provided valuable feedback at times.

Moreover, I am grateful to NICTA (by the time of writing already known as Data61) and the Australian National University (ANU), for taking me on as a postgraduate student and funding me all throughout my research journey. Special thanks are due to my colleagues at the NICTA lab in Canberra, Patrik Haslum, Karsten Lehmann, Michael Norrish, as well as Gil Vegliach, for the many brain-teasing discussions about some of the mathematical proofs contained in this thesis. I am impressed by you providing me with valuable help, even though the problems were mostly unrelated to your own research. There are also Josh Bax, Mohammad Abdulaziz, Paul Scott and Alban Grastien. I thank you for many inspiring conversations and making NICTA such a fun place to work.

Furthermore, I would like to point out Gernot Heiser and Aaron Carroll at NICTA in Sydney, for their help and guidance on the systems angle of my work. Mark Staples deserves special mention, who has been a valuable tutor, as he pointed me to various interesting cross-disciplinary research happenings across NICTA and introduced me to numerous people. Ralf Huuck, Franck Cassez, and the rest of the Goanna team should not be forgotten, who I had the pleasure to join in Sydney—unfortunately for a way too short time. Also, I thank Jan Michael Auer and Pablo González de Aledo for their curiosity and interest around my research in runtime verification. At ANU, I would like to mention Rajeev Goré, for his support beyond the PhD monitorings and, especially, Alwen Tiu, for some insightful discussion about first-order monitoring mostly via email after he has moved to Singapore.

I owe special thanks to several people that proofread my various paper drafts and this thesis: Alexa, Alban, Josh, Karsten, Luke, Mark, Peter, Ralf, and Wayne.

Another great source of support is the Level Up Analytics Team, namely Tobias Ullrich, Thorsten Krüger, Lucian Vlad Lita, Jonathan Goldman, and Anu Tewary, who provided me with the opportunity to work with them part-time and gave me this way the necessary distraction from my daily research duties.

Last but not least, I thank all my family and friends back home, who I seldom had the chance to visit in the last few years and might have felt neglected, but who have not forgotten me during my time in Australia.
Abstract

Malfunctioning software systems can cause severe loss of money, sensitive data, or even human life. The ambition is therefore to verify these systems not only statically, but also monitor their behaviour at runtime. For the latter case, the temporal logic LTL—a de facto standard specification formalism in runtime verification—is widely used and well-understood. However, propositional variables are usually not a natural nor sufficient model to represent the behaviour of complex, interactive systems that can process arbitrary input values. Consequently, there is a demand for more expressive formalisms that are defined wrt. what we call traces with data, i.e., traces that contain propositions enriched with values from a (possibly) infinite domain.

This thesis studies the runtime monitoring with data for a natural extension of LTL that includes first-order quantification, called LTL$^{FO}$. The logic’s quantifiers range over values that appear in a trace. Under assumptions laid out of what should arguably be considered a “proper” runtime monitor, this thesis first identifies and analyses the underlying decision problems of monitoring properties in LTL and LTL$^{FO}$. Moreover, it proposes a monitoring procedure for the latter. A result is that LTL$^{FO}$ is undecidable, and the prefix problem too, which an online monitor has to preferably solve to coincide with monotonicity. Hence, the obtained monitor cannot be complete for LTL$^{FO}$; however, this thesis proves the soundness of its construction and gives experimental results from an implementation, in order to justify its usefulness and efficiency in practice. The monitor is based on a new type of automaton, called spawning automaton; it helps to efficiently decide what parts of a possibly infinite state space need to be memorised at runtime. Furthermore, the problem occurs that not every property can be monitored trace-length independently, which is possible in LTL. For that reason, a hierarchy of effectively monitorable properties is proposed. It distinguishes properties for which a monitor requires only constant memory from ones for which a monitor inevitably has to grow ad infinitum, independently of how the future of a trace evolves.

Last but not least, a proof of concept validates the monitoring means developed in this thesis on a widely established system with intensive data use: Malicious behaviour is checked on Android devices based on the most comprehensive malware set presently available. The overall detection and false positive rates are 93.9% and 28%, respectively. As a means of conducting the experiments and as a contribution in itself, an application-agnostic logging-layer for the Android system has been developed and its technical insights are explained. It aims at leveraging runtime verification techniques on Android, like other domain-specific instrumentation approaches did, such as AspectJ for Java.
## Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 An overview of the runtime verification process</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Detailed problem statement</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Monitoring beyond propositional expressiveness</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Desired properties of a monitor</td>
<td>6</td>
</tr>
<tr>
<td>1.2.3 System instrumentation as a domain-specific task</td>
<td>8</td>
</tr>
<tr>
<td>1.2.4 The benefit of runtime verification on mobile platforms</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Contributions of this thesis</td>
<td>11</td>
</tr>
<tr>
<td>1.3.1 Publications</td>
<td>13</td>
</tr>
<tr>
<td>1.3.2 Practical realisations</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Results of this thesis</td>
<td>14</td>
</tr>
<tr>
<td>1.5 Structure of this thesis</td>
<td>15</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td>17</td>
</tr>
<tr>
<td>2.1 Words over alphabets</td>
<td>17</td>
</tr>
<tr>
<td>2.2 Representation of system behaviour</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Automata as acceptors of finite or infinite words</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Propositional linear-time temporal logic</td>
<td>22</td>
</tr>
<tr>
<td>2.5 LTL semantics for runtime verification</td>
<td>26</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>29</td>
</tr>
<tr>
<td>3 Logic and complexity</td>
<td>31</td>
</tr>
<tr>
<td>3.1 LTL\textsuperscript{FO}—Formal definitions and notation</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Complexity of monitoring</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 Propositional case</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 First-order case</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Summary</td>
<td>40</td>
</tr>
<tr>
<td>4 Monitoring algorithm</td>
<td>43</td>
</tr>
<tr>
<td>4.1 Spawning automaton</td>
<td>43</td>
</tr>
<tr>
<td>4.1.1 General definition</td>
<td>43</td>
</tr>
<tr>
<td>4.1.2 A construction for LTL\textsuperscript{FO}-formulae</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Monitor construction</td>
<td>48</td>
</tr>
</tbody>
</table>
# Contents

4.3 Optimisations ........................................ 52  
4.4 Experiments ....................................... 53  
  4.4.1 SA-based monitoring vs. formula progression ... 54  
  4.4.2 SA-based monitoring vs. optimisations ......... 57  
4.5 Related monitoring approaches ................... 58  
4.6 Summary ........................................... 68  

5 Towards a hierarchy of effectively monitorable languages 69  
  5.1 Monitorability ..................................... 69  
  5.2 Trace-length independence ....................... 70  
  5.3 Strong trace-length dependence .................. 71  
  5.4 Summary ......................................... 73  

6 Proof of concept: Android malware detection 75  
  6.1 Android security concepts in a nutshell .......... 76  
  6.2 A monitoring tool for the Android platform .... 77  
  6.2.1 System call interception ....................... 78  
  6.2.2 Unmarshalling .................................. 80  
  6.2.3 Netlink communication ......................... 83  
  6.3 Specifying malware behaviour ................... 83  
  6.3.1 System model of app behaviour ............... 83  
  6.3.2 A user manual: How to specify malware behaviour in LTL* ...... 85  
  6.3.3 Common, recurring malware patterns .......... 87  
  6.4 Identifying malware behaviour .................. 91  
  6.4.1 Experimental results ......................... 92  
  6.4.2 False positives analysis ...................... 93  
  6.4.3 Performance and portability ................. 95  
  6.5 Related work ................................... 98  
  6.6 Summary ...................................... 100  

7 Conclusions ........................................ 101  
  7.1 Future research .................................. 101  
  7.2 Open problems .................................. 102  

A Detailed proofs ................................... 123  
  A.1 Complexity results .............................. 123  
  A.2 Spawning automaton .............................. 129  
  A.3 Monitor construction ............................ 133  

B Proof of concept: Additional experiment data .... 135  
  B.1 Information stealing ............................. 137  
  B.2 False positives ................................. 138  
  B.3 Intercepted interface names ...................... 140
In today's society, software systems grow inevitably in scale and functionality while at the same time performing increasingly critical work. For example, in cars, which evolve towards self-driving [Halleck, 2015], a growing range of complex control tasks is handled by software whose malfunction can cause severe injury or even death to human beings [cf. Knight, 2002]. Not long ago, many consumer devices, such as mobile phones, involved almost no software. Nowadays, they are powerful computing devices, connect to the internet, and contain several built-in sensors dedicated to tracking a user's location, taking photos or scanning fingerprints. These innovations in hardware open up the door to many useful software applications, such as mobile payments, or location based services. However, unauthorised access to sensitive data (e.g., through exploits of a phone's operating system [cf. Davi et al., 2010]) can cause the loss of valuable assets, including real money or users' identity [cf. Dagon et al., 2004]. The literature usually distinguishes systems as being safety- or security critical; safety refers to the inability of the system to have an undesirable effect on its environment, and security, vice versa, to the inability of the environment to have an undesirable effect on the system [Line et al., 2006].

Due to the increased complexity and growing interrelation of security and safety, it is a main goal in software and systems engineering to improve on verification techniques for such critical systems; verification, according to IEEE [2005a], comprises all techniques suitable for showing that a system satisfies its specification. Ideally verification is achieved statically, i.e., without executing a program (e.g., based on its source code alone). Traditional static verification techniques include model checking [cf. Clarke et al., 2001; Baier and Katoen, 2008], theorem proving [cf. Bertot and Castéran, 2004], and testing [cf. Myers et al., 2012]. Despite their great advantages to program correctness, these techniques have shortcomings: Model checking faces the state explosion problem [cf. Clarke, 2009], so that for very large systems the computational power or memory needed is beyond what is practically feasible today. On the other hand, theorem proving does not scale well as it typically requires manual effort [Kaufmann and Moore, 2004]. Moreover, there is no hope to statically verify programs or systems comprehensively, as this is in general an undecidable problem.

1 Testing can be dynamic and static. In the latter case, it includes program inspections or code walkthroughs [Myers et al., 2012 §3], which are usually not computer-based.
Introduction

Turing [1936]. Even if all errors could be eliminated prior to execution, the underlying hardware might behave in an unexpected manner, or assumptions made about the execution environment prove themselves inadequate or incomplete in practice.

Hence, the non-occurrence of problems at runtime cannot be guaranteed, so that the combination of static and dynamic analysis has become the norm and often even mandatory for critical systems [cf. Bowen and Stavridou 1993]. Generally speaking, dynamic analysis approaches check single execution paths; thus, they are by definition incomplete, as not all errors can be detected (false negatives). However, their advantage is the access to “real” execution data. This can drastically limit false positives and, therefore, complement static analysis techniques, as false positives often occur due to working with an abstraction of the system [cf. Chen et al. 2004]. Another advantage is that dynamic analysis can be applied to “black boxes” (i.e., without having access to the source code). In this case static verification becomes impossible.

Dynamic techniques can be used both for testing before deployment of the software, in which case test case runs are checked to uncover a bug [Artho et al. 2003], and after—to continuously monitor a running system in production. This thesis deals with studying challenges and formal approaches to the latter.

1.1 An overview of the runtime verification process

This section provides a brief overview of the research discipline and the major steps involved in the process to formally and continuously monitor a system.

Leucker and Schallhart [2009] define runtime verification as the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether an execution of a system under scrutiny satisfies or violates a given correctness property. Thus, the inputs to a runtime verification framework are: (1) a system to be checked, and (2) a set of correctness properties that is checked against the execution of the system. Runtime verification is nowadays commonly associated with only the detection of a property’s satisfaction or violation (i.e., only passively observes a running system), unlike runtime enforcement [cf. Falcone et al. 2012b; Schneider 2000], which additionally studies the “repairing” (i.e., delaying, stopping or changing) of executions. Runtime enforcement is out of scope of this thesis. The runtime verification process typically involves four aspects, whose relation is depicted in Fig. 1.1, and which are introduced in the following [cf. Bauer 2007, Falcone et al. 2013]. The first two aspects are concerned with checking system traces, i.e., designing adequate specification languages to formulate properties, and building algorithms to check those (from top left to the middle). The last two aspects (from bottom to top right) are concerned with the practical side of generating the traces and running the analysis, i.e., extracting the necessary observations from a system, and finally, forwarding them to monitors as well as interpreting their results.
§1.1 An overview of the runtime verification process

Figure 1.1: An overview of the runtime verification process.

Property specification. A set of correctness properties is usually expressed in a formal specification language (e.g., a logic-based formalism with mathematically defined syntax and semantics) that suits for unambiguous description and reasoning, so that the latter can be performed by a computing device. As writing specifications is even a difficult and error prone task for experts, one aims at the same time for conciseness and intuitive understanding of the specification language. For this purpose, there exists work on extensions, which do not make a language more powerful but improve its usability [cf. Bauer and Leucker 2011; IEEE 2005b].

Depending on what needs to be checked, the requirements for the specification formalism vary. However, their diversity shows that, not surprisingly, there exists no “silver bullet” formalism and associated monitoring technique yet [cf. Falcone et al. 2013]. Formalisms adequate for runtime verification usually support reasoning about sequences of temporally separated program states (and not just assertions about a single program state). This means they include some kind of temporal quantification, e.g., in form of temporal operators. These can refer to points in time, so that properties such as “whenever a user logs into a system, she eventually logs out” can be expressed. Some formalisms are called untimed, i.e., only allow referring to time in terms of a state ordering relationships [cf. Pnueli 1977], while others go further and allow stating relative or absolute time values [cf. Raskin 1999]. An often additionally required language feature is called data quantification. It allows binding and referral (forward or backward in time) to values across states [Havelund and Goldberg 2005].

This thesis deals in particular with a natural extension of LTL that allows data quantification, called LTL[0] (see §3.1).
Monitor generation. From each property a so-called monitor is generated, i.e., a decision procedure for the property. It takes representations of system executions as inputs. The monitor’s task then is to yield a verdict that tells if the property has been satisfied (or violated) by the observed system behaviour. The monitor usually informs the user after each received event, while taking into account the history of all events received so far. Monitor generation is usually automated, i.e., no user interaction is necessary. It is a challenging trade off between designing the expressiveness of a language and constructing effective as well as efficient monitors that can check every property specifiable in it.

System instrumentation. The purpose of this stage is to generate from native occurring system events—to give some examples, these can be performed operations, function calls, acquired locks, or assignments to variables—so-called actions that can be understood and processed by a monitor for analysis. As it has to be assured that all observations necessary for checking a property can be technically retrieved, this stage depends on the ones above. In other words, the stream of captured events is the sole representation of a system’s execution on which the analysis is based; hence, it is crucial to gather all relevant actions, as the consequence of missed observations likely result in incorrect outcome of monitor verdicts. On the other hand, capturing irrelevant observations causes unnecessary performance overhead to the system environment. This stage can also influence the stages above if one has to deal with a given, fixed set of possible observations, and one needs to find under this assumption an adequate formalism for reasoning.

Execution analysis. Finally, the system is executed and generates events that are consumed by the monitor(s) whose outcome is interpreted and from which according actions are to be derived. Overhead is induced by the system instrumentation, as well as the monitors. The latter might run internally (i.e., on the same execution environment as the system), where they influence its efficiency, or externally, where performance is probably less crucial. In synchronous monitoring, whenever a relevant observation is produced by the system, further execution is stopped until the checker confirms that no violation has occurred. However, synchronous monitoring is only necessary if reconfiguration, repairing or giving feedback to the system based on the monitor’s results is desired and time-critical.

1.2 Detailed problem statement

Runtime verification for propositional temporal specification formalisms, such as LTL has been widely studied since the early 2000s and is nowadays well-understood.

---

2In the literature, the term action and event are often used interchangeably. In this thesis we will use the term system event, to make clear whenever we mean a native event, action for their formal representation, and simply event to denote a set of actions, which represents all necessary information about a system state at a time, for example.
A key result of monitoring LTL properties is that the prefix problem (see Def. 3.2.3) can be solved by a decision procedure in form of a Moore machine [Bauer et al., 2011] that is optimal wrt. space complexity (i.e., is the unique solution with the fewest states).

However, while LTL is sufficient to monitor embedded systems, there is a growing demand to monitor interactive and structural more complex, high-level systems that process arbitrary input values. A use case is to check the latter, for instance, against compliance and security regulations [cf. Basin et al., 2014]. Consequently, research has shifted towards formalisms with an adequate expressiveness. These are defined over what we call in the following traces with data, i.e., traces containing actions enriched with values from a (possibly) infinite domain. In this context, dealing with decidable monitoring problems, building finite state-space algorithms, or finding an optimal monitor is generally not—or only under tight restrictions—possible.

### 1.2.1 Monitoring beyond propositional expressiveness

When using propositional runtime verification, one assumes that information about a system’s state is representable in form of propositional variables; or more precisely conjunctions of propositional literals. There exist various practically useful properties that are expressible in this kind of formalism. Consider, for example, the dining philosopher problem introduced by Dijkstra [1971], which illustrates synchronisation issues of concurrent systems: having \( n \) philosophers and \( n \) chopsticks, each philosopher can be in a state of waiting for up to two chopsticks (represented by proposition \( \text{wait}_i \)), eating (if occupying two chopsticks), or releasing both chopsticks after finishing eating. An occupied chopstick is represented by \( \text{occupied}_i \). The property

\[
G \neg \left( \bigwedge_{0 \leq i < n} \text{wait}_i \land \bigwedge_{0 \leq i < n} \text{occupied}_i \right),
\]

written here in LTL (see § 2.4 for its formal syntax and semantics), describes deadlock freedom, i.e., it should be never the case that all philosophers are waiting while all chopsticks are occupied (as this means exactly one chopstick is occupied by each philosopher, so that no one can proceed).

However, if a system’s internal state structure is complex, or it processes a priori unknown input values, propositional logic might not be an adequate formalism for verification. In the best case, properties are just very long and inefficient to write and monitor. Imagine in the example above having thousands of philosophers (or processes, as an equivalent in a software system). In this case the property would be still expressible as long as the number of philosophers is known at the time of specification. However, in other cases, arbitrary values from an infinite domain can occur (such as the integers or strings), which cannot be stated as a propositional property of finite length. Consider for example a web server, for which we do not know the number of expected users logged in at any given time. The property

\[
G \forall x : \text{login}. X(\neg \text{login}(x)U \text{logout}(x))
\]
overcomes this problem, as it is able to reason over traces with data. A trace contains instead of propositions login and logout, which are true (i.e., appear in the trace) if a login happens, ground atoms such as login(d) and logout(d′), where d, d′ ∈ N₀ represent user IDs. The quantifier in the formula above then binds the values from ground atoms of each event to the variable x for which the formula X(¬login(x)Ulogout(x)) must hold, respectively. This means a user identified by the natural number 3, for example, should not log in twice, unless she logs out (i.e., logout(3) appears in the trace) in between. For the formal syntax and semantics of this property, specified here in LTL(position), see §3.1.

There exist a wide range of different approaches that are capable of monitoring traces with data, but these vary in expressiveness and efficiency. Some allow explicit, but often only implicit or restricted quantification, can only monitor a syntactical safety fragment, or do not allow for arbitrary computable predicates or functions as we shall see when discussing related work in §4.5. One of the most efficient approaches is JavaMOP, which can solve the login-logout-problem above via so-called trace slicing [Chen and Rosu, 2009]. However, we will see that many other properties cannot be monitored by it. Furthermore, some works do not agree with the properties an online monitor should arguably have, and which are described in detail in the next section.

1.2.2 Desired properties of a monitor

As there exist many ways in which a system can be monitored in the abstract sense described in §1.1, we are going to put forth six specific assumptions concerning the properties and inner workings of what is considered a “proper” monitor.

Property 1 (Online monitoring): When monitoring reactive systems, such as operating systems, web servers or mobile phones, the assumption is that those ideally do not terminate. In this case, we need to monitor online, i.e., process executions from a system while it is still evolving—knowing that there is always eventually a next event. In other words, a monitor can not wait until the system has finished executing to process observations, and therefore incrementally consumes events in a step-by-step manner—whenever a next, new observation occurs. It is common practice that the monitor yields a verdict after every observation; that is, to inform the user as early as possible about the satisfaction (resp. violation) of the property.

On the other hand, when monitoring offline, one usually records events from a system under scrutiny for some fixed amount of time or until it terminates (e.g., in the case of a non-reactive system or algorithm, such as a sorting function), stores them, for example, in a log file, and applies a monitor to analyse the sequence of executions post hoc—as a whole. In this case, we say the sequence of observed events is complete, i.e., no further events of the system are expected, and no further knowledge about the system executing can be gained. In offline monitoring, it is sufficient for a monitor to yield a verdict at the end of the trace instead of providing intermediate results.
In this thesis, we require a monitor of being capable to perform at least online monitoring, which we will see is the much harder problem to solve.

**Property 2 (Trace-length independence):** Our second property states that an online monitor should not try to store an ever growing trace, or otherwise the monitor’s efficiency will inevitable decline with an increasing number of observations. In other words, ideally, the amount of information that needs to be stored by a monitor as its inner state should depend only upon the property to be monitored and not upon the number of already processed events. While many monitors in propositional runtime verification fulfil this property, we shall see that this is generally not possible when monitoring traces with data. More precisely, trace-length independence becomes a property both of formulae and monitors, where the later should be trace-length independent if likewise the formulae being monitored allow it.

In a taxonomy of monitoring properties introduced by Rosu and Havelund [2005], a monitor violating this property was named trace storing, or non-storing if being trace-length independent.

**Property 3 (Monotonicity):** The property called monotonicity of entailment in a logical system such as classical first-order logic, means that if a sentence \( \varphi \) can be inferred from a set of premises \( \Gamma \), then it can also be inferred from any set \( \Delta \) of premises containing \( \Gamma \) as a subset. In other words, additional knowledge should never change the truth value of already inferred results. Kleene [1952] introduced monotonicity (back then called regularity) in the context of dealing with undefined (i.e., unknown) truth values, when extending Boolean logic to a 3-valued logic with truth value “inconclusive”. He stated that the value of any formula should never change from true to false or from false to true, though a change from “inconclusive” to one of false or true is allowed—complying this way with the meaning of true and false in the 2-valued Boolean logic.

Similarly, since observed behaviour at runtime can only ever be a finite prefix of an ideally infinite behaviour, a consequence from online monitoring is that knowledge (i.e., a trace) is always incomplete albeit extended with each new observation. Hence, with the desire in runtime verification to avoid misunderstanding wrt. the meaning of true and false in classical logics, we require that these truth-value’s semantics ultimately should be preserved; that is, the satisfaction (in the case of a monitor returning true) or violation (in the case of a monitor returning false) of a property independent of more knowledge gained—implying also that a monitoring device can stop observing in those cases. In other words, we demand a monitor in this thesis to be monotonic wrt. reporting verdicts for a specification, meaning that once the monitor returns “SAT” to the user, additional observations do not lead to it returning “UNSAT” (and vice versa).

**Property 4 (Impartiality):** Refining the idea of monotonicity, Bauer et al. [2007] and Dong et al. [2008] have stated the principle of impartiality, which requires that a finite trace is not evaluated to true or false if there still exists an (infinite) continuation leading to another verdict. Obviously, this principle implies online monitoring, as
there are no continuations when working with complete traces. This principle was
dated having a concrete semantics for runtime verification in mind, namely LTL_3,
which we will investigate further in §2.5.

**Property 5 (Anticipation):** Generally speaking, this principle desires a monitor to
return a conclusive verdict as early as possible, so that the user can react timely on
a system fault, for example. However, the principle of anticipation was articulated
having the detection of good and bad prefixes in mind, thus requires that once every
(infinite) continuation of a finite trace leads to the same verdict, then the finite trace
evaluates to this verdict [Dong et al. 2008]. Note that requesting the principle of
impartiality alone would allow for a trivial monitor; that is, one that always yields the
verdict “unknown”. But this monitor would not be anticipatory, so both principles
are required together.

Note as well that historically, in a taxonomy of monitor properties proposed by
Rosu and Havelund [2005], this property was called synchronous monitoring. However,
it did not distinguish between an offline monitor, which generally cannot detect a
violation in a timely manner, and an online monitor, which detects violations several
but finitely many steps later than those occurred.

**Property 6 (Minimal or low runtime overhead):** We understand monitoring or run-
time overhead as the time (or number of computation steps) that it takes the monitor to
yield a verdict, starting from reading the current event as input until being ready to
process the next event. However, the verdict should not depend on the current event
alone, but must obviously take into account the history of all events received so far
(for which purpose the monitor keeps an inner state). For the case of LTL properties,
Bauer et al. [2006, 2011] have proposed a monitor with constant runtime overhead,
but whose construction is double-exponential. However, a common assumption in
runtime verification is to neglect the cost of monitor generation ahead of time (un-
less, of course, the monitor for very long properties cannot practically be built), as
it is comparably low in relation to the runtime overhead that occurs over and over
again for each new event.

Note that this property is somewhat stricter than demanding a monitor to be
only trace-length independent, since minimal (or low) overhead implicitly claims
that a monitor is not allowed to process the whole trace after each new event.

1.2.3 System instrumentation as a domain-specific task

As we have pointed out in §1.1, runtime verification with data requires collecting
detailed information about a system executing. Therefore, developing techniques to
automatically instrument systems—with probes for event extraction—is a practical
and important problem the runtime verification community is concerned with; espe-
cially, since not every system comes with a built-in logging mechanism, or what is
logged cannot be configured flexibly after a program is compiled.

However, in contrast to work pursued on monitoring algorithms and specification
languages, instrumentation is a highly domain specific task that cannot be solved
without knowing what data is available to be intercepted, nor understanding how information flows between components of a system, so that events are not missed. Furthermore, it is necessary to take the characteristics of the targeted programming language or paradigm into account, which also becomes apparent when considering the various tools available: For Java, there exists for example AspectJ [Kiczales et al., 2001], and the AspectJ compiler, called ajc\footnote{http://www.eclipse.org/aspectj/doc/released/devguide/ajc-ref.html} which is used by many runtime verification frameworks to “weave” log statements into the source code or even bytecode of a program. For programs written in C++, there are approaches such as AspectC++\footnote{http://www.aspectc.org/} [Spinczyk and Lohmann, 2007], for C the tool Movec\footnote{http://svlab.nu.edu.cn/zchen/projects/movec/} [Chen et al., 2016b], or Arachne\footnote{http://web.emn.fr/x-info/arachne/index.html} [Douence et al., 2006], which is a dynamic “weaver” for binary code of C applications.

Challenges of instrumenting the Android system. The Android operating system (OS) is the fastest growing mobile operating system in the world (Fig. B.1), which has been under development since 2003. With 86% global market share (Fig. B.1), and in use not just on mobile phones or tablets, but also on televisions, or in cars, it is nowadays established world-wide; hence, there is arguably a need to have an instrumentation mechanisms for it, in order to undertake debugging\footnote{https://developer.android.com/studio/debug/index.html} or profiling \cite{Yoon2012,Wei2012} or security hardening \cite{Enck2010}, to name just some applications. Note that in this thesis we use the term Android system when we mean the Android OS and the software applications (called apps) running on top of it. In other words, the term Android system denotes here the complete software stack that runs on a mobile device.

Although apps are written mainly in Java, instrumenting them cannot be simply solved by applying AspectJ to them. First of all, apps running on the Android OS are usually developed by third-party vendors, so that their source code is not directly accessible and often heavily obfuscated \cite{Moser2007}. Even though placing probes in bytecode is possible, modifying apps would break the licence agreement and also their signature. Furthermore, one has to be concerned with dynamic loading of executable code via reflection \cite{Forman2004}, which is not present at instrumentation time, or C libraries that an app can make use of via the \textit{Java Native Interface} (JNI). Consequently, it might be preferred to consider apps as “black boxes” and capture events outside the apps—as part of the Android OS. Depending on the range of events to capture one might need to tie into the platform at a low level (e.g., the Linux kernel), where one is concerned with data formats that are not human-readable anymore; that is, the unknown semantics of bytestreams has to be decoded in some way. Although the Android platform stack is freely available\footnote{https://source.android.com/}, and arbitrary modifications to the Android OS are possible, users usually run a so-called...
Introduction

stock ROM (i.e., the initial firmware on a device provided by the manufacturer or mobile carrier), and do not want to be concerned with deploying their own version. Therefore, statically weaving and recompiling the Android Open Source Project (AOSP) is not a choice as it restricts usability.

Consequently, having an instrumentation approach for Android—a data-intensive and security-critical platform—is challenging but is expected to be of interest to the runtime verification community.

1.2.4 The benefit of runtime verification on mobile platforms

The dominance of the Android OS, and especially its security relevance due to managing a wide range of sensitive data—such as location information or two-step authentication tokens to access financial services—have made it a target of a steadily growing range of attacks, such as so-called spyware or banking trojans. In its Q1/2011 malware report, security firm Kaspersky remarks that “since 2007, the number of new antivirus database records for mobile malware has virtually doubled every year.”, and in August 2015 McAfee still recognised a growth of 17% quarterly (see Fig. B.2)—having captured over 8 million mobile malwares in total. Furthermore, in case of the Android platform, security firm McAfee asserts in its Q2/2011 threats report that, in fact, “Android OS-based malware became the most popular target for mobile malware developers.”

Besides traditional virus scanners tackling this problem, static analysis techniques have been increasingly employed, such as model checking and theorem proving, in order to perform malware detection on a semantic-based level [cf. Kinder et al., 2005; Preda et al., 2008]. While it has been shown that these systems are effective in identifying current malware, code obfuscation schemes [cf. Moser et al., 2007] and the fundamental limits in what can be decided statically, render static analysis techniques alone insufficient to detect malicious code. For this reason, automated dynamic analysis systems have been explored that can deal with the large amounts of malware and its diversity. However, those techniques on the other hand, which are often used as a form of testing (i.e., run on an emulator or other kind of artificial test environment for a short period of time), face difficulties with malware samples that employ recent emulator-detection techniques [Vidas and Christin, 2014]. Therefore, it is expected that online monitoring is a complementing option lending itself well for mobile security and helping to secure mobile systems running directly on end-users’ phones. The latter scenario had not been comprehensively explored before, i.e., no vast amount of real malware behaviour has been exhaustively monitored to support this hypothesis. Furthermore, questions arise whether it scales to effectively discover malicious behaviour by specifications and if detection and false positive rates are promising. Also, while runtime verification has been evaluated for the use of monitoring safety [cf. Brat et al., 2004; Pike et al., 2011] as well as security policies [cf. Basin et al., 2014], it is of interest to study what language features are suitable

for mobile platforms, for example, to facilitate reasoning over the various appearing forms of data to a great extent.

1.3 Contributions of this thesis

This thesis contributes both in a theoretical and practical way to the research field of runtime verification. Under the assumption of modelling system behaviour in form of traces with data, it investigates the computational complexity of online monitoring and provides an efficient runtime monitor for this setting that complies with properties stated in §1.2.2. Furthermore, it provides a useful instrumentation approach for a widely used platform with data—the Android system—and evaluates the application of runtime verification to Android in terms of effectiveness and efficiency for mobile security.

Contributions to the foundations of runtime verification. After first summarising some well-known computational complexity results for decision problems around monitoring LTL properties—such as the satisfiability, word, model checking and prefix problem—we look at this decision problems again, in comparison, for LTL$^{FO}$. In this new setting we will see that what we understand by solving the online monitoring problem, based on the desired properties of a monitor laid out in §1.2.2 is undecidable. Furthermore, additional problems arise that have been irrelevant or non-existent in the propositional case, such as determining whether a property is trace-length (in)dependent; that is, if the amount of information a monitor has to keep about the trace during monitoring has an upper bound. Note that in the propositional case only constant amount of data depending on trace needs to be stored \cite{Bauer}. For this purpose, we propose formal definitions of a monitoring hierarchy. It distinguishes property classes for which a monitor exists that has constant size, from properties for which it is unavoidable for any monitor to store a strictly monotonic growing amount of data (independent of how the trace evolves and which properties therefore have the inherent problem of being practically infeasible to monitor).

Contributions to monitor synthesis. The decision procedure developed in this thesis for LTL$^{FO}$ is neither restricted to implicit quantification (i.e., both universal and existential quantifiers can be arbitrarily nested) nor syntactical fragments (instead every formula definable in the language can in principle be monitored), allows using arbitrary computable predicates and functions, and furthermore agrees with the postulated, desired properties of a monitor by tackling the prefix problem (see Def. 3.2.3)—which we will see is computationally more involved than solving the easier and decidable word problem (see Def. 3.2.1). The monitor construction is based on a novel type of automaton, called a spawning automaton (SA), which is an acceptor for LTL$^{FO}$; that is, it only has an accepting run if the according trace is a model of LTL$^{FO}$. The automaton construction as well as the monitor construction is
proven to be sound albeit the latter being incomplete, as the prefix problem of $\text{LTL}^{\text{FO}}$ is undecidable. However, experiments show that this is more of a theoretical problem, as in practice a wide range of relevant properties can be efficiently monitored. The construction resembles a similar idea as in Bauer et al. [2011], but since an SA has an infinite state space, the approach for $\text{LTL}^{\text{FO}}$ is more complex. We can merely create its structure ahead of time serving as a pattern, for which at runtime necessary assignments are efficiently “book-kept” and “garbage collected”.

**Contributions to system instrumentation.** Chen et al. [2016b] argue that runtime verification for Java became popular especially due to the success and availability of the AspectJ compiler. Thus, inspired by AspectJ, they ported it for the C language, hoping that their instrumentation leverages runtime verification applications for this language, too. With a similar hypothesis in mind, this thesis proposes an application-agnostic logging library for the Android platform that overcomes certain problems in this field: It is modular in the sense that no Android system modifications are required, and portable, i.e., works for a wide range of different Android versions running on various hardware. This way it addresses the Android fragmentation problem [cf. Zhou et al., 2014]. It has been successfully tested on devices of the Google Nexus family (Nexus S, 7 and 5) and for early as well as recent Android versions (2.3.6, 4.3 and 5.0.1).

Compared to other approaches, which are usually either app- or platform-centric (see discussion in §6.5), the major advantage is that it, conceptually, combines the strengths of both of these “worlds”: While it is similarly easy to install on a user’s “off-the-shelf” device as app-centric approaches [cf. Backes et al., 2013; Xu et al., 2012], which recompile apps under scrutiny but not the Android OS, it is also as powerful as platform-centric approaches that have access to a wide range of system events, but therefore need to modify the system [cf. Enck et al., 2010; Bugiel et al., 2012]. However, the approach in this thesis neither modifies the Android OS nor the apps running on it but it requires devices to be rooted to load a Linux kernel module. This may seem restrictive, but one should keep in mind that it has become common practice by now. The kernel module can be loaded even into a currently running Android system, yet is able to trace app (even pre-installed Google apps that cannot be rewritten or hidden spyware that the user is unaware of) and other Android system interactions all the way down to the OS kernel level.

**Feasibility and limitations.** As a proof of concept, we validate runtime verification for security purposes on mobile platforms: In the most comprehensive study on Android attacks so far (undertaken by the *Android Malware Genome Project* (AMGP)), the behaviour of more than 1,200 malwares was analysed and categorised into common, recurring groups of attacks. Based on this work (and the corresponding actual malware files), we specify and identify these (and similar) attacks using runtime verification. For conducting the experiments, Küster and Bauer [2015] developed a standalone monitoring app, which combines the monitor construction and the Android system instrumentation approach mentioned above. Even though there have
been many monitoring accomplishments to mobile security based on formal specifications, and especially Android, to the best of the author’s knowledge, this thesis is the first undertaking a broad study of comprehensively monitoring a vast set of real malware collected by the AMGP on a real Android stock device.

1.3.1 Publications

The majority of research results presented throughout this monograph have been published in proceedings of international conferences and journals: The detailed theoretical foundations of LTL$^{FO}$ and complexity results in §3, the introduction of an SA, the efficient and sound monitor approach for LTL$^{FO}$, as well as the comparison with a somewhat naive approach in §4, can be found in Bauer, Küster, and Vegliach [2013]. The journal article by Bauer, Küster, and Vegliach [2015] extends the conference paper by proposing an optimised monitor construction for LTL$^{FO}$, also contained in §4 as well as refining ideas on trace-length independence, which led towards a hierarchy of effectively monitorable languages in §5.

The modular, portable, and application-agnostic logging library for the Android platform as well as the malware study in §6, for which the monitoring app was developed, has been published as a technical report (see [Küster and Bauer, 2014]) and in Küster and Bauer [2015]. The first idea and study of applying runtime verification to Android security was published in Bauer, Küster, and Vegliach [2012]—at that time having implemented a somewhat naive monitoring approach based on formula rewriting for LTL$^{FO}$ and having only a non-modular instrumentation prototype for the Android system with limited access to a selection of system events.

1.3.2 Practical realisations

For undertaking the research of this thesis, three different tools have been developed. They are available to use free of charge. Furthermore, Ltlfo2mon and DroidTracer are free software, i.e., published under the GNU General Public License (GPL).

Ltlfo2mon (https://github.com/jckuester/ltlfo2mon). The decision procedure introduced in this thesis, for solving the prefix problem of properties specified in LTL$^{FO}$, is implemented in Scala and available as the project Ltlfo2mon on github. Ltlfo2mon allows to programmatically define arbitrary computable functions and predicates, contains further an optimised version of the algorithm, which is also presented in this thesis, and for comparison, allows switching to a procedure based on formula rewriting. Ltlfo2mon can be used as a standalone tool without installing Scala. It is ready-to-use via the command line in form of a Java Archive (JAR) file.

DroidTracer (https://github.com/jckuester/droidtracer-module). The application-agnostic logging library for intercepting system events on the Android system is also freely available on github. The project is wrapped into an Android Studio module that can be imported in someone’s own app development process if its func-
fionality is needed. DroidTracer has an API, which Javadoc can be found under http://jckuester.github.io/droidtracer-module/ It allows other developers of third-party apps to integrate DroidTracer and build their own analyses on top. More precisely, the API allows a developer to register a callback method (which is called whenever a new event occurs), start and stop tracing certain apps or all apps, and add specific events to a white or black list (which always or never should be captured, respectively, independent of what apps are currently intercepted).

MonitorMe (https://github.com/jckuester/monitorme-app). The monitoring app is called MonitorMe. Technically, it uses DroidTracer as a library and combines it with Ltlfo2mon, to run on an Android phone or tablet device. MonitorMe is able to run in two (not mutually exclusive) modes: (1) monitor LTL\textsuperscript{FO} specifications online, and (2) persist events in an SQLite database for offline analysis, repeatability of experiments, or to share traces with other researchers. MonitorMe is still under active development. A range of already compiled kernel modules working for various devices is available as well.

1.4 Results of this thesis

In summary, the particular results developed in this thesis are:

- The formal foundations of a custom first-order temporal logic, called LTL\textsuperscript{FO}, which is a natural extension of propositional future LTL with quantification—to allow reasoning over traces carrying data.

- A summary of known complexity results for decision problems around monitoring in the propositional case (LTL) and proofs of their pendants’ complexity, when “lifting” the setting to first-order (LTL\textsuperscript{FO}).

- A translation from LTL\textsuperscript{FO} to a novel type of automaton, called SA which is proven to be an acceptor for LTL\textsuperscript{FO}; that is, it recognises exactly the language (i.e., all models) of the LTL\textsuperscript{FO} formula it is constructed for.

- A monitor construction based on an SA which is proven to be sound albeit inherently incomplete (due to the undecidability of LTL\textsuperscript{FO}). However, it is efficient since it is able to precompute the state space structure required at runtime.

- Formal definitions towards a general categorisation of so-called effectively monitorable languages, which is closely related to this notion of “growth-inducing” (i.e., trace-length dependent) formulæ. It relates to the well-known safety-progress hierarchy [Manna and Pnueli, 1990], yet is orthogonal to it.

- Instrumentation means of a native logging layer for the Android platform, which is modular, portable to many versions and devices, and efficient. It is
application-agnostic, so that one can build whatever analysis on top, and has a functional-rich API to receive events without polling as well as to control what apps and event types should be intercepted.

- A proof of concept showing that the first-order runtime verification approach with data is effective to solve a "real world problem" in an efficient way. The study provides detection and false positive rates by using the AMGP as a benchmark. It strengthens the idea that runtime verification lends itself well for mobile security.

- Tool-support in the form of Ltlfo2mon, DroidTracer, and MonitorMe.

1.5 Structure of this thesis

This section briefly summarises the remaining chapters in this thesis.

Chapter 2—Preliminaries. This chapter outlines the formal foundations and terminology used in this thesis. It introduces notions and notations from the theory of formal language, which are needed to mathematically precisely model both propositional and data-carrying traces. Furthermore, this chapter recalls the concept of automata as acceptors for finite and infinite words, as the monitoring procedure for LTL$^\text{FO}$ is based on these. It follows the formal syntax and semantics of LTL, and an overview on different semantics suitable and arguably non-suitable for online monitoring. They have an impact on the decision problem that a monitor is required to solve in this thesis.

Chapter 3—Logic and complexity. This chapter first formally introduces LTL$^\text{FO}$. Second, it studies the complexity of decision problems revolving around runtime verification, namely the satisfiability, model checking, word, and prefix problem—first in the propositional case, for LTL, and then for its first-order extension, LTL$^\text{FO}$.

Chapter 4—Monitoring algorithm. First, this chapter introduces definitions of the new SA, and provides a translation for LTL$^\text{FO}$ formulae to an SA which is proven correct. Second, it introduces the monitoring algorithm for LTL$^\text{FO}$ formulae based on an SA as well as optimisations for its implementation, and provides a soundness proof for both. Furthermore, experimental results from an implementation are discussed. Last but not least, a comprehensive study of monitoring approaches in the literature that are also able to handle data in a broader sense is provided.

Chapter 5—Towards a hierarchy of effectively monitorable languages. This chapter sketches a categorisation of so-called effectively monitorable languages, which is closely related to this notion of “growth-inducing” (that is, trace-length dependent) formulae. Furthermore, it discusses example formulae for each of the categories.
Chapter 6—Proof of concept: Android malware detection. This chapter first details on the Android architecture and its built-in security mechanism, to then introduce conceptual and technical details of the proposed Android logging-library as well as the monitoring app. Furthermore, this chapter presents a specification manual that explains how to best specify malware characteristics and behaviour in LTL$^\text{FO}$, as well as discusses formulae that describe malware behaviour derived from the AMGP. It follows experimental results in form of detection rates for monitored malware samples from the AMGP, a discussion about false positives, as well as the examination of performance and portability. Finally, related Android instrumentation approaches in the literature are discussed.

Chapter 7—Conclusions. Conclusions are drawn from the work provided in this monograph, as well as further research directions and open problems are discussed in the last chapter of this thesis.

Appendix A—Detailed proofs. The undecidability, complexity, and soundness proofs, whose ideas are only outlined in §3 and §4, are given in this appendix in their full length.

Appendix B—Proof of concept: Additional experiment data. Comprehensive experimental results from the malware analysis on the Android platform are contained in this appendix.
The aim of this chapter is to recall formal foundations and define some terminology used in the remainder of this thesis.

Section §2.1 introduces the notions and notations of finite and infinite words from the theory of formal language, and §2.2 details on their use as an underlying model of system executions. Furthermore, §2.4 formally describes the syntax and semantics of LTL—originally defined in terms of infinite words—that is used to formulate system properties for static as well as dynamic verification. Section §2.4 further explains the meaning of some LTL formulae based on examples from the well-known specification patterns. Since many monitoring approaches for LTL, similar to the one in this thesis, are based on executable automata, §2.3 presents how words are recognised by means of automata theory. It focuses on regular and \( \omega \)-regular languages, as languages defined in terms of LTL are a strict subset of the latter. Finally, §2.5 gives an overview of different LTL semantics over finite words—some arguably suitable and others non-suitable for online monitoring—as these determine what decision problem a monitor should solve; hence, the semantics ultimately influences the monitor construction provided in §4.

2.1 Words over alphabets

An \textit{alphabet} \( \Sigma \) is a finite non-empty set of symbols. A finite \textit{word} (or string) over \( \Sigma \) is a sequence of symbols drawn from \( \Sigma \), i.e., finite words are of the form \( u = u_0 \ldots u_i \ldots u_n \), where \( n, i \in \mathbb{N}_0, 0 \leq i \leq n \) and \( u_i \in \Sigma \). Depending on the context, we give the symbols a concrete name, e.g., we speak of \textit{events} when we use words to represent system executions. Also, if we refer to a word not just in its abstract sense of a language, but mean the representation of some concrete system behaviour or executions, we might use the term \textit{trace} instead.

The length of a finite word \( u = u_0 u_1 \ldots u_n \), usually written \( |u| \), is \( n + 1 \) (i.e., the number of all symbols occurring in \( u \)). The \textit{empty word}, denoted \( \varepsilon \), is the word of length 0. Given an alphabet \( \Sigma \), we write \( \Sigma^n \) to denote all words with length \( n \) that can be constructed from \( \Sigma \). The set of all finite words is indicated using the Kleene star operator, \( \Sigma^* := \bigcup_{i \in \mathbb{N}_0} \Sigma^i \). Furthermore, the Greek letter \( \omega \) (omega) is used to denote “infinity”, i.e., represents the smallest infinite ordinal \( \omega = \{0,1,2,\ldots\} \), so
that we write $\Sigma^\omega$ to denote the set of all infinite words over the alphabet $\Sigma$. We call an infinite word $w = w_0w_1w_2\ldots \in \Sigma^\omega$ also an $\omega$-word. As a convention, we use $u, u', v, \ldots$ to denote finite words, by $\sigma$ the word of length 1, and $w$ for infinite ones or where the distinction is of no relevance. Furthermore, we define $\Sigma^\omega$ to be the union of the sets of all infinite and finite words, and $\Sigma^+ \defeq \Sigma^* \setminus \{\epsilon\}$ the set of all non-empty finite words, over $\Sigma$ respectively.

A prefix of an infinite word $w = w_0w_1w_2\ldots$ is a word $v$ of the form $v = w_0w_1\ldots w_i$, where $i \in \mathbb{N}_0$, or in the case of a finite word $w = w_0w_1\ldots w_n$, if $0 \leq i \leq n$. Note that $\epsilon$ is a prefix of any finite or infinite word. For two words $u$ and $v$, where $u$ is finite, we write $uv$ when we mean their concatenation, which itself is another word; more precisely an $\omega$-word if $v$ is an $\omega$-word, but finite if $v$ is finite. For a word $w = w_0w_1\ldots$, the word $w^i$ is defined as $w_iw_{i+1}\ldots$.

A language is any set of words over some fixed alphabet $\Sigma$, denoted by $L$, and $L \subseteq \Sigma^*$ therefore is a language of finite and $L \subseteq \Sigma^\omega$ of infinite words. Based on concatenation of single words, we define the concatenation of finite languages, $L L' \defeq \{uv \in \Sigma^* \mid u \in L \text{ and } v \in L'\}$, and the Kleene-closure or star of a language inductively as $L^* \defeq \bigcup_{i\in \mathbb{N}_0} L^i$, where $L^0 \defeq \{\epsilon\}$, $L^1 \defeq L$, and $L^i = L \cdots L$ (the concatenation of $i$ copies of $L$). The union and intersection of two languages $L$ and $L'$, denoted as $L \cup L'$ and $L \cap L'$ respectively, is defined as for usual sets; thus $L \cup L'$ contains all words in either $L$ or $L'$, or both, whereas $L \cap L'$ contains all words that are only in both languages.

## 2.2 Representation of system behaviour

This section details on some common, propositionally-based representations of system executions in static and dynamic verification methods, such as model checking or runtime verification. These representations are based on the notions of finite and infinite words in \[ \text{(2.1)} \]

A Kripke structure is a transition system, or digraph. It is used in model checking to represent all possible, infinite system executions of a reactive system. States of the structure correspond to system states, and executions correspond to paths through the structure. Clarke et al. [2001] define a Kripke structure as follows.

**Definition 2.2.1** (Kripke structure). Let $AP$ be a set of atomic propositions. A Kripke structure over $AP$ is a tuple $K = (S, s_0, \lambda, \rightarrow)$, where

- $S$ is a finite set of states,
- $s_0 \in S$ a distinguished initial state,
- $\lambda : S \rightarrow 2^{AP}$ a labelling function assigning propositions to states (where $2^{AP}$ is the power set of $AP$), and
- $\rightarrow \subseteq S \times S$ a left-total transition relation, (i.e., $\forall s \in S. \exists s' \in S. (s, s') \in \rightarrow$).

\[1\] Words can be also defined as a function, i.e., $u : \{0,1,\ldots,n\} \rightarrow \Sigma$ for finite ones or $w : \mathbb{N}_0 \rightarrow \Sigma$ for infinite ones, which leads to the notion $u = u(0)u(1)u(2)\ldots u(n)$ and $w = w(0)w(1)w(2)\ldots$
There exist a transition from state \( s \in S \) to state \( s' \in S \) if and only if \((s, s') \in \rightarrow\).

A run \( \rho \) through the structure \( K \) is a sequence of states \( \rho = s_0 s_1 s_2 \ldots \) such that for \( i \in \mathbb{N}_0 \) it holds that \((s_i, s_{i+1}) \in \rightarrow\), and \( s_0 \) is the initial state. The infinite trace \( w = \lambda(s_0) \lambda(s_1) \lambda(s_2) \ldots \) over a run \( \rho \) is then an \( \omega \)-word over the alphabet \( 2^{AP} \).

Recall that we call the sets of atomic propositions events. The occurrence of some native system event is then modelled by a single atomic proposition if it is contained in such set. Since \( \rightarrow \) is left-total, there is always an infinite run through the structure. In other words, the language defined by a Kripke structure is an \( \omega \)-language. A deadlock state \( s \) can be modeled by having for \( s \) only a single outgoing and self-looping transition, which is of the form \((s, s) \in \rightarrow\).

We say that \( K \) is a linear Kripke structure, if each state of \( K \) has at most one successor and the transition relation is loop-free. In this case \( K \) defines a single, finite run and it models a sequence of observed system executions in runtime verification, for example.

### 2.3 Automata as acceptors of finite or infinite words

Compared to Turing machines \cite{Turing1936}—a useful mathematical model to study what can and what cannot be computed in terms of real computers—automata are a simpler form of abstract “machines”. Conceptually, a finite automaton might represent a computer program. It is at all times in one of a finite number of states, whose purpose is to “remember” relevant parts of the program’s history. Since the complete history of reactive systems cannot be remembered with finite resources, the states are a helpful concept to design a program in such a way that it has to “forget” what is not important. In this regard an automaton is a useful concept in runtime verification; that is, to build monitoring procedures, and also study the amount of information these keep at most at any time in order to yield a result.

Note that wrt. generating a monitor, an automaton can serve as a so called acceptor, i.e., indicate whether or not a received input is accepted (for this purpose states are either marked as non-accepting or accepting). Formally, an automaton can also be considered as the representation of a language, which contains every word it accepts but none of the rejected ones.

**Automata over finite words.** Let us first recall automata over finite words, which accept what are called the regular languages.

**Definition 2.3.1 (Non-deterministic finite automaton).** A non-deterministic finite automaton (NFA) is a tuple \( A = (\Sigma, Q, Q_0, \delta, F) \), where

- \( \Sigma \) is a finite alphabet,
- \( Q \) a finite set of states,
- \( Q_0 \subseteq Q \) a set of distinguished initial states,
- \( \delta : Q \times \Sigma \rightarrow 2^Q \) a transition function, and
• \( F \subseteq Q \) a distinguished set of final (or accept) states, also called acceptance set.

**Definition 2.3.2** (Complete NFA). An NFA \( A \), is called complete, if for all \( q \in Q \) and \( \sigma \in \Sigma \), there exists a state \( q' \in Q \) with \( q' \in \delta(q, \sigma) \).

Note that for every non-complete NFA, \( A \), there exists a complete NFA, \( A' \), such that \( L(A) = L(A') \) [cf. Hopcroft and Ullman, 1979].

**Definition 2.3.3** (Finite run). Given a finite word \( u = u_0u_1 \ldots u_n \in \Sigma^* \), a finite run\(^2\) of \( A \) over \( u \) is defined as a mapping \( \rho : \{0, \ldots, n\} \rightarrow Q \), such that

1. \( \rho(0) \in Q_0 \), and
2. \( \rho(i + 1) \in \delta(\rho(i), u_i) \), where \( i, n \in \mathbb{N}_0 \) and \( i < n \).

**Definition 2.3.4** (Accepting run). A run \( \rho = \rho(0)\rho(1)\ldots\rho(n) \) is called accepting if \( \rho(n) \in F \), i.e., \( \rho \) ends in an accepting state.

As said above, automata can be seen as acceptors of words of a language, where the accepted language of \( A \) is defined as

\[
L(A) := \{ u \in \Sigma^* \mid \text{there exists an accepting run of } A \text{ over } u \}.
\]

The size of \( A \), denoted \( |A| \), is defined as the numbers of states and transitions of \( A \), i.e.,

\[
|A| = |Q| + \sum_{q \in Q} \sum_{\sigma \in \Sigma} |\delta(q, \sigma)|.
\]

\( A \) is called a deterministic finite automaton (DFA), if \( |Q_0| \leq 1 \) and \( |\delta(q, \sigma)| \leq 1 \) for all \( q \in Q \) and \( \sigma \in \Sigma \). It is well-known that deterministic and non-deterministic finite automata are equally expressive; that is, a language can be recognised by an NFA iff it can be recognised by a DFA, this can be proven via the well-known powerset construction by [Rabin and Scott 1959]. Therefore, we speak only of finite automata, and omit their characterisation of being (non-)deterministic, if it is irrelevant or clear from the context.

Another language-defining notation to describe regular languages are regular expressions [Kleene, 1956], these are outlined here briefly for completeness, since LTL is a strict subset of their extension to the \( \omega \)-regular languages, but which are otherwise of no specific relevance in the rest of this thesis.

**Definition 2.3.5** (Regular expressions). The set of regular expressions over an alphabet \( \Sigma \), denoted by \( RE(\Sigma) \), is inductively defined as follows:

\[
e, \emptyset \in RE(\Sigma), \text{ where } e \text{ and } \emptyset \text{ are constant symbols,}
\]

\[
\text{if } \sigma \in \Sigma, \text{ then } \sigma \in RE(\Sigma),
\]

\[
\text{if } E, E' \in RE(\Sigma), \text{ then } E + E', EE', E^* \in RE(\Sigma).
\]

\(^2\)Alternatively, a finite run can be defined as a finite word (or sequence of states) \( q_0q_1 \ldots q_n \in Q^* \), such that \( q_0 \in Q_0 \) and \( q_{i+1} \in \delta(q_i, u_i) \) for \( i, n \in \mathbb{N}_0 \) and \( i < n \).
2.3 Automata as acceptors of finite or infinite words

The language defined by a regular expression is defined inductively by the following rules:

- $L(\emptyset) := \emptyset$, $L(\epsilon) := \{\epsilon\}$,
- $L(\sigma) := \{\sigma\}$,
- $L(E + E') := L(E) \cup L(E')$, $L(EF) := L(E)L(F)$,
- $L(E^*) := (L(E))^*$.

The equivalence of finite automata and regular expressions is known by Kleene’s famous theorem:

**Theorem 2.3.1 (Kleene theorem [Kleene, 1956]).** A language $L \subseteq \Sigma^*$ is regular iff there is a regular expression $E \in RE(\Sigma)$ such that $L = L(E)$.

Another concept we refer to in this thesis is a deterministic Moore machine [Moore, 1956]. It is a finite state transducer, which—in contrast to a finite automaton—has an output alphabet and tape; thus, it can print out some word, which is for a Moore machine solely determined by its visited states. A Moore machine is used by the monitor of Bauer et al. [2011] to print a verdict after processing each event.

**Definition 2.3.6 (Moore machine).** A Moore machine is a 6-tuple $M = (\Sigma, Q, Q_0, \delta, \lambda, \Lambda)$, where the first four symbols are defined as for the finite automaton above, and

- $\Lambda$ is an output alphabet, and
- $\lambda : Q \rightarrow \Lambda$ a labelling function.

**Automata over infinite words.** Automata that serve as acceptors for $\omega$-words are called $\omega$-automata. There exist various classes of $\omega$-automata, such as the ones proposed by [Büchi, 1962], [Rabin, 1969], [Streett, 1982], or [Müller, 1963]—each in a deterministic or non-deterministic form. They are structurally identical to finite automata (i.e., are defined by the same five-tuple), but in order to accept infinite words, they differ in the accepting condition. Furthermore, they all have the same expressiveness (i.e., are able to accept what is called the $\omega$-regular languages), except the deterministic Büchi automaton, which is strictly less expressive. However, we consider only the non-deterministic Büchi automaton (NBA) [Büchi, 1962] any further, as this is the standard acceptors for LTL-definable languages (i.e, the star-free regular languages [cf. Lichtenstein et al., 1985], which are a strict subset of the $\omega$-regular languages), and also an underlying concept of the monitor construction in this thesis. We omit the reference “non-deterministic”, thus speak of a BA whenever we mean an NBA.

**Definition 2.3.7 (Infinite run).** Given an $\omega$-word $w = w_0w_1w_2\ldots \in \Sigma^\omega$, an infinite run of $\omega$-automaton $A$ over $w$ is defined as a mapping $\rho : \mathbb{N}_0 \rightarrow Q$, such that

- $\rho(0) \in Q_0$, and
- $\rho(i + 1) \in \delta(\rho(i), w_i)$, where $i \in \mathbb{N}_0$.

For a run $\rho$ and $\omega$-automaton $A$, the set of states visited infinitely often is denoted by

$$\text{Inf}(\rho) := \{q \in Q \mid \rho(i) = q \text{ for infinitely many } i \in \mathbb{N}_0\}.$$
Definition 2.3.8 (Non-deterministic Büchi automaton). An NBA is an an ω-automaton $A$, of which a run $ρ$ is accepting if $ρ(i) \in F$ holds for infinitely many indices $i \in \mathbb{N}_0$, i.e.,

$$\text{Inf}(ρ) \cap F \neq \emptyset$$ (Büchi acceptance).

A further concept we need is that of a generalised (non-deterministic) Büchi automaton (GBA). It is used for the standard translation of LTL to ω-automata [Gerth et al., 1995], as it often results in smaller size than directly constructed NBAs [Gastin and Oddoux, 2001]. A simple translation of a GBA into an NBA exists in form of the well-known counting construction [cf. Gerth et al., 1995; Gastin and Oddoux, 2001, §5].

Definition 2.3.9 (Generalised BA). A GBA, is a tuple $G = (Σ, Q, Q_0, δ, F)$, where $Σ$, $Q$, $Q_0$, and $δ$ are defined as for an NBA, and $F = \{F_1, \ldots, F_n\}$ is a set of acceptance sets. Its acceptance condition requires a run $ρ$ to visit each of the sets $F_1, \ldots, F_n$ infinitely often:

$$∀F_i \in F : \text{Inf}(ρ) \cap F_i \neq \emptyset$$ (generalised Büchi acceptance).

The (infinite) accepted language of an ω-automaton is defined as

$$L(A) := \{ w \in Σ^ω | \text{there exists an accepting run of } A \text{ over } w \}.$$ 

Definition 2.3.10 (ω-regular languages). A language $L \subseteq Σ^ω$ is called an ω-regular language, if $L := \bigcup_{1 \leq i \leq n} L_i L_i^ω$, for some $n \in \mathbb{N}$, where $L_i, L_i'$ are regular languages, $L_i'$ is non-empty and does not contain the empty word $ε$, and $L_i^ω := \{ v_0 v_1 \ldots | v_j \in L_i' \text{ for all } j \in \mathbb{N}_0 \}$.

It is noteworthy that Kleene’s theorem carries over in straightforward manner to the ω-regular languages:

Theorem 2.3.2 (Büchi [1962]). An ω-language $L \subseteq Σ^ω$ is recognised by a BA if and only if $L$ is ω-regular.

The ω-regular languages and BAs—like regular languages and finite automata—are closed under negation, union, and intersection; that is, if $L_1, L_2 \subseteq Σ^ω$ are two ω-regular languages, then $Σ^ω \setminus L_1$, $L_1 \cup L_2$, and $L_1 \cap L_2$ are regular ω-languages, respectively [cf. Baier and Katoen, 2008].

### 2.4 Propositional linear-time temporal logic

Linear-time temporal logic (LTL) is often more intuitive than writing ω-regular expressions or even notions of automata directly. It was first proposed by Pnueli [1977] as a formalism to program verification of reactive and concurrent systems and adds temporal operators to propositional logic. While propositional logic is limited to specify conditions about a single state (or point in time), LTL extends this concept to sequences of states (and therefore permits truth values of propositions to vary...
over time). Note that temporal in this context does not refer to a global external clock that determines when exactly events occur, but merely their relative order. In this setting time points can then be modelled by monotonically increasing natural numbers, which is sufficient to express for example the property “no second login should happen until a logout”. Moreover, the underlying perspective on time is linear, so that every moment has exactly a single successor moment—in contrast to branching time, where multiple successors, in a tree-like manner, are allowed (see for example computational tree logic (CTL) \cite{Baier:2008}). As the discipline of run-time verification reasons over single paths of executions, linear time is the only and sufficient time model considered here.

In this section we recall the formal syntax and semantics of LTL.\footnote{In the literature, many variants of LTL exist, for example with past operators \cite{Laroussinie:2002}, but which are not further discussed in this thesis.}

We also explain the meaning of some common examples of LTL properties.

**Definition 2.4.1** (LTL syntax). Given a set $AP = \{p_1, \ldots, p_n\}$ of atomic propositions, the set of well-formed LTL formulae over $AP$, denoted by $\mathsf{LTL}(AP)$, is defined by the following induction:

- if $p \in AP$, then $p \in \mathsf{LTL}(AP)$,
- if $\phi, \psi \in \mathsf{LTL}(AP)$, then $\neg \phi, \phi \land \psi, \phi U \psi \in \mathsf{LTL}(AP)$.

If a concrete set $AP$ is irrelevant or clear from the context, we write simply LTL instead of $\mathsf{LTL}(AP)$. LTL formulae are usually interpreted over infinite traces of sets of atomic propositions, i.e., elements from the alphabet $\Sigma = 2^{AP}$. Interpretations over finite traces exist too, such as the one originally introduced by Kamp \cite{1968}. To reduce redundancy, we provide in the following a combining definition, since both forms are referred to in this thesis.

**Definition 2.4.2** (LTL semantics). Let $\phi \in \mathsf{LTL}(AP)$, $w = w_0 w_1 \ldots \in \Sigma^\infty$, where $\Sigma = 2^{AP}$, a non-empty trace, and $i \in \mathbb{N}_0$ a time point. Then the relation $w, i \models \phi$ is defined inductively as follows:

- $w, i \models p$ iff $p \in w_i$, where $p \in AP$,
- $w, i \models \neg \phi$ iff $w, i \not\models \phi$,
- $w, i \models \phi \land \psi$ iff $w, i \models \phi$ and $w, i \models \psi$,
- $w, i \models X \phi$ iff $|w| > i$ and $w, i+1 \models \phi$,
- $w, i \models \phi U \psi$ iff there is a $k$ s.t. $i \leq k < |w|, w, k \models \psi$, and for all $i \leq j < k, w, j \models \phi$.

If $w, 0 \models \phi$ holds, we usually write $w \models \phi$ instead, and we say $w$ is a model of (or satisfies) $\phi$. Symmetrically, if $w \not\models \phi$, we say $w$ violates $\phi$, in which case $w$ is not a model of $\phi$. The set $\mathcal{L}(\phi) := \{w \in \Sigma^\infty \mid w \models \phi\}$ is called the language of $\phi$. If $\mathcal{L}(\phi) = \emptyset$, then $\phi$ is called unsatisfiable, or otherwise, if $\phi$ has a model, satisfiable. Note that LTL is a decidable logic; in fact, the satisfiability problem for LTL is known to be PSpace-complete \cite{Sistla:1985}.
Although this semantics, which was also proposed by Markey and Schnoebelen [2003], gives rise to mixed languages, i.e., languages consisting of finite and infinite words, we shall only ever be concerning ourselves with either finite-trace or infinite-trace languages, but not mixed ones. It is easy to see that over infinite traces this semantics matches the definition of standard LTL. If necessary we use a subscript and write $|=F$ or $|=\omega$, to distinguish whether we mean the finite or infinite semantics, respectively.

When using LTL in proofs or other formal contexts, we restrict the Boolean operators to $\wedge$ and $\neg$, as those are sufficient to make the logic functionally complete in terms of Boolean operators. However, in other parts throughout this thesis, for readability, we may make use of the following derived operators:

$$\text{false} := p \land \neg p,$$

$$\varphi \lor \psi := \neg(\neg\varphi \land \neg\psi),$$

$$\varphi \leftrightarrow \psi := (\varphi \Rightarrow \psi) \land (\psi \Rightarrow \varphi),$$

$$\varphi \Rightarrow \psi := \neg\varphi \lor \psi,$$

$$\varphi \oplus \psi := (\varphi \lor \psi) \land \neg(\varphi \land \psi).$$

Some further and commonly seen LTL operators are the following derivations, which add "syntactic sugar" to the standard syntax. They do not add expressiveness to LTL—as defined in terms of the standard operators—but are practical useful to write shorter, often more concise and intuitive formulae:

$$F\varphi := \text{true} U \varphi$$

$$(\text{eventually}),$$

$$G\varphi := \neg F \neg \varphi \ (\text{or } q W \text{false})$$

$$(\text{globally}),$$

$$\varphi W \psi := (\varphi U \psi) \lor G \varphi$$

$$(\text{weak until}),$$

$$\varphi R \psi := \neg(\neg \varphi U \neg \psi)$$

$$(\text{release}).$$

Note that the $G$-operator is dual to the $F$-operator, the $U$-operator dual to the $R$- and $W$-operator, respectively, and the $X$-operator is dual to itself.\(^4\)

$$\neg G \varphi = F \neg \varphi,$$

$$\neg F \varphi = G \neg \varphi,$$

$$\neg(\neg \varphi U \neg \psi) = \varphi R \psi,$$

$$\neg(\neg \varphi R \neg \psi) = \varphi U \psi,$$

$$\neg(\varphi U \psi) = (\varphi \land \neg \psi) W (\neg \varphi \land \neg \psi),$$

$$\neg(\varphi W \psi) = (\varphi \land \neg \psi) U (\neg \varphi \land \neg \psi),$$

$$\neg X \varphi = X \neg \varphi.$$

**Examples of common LTL specifications.** Let us consider some examples of widely-used LTL formulae. They are chosen from the well-known specification patterns proposed by Dwyer et al. [1999], who collected these by surveying over 500 properties for finite-state verification tools. Each pattern has a scope, which is the extent of a program execution over which the pattern must hold. There are five basic scopes,\(^4\)

\(^4\)Note that the $X$-operator is only dual to itself in the infinite trace semantics.
highlighted in Fig. 2.1 in grey: global (the entire program execution), before (the execution up to a given event), after (the execution after a given event), between (any part of an execution from one given event to another), and after until (like between but the designated part of the execution continues even if the second event does not occur). The start and ending of a scope is defined by so-called state formulae; that is, if those evaluate to true over a given event. State formulae are denoted here by capital letters (e.g., $P$, $Q$, $R$, or $S$) and represent Boolean combinations of propositions. However, they can sometimes also contain temporal operators without changing the semantics of the pattern (see for example Bauer and Leucker [2011] and the definition of the stop operator for this purpose). Here, the scope begins with the start event, whereas the end event lies outside the scope. However, it is possible to define open-left and closed-right scopes as well [cf. Bauer and Leucker, 2011].

The patterns themselves are organised into two groups, of which the first semantically deals with the occurrence of events, i.e., require a state formula to evaluate to true or not over an event: absence (a given event does not occur within a scope), existence (a given event must occur within a scope, which is dual to the absence pattern), bounded existence (a given event must occur at most $k$ times within a scope), and universality (a given event occurs all throughout a scope). The second group of patterns deals with the ordering of events, i.e., events must appear in a defined order: precedence (a given event for which $P$ holds must always be preceded by one for which a given $Q$ holds within a scope), response (a given event for which $P$ holds must always be followed by one for which a given $Q$ holds within a scope, which is converse to the precedence pattern), as well as chain precedence and chain response (a generalisation of the precedence and response pattern, i.e., a sequence of events for which $P_1$, $\ldots$, $P_n$ holds must always be preceded or followed by one for which $Q_1$, $\ldots$, $Q_n$ holds, respectively).

Table 2.1 lists the LTL formulae for a selection of the discussed patterns over different scopes. A common use case for the absence pattern is mutual exclusion,
Table 2.1: Example LTL formulae from the specification patterns.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Scope</th>
<th>LTL formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absence (P is false)</td>
<td>Globally</td>
<td>$G(\neg P)$</td>
</tr>
<tr>
<td>Existence (P becomes true)</td>
<td>After Q until R</td>
<td>$G(Q \land \neg R \Rightarrow (\neg R U (P \land \neg R)))$</td>
</tr>
<tr>
<td>Bounded existence (P occurs max. twice)</td>
<td>After Q until R</td>
<td>$G(Q \Rightarrow ((\neg P \land \neg R) U (R \lor ((P \land \neg R) U (R \lor ((P \land \neg R) U (R \lor \neg (P W R) \lor G P))))))))$</td>
</tr>
<tr>
<td>Universality (P is true)</td>
<td>Between Q and R</td>
<td>$G((Q \land \neg R \land FR) \Rightarrow (P U R))$</td>
</tr>
<tr>
<td>Precedence (S precedes P)</td>
<td>After Q until R</td>
<td>$G(Q \land \neg R \Rightarrow (\neg P W (S \lor R)))$</td>
</tr>
<tr>
<td>Response (S responds to P)</td>
<td>After Q</td>
<td>$G(Q \Rightarrow G(P \Rightarrow F S))$</td>
</tr>
<tr>
<td>Precedence chain (P precedes S, T)</td>
<td>Before R</td>
<td>$FR \Rightarrow ((\neg (S \land (\neg R)) \land X (\neg R U (T \land \neg R))) U (R \lor P))$</td>
</tr>
<tr>
<td>Response chain (P responds to S, T)</td>
<td>After Q</td>
<td>$G(Q \Rightarrow G(S \land X F T \Rightarrow X (\neg T U (T \land F P))))$</td>
</tr>
</tbody>
</table>

where $P$ describes for example that orthogonal traffic lights at a crossing show a green light at the same time. The precedence pattern can be used to express that a resource (e.g., a lock) is only granted in response to a request, whereas the response pattern to describe that a resource must be granted after it is requested. Bounded existence is useful to describe that a process $A$ can enter its critical section at most twice while $B$ is waiting, in which case a between scope is used and delimited by $B$ entering and exiting its waiting region.

2.5 LTL semantics for runtime verification

After having introduced in the previous section the specification language LTL, a model checker and monitor’s task can be described to decide for a given property of that language whether it is satisfied (or violated) by all traces of some given set of traces, or a given single trace, respectively. However, while a model checker, in terms of reactive systems, can evaluate traces according to the standard semantics of LTL\(^5\)—since both are defined over infinite traces—a monitor must use a finite LTL semantics, as observations are merely prefixes of infinite traces.

\(^5\)For bounded model checking, obviously, also finite trace semantics are used \cite{Biereetal2003}.
Runtime verification is usually performed to complement model checking, and therefore the finite LTL semantics should preferably coincide with the infinite one; that is, a monitor should only return true denoted by $\top$ (or false denoted by $\bot$) if the property is satisfied (or violated, respectively) for all continuations. Recall that this is exactly the definition of impartiality, described by Property 4 in §1.2.2. Note that this property is easy to comply with for an offline monitor, as there are no continuations, and verdicts are returned only once at the end of a trace. In other words, a trace represents complete knowledge—as no further observations of a system are to be expected—so that under this assumption monotonicity is not an issue. An offline monitor, therefore, can for example implement the finite LTL semantics introduced in Def. 2.4.2: Intuitively $\varphi U \psi$ is interpreted as false if $\psi$ never evaluates to true on a finite trace, and $X \varphi$ is interpreted as false at the end of the trace if no further event exists—called the strong semantics of the $X$-operator. This idea is extended by Manna and Pnueli [1995], who add a weak view of the $X$-operator (being dual to the strong one, and which we denote here with $X!$) that is interpreted as “if a next event exists then for this event $\varphi$ must hold”.

In contrast, during online monitoring, traces always resemble only incomplete albeit expanding knowledge (and are therefore also called truncated). In this section we will see that the finite LTL semantics from above and similar variations based on a 2-valued semantics (i.e., with only true and false as truth values) are inadequate for that purpose since these are inherently not able to comply with monotonicity.

**Weak, strong, and neutral finite path semantics.** While the syntax and semantics of LTL for checking complete traces is well accepted in the literature, there is no consensus on defining LTL over truncated traces. Eisner et al. [2003] define for example $LTL^{\text{trunc}}$, which is an LTL semantics defined wrt. finite or infinite traces, and an indicator for the strength of the interpretation; it can be either weak, strong or neutral in relation to the standard finite semantics by Kamp [1968] in Def. 2.4.2 (denoted by $[w \models \varphi]_-$, $[w \models \varphi]_+$, and $[w \models \varphi]_F$, respectively). We call the resulting logics $LTL^-$, $LTL^+$, and FLTL, respectively. Intuitively, in the weak view $U$ acts like $W$, $X$ like $X!$; and in the strong view (which is dual to the weak one) $W$ acts like $U$, and $X!$ like $X$; whereas the neutral one leaves the operators unchanged. In other words, the weak view has a preference for false positives, the strong view for false negatives, and the neutral view desires to see as much evidence as can reasonably be expected from a finite trace.

Table 2.2 exemplifies how these three semantics behave differently for a trace consisting of a single event. An interesting corollary shows that if $w$ is infinite then $[w \models \varphi]_F$ iff $[w \models \varphi]_- \iff [w \models \varphi]_+$; that is, the strength indicators behave identical over an infinite trace [Eisner et al., 2003, Corollary 5]. However, the inherent problem with all three proposed finite semantics for truncated paths is that those must evaluate to true or false prematurely since these cannot reflect the “not yet known”-case properly to avoid either false positives or negatives.
Table 2.2: The evaluations of a trace wrt. the weak, strong and natural view.

<table>
<thead>
<tr>
<th>neutral</th>
<th>strong</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a} ⫋ aUb</td>
<td>{a} ⫋ aUb</td>
<td>{a} ⊨ aUb</td>
</tr>
<tr>
<td>{a} ⫋ Fb</td>
<td>{a} ⫋ Fb</td>
<td>{a} ⊨ Fb</td>
</tr>
<tr>
<td>{a} ⫋ Ga</td>
<td>{a} ⫋ Ga</td>
<td>{a} ⊨ Ga</td>
</tr>
<tr>
<td>{a} ⫋ Xb</td>
<td>{a} ⫋ Xb</td>
<td>{a} ⊨ Xb</td>
</tr>
</tbody>
</table>

**LTL\textsubscript{3} semantics.** Therefore, [Bauer et al. 2011] have introduced a 3-valued prefix semantics (i.e., it cannot be defined inductively as in Def. 2.4.2), defined as follows.

**Definition 2.5.1** (LTL\textsubscript{3} semantics).

\[ [u \models \varphi]_3 = \begin{cases} \top & \text{if } \forall w \in \Sigma^\omega : uw \models \varphi \\ \bot & \text{if } \forall w \in \Sigma^\omega : uw \not\models \varphi \\ ? & \text{otherwise.} \end{cases} \]

It agrees with the property of impartiality and anticipation stated in §1.2.2. Note that the truth domain here is a set \( B_3 = \{ \top, \bot, ? \} \), where ? denotes the inconclusive case. A monitor following this semantics must return ? for a prefix as long as there exist continuations of which some satisfy and some violate the property \( \varphi \). However, as soon as there are only continuations of one sort, it must return a conclusive verdict.

**RV-LTL semantics.** [Bauer et al. 2010] further introduced a 4-valued prefix semantics, called RV-LTL. It refines the inconclusive case of LTL\textsubscript{3} by “presumably true” (\( \top^P \)) and “presumably false” (\( \bot^P \)); that is, helps with so-called ugly prefixes (i.e., prefixes whose expansions all lead to a monitor returning ? but never a conclusive verdict) by providing at least some information to which degree a formula is considered satisfied or not.

**Definition 2.5.2** (RV-LTL semantics).

\[ [u \models \varphi]_{RV} = \begin{cases} \top & \text{if } [u \models \varphi]_3 = \top \\ \bot & \text{if } [u \models \varphi]_3 = \bot \\ \top^P & \text{if } [u \models \varphi]_3 = ? \text{ and } [u \models \varphi]_F = \top \\ \bot^P & \text{if } [u \models \varphi]_3 = ? \text{ and } [u \models \varphi]_F = \bot \end{cases} \]

Note that RV-LTL complies with a property that is not further considered in this thesis: complementation by negation, i.e., a formula \( \neg \varphi \) of a logic should yield a complementary and different truth value than \( \varphi \) when evaluated over the same trace. This is not the case for traces leading to ? wrt. the LTL\textsubscript{3} semantics, as ? is complementary to itself, in contrast to \( \top \), \( \bot \) or \( \top^P \), \( \bot^P \) being complementary to each other, respectively.
Table 2.3 summarises the different semantics and their most relevant properties to runtime monitoring [Bauer et al., 2010]. A conclusion drawn from the discussion in this section is that we prefer an online monitor to check traces—and similarly traces with data when LTL is lifted to LTL$^{FO}$ in the following chapters—at least wrt. a 3-valued semantics such as LTL$_3$.

### 2.6 Summary

This chapter briefly outlines formal foundations and terminology used in this thesis. It details on notions and notations from theory of formal language, which provides the basic framework to define propositional as well traces with data; in fact we will see a precise definition of the latter in the next chapter. It recalls the standard syntax and semantics of LTL over finite and infinite traces, since the next chapter researches on its extension to first-order quantification. Furthermore, it discusses finite LTL semantics used in runtime verification—namely, the neutral, strong and weak finite LTL semantics, as well as the prefix semantics LTL$_3$ and RV-LTL. While the first three all cannot comply with monotonicity, the latter fulfil this property; thus, we consider the prefix problem as the central decision problem to solve in online monitoring.
Chapter 3

Logic and complexity

This chapter explores and compares the complexity of decision problems around runtime monitoring—namely the satisfiability, word, model checking, and prefix problem both for propositional LTL (see §3.2.1) as well as LTL\(^{FO}\) (see §3.2.2). While the former is the de facto standard specification language in runtime verification, the latter can be considered as natural extension with first-order quantifiers. In a nutshell, the quantifiers bind domain values from parameterised actions appearing in a trace, and not directly from some “external” (possibly infinite) domain. LTL\(^{FO}\) is introduced in §3.1. Based on the desired properties of a monitor in §1.2.2 and the different LTL semantics laid out in §2.5, we will see that the prefix problem a runtime monitor should arguably solve is undecidable in LTL\(^{FO}\). In contrast, the word problem, which should be solved by offline monitors is decidable both in LTL and LTL\(^{FO}\) (and in LTL also computationally less involved). Furthermore, this chapter discusses that the actual problem wrt. online monitoring is difficult to state in terms of complexity theory at all, but the prefix problem helps to find at least a lower bound.

3.1 LTL\(^{FO}\)—Formal definitions and notation

Let us now introduce the first-order specification language LTL\(^{FO}\) and related concepts in more detail. The first concept we need is that of a (sorted first-order) signature, given as \(\Gamma = (S, F, R)\), where \(S\) is a finite non-empty set of sorts, \(F\) a finite set of function symbols and \(R = U \cup I\) a finite set of a priori uninterpreted and interpreted predicate symbols, s.t. \(U \cap I = \emptyset\) and \(R \cap F = \emptyset\). The former set of predicate symbols are referred to as U-operators and the latter as I-operators. As is common, 0-ary function symbols are also referred to as constant symbols, and 0-ary predicate symbols as propositions. We assume that all operators in \(\Gamma\) have a given arity that ranges over the sorts given by \(S\), respectively. We also assume infinite supplies of variables, one for each sort, collectively referred to as a set \(V\), where \(V \cap (F \cup R) = \emptyset\). Let us refer to the first-order language determined by \(\Gamma\) as \(L(\Gamma)\). The set of terms \(T(\Gamma)\) of \(L(\Gamma)\) is inductively defined as follows: Any \(x \in V\) and any constant is a term, and if \(f \in F\) is an \(n\)-ary function symbol, and \(t_1, \ldots, t_n\) are terms of the proper sort, then \(f(t_1, \ldots, t_n)\) is a term.
Definition 3.1.1 (Syntax). Formulae of \( \mathcal{L}(\Gamma) \) are defined inductively as follows:

\[
\begin{align*}
\text{if } p \in U \text{ and } t_1, \ldots, t_n \in T(\Gamma), \text{ then } p(t_1, \ldots, t_n) & \text{ is a } \mathcal{L}(\Gamma) \text{ formula}, \\
\text{if } r \in I \text{ and } t_1, \ldots, t_n \in T(\Gamma), \text{ then } r(t_1, \ldots, t_n) & \text{ is a } \mathcal{L}(\Gamma) \text{ formula}, \\
\text{if } \psi, \phi \text{ are } \mathcal{L}(\Gamma) \text{ formulae, then} & \\
\neg \phi, \phi \land \psi, \phi \land \psi, & \text{ are } \mathcal{L}(\Gamma) \text{ formulae.}
\end{align*}
\]

Note that the symbols in \( \Gamma \) (i.e., the non-logical symbols) are the only thing that distinguishes one first-order language \( \mathcal{L}(\Gamma) \) from another.

As variables are sorted, in the quantified formula \( \forall(x_1, \ldots, x_n) : p. \phi \), the \( U \)-operator \( p \) with arity \( \tau_1 \times \cdots \times \tau_n \) implicitly defines the sorts of variables \( x_1, \ldots, x_n \) to be \( \tau_1, \ldots, \tau_n \), with \( \tau_i \in S \), respectively. For terms \( t_1, \ldots, t_n \), we say that \( p(t_1, \ldots, t_n) \) is well-sorted if the sort of every \( t_i \) is \( \tau_i \). This notion is inductively applicable to terms. Moreover, we consider only well-sorted formulae and refer to the set of all well-sorted \( \mathcal{L}(\Gamma) \) formulae over a signature \( \Gamma \) in terms of \( \text{LTL}^\Gamma \). When a specific signature \( \Gamma \) is either irrelevant or clear from the context, we will simply write \( \text{LTL}^\Gamma \) instead. When convenient and a certain index is of no importance in the given context, we also shorten notation of a vector \((x_1, \ldots, x_n)\) by a (bold) \( x \).

A \( \Gamma \)-structure, or just (first-order) structure is a pair \( \mathfrak{A} = (|\mathfrak{A}|, I) \), where \( |\mathfrak{A}| = [|\mathfrak{A}|_1 \cup \ldots \cup |\mathfrak{A}|_n] \), is a non-empty set called domain, s.t. every sub-domain \( |\mathfrak{A}|_i \) (one for each sort, pairwise disjoint) is either a non-empty finite or countable set (e.g., set of all integers or strings) and \( I \) an interpretation. The purpose of \( I \) is to assign to each sort \( \tau_i \in S \) a specific sub-domain \( \tau^I_i = |\mathfrak{A}|_i \), to each function symbol \( f \in F \) of arity \( \tau_1 \times \cdots \times \tau_l \rightarrow \tau_m \) a function \( f^I : |\mathfrak{A}|_1 \times \cdots \times |\mathfrak{A}|_l \rightarrow |\mathfrak{A}|_m \), and to every \( I \)-operator \( r \) with arity \( \tau_1 \times \cdots \times \tau_n \) a relation \( r^I \subseteq |\mathfrak{A}|_1 \times \cdots \times |\mathfrak{A}|_m \). We restrict ourselves, if not explicitly stated, to computable structures [cf. Harizanov 1998]; that is, there must be a computable enumeration of \( |\mathfrak{A}| \) and relations and functions must be computable (i.e., \( f, r \) are computable if there is a procedure which effectively computes the result of \( f^I \), or answers whether \( r^I \subseteq r^I \) or \( r^I \notin r^I \) in a finite number of steps, respectively). In that regard, we can think of \( I \) as a mapping between \( I \)-operators (resp. function symbols) and the corresponding algorithms which compute the desired return values, each conforming to the symbols’ respective arities. Note that the interpretation of \( U \)-operators is rather different from \( I \)-operators, as it is closely tied to what we call a trace and therefore discussed after we introduce the necessary notions and notation.

We model observed system behaviour in terms of actions: Let \( p \in U \) with arity \( \tau_1 \times \cdots \times \tau_m \) and \( d \in D_p = [|\mathfrak{A}|_1 \times \cdots \times |\mathfrak{A}|_m] \), then we call \((p, d)\) an action. We refer to finite sets of actions as events. A system’s behaviour is therefore a finite trace of events, which we also denote as a sequence of sets of ground terms \( \{\text{sms}(1234)\} \{\text{login}(“user”)\} \ldots \) when we mean the sequence of tuples \( \{(\text{sms}, 1234)\} \{(\text{login}, “user”)\} \ldots \) Therefore the occurrence of some action \( \text{sms}(1234) \) in the trace (which can be informally seen as a parameterised proposition) at position \( i \in \mathbb{N}_0 \), written \( \text{sms}(1234) \in w_i \), indicates that, at time \( i \), \( \text{sms}(1234) \) holds (or, from a practical point of view, a Short Message Service (SMS) message was sent to number 1234 on a mobile phone). We follow the assumption that only symbols from \( U \) appear
§3.1 LTL\textsuperscript{FO}—Formal definitions and notation

in a trace, which therefore gives these symbols their respective interpretations. The following formalises this notion.

A \textit{(first-order) temporal structure} is a tuple \((A, w)\), where \(A = (|A_0|, I_0)(|A_1|, I_1)\) \ldots is a (possibly infinite) sequence of first-order structures and \(w = w_0w_1\ldots\) a corresponding trace. We demand that for all \(A_i\) and \(A_{i+1}\) from \(A_i\), it is the case that \(|A_i| = |A_{i+1}|\), for all \(f \in F\), \(f^{l_{i+1}} = f^{l_i}\), and for all \(\tau \in S\), \(\tau^{l_i} = \tau^{l_{i+1}}\). For any two structures, \(A\) and \(A'\), which satisfy these conditions, we write \(A \sim A'\). Moreover given some \(A\) and \(A\), if for all \(A\) from \(A\), we have that \(A_i \sim A\), we also write \(A_i \sim A\). In other words, the latter notation states that the same domain as well as interpretation of functions and sorts, defined in \(A\), is used throughout all first-order structures of the sequence \(A\). Finally, the interpretation of a \(U\)-operator \(p\) with arity \(\tau_1 \times \ldots \times \tau_m\) is then defined wrt. a position \(i\) in \(w\) as \(p^i = \{d | (p, d) \in w_i\}\). Essentially this means that, unlike function symbols, \(U\)- and \(I\)-operators do not have to be rigid.

Note also that from this point forward, we consider only the case where the specification to be monitored is given as a \textit{closed} formula (i.e., a sentence), which means it contains no free variables. If a variable occurrence is not free we call it \textit{bound}. Furthermore, if \(x\) is the tuple of all the free variables of \(\varphi\), we write \(\varphi(x)\). This is closely related to our means of quantification: a quantifier in LTL\textsuperscript{FO} is restricted to those elements that appear in the trace, and not arbitrary elements from a (possibly infinite) domain. While certain policies cannot be expressed with this restriction (e.g., "for all phone numbers \(x\) that are not in the contact list, \(r(x)\) is true"), this restriction bears the advantage that, when examining a given trace, functions and relations are only ever evaluated over known objects. The advantages of this type of quantification in monitoring first-order languages have also been pointed out in Halle and Villemaire [2008] and Bauer et al. [2009b]. In other words, had we allowed free variables (i.e., quantification over arbitrary domains), a monitor might end up having to "try out" all the different domain elements in order to evaluate such policies, which runs counter to our design rationale of quantification.

\textbf{Definition 3.1.2 (Semantics).} Let us fix a particular signature \(\Gamma\). The semantics of LTL\textsuperscript{FO} can now be defined wrt. a quadruple \((A, w, v, i)\) as follows, where \(i \in \mathbb{N}_0\), and \(v\) is an (initially empty) set of valuations assigning domain values to variables:

\begin{align*}
(\overline{A}, w, v, i) &\models p(t_1, \ldots, t_n) \iff (t^i_1, \ldots, t^i_n) \in p^i, \\
(\overline{A}, w, v, i) &\models r(t_1, \ldots, t_n) \iff (t^i_1, \ldots, t^i_n) \in r^i, \\
(\overline{A}, w, v, i) &\models \neg \varphi \iff (\overline{A}, w, v, i) \not\models \varphi, \\
(\overline{A}, w, v, i) &\models \varphi \land \psi \iff (\overline{A}, w, v, i) \models \varphi \text{ and } (\overline{A}, w, v, i) \models \psi, \\
(\overline{A}, w, v, i) &\models \varphi \text{ iff } |w| > i \text{ and } (\overline{A}, w, v, i + 1) \models \varphi, \\
(\overline{A}, w, v, i) &\models \varphi \text{ iff there is a } k \text{ s.t. } i \leq k < |w|, (\overline{A}, w, v, k) \models \psi, \\
&\text{ and } (\overline{A}, w, v, j) \models \varphi \text{ for all } i \leq j < k, \\
(\overline{A}, w, v, i) &\models \forall(x_1, \ldots, x_n) : \exists \varphi \text{ iff for all } (p, d_1, \ldots, d_n) \in w_i, \\
&\text{ and } (\overline{A}, w, v) \cup \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}, i) \models \varphi,
\end{align*}

where terms are evaluated inductively, and \(x^i\) treated as \(v(x)\).

If \((\overline{A}, w, v, 0) \models \varphi\), we write \((\overline{A}, w, v) \models \varphi\), and if \(v\) is irrelevant or clear from
the context, \( (\mathfrak{A}, w) \models \varphi \). Later we will also make use of the (possibly countably infinite) set of all actions (resp. events) wrt. \( \mathfrak{A} \), given as
\[
(\mathfrak{A})\text{-Act} = \bigcup_{p \in \mathbb{U}} \{(p, d) \mid d \in D_p\} \quad \text{(resp. } (\mathfrak{A})\text{-Ev} = 2^{\text{Act}}) \]
and take the liberty to omit the trailing \( (\mathfrak{A}) \) whenever a particular \( \mathfrak{A} \) is either irrelevant or clear from the context. We can then describe the generated language of \( \varphi \), \( \mathcal{L}(\varphi) \) (or simply the language of \( \varphi \), i.e., the set of all logical models of \( \varphi \)) compactly as
\[
\mathcal{L}(\varphi) := \{(\mathfrak{A}, w) \mid w_i \in \text{Ev and } (\mathfrak{A}, w) \models \varphi\},
\]
although, as before, we shall only ever concern ourselves with either infinite- or finite-trace languages, but not mixed ones (and like for the finite and infinite LTL semantics, we distinguish those if necessary also for LTL\textsuperscript{FO} with a subscript, i.e., write \( \models_F \) for the former and \( \models_\omega \) for the latter, respectively). Finally, we will use common syntactic “sugar”, which was mentioned already in §2.4 for LTL, and additionally
\[
\exists (x_1, \ldots, x_n) : p. \varphi := \neg (\forall (x_1, \ldots, x_n) : p. \neg \varphi).
\]

**Example 3.1.1.** See [Bauer, Küster, and Vegliach, 2012], [Küster and Bauer, 2015] and §6.3 for various example policies formalised in LTL\textsuperscript{FO}. However, to give in this section already an intuition, let us pick up the idea of monitoring Android apps again, and specify that these must not send SMS messages to numbers not in a user’s contact database. Assuming there exists a \( \mathbb{U} \)-operator \( \text{sms} \), which is true (i.e, appears in the trace), whenever an app sends an SMS message to phone number \( x \), we could formalise said policy in terms of \( \mathsf{G} \forall x : \text{sms}. \text{contact}(x) \). Note how in this formula the meaning of \( x \) is given implicitly by the arity of \( \text{sms} \) and must match the definition of the \( \mathbb{I} \)-operator \( \text{contact} \) in each event, i.e., \( x \) is not just any domain element, but a numerical value, which \( \text{sms} \) uses to capture the phone number associated with an incoming SMS. Also note how \( \text{sms} \) itself is interpreted indirectly via its occurrence in the trace, whereas \( \text{contact} \) never appears in the trace, even if true. The \( \mathbb{I} \)-operator \( \text{contact} \) can be thought of as interpreted via a program that queries a user’s contact database, whose contents may change over time.

### 3.2 Complexity of monitoring

This section discusses and compares the complexity of monitoring in the propositional and first-order case, based on LTL and LTL\textsuperscript{FO}, respectively.

#### 3.2.1 Propositional case

As there are no commonly accepted rules for what qualifies as a monitor (not even in the runtime verification community), we have explicitly argued in §1.2.2 what properties a runtime monitor should fulfil. However, as a consequence, there exist in the literature a myriad of different approaches to checking that an observed behaviour satisfies (resp. violates) a formal specification. Some of these [cf. Havelund and Rosu, 2004; Bauer et al., 2009b] consist in solving the word problem (see Definition 3.2.1). A monitor following this idea can either be an offline monitor, i.e., first record the entire system behaviour in form of a finite trace \( u \in \Sigma^* \), or an online monitor, i.e., process
the events incrementally as they are emitted by the system under scrutiny (recall §1.2.2, Property 1). Both approaches are documented in the literature [cf. Havelund and Rosu, 2004; Genon et al., 2006; Hallé and Villemaire, 2008; Bauer et al., 2009b; Basin et al., 2014; Maler and Nickovic, 2004; D’Angelo et al., 2005], but only the second one is suitable to properly monitor reactive systems running in production and detect property violations (resp. satisfaction) right when they occur.

Definition 3.2.1 (Word problem). The word problem for LTL is defined as follows.

Input: A formula $\varphi \in \text{LTL}(\text{AP})$ and some trace $u \in (2^\text{AP})^+$.  

Question: Does $[u| = \varphi]_F$ hold?

Markey and Schnoebelen [2003] presented a bilinear algorithm for this problem (an even more efficient solution was given by Kuhtz and Finkbeiner [2012]). Hence, the first sort of monitor (offline monitor), which is really more of a test oracle than a monitor, solves a classical decision problem [cf. Garey and Johnson, 1979]. A test oracle is a mechanism known from testing [Richardson et al., 1992]. It checks whether the output for all test-case executions of a system (i.e., a finite set of finite input sequences) is correct. Therefore, in terms of runtime verification the oracle works like an offline monitor.

The second sort of monitor (online monitor), in which we are ultimately interested in this thesis, however, solves an entirely different kind of problem; it cannot be stated in complexity-theoretical terms at all (also pointed out by Baader and Lippmann [2014]): Its input is an LTL formula and a finite albeit unbounded trace which grows incrementally. This means that this monitor solves the word problem for each and every new event that is added to the trace at runtime over and over again. It solves a sequence of decision problems

$$(\varepsilon, \varphi), (\sigma_0, \varphi), (\sigma_0\sigma_1, \varphi), (\sigma_0\sigma_1\sigma_2, \varphi), \ldots,$$

where $(\sigma_0 \ldots \sigma_n, \varphi)$, is the input to the word problem. We can therefore say that the word problem acts as a lower bound on the complexity of the monitoring problem that such a monitor solves; or, in other words, the problem that the online monitor solves is at least as hard as the problem that the offline monitor solves. As one clearly wants not answer the sequence of decision problems indicated above independently from each other, there are approaches to build efficient (i.e., trace-length independent) monitors that repeatedly answer the word problem [cf. Havelund and Rosu, 2004], and which comply with Property 2 in §1.2.2.

However, such online approaches violate Property 3 in §1.2.2 in that they are necessarily non-monotonic. Recall that monotonicity is not a concern when monitoring finite, complete traces, so that solving the word problem is actually what is desired in offline monitoring. To see this, consider $\varphi = aU b$ and some trace $u = \{a\}\{a\} \ldots \{a\}$ of length $n$. Using the finite-trace interpretation in Def. 2.4.2 [or the strong interpretation in §2.5], $[u \not|= \varphi]_F$. However, if we add $u_{n+1} = \{b\}$, we get $[u \models \varphi]_F$.  

\footnote{Note that this effect is not particular to the choice of finite-trace interpretation, but lies rather in the nature of monitoring truncated traces and using 2-valued logics to monitor them. Had we used, e.g.,}
the users, this essentially means that they cannot trust the verdict of the monitor as it may flip in the future, unless of course it is obvious from the start that, e.g., only safety properties are monitored and the monitor is built merely to detect violations, i.e., bad prefixes [cf. Basin et al. 2015b]. However, if we take other monitorable languages into account as we do in this thesis, i.e., those that have either good or bad prefixes (or both) (see Def. 3.2.2), we need to distinguish between satisfaction and violation of a property (and want the monitor to report either occurrence truthfully).

**Definition 3.2.2** (Good and bad prefixes [Kupferman and Vardi 2001]). Given a language \( L \subseteq \Sigma^\omega \) of infinite words over \( \Sigma \), a finite word \( u \in \Sigma^* \) is called

- a **good** prefix of \( L \), if for all \( w \in \Sigma^\omega \) it holds that \( uw \in L \),
- a **bad** prefix of \( L \), if for all \( w \in \Sigma^\omega \) it holds that \( uw \notin L \).

We shall use \( \text{good}(L) \subseteq \Sigma^* \) (resp. \( \text{bad}(L) \)) to denote the set of good (resp. bad) prefixes of \( L \). For readability, we also write \( \text{good}(\varphi) \) instead of \( \text{good}(L(\varphi)) \), and do the same for \( \text{bad}(L(\varphi)) \). A good (resp. bad) prefix \( u \) is **minimal** if every prefix \( v \) of \( u \) that is strictly shorter (i.e., \(|v| < |u|\)) is not a good (resp. bad) prefix anymore. Note that every finite continuation of \( uv \) is a good (resp. bad) prefix as well.

A monitor that detects good (resp. bad) prefixes has been termed impartial (recall §1.2.2, Property 4) as it not only states something about the past, but also about the future: once a good (resp. bad) prefix has been detected, no matter how the system would evolve in an indefinite future, the property would remain satisfied (resp. violated). In that sense, impartial monitors are monotonic by definition. A further monitor characteristic is anticipation (§1.2.2, Property 5), which demands detection of minimal good or bad prefixes. In other words, an impartial but not anticipatory monitor is allowed to return ? even though a (resp. bad) prefix has been observed. While Bauer et al. [2011] give a construction that yields a trace-length independent (even optimal) impartial and anticipatory monitor for an LTL formula as well as a timed extension called TLTL, we shall see in §3.2.2 that obtaining anticipatory monitors for first-order temporal specifications is generally impossible. The obtained monitor for LTL specifications basically returns \( \top \) to the user if \( u \in \text{good}(\varphi) \) holds, \( \bot \) if \( u \in \text{bad}(\varphi) \) holds, and ? otherwise; and as such agrees with the LTL3 semantics \([u |= \varphi]_3 \) stated in Def. 2.5.1. Not surprisingly though, the monitoring problem such a monitor solves is computationally more involved than the word problem. It solves what we call the prefix problem (of LTL), which can easily be shown PSpace-complete by way of LTL satisfiability.

**Definition 3.2.3** (Prefix problem). The prefix problem for LTL is defined as follows.

**Input:** A formula \( \varphi \in \text{LTL}(\text{AP}) \) and some trace \( u \in (2^{\text{AP}})^* \).

**Question:** Does \( u \in \text{good}(\varphi) \) (resp. \( \text{bad}(\varphi) \)) hold?

what is known as the weak finite-trace semantics, discussed in §2.5 we would first have had \([u |= \varphi]_F \) and if \( u_{n+1} = \emptyset \), subsequently \([u \neq \varphi]_F \).

Safety properties are those of which all counterexamples exhibit a bad prefix [cf. Alpern and Schneider 1985]. This is an important characteristic exploited in runtime verification, since all violations of safety properties can be detected by finite prefixes.
Theorem 3.2.1. The prefix problem for LTL is PSpace-complete.

Proof. Let us first focus on bad prefixes. It is easy to see that $u = u_0 \ldots u_n \in \text{bad}(\varphi)$ iff $L(u_0 \land Xu_1 \land XXu_2 \land \ldots X^nu_n \land \varphi) = \emptyset$, where $X^n$ means the next operator written down $n$ times. Constructing this conjunction takes polynomial time and the corresponding emptiness check can be performed in PSpace [Sistla and Clarke, 1985]. For hardness, we proceed with a reduction of LTL satisfiability. Again, it is easy to see that $L(\varphi) \neq \emptyset$ iff $\sigma \notin \text{bad}(X\varphi)$ for any $\sigma \in 2^{2AP}$. This reduction is linear, and as PSpace = co-PSpace [cf. Papadimitriou, 1994], the statement follows.

The proof is symmetric for good prefixes, i.e., $u = u_0 \ldots u_n \in \text{good}(\varphi)$ iff $L(u_0 \land Xu_1 \land XXu_2 \land \ldots X^nu_n \land \neg \varphi) = \emptyset$, and $L(\varphi) \neq \emptyset$ iff $\sigma \notin \text{good}(X\neg \varphi)$ for any $\sigma \in 2^{2AP}$.

Note that the automaton-based LTL$_3$ monitor of Bauer et al. [2011] even processes every incoming event in constant time, and therefore has a minimal runtime overhead (§1.2.2, Property 6). Other LTL monitors, based on rewriting of formulae (see for example the first five basic rewriting rules in Bacchus and Kabanza [1998, Table 1], which can be used for LTL if the interval is $[0, \infty)$), are trace-length independent, and can also achieve anticipation by using a validity checker [cf. Havelund and Rosu, 2004]. The checker must be called on the rewritten formula after each processed event [Rosu and Havelund, 2005]. However, this means solving a PSpace-complete problem each time, which disagrees with our minimal runtime overhead requirement. Even without the validity check, the runtime overhead of rewriting based approaches is in the worst case exponential with respect to size of the formula [Rosu and Havelund, 2005]. Bauer et al. [2011] cannot avoid an exponential overhead either (since we know the prefix problem is PSpace-complete), but for them it occurs while generating the monitor—ahead of monitoring. In general, in runtime verification we prefer the overhead during generation, as it is not recurring. On the other hand the rewriting rules have the major advantage that their generation effort is constant. This becomes a real disadvantage for automaton-based approaches if monitors are intractable to create from large formulae.

Finally, we would like to point out the possibility of building an impartial and anticipatory though trace-length dependent LTL monitor using an “off the shelf” model checker, which accepts a propositional Kripke structure (see Def. 2.2.1) and an LTL formula as input.

Definition 3.2.4. The model checking problem for LTL is defined as follows.

Input: A formula $\varphi \in \text{LTL}(AP)$ and a Kripke structure $\mathcal{K}$ over $2^{2AP}$.

Question: Does $L(\mathcal{K}) \subseteq L(\varphi)$ hold?

As in LTL the model checking and the satisfiability problems are both PSpace-complete [Sistla and Clarke, 1985], we can use a model checking tool as monitor: given that it is straightforward to construct $\mathcal{K}$, s.t. $L(\mathcal{K}) = \{uw \mid w \in (2^{2AP})^w\}$, in no more than polynomial time; that is, by adding to the last state of a linear Kripke structure wrt. $u$, self-looping transitions for every subset of $2^{2AP}$. Then we return $\top$ to the user if $L(\mathcal{K}) \subseteq L(\varphi)$ holds, $\bot$ if $L(\mathcal{K}) \subseteq L(\neg \varphi)$ holds, and $?$ if neither
holds. One could therefore be tempted to think of monitoring merely in terms of a model checking problem, but we shall see that as soon as the logic in question has an undecidable satisfiability problem this reduction fails. Besides, it can be questioned whether monitoring as model checking leads to a desirable monitor with its obvious trace-length dependence and having to repeatedly solve a PSpace-complete problem for each new event. See also Schnoebelen [2002] for a survey of complexity results for model checking other temporal logics than LTL.

### 3.2.2 First-order case

First and foremost, LTL\(_{FO}\) as defined in §3.1 is undecidable as can be shown by way of the following lemma. It basically helps us reduce finite satisfiability of standard first-order logic to LTL\(_{FO}\).

**Definition 3.2.5** (Finite structure). A structure A is called finite if its domain \(|A|\) is a finite set [cf. Libkin 2004, Definition 2.1].

**Lemma 3.2.1.** Let \(\varphi\) be a sentence in first-order logic, then we can construct a corresponding \(\psi \in \text{LTL}_{FO}\) s.t. \(\varphi\) has a finite model iff \(\psi\) is satisfiable.

*Proof idea (full proof on page 124).* By constructing a \(\psi\) in LTL\(_{FO}\) (which is essentially a formula of first-order logic, i.e., without temporal operators) in such a way that a model \((A, \sigma)\) of \(\psi\) contains a \((d, e) \in \sigma\) for every domain value \(e\) involved in satisfying \(\psi\) (and where \(d\) is a freshly introduced U-operator). The finitely many \(e\)’s then make up the domain satisfying \(\varphi\).

**Theorem 3.2.2.** LTL\(_{FO}\) is undecidable.

*Proof.* Assume it is decidable whether \(\varphi \in \text{LTL}_{FO}\) is satisfiable. Let \(\varphi\) be a sentence in first-order logic, then by Lemma 3.2.1, we can construct some \(\psi \in \text{LTL}_{FO}\), s.t. \(\varphi\) has a finite model iff \(\psi\) is satisfiable, so that we can test whether \(\varphi\) has a finite model. However, by Trakhtenbrot’s Theorem, testing if \(\varphi\) has a finite model is generally undecidable [cf. Libkin 2004, §9]. Contradiction.

Alternatively, undecidability can be shown by reducing the halting problem of a deterministic Turing machine to the satisfiability problem of LTL\(_{FO}\), as shown by Lemma A.1.1 in the appendix (full proof on page 125).

Let us now define what is meant by Kripke structures in our new setting and the generated language of them. As in Def. 2.2.1, Kripke structures we consider either give rise to infinite-trace languages (i.e., have a left-total transition relation), or represent finite traces (i.e., each state has at most one successor and the transition relation is loop-free). We restrict to the more general definition of the former. Note that we will also skip detailed redefinitions of the decision problems discussed in §3.2.1 since the concepts transfer in a straightforward manner.

**Definition 3.2.6** (First-order Kripke structure). Given some A, a (A)-Kripke structure, or just first-order Kripke structure, is a tuple \(K_{FO} = (S, s_0, \lambda, \rightarrow)\), defined as in Def. 2.2.1 but with \(\lambda : S \rightarrow \hat{A} \times \text{Ev}\), where \(\hat{A} = \{\hat{A}' \mid A' \sim A\}\).
§3.2 Complexity of monitoring

Definition 3.2.7 (First-order Kripke language). For a \((\mathfrak{A})\)-Kripke structure \(K_{FO}\) with states \(s_0, \ldots, s_n\), its generated language is \(L(K_{FO}) = \{ (\mathfrak{A}, w) \mid (\mathfrak{A}_0, w_0) = \lambda(s_0) \text{ and for all } i \in \mathbb{N} \text{ there is some } j, k \in \{0, \ldots, n\} \text{ s.t. } (\mathfrak{A}_i, w_i) = \lambda(s_j), (\mathfrak{A}_{i-1}, w_{i-1}) = \lambda(s_k) \text{ and } (s_k, s_j) \in \rightarrow \}\).

The inputs to the LTL\(^{FO}\) word problem are therefore an LTL\(^{FO}\) formula and a linear first-order Kripke structure, representing a finite input trace. Unlike in standard LTL,

Theorem 3.2.3. The word problem for LTL\(^{FO}\) is PSpace-complete.

Proof idea (full proof on page 127). Membership by a depth-first search according to the inductive definition of LTL\(^{FO}\). Hardness by reducing the Quantified Boolean Formula Problem [cf. Garey and Johnson, 1979; Hopcroft and Ullman, 1979, §11.3.2-11.3.4] to the word problem of LTL\(^{FO}\).

The inputs to the LTL\(^{FO}\) model checking problem, in turn, are a left-total first-order Kripke structure, which gives rise to an infinite-trace language, and an LTL\(^{FO}\) formula.

Theorem 3.2.4. The model checking problem for LTL\(^{FO}\) is in ExpSpace.

Proof idea (full proof on page 127). For a given \(\varphi \in LTL^{FO}\) and \((\mathfrak{A})\)-Kripke structure \(K_{FO}\) defined as usual, where \(\mathfrak{A} = (|\mathfrak{A}|, I)\), we construct a propositional Kripke structure \(K'\) and \(\varphi' \in LTL\), s.t. \(L(K_{FO}) \subseteq L(\varphi) \iff L(K') \subseteq L(\varphi')\) holds.

The reason for this result is that we can devise a reduction of that problem to LTL model checking in exponential space. While the PSpace-lower bound is easy, e.g., via reduction of the LTL\(^{FO}\) word problem, we currently do not know how tight these bounds are and, therefore, leave this as an open problem. Note also that the results of both Theorem 3.2.3 and Theorem 3.2.4 are obtained even without taking into account the complexities of the interpretations of function symbols and I-operators; that is, for these results to hold, we assume that interpretations do not exceed polynomial, resp. exponential space.

We have seen in §3.2.1 that the prefix problem lies at the heart of an impartial monitor. While in LTL it was possible to build an impartial and anticipatory monitor using a model checker (albeit a very inefficient one), the following shows that this is no longer possible.

Lemma 3.2.2. Let \(\mathfrak{A}\) be a (first-order) computable structure and \(\varphi \in LTL^{FO}\), then \(L(\varphi)_\mathfrak{A} = \{ (\mathfrak{A}, w) \mid \mathfrak{A} \sim \mathfrak{A}, w \in Ev^\omega, \text{ and } (\mathfrak{A}, w) \models \varphi \}\). Testing if \(L(\varphi)_\mathfrak{A} \neq \emptyset\) is generally undecidable.

Proof idea (full proof on page 129). By a reduction from Post’s Correspondence Problem [cf. Post, 1946; Hopcroft and Ullman, 1979, §9.4].
Theorem 3.2.5 (Prefix problem). The prefix problem for LTL$^{FO}$ is undecidable.

Proof idea (full proof on page [29]). By way of a similar reduction as in Theorem 3.2.1, but here based on Lemma 3.2.2, i.e., $(\mathcal{A}, \sigma) \in \text{bad}(X \varphi)$ iff $L(\varphi)_{\mathcal{A}} = \emptyset$ for any $\sigma \in \text{Ev}$.

There are essentially two ways trying to achieve a decidable prefix problem: Restricting the structure (i.e., the interpretations of predicate symbols allowed) or syntax of formulae that can be used for monitoring. However, from Corollary 3.2.1 can be concluded that seeking decidable fragments under computational structures is an impossible task. Already a single existential quantifier and unary predicate symbol make the problem be undecidable.

Corollary 3.2.1. The prefix problem for LTL$^{FO}$ under computable structures with only a single existential quantifier, and one unary interpreted predicate symbol, is undecidable.

Proof. Follows from constructed formula in the proof of Lemma 3.2.2

If we do not allow any predefined predicates or functions, but do not restrict the syntax, the prefix problem is also undecidable.

Corollary 3.2.2. LTL$^{FO}$, where $\Gamma$ is a relational signature (i.e., $\Gamma$ contains only U-operators and constants), with at least one binary U-operator, is undecidable.

Proof. Follows directly from Lemma 3.2.1 and Trakhtenbrot’s Theorem, which holds for relational signatures with at least one binary relation.

Note that the prefix problem for finite structures is decidable. To show this, a model checker can be used as a monitor, similarly as for LTL in §3.2.1; that is, we construct a linear first-order Kripke structure, with a self-loop in the last state for every subset of $\{(p,d) \mid p \in U, \text{ and } d \in \mathcal{A}_0 \times \ldots \times \mathcal{A}_n\}$.

3.3 Summary

The first part of this chapter formally defines LTL$^{FO}$, a natural extension of LTL with first-order quantification. Quantifiers are restricted to bind only data values from actions appearing in the trace (coming from an possible infinite and “external” source). This trace-centred view of quantification makes LTL$^{FO}$ arguably a natural formalism to use for the purpose of runtime verification. The second part of the chapter details on complexity results revolving around monitoring in LTL as well as LTL$^{FO}$. Table 3.1 summarises the results. While the word problem (which is the underlying decision problem to be solved in offline monitoring) is shown to be PSpace-complete in LTL$^{FO}$, the prefix problem (to be solved by an online monitor), is undecidable. However, a decision procedure for tackling the prefix problem is subject of the next chapter.
Table 3.1: Overview of complexity results.

<table>
<thead>
<tr>
<th></th>
<th>LTL</th>
<th>LTL\textsuperscript{fo}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Satisfiability</strong></td>
<td>PSpace-complete</td>
<td>Undecidable</td>
</tr>
<tr>
<td><strong>Word problem</strong></td>
<td>&lt; Bilinear-time</td>
<td>PSpace-complete</td>
</tr>
<tr>
<td><strong>Model checking</strong></td>
<td>PSpace-complete</td>
<td>ExpSpace-membership, PSpace-hard</td>
</tr>
<tr>
<td><strong>Prefix problem</strong></td>
<td>PSpace-complete</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>

Experiments of the next chapter also show that a decidable prefix problem is merely of theoretical interest, as one usually does not encode undecidable problems into formulae to be answered at runtime.
Chapter 4

Monitoring algorithm

A corollary of Theorem 3.2.5 is that there cannot exist a complete monitor for LTL$^{FO}$-definable infinite trace languages. Yet the main contributions of this chapter is to show that one can build a sound and efficient LTL$^{FO}$ monitor that tackles the prefix problem, using a new kind of automaton. Before we go into the details of the actual monitoring algorithm (§4.2), let us first consider the model of an SA (§4.1), which potentially “spawn” a positive Boolean combination of “children SAs” (i.e., subautomata) in each such step. The entire number of subautomata is infinite wrt. a trace, thus those cannot be precomputed, unlike a BA for LTL. However, the symbolic structure of subautomata (i.e., their states and transitions containing variables, but not assignments) can be generated prior to commencing monitoring. The structure helps to efficiently decide what needs to be stored from events at runtime (i.e., by spawning the right amount of SAs in each step and assigning values of ground atoms to variables in states), and what can be “garbage collected” (i.e., by pruning entire subautomata).

We then consider optimisations to reduce the state space of the monitor (§4.3) and experimental results of its implementation (§4.4). Finally, a comprehensive survey of related monitoring approaches in the literature (i.e, those that allow reasoning over traces with data in a broader sense) is given (§4.5).

4.1 Spawning automaton

First, we provide the general definition of an SA and then detail on the construction for LTL$^{FO}$ formulae. For an intuitive understanding of the new automaton model, see Example 4.1.1.

4.1.1 General definition

For the output of a spawning function in Def. 4.1.1, we need to introduce the concept of a positive Boolean formula: Let $B^+(X)$ denote the set of all positive Boolean formulae over the set $X$ (i.e., Boolean formulae that are created from $X$ and use only $\land$ and $\lor$, but not $\neg$). We say that some set $Y \subseteq X$ satisfies a formula $\beta \in B^+(X)$,
written \( Y \models \beta \), if the truth assignment that assigns true to all elements in \( Y \) and false to all \( X - Y \) satisfies \( \beta \).

**Definition 4.1.1** (Spawning automaton). An **SA** is given by \( \mathcal{A} = (\Sigma, l, Q, Q_0, \delta_{\rightarrow}, \delta_{\downarrow}, F) \), where

- \( \Sigma \) is a countable set called alphabet,
- \( l \in \mathbb{N}_0 \) the level of \( \mathcal{A} \),
- \( Q \) a finite set of states,
- \( Q_0 \subseteq Q \) a set of distinguished initial states,
- \( \delta_{\rightarrow} \) a transition relation,
- \( \delta_{\downarrow} \) what is called a spawning function, and
- \( F = \{ F_1, \ldots, F_n \mid F_i \subseteq Q \} \) an acceptance condition (to be defined later on).

We have \( \delta_{\rightarrow} : Q \times \Sigma \rightarrow 2^Q \) and \( \delta_{\downarrow} : Q \times \Sigma \rightarrow B^+ (\mathcal{A}^{<l}) \), where \( \mathcal{A}^{<l} = \{ \mathcal{A}' \mid \mathcal{A}' \text{ is an SA with level less than } l \} \).

An **SA** can be seen as an extension of a **GBA** (see Def. 2.3.9), where the level and the spawning function is added to the usual 5-tuple. The spawning function of an SA with level \( l \) spawns only Boolean combinations of "children SAs" that have a strictly lower level than \( l \).

**Definition 4.1.2** (Run of an SA). A run of an SA, \( \mathcal{A} \), over input \( w \in \Sigma^\omega \) is a mapping \( \rho : \mathbb{N}_0 \rightarrow Q \), s.t. \( \rho(0) \in Q_0 \) and \( \rho(i + 1) \in \delta_{\rightarrow}(\rho(i), w_i) \) for all \( i \in \mathbb{N}_0 \). When clear from the context, we just say "run".

**Definition 4.1.3** (Locally accepting run). A run \( \rho \) of an SA, \( \mathcal{A} \), is locally accepting if \( \text{Inf}(\rho) \cap F_i \neq \emptyset \) for all \( F_i \in F \), where \( \text{Inf}(\rho) \) denotes the set of states visited infinitely often.

**Definition 4.1.4** (Accepting run of an SA). A run \( \rho \) of \( \mathcal{A} \) over input \( w \in \Sigma^\omega \) is called accepting if \( l = 0 \) and it is locally accepting. If \( l > 0 \), \( \rho \) is called accepting if it is locally accepting and for all \( i \in \mathbb{N}_0 \) there is a set \( Y \subseteq \mathcal{A}^{<l} \), s.t. \( Y \models \delta_{\downarrow}(\rho(i), w_i) \) and all SAs \( \mathcal{A}' \in Y \) have an accepting run, \( \rho' \), over \( w^i \).

When clear from the context, we just say "accepting run". While not inherent in the automaton model, notice that the above defined local acceptance basically resembles a generalised Büchi acceptance. This means that the **SAs** with level 0 are equally expressive as **GBAs**.

While the **accepted language of an SA** \( \mathcal{A} \), \( L(\mathcal{A}) \), is defined as usual for automata seen in §2.3 (i.e., it consists of all \( w \in \Sigma^\omega \), for which \( \mathcal{A} \) has at least one accepting run), we additionally define the local language of \( \mathcal{A} \) as

\[
L_{\text{local}}(\mathcal{A}) := \{ w \in \Sigma^\omega \mid \text{there exists a local accepting run of } \mathcal{A} \text{ over } w \}.
\]
4.1.2 A construction for $\text{LTL}^{\text{FO}}$-formulae

Given some $\varphi \in \text{LTL}^{\text{FO}}$, let us now examine in detail how to build the corresponding SA, $A_\varphi = (\Sigma, I, Q, Q_0, \delta, \delta_i, F)$ s.t. $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$ holds. To this end, we set $\Sigma = \{(\mathfrak{A}, \sigma) \mid \sigma \in (\mathfrak{A})-\text{Ev}\}$. If $\varphi$ is not a sentence, we write $A_{\varphi,v}$ to denote the spawning automaton for $\varphi$ in which free variables are mapped according to a finite set of valuations $v[1]$. To define the set of states for an SA, we make use of a restricted subformula function, $sf|_v(\varphi)$, which is defined like a generic subformula function (Def. 4.1.5), except if $\varphi$ is of the form $\forall x : p. \, \psi$, we have $sf|_v(\varphi) = \{\varphi\}$.

**Definition 4.1.5** (Generic subformula function). The generic subformula function is a mapping, $sf : \text{LTL}^{\text{FO}} \rightarrow 2^{\text{LTL}^{\text{FO}}}$, which is inductively defined as follows:

- $sf(p(t)) = \{p(t)\}$,
- $sf(r(t)) = \{r(t)\}$,
- $sf(\neg \varphi) = \{\neg \varphi\} \cup sf(\varphi)$,
- $sf(\varphi \land \psi) = \{\varphi \land \psi\} \cup sf(\varphi) \cup sf(\psi)$,
- $sf(\varphi) = \{\varphi\} \cup sf(\varphi)$,
- $sf(\varphi \cup \psi) = \{\varphi \cup \psi\} \cup sf(\varphi) \cup sf(\psi)$, and
- $sf(\forall x : p. \, \varphi) = \{\forall x : p. \, \varphi\} \cup sf(\varphi)$.

This essentially means that an SA for a formula $\varphi$ on the topmost level looks like the GBA for $\varphi$, where quantified subformulae have been interpreted as atomic propositions.

For example, if $\varphi = \psi \land \forall x : p. \, \psi'$, where $\psi$ is a quantifier-free formula, then $A_{\varphi, v}$, at the topmost level $n$, is like the GBA for the LTL formula $\psi \land a$, where $a$ is an atomic proposition; or in other words, $A_\varphi$ handles the subformula $\forall x : p. \, \psi'$ separately in terms of a subautomaton of level $n - 1$ (see also definition of $\delta_1$ below).

Finally, we define the closure of $\varphi$ wrt. $sf|_v(\varphi)$ as

$$\text{cl}(\varphi) = \{\neg \psi \mid \psi \in sf|_v(\varphi)\} \cup sf|_v(\varphi),$$

i.e., the smallest set containing $sf|_v(\varphi)$, which is closed under negation. The set of states of $A_\varphi$, $Q$, consists of all complete subsets of $\text{cl}(\varphi)$; that is, a set $q \subseteq \text{cl}(\varphi)$ is complete iff
- for any $\psi \in \text{cl}(\varphi)$ either $\psi \in q$ or $\neg \psi \in q$, but not both; and
- for any $\psi \land \psi' \in \text{cl}(\varphi)$, we have that $\psi \land \psi' \in q$ iff $\psi \in q$ and $\psi' \in q$; and

---

1Considering free variables, even though the runtime policies can only ever be sentences, is necessary, because an SA for a formula $\varphi$ is inductively defined in terms of SAs for its subformulae (i.e., $A_\varphi$’s subautomata), some of which may contain free variables.
for any \( \psi U \psi' \in \text{cl}(\varphi) \), we have that if \( \psi U \psi' \in q \) then \( \psi' \in q \) or \( \psi \in q \), and if
\( \psi U \psi' \notin q \), then \( \psi' \notin q \).

Let \( q \in Q \) and \( \mathfrak{A} = (|\mathfrak{A}|, I) \). The transition function \( \delta_{\rightarrow}(q, (\mathfrak{A}, \sigma)) \) is defined iff

- for all \( p(t) \in q \), we have \( t^l \in p^l \) and for all \( \neg p(t) \in q \), we have \( t^l \notin p^l \),
- for all \( r(t) \in q \), we have \( t^l \in r^l \) and for all \( \neg r(t) \in q \), we have \( t^l \notin r^l \).

In which case, for any \( q' \in Q \), we have that \( q' \in \delta_{\rightarrow}(q, (\mathfrak{A}, \sigma)) \) iff

- for all \( X \psi \in \text{cl}(\varphi) \), we have \( X \psi \in q \) iff \( \psi \in q' \), and
- for all \( \psi U \psi' \in \text{cl}(\varphi) \), we have \( \psi U \psi' \in q \) iff \( \psi' \in q \) or \( \psi \in q \) and \( \psi U \psi' \in q' \).

This is similar to the well known syntax directed construction of GBA's \[\text{cf. Bate and Katoen, 2008, \S5.2}\], except that we also need to cater for quantified subformulae.

For this purpose, an inductive \textit{spawning function} is defined as follows. If \( l > 0 \), then \( \delta_1(q, (\mathfrak{A}, \sigma)) \) yields

\[
\left( \bigwedge_{\forall x: p. \psi \in q} \left( \bigwedge_{(p, d) \in \sigma} \mathcal{A}_{\psi, \psi'} \right) \right) \land \left( \bigwedge_{\neg \forall x: p. \psi \in q} \left( \bigvee_{(p, d) \in \sigma} \mathcal{A}_{\neg \psi, \psi'} \right) \right),
\]

where \( \nu' = \nu \cup \{x \mapsto d\} \) and \( \nu'' = \nu \cup \{x \mapsto d\} \) are sets of valuations, otherwise \( \delta_1(q, (\mathfrak{A}, \sigma)) \) yields \( \top \). Moreover, we set \( Q_0 = \{q \in Q \mid \varphi \in q\} \), \( F = \{F_{\psi U \psi'} \mid \psi U \psi' \in \text{cl}(\varphi)\} \) with \( F_{\psi U \psi'} = \{q \in Q \mid \psi' \in q \lor (\psi U \psi' \notin q)\} \), and \( l = \text{depth}(\varphi) \), where \( \text{depth}(\varphi) \) is called the \textit{quantifier depth} of \( \varphi \).

**Definition 4.1.6 (Quantifier depth).** The quantifier depth is a mapping, \( \text{depth} : \text{LTL}^{FO} \rightarrow \mathbb{N}_0 \), which is inductively defined as follows:

- \( \text{depth}(\varphi) = 0 \) iff \( \varphi \in \text{LTL}^{FO} \) is a quantifier free formula,
- \( \text{depth}(\forall x : p. \psi) = 1 + \text{depth}(\psi) \)
- \( \text{depth}(\varphi \land \psi) = \text{depth}(\psi U \psi) = \text{max}(\text{depth}(\varphi), \text{depth}(\psi)) \), and
- \( \text{depth}(\neg \varphi) = \text{depth}(X \varphi) = \text{depth}(\varphi) \) for all \( \varphi, \psi \in \text{LTL}^{FO} \).

**Correctness.** We now assert formally in Theorem 4.1.1 that for a given \( \varphi \in \text{LTL}^{FO} \), the constructed SA, \( A_{\varphi} \), as outlined above is correct in the sense that for any sentence \( \varphi \in \text{LTL}^{FO} \), we have that \( \mathcal{L}(A_{\varphi}) = \mathcal{L}(\varphi) \). The following lemma is needed in order to prove this statement.

**Lemma 4.1.1.** Let \( \varphi \in \text{LTL}^{FO} \) (not necessarily a sentence) and \( \nu \) be a valuation. For each accepting run \( \rho \) in \( A_{\varphi, \nu} \) over input \( (\mathfrak{A}, \nu) \), \( \psi \in \text{cl}(\varphi) \), and \( i \geq 0 \), we have that \( \psi \in \rho(i) \) iff \( (\mathfrak{A}, w, \nu, i) \models \psi \).

**Proof idea (full proof on page 129).** By nested induction on \( \text{depth}(\varphi) \) and the structure of \( \psi \in \text{cl}(\varphi) \).

\( \square \)
Theorem 4.1.1. The constructed SA is correct in the sense that for any sentence $\phi \in \text{LTL}^\Sigma$, we have that $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$.

Proof idea (full proof on page 132). $\subseteq$ by Lemma 4.1.1 The other direction uses induction on depth($\phi$).

Example 4.1.1. Consider the graphical representation of an SA for $\phi = G(\forall(u, ip) : \text{login. } ((\forall(u', ip') : \text{send. } (u = u') \Rightarrow (ip = ip')) \text{logout}(u, ip))$ in Fig. 4.1. For readability, note that we write the interpreted predicate for equality as infix notation. In a nutshell, $\phi$ specifies that once user $u$ has logged in to the system from Internet Protocol (IP) address $ip$, she must not send anything from an IP address other than $ip$ until logged out. While $\phi$ is not meant to represent a realistic security policy, it does help highlight the features of an SA: We first note that level 1 of $A_\phi$ is given by depth($\phi$) = 2. As $\phi$ is of the form $G\forall(u, ip) : \text{login}$. $\forall$, $A_\phi$'s state space is de facto that of an ordinary GBA for an LTL formula of the form $G\forall$.

Let us now assume that $\sigma = \{\text{login}(1,2.3.4.1), \text{login}(2,2.3.4.2), \text{send}(3,2.3.4.3), \text{send}(1,5.6.7.8)\}$ is an event, which we want $A_\phi$ to process. Due to $\phi$'s outmost quantifier, the two login-actions will lead to the spawning of a conjunction of two subautomata of respective levels $l - 1$ (downward dotted lines). The state space of these subautomata is de facto that of an ordinary GBA for an LTL formula of the form $aUb$ as one can see in Fig. 4.1 level 1. These SAs also keep track of a quantified formula, hence the two send-actions will also spawn a conjunction of subautomata, basically, to check if $(u = u') \Rightarrow (ip = ip')$ holds. The respective valuations are given below each SA, whereas the respective current states are marked in grey.

Let us now consider the accepting run of this SA. First note that the SA on level 2 has a locally accepting run over any trace. This is because its state does not contain any
predicates. However, spawned conjunctions based on the state must be satisfied for a run to be also accepting. In this example, the spawned conjunction of the initial state is satisfied if the SAs on level 1 both have an accepting run. Note that if $\phi$’s outermost quantifier were an existential quantifier, a disjunction would be spawned, so that only one of the SAs on level 1 would need to have an accepting run. Recursively, the spawned conjunctions of the SAs on level 1 must be satisfied. However, for the second SA from the left on level 0, there cannot exist any (locally) accepting run over $\sigma w$ for any $w \in \Sigma^\omega$. This is because the predicates in both initial states do not evaluate to true for the according send-action. It follows that the spawned conjunction of the left automaton on level 1 cannot be satisfied, and therefore, it cannot have an accepting run. Consequently, the SA on level 2 cannot have an accepting run over any trace starting with $\sigma$.

**Size of an SA.** Given an SA for $\phi \in \text{LTL}^{\text{FO}}$, the size of the top-most level as well as each spawned SA is $O(2^{\|\phi\|})$ (i.e., essentially the size of a GBA). Each SA spawns $O(|\phi| \cdot |\sigma|)$ different subautomata, where $\sigma$ is the current event and $O(|\phi|)$ the number of quantified subformulae. This means, in total the size of an SA at time 0 is bounded by $O((|\phi| \cdot |\sigma|)^{\text{depth}(\phi)} \cdot 2^{\|\phi\|})$. However, an SA is not a finite state space automaton; thus, it grows ad infinitum over time. A consequence is that a monitor cannot build naively based on this type of automaton but must prune spawned subautomata that are not needed for evaluation anymore.

### 4.2 Monitor construction

Before we look at the actual monitor construction in particular, let us first introduce some additional concepts and notation: For a finite run $\rho$ of $A_{\phi}$ over $(\mathcal{A}, u)$, we call $\delta_i(\rho(j), (\mathcal{A}_j, u_j)) = \text{obl}_j$ an obligation, where $0 \leq j < |u|$, in that $\text{obl}_j$ represents the language to be satisfied after $j$ inputs. That is, $\text{obl}_j$ refers to the language represented by the positive Boolean combination of spawned SAs. We say $\text{obl}_j$ is met by the input, if $(\mathcal{A}_j, u_j) \in \text{good}(\text{obl}_j)$ and violated if $(\mathcal{A}_j, u_j) \in \text{bad}(\text{obl}_j)$. Furthermore, $\rho$ is called potentially locally accepting, if it can be extended to a run $\rho'$ over $(\mathcal{A}, u)$ together with some infinite suffix, such that $\rho'$ is locally accepting.

The monitor for a formula $\phi \in \text{LTL}^{\text{FO}}$ can now be described in terms of two mutually recursive algorithms: The main entry point is Algorithm M. It reads an event and issues two calls to a separate Algorithm T: one for $\phi$ (under a possibly empty valuation $v$) and one for $\neg \phi$ (under a possibly empty valuation $v$). The purpose of Algorithm T is to detect bad prefixes wrt. the language of its argument formula, call it $\psi$. It does so by keeping track of those finite runs in $A_{\psi,v}$ that are potentially locally accepting and where its obligations have not been detected as violated by the input. If at any time no such run exists, then a bad prefix has been encountered. Algorithm T, in turn, uses Algorithm M to evaluate if obligations of its runs are met or violated by the input observed so far (i.e., it inductively creates submonitors): after the $i$th input, it instantiates Algorithm M with argument $\psi'$ (under corresponding valuation $v'$) for each $A_{\psi',v'}$ that occurs in $\text{obl}_i$ and forwards to it all observed events
from time \(i\) on.

**Algorithm M** *(Monitor).* The algorithm takes a \(\varphi \in \text{LTL}^\to\) (under a possibly empty valuation \(v\)). Its intuitive behaviour is as follows: Let us assume an initially empty first-order temporal structure \((\overline{A}, u)\). Algorithm M reads an event \((\overline{A}, \sigma)\), prints “\(\top\)” if \((\overline{A}, u) \in \text{good}(\varphi)\) (resp. “\(\bot\)” for \(\text{bad}(\varphi)\)), and returns. Otherwise it prints “?”, whereas we now assume that \((\overline{A}, u) = (\overline{A}, u^\sigma)\) holds.

**M1.** [Create instances of Algorithm T.] Create two instances of Algorithm T: one with \(\varphi\) and one with \(\neg \varphi\), and call them \(T_{\varphi,v}\) and \(T_{\neg \varphi,v}\), respectively.

**M2.** [Forward next event.] Wait for next event \((\overline{A}, \sigma)\) and forward it to \(T_{\varphi,v}\) and \(T_{\neg \varphi,v}\).

**M3.** [Communicate verdict.] If \(T_{\varphi,v}\) sends “no runs”, print \(\bot\) and return. If \(T_{\neg \varphi,v}\) sends “no runs”, print \(\top\) and return. Otherwise, print “?” and go back to M2.

**Algorithm T** *(Track runs).* The algorithm takes a \(\varphi \in \text{LTL}^\to\) (under a corresponding valuation \(v\)), for which it creates an SA, \(\mathcal{A}_{\varphi,v}\). It then reads an event \((\overline{A}, \sigma)\) and returns if \(\mathcal{A}_{\varphi,v}\), after processing \((\overline{A}, \sigma)\), does not have any potentially locally accepting runs, for which obligations have not been detected as violated. Otherwise, it saves the new state of \(\mathcal{A}_{\varphi,v}\) waits for new input, and then checks again, and so forth.

**T1.** [Create SA.] Create an SA \(\mathcal{A}_{\varphi,v}\).

**T2.** [Wait for new event.] Let \((\overline{A}, \sigma)\) be the event that was read.

**T3.** [Update potentially locally accepting runs.] Let \(B\) and \(B'\) be (initially empty) buffers. If \(B = \emptyset\), for each \(q \in Q_0\) and for each \(q' \in \delta_\to(q, (\overline{A}, \sigma))\): add \((q', [\delta_\to(q, (\overline{A}, \sigma))] )\) to \(B\). Otherwise, set \(B' = B\), and subsequently \(B = \emptyset\). Next, for all \((q, [obl_1, \ldots, obl_n]) \in B'\) and for all \(q' \in \delta_\to(q, (\overline{A}, \sigma))\): add \((q', [obl'_{\text{new}}, obl_1, \ldots, obl_n])\) to \(B\), where \(obl'_{\text{new}} = \delta_\to(q, (\overline{A}, \sigma))\).

**T4.** [Create submonitors.] For each \((q, [obl'_{\text{new}}, obl_1, \ldots, obl_n]) \in B\): call Algorithm M with argument \(\psi\) (under corresponding \(v')\) for each \(\mathcal{A}_{\varphi',v'}\) that occurs in \(obl'_{\text{new}}\).

**T5.** [Iterate over candidate runs.] Assume \(B = \{b_0, \ldots, b_m\}\). Create a counter \(j = 0\) and set \((q, [obl_1, \ldots, obl_n]) = b_j\) to be the \(j\)th element of \(B\).

**T6.** [Send, receive, replace.] For all \(0 \leq i \leq n\): send \((\overline{A}, \sigma)\) to all submonitors corresponding to SAs occurring in \(obl_i\), and wait for the respective verdicts. For every returned \(\top\) (resp. \(\bot\)) replace the corresponding SA in \(obl_i\) with \(\top\) (resp. \(\bot\)).

**T7.** [Corresponding run has violated obligations?] For all \(0 \leq i \leq n\): if \(obl_i = \bot\), remove \(b_j\) from \(B\) and go to T9.

**T8.** [Obligations met?] For all \(0 \leq i \leq n\): if \(obl_i = \top\), remove \(obl_i\).

**T9.** [Next run in buffer.] If \(j \leq m\), set \(j\) to \(j + 1\) and go to step T6.

**T10.** [Communicate verdict.] If \(B = \emptyset\), send “no runs” to the calling Algorithm M and return, otherwise send “some run(s)” and go back to T2.

---

2Obviously, the monitor does not really keep \((\overline{A}, u)\) around, or it would be necessarily trace-length dependent. \((\overline{A}, u)\) is merely used here to explain the inner workings of the monitor.

3Note that we build the SA in such a way that it contains only states from which exist a loop through some final states; i.e., every run in the SA is potentially locally accepting.
Figure 4.2: Monitor processing the event \( \{ login(1,2.3.4.1), login(2,2.3.4.2), send(3,2.3.4.3), send(1,5.6.7.8) \} \).

Note that the idea to use two instances of Algorithm T in step M1 is conceptually similar to the idea put forward in [Bauer et al. 2011]. They construct for a given LTL formula \( \varphi \), a BA for \( \varphi \) and \( \neg \varphi \), respectively. Based on the BAs, two NFAs are created. The NFA wrt. \( \varphi \) helps to detect bad prefixes. This means, it has essentially the structure of the according BA for \( \varphi \), where a state is marked as accepting iff there exists an accepting run starting from that same state in the BA (i.e., the language is not empty). If the NFA for \( \varphi \) does not accept a prefix, a bad prefix has been found. Inversely, the NFA for \( \neg \varphi \) is built to detect good prefixes.

For a given \( \varphi \in \text{LTL}\) and \( (\mathfrak{A},u) \), let us use \( M_\varphi(\mathfrak{A},u) \) to denote the successive application of Algorithm M for formula \( \varphi \), first on \( u_0 \), then \( u_1 \), and so forth. We then get

**Theorem 4.2.1** (Impartiality). \( M_\varphi(\mathfrak{A},u) = \top \Rightarrow (\mathfrak{A},u) \in \text{good}(\varphi) \) (resp. for \( \bot \) and \( \text{bad}(\varphi) \)).

**Proof idea** (full proof on page 133). By nested induction over depth(\( \varphi \)) and the length of \( (\mathfrak{A},u) \). \(\square\)

**Example 4.2.1.** Let us recall the policy used in Example 4.1.1 which we are going to use in the following to exemplify the algorithm’s behaviour. Given the previously used event \( \sigma = \{ login(1,2.3.4.1), login(2,2.3.4.2), send(3,2.3.4.3), send(1,5.6.7.8) \} \), \( M_\varphi \) creates two instances \( T_\varphi \) and \( T_{\neg \varphi} \), respectively. Note that there is no explicit valuation passed as an
argument as $\varphi$, being the original user-specification, can be assumed to be a sentence. We have illustrated this in Fig. 4.2 (see level 2).

Both $T_{\varphi}$ and $T_{\neg \varphi}$ then create their respective SAs, $A_{\varphi}$ and $A_{\neg \varphi}$. More precisely, in this very step, they create only the topmost level of respective SAs which, in case of $A_{\varphi}$, coincides with level 2 of the SA depicted in Fig. 4.1. We omit details for the creation of $A_{\neg \varphi}$, as it is done similarly. Instead of naively unfolding the respective SAs, Algorithm T has a local argument as $\varphi$, which keeps track of potentially locally accepting runs; that is, $T_{\varphi}$ initialises $B$ with $(q_0, [\delta_1(q_0, (\mathcal{A}, \sigma))] = A_{\varphi, 0} \land A_{\varphi, v'})$, where $q_0$ denotes the one and only state of $A_{\varphi}$ on level 2 and $v, v'$ are the valuations reflecting the contents of $\sigma$. In other words, instead of unfolding $A_{\varphi}$ (resp. $A_{\neg \varphi}$) directly, we store $\delta_1(q_0, (\sigma, \mathcal{A}))$ (resp. for $A_{\neg \varphi}$) in $B$ and then let another monitor deal with its results.

Therefore, as far as $T_{\varphi}$ (resp. $T_{\neg \varphi}$) is concerned, it treats the quantified part of $\varphi$ (resp. $\neg \varphi$) as a proposition, say, $p$, whose truth value is determined by some oracle (i.e., other monitor). This means, $\varphi$ is of the form $Gp$, for which a corresponding GBA only has a single looping state over proposition $p$. Hence, $T_{\varphi}$ (resp. $T_{\neg \varphi}$) needs to remember only one potentially locally accepting run inside $B$—the one that loops over $p$. From this point of view, the name “potentially locally accepting” seems quite fitting, in that a positive answer by the oracle (i.e., submonitor) will, indeed, confirm that the run is a suitable prefix of a satisfying run, if only it continued like that.

Consequently—in a mutually recursive manner—$T_{\varphi}$ then creates monitors, one for $A_{\varphi, 0}$ and one for $A_{\varphi, v'}$, which we refer to as $M_{\varphi, 0}$ and $M_{\varphi, v'}$, respectively (see Fig. 4.2 level 1). To stick with our analogy: these monitors serve as the oracles for the simple $Gp$ automaton. (The same process is happening for $T_{\neg \varphi}$, of course, which we disregard, as it is similar.) By their recursive definition, these new monitors behave like $M_{\varphi}$, except that they do not start with an empty valuation. Moreover, the mutual recursion eventually ends, once there are no further quantifiers to create new Ts and Ms for. It is therefore obvious that Algorithm M terminates, but not necessarily that the respective buffers are not growing unboundedly with increasing trace lengths (and therefore potentially unbounded number of potentially locally accepting runs). The following discussion illustrates this point.

Let us consider the newly created $T_{\varphi, 0}$ on level 1, Fig. 4.2, whose buffer is initially as follows: $(q_0, [\delta_1(q_0, (\mathcal{A}, \sigma))] = A_{\varphi, 0} \land A_{\varphi, v''})$, where $A_{\varphi, 0}$ and $A_{\varphi, v''}$ are a reference to the two leftmost SAs on level 0 in Fig. 4.1. Again, if we interpret these as propositions, we need two further monitors that yield their truth value, which are $M_{\varphi, 0}$ and $M_{\varphi, v''}$, respectively. These, in turn, yield four instances of Algorithm T, $T_{\varphi, 0}$, $T_{\varphi, v''}$, $T_{\neg \varphi, 0}$, and $T_{\neg \varphi, v''}$, respectively (four leftmost instances on level 0, Fig. 4.2). But as we have reached the end of our recursion and there are no further quantifiers left, the respective buffers of these four instances will never grow and, in fact, both $T_{\varphi, 0}$ and $T_{\neg \varphi, 0}$ will have no locally accepting runs. Therefore, $\delta_1(q_0, (\mathcal{A}, \sigma)) = T$, and the buffer becomes $(q_0, [])$. Note that we have, again, omitted some details for the negated policy. However, the point we would like to make here is that potentially locally accepting runs, stored inside the respective buffers, do not necessarily have to be memorised over the entire lifetime of a monitor and can be removed in a process similar to “garbage collection” known from programming languages [cf. Jones and Lins 1996].
4.3 Optimisations

After we have developed in Example 4.2.1 an intuitive understanding of how the monitoring algorithm works, let us discuss some obvious potential for optimisations. We can exploit the idea that the individual levels of an SA are merely GBAs of effectively propositional LTL formulae and are therefore able to use well-known transformations on automata to reduce the overall state space of the monitor. Some of those were presented in the context of runtime verification in Bauer et al. [2011].

We have seen, in particular, in Example 4.2.1 that monitoring a formula, such as \( \varphi = G\forall(u, ip): login. \psi \), corresponds to building a hierarchy of submonitors, one for each quantified subformula (and observed action, naturally). On the highest level of this hierarchy, the corresponding monitor will effectively use two GBAs, one for a formula of the form \( Gp \) and one for a formula of the form \( \neg(Gp) \), where we use \( p \) merely as a reference to the submonitors checking the \( \forall(u, ip): login. \psi \) part of the original formula, and so forth. In other words, on the topmost level, we have two GBAs for propositional specifications, \( Gp \) and \( \neg(Gp) \), and on the next level down, one for \( \psi \) and one for \( \neg\psi \) (as well as for each observed action), and so forth.

This opens up the door to the following automata optimisations, which are expected to help make the monitor more efficient. We first convert the individual GBAs into BAs, using the well-known counting construction, which we mentioned already in §2.3. Let \( \tilde{A}_\varphi = (\Sigma, Q, Q_0, \delta, F) \) denote a complete BA (see Def. 2.3.2) obtained this way for \( \varphi \), where \( \Sigma \) corresponds to the propositional alphabet of the automaton (thus, completely ignoring the fact that it is used within the context of an SA), \( Q \) is its set of states, \( Q_0 \subseteq Q \) a set of initial states, \( \delta \subseteq Q \times \Sigma \times 2^Q \) the transition relation and \( F \subseteq Q \) the set of final states. Then we turn \( \tilde{A}_\varphi \) into an NFA (see Def. 2.3.1), \( \hat{A}_\varphi = (\Sigma, Q, Q_0, \delta, \hat{F}) \), where \( \hat{F} \supseteq F \) is the set of states for which there exists a path in \( \tilde{A}_\varphi \) s.t. a strongly connected component can be reached, which contains at least one state from \( F \). Strongly connected components can be found, for example, by Tarjan’s algorithm [Tarjan, 1972], which performs a depth-first search. In Bauer et al. [2011], it was shown that \( L(\hat{A}_\varphi) = \{ u \in \Sigma^* \mid \) there exists a \( w \in \Sigma^\omega \) s.t. \( uw \in L(\tilde{A}_\varphi) \} \). In other words, \( \hat{A}_\varphi \) accepts prefixes of elements in \( L(\tilde{A}_\varphi) \); that is:

**Proposition 4.3.1.** Every \( u \in L(\hat{A}_\varphi) \), has a potentially locally accepting run in \( A_\varphi \).

*Proof.* Follows straight from the definitions. □

Since \( \hat{A}_\varphi \) is an ordinary NFA, it can be made deterministic and minimal in a language-preserving manner (see again §2.3). Moreover, we can build the same automaton for \( \neg\varphi \), and instead of using two automata in parallel, one for \( \varphi \) and one for \( \neg\varphi \), we can build the synchronous product of these two [cf. Hopcroft and Ullman 1979; Bauer et al., 2011] and use only this one automaton per submonitor—which on top of it all is minimal and deterministic. Let \( P_\varphi = (\Sigma, Q^\varphi \times Q^{\neg\varphi}, Q^\varphi_0 \times Q^{\neg\varphi}_0, \delta') \) be the product automaton obtained this way and \( \delta' \) defined as expected. The following proposition formally sums up and gives argument for soundness of the proposed optimisation.
Proposition 4.3.2. If a run reaches a state \((p,q) \in Q^\varphi \times Q^{-\varphi}\), s.t. \(q \not\in \hat{F}^{\neg \varphi}\), then it is potentially locally accepting only in \(A_\varphi\); if \(p \not\in \hat{F}^\varphi\), then it is potentially locally accepting only in \(A_{\neg \varphi}\); and if \(p \in \hat{F}^\varphi\) and \(q \in \hat{F}^{\neg \varphi}\) holds, it is potentially locally accepting in \(A_\varphi\) and \(A_{\neg \varphi}\).

Proof. Follows from Proposition 4.3.1 and soundness of the product construction. \(\square\)

The concrete changes to the algorithms in §4.2, resulting from these optimisations, are now relatively straightforward to describe. Algorithm M, instead of creating two instances of Algorithm T, merely creates one and interprets its result accordingly (see below). Since for a given \(\varphi\), Algorithm M no longer calls Algorithm T for both \(\varphi\) and \(\neg \varphi\), Algorithm T does not build \(A_\varphi\) in T1, but \(P_\varphi\). Moreover, it performs subsequent operations on this product automaton instead of \(A_\varphi\). Finally, in T10 and in accordance with Proposition 4.3.2, if all runs \((q, [obl_{new}, obl_1, \ldots, obl_n]) \in B\) are s.t. that they are only locally accepting in \(A_{\neg \varphi}\), Algorithm T sends “no runs in \(A_\varphi\);” if they are only locally accepting in \(A_\varphi\), it sends “no runs in \(A_{\neg \varphi}\);” and if neither holds, it sends “runs in \(A_\varphi\) and \(A_{\neg \varphi}\).” Algorithm M, in instruction M3, then prints \(\bot\), \(\top\), \(?\) in either event, respectively.

4.4 Experiments

In order to demonstrate feasibility of the monitor construction and to get an intuition on its runtime performance (i.e., space consumption at runtime), the author has implemented the algorithm above both as laid out in §4.2 as well as with the in §4.3 explained optimisations.\(^4\) Let us refer to the former in the following as the “unoptimised” version of the algorithm.

The only divergence from its description is that the GBAs according to the rules laid out in §4.2 are not manually constructed. We have argued that the GBAs are basically ordinary propositional automata, hence there is no reason why we cannot employ an “off the shelf” GBA generator, such as lbt.\(^5\) Similarly, for the optimised version, we used the LTL3-Tools to construct the product automata.\(^6\)

It should be obvious that this does not change any of the results, but instead makes the approach a lot easier to implement. Moreover, the proposed algorithm has the advantage that it is possible to precompute all the SAs (resp. product automata) that are required at runtime, i.e., we replaced step T1 in Algorithm T with a look-up in a precomputed table of SAs (resp. product automata) and merely use a new valuation each time.

In §4.4.1 we first compare the unoptimised implementation with the somewhat naive formula progression and in §4.4.2 we analyse explicitly if the optimisation leads to effectively smaller monitor sizes—and therefore more efficient monitors—at runtime.

\(^4\)Available as open source Scala project on https://github.com/jckuester/ltlfo2mon
\(^5\)http://www.tcs.hut.fi/Software/maria/tools/lbt/
\(^6\)http://ltl3tools.sourceforge.net/
Figure 4.3: Difference in space consumption at runtime: SA-based monitor vs. progression.

4.4.1 SA-based monitoring vs. formula progression

Let us compare the SA-based monitor implementation with the arguably easier to construct approach of monitoring LTL\textsuperscript{FO} formulae, which has been described by Bauer, Küster, and Vegliach [2012]. In that work was used the well-known concept of formula rewriting known for LTL, which we have mentioned already in \S 3.2.1. It is sometimes also referred to as progression [cf. Bacchus and Kabanza [1998] Table 1]: a function, \textit{Progress} : LTL × Σ \rightarrow LTL, continuously “rewrites” a formula \( ϕ \in LTL \) using an observed event, \( σ \), s.t., \( σw \models ϕ ↔ w \models \textit{Progress}(ϕ, σ) \) holds. Assume \( ϕ' \in \textit{Progress}(ϕ, σ) \). If \( ϕ' = ⊤ \), then \( σ \) is good\((ϕ) \), if \( ϕ' = ⊥ \) then \( σ \) is bad\((ϕ) \), otherwise the thereby realised monitor waits for further events to apply its progression function to. \textit{Progress} rewrites according to the well-known fixpoint characterisations of LTL operators. For the temporal operators, the basic rules are therefore \( \text{Progress}(Xϕ, σ) = ϕ \), and \( \text{Progress}(ϕUψ, σ) = \text{Progress}(ψ, σ) \lor (\text{Progress}(ϕ, σ) \land ϕUψ) \).

For LTL\textsuperscript{FO} was added another rule that handles the quantifier as well as the progression function takes another input: a valuation \( v \). The new rule rewrites the quantifier into a conjunction according to the actions occurring in an event \((\mathfrak{A}, σ)\): \( \text{Progress}(∀x : p. ϕ, (\mathfrak{A}, σ), v) = \bigwedge_{(p, d) \in r} \text{Progress}(ϕ, (\mathfrak{A}, σ), v \cup \{x \mapsto d\}) \).

As a benchmark for all of the tests were used several formulae derived from the well-known specification patterns discussed in \S 2.4. Quantification was added to
§4.4 Experiments

Figure 4.4: Size of precomputed look-up tables.

crucial positions in the formulae. Some results of the comparison between progression and the monitor in this thesis are visualised in Fig. 4.3. For each LTL\(^{\text{FO}}\) formula corresponding to a pattern, 20 traces of lengths 100, 1000, and 10000 have been randomly generated, respectively, and were passed to both algorithms. The number of actions per event is uniquely distributed between 0 and 5 and the domain values of ground terms are log-normal distributed.

The average space consumption of each algorithm (i.e., size of the monitor) was measured at different trace lengths. The x-axis marks the difference between the two approaches after reading a trace of a given length. A positive difference (i.e., if the small shape sits in the right part of the diagram) indicates that the SA-based monitor is on average strictly smaller than progression after reading a trace of a set length (and vice versa, negative values indicate that progression is on average smaller). The space consumption of progression is measured simply in terms of the length of the formula at a given time, whereas for the SA-based monitor \(M_{\varphi,v}\) it is determined recursively as follows: Recall, \(M_{\varphi,v}\) first creates two instances of Algorithm T, \(T_{\varphi,v}\) and \(T_{\neg \varphi,v}\), each of which creates a buffer, call it \(B_{\varphi}\), resp. \(B_{\neg \varphi}\). Let \(B = B_{\varphi} \cup B_{\neg \varphi}\), and \((q_i, [obl_{i,0}, \ldots, obl_{i,n_i}])\) be the \(i\)-th element of \(B\), then

\[\text{eq:recursive-space}

---

\(^7\)All traces used in this chapter, the definitions of I-operators appearing in formulae, as well as the experiments’ results in its full extent can be found on [https://github.com/jckuester/ltlfo2mon/tree/master/experiments](https://github.com/jckuester/ltlfo2mon/tree/master/experiments).
\[ |M_{\psi,v}| = \sum_{i=0}^{\langle B \rangle - 1} (|obl_{i,0}| + \ldots + |obl_{i,n}|) = \sum_{i=0}^{\langle B \rangle - 1} (1 + |obl_{i,0}| + \ldots + |obl_{i,n}|), \]
where \[ |obl_{i,j}| = |obl_{i,j}| + \sum_{A \in \psi} |M_{\psi,v}|, \]
i.e., the sum of the top-level monitor’s constituents as well as that of all of its submonitors. Finally, we also need to add the total size of the precomputed GBA look-up table. This is measured as \[ |A_{\psi} + |A_{\neg \psi}| + \sum_{\forall x \cdot p. \psi \in \sf(f)} (|A_{\psi}| + |A_{\neg \psi}|), \]
where \[ |A_{\psi}| \]
is the size of the obtained GBA when we run lbt on input \( \psi' \) with quantified subformulae interpreted as propositions; that is, the sum of the number of its states and transitions as well as the number of literals contained in states. The absolute size of the look-up table for each formula is represented as a black coloured bar in Fig. 4.4 respectively. We find the smallest look-up table with size 17 for the shortest of all formulae, \( \psi_1 \), which contains GBAs of the form \( Gp, \neg Gp, p \) and \( \neg p \). On the other hand, the biggest look-up table is the one for \( \psi_7 \) with size 1785, which is also the longest formula of the patterns used in the experiments.

The end markers on the left of each horizontal bar show how much bigger in the worst case an SA-based monitor is for a given formula compared to the corresponding progression-based monitor (and vice versa for the right markers). The small shapes in the middle denote the average size difference of the two monitors over the whole length of a trace. This difference is most striking for \( \psi_2 \) on longer traces (e.g., \( \Delta \geq 10000 \) for traces of length 10000), where the average almost coincides with the worst case. As such, this example brings to surface one of the potential pitfalls of progression, namely that a lot of redundant information can accumulate over time: If \( \exists x : w. q(x) \) ever becomes true, then Progress, which operates purely on a syntactic level, will produce a new conjunct \( G \forall y : w. \neg p(y) \) for each new event, even though semantically it is not necessary (or, to use our analogy of before: progression is not very good at “garbage collection”). Hence, the longer the trace, the greater the average difference in size (similar in case of \( \psi_3 \) and \( \psi_4 \)).

At first glance, it may seem a curious coincidence that the left markers of each bar align perfectly, because this indicates that for all three traces that belong to a given formula, the SA-based monitor is in the worst case by exactly the same constant \( k \) bigger than the progression-based one, irrespective of the trace. However, it makes sense if we consider when this worst case occurs: It is whenever the SA-based monitor (and consequently also the rewriting-based one) does not have to memorise any data at all, in which case the size of the SA-based monitor’s look-up table weighs the most; that is, the size of the look-up table is almost equal to \( k \). Usually this happens when monitoring commences; hence, there is a perfect alignment on all traces. On the other hand, the worst case for progression occurs whenever the amount of data to be memorised by the monitor has reached its maximum. As this depends not only on the formula, but also on the content of the randomly generated traces (and in some of the examples also on their lengths, as seen in the previous example), we generally do not observe alignment on the right.

For those examples that, on average, favour progression, note that the difference in size is less dramatic—a fact, which may be slightly obfuscated by the pseudo-logarithmic scale of the x-axis. Again, the differences in these examples (\( \psi_1, \psi_5, \psi_6, \) and \( \psi_7 \)) can be mostly explained by the fact that an SA-based monitor generally
§4.4 Experiments

4.4.2 SA-based monitoring vs. optimisations

Let us now evaluate if the state space reduction, aimed for by the optimisations, actually affects the size of the monitor in practice. Therefore, the optimised version of the monitor has been executed on the same set of formulae and traces as discussed in §4.4.1 to compare it against the unoptimised version.

Fig. 4.5 shows the results in the same type of diagram, which was used to compare SA-based monitoring with a progression-based monitor. Again, as we measure the difference between both monitor implementations at runtime, positive values on the x-axis mean that the optimised monitor is smaller in size than the unoptimised one (and vice versa for negative values). One can clearly see that the optimised monitor, in the majority of cases, is on average much smaller than the unoptimised one. However, in the worst case (left markers) it tends to be often bigger than the unoptimised one, as we can observe for formulae \( \varphi_3, \varphi_4, \varphi_5, \varphi_6 \) and \( \varphi_8 \). The perfect alignment of markers on the left, which we have seen before in Fig. 4.3, again indicate that the worst case occurs independently of the trace-length; that is, the optimised monitor is bigger by a constant \( k \). In this case \( k \) correlates exactly with the difference in size of the precomputed look-up table for the optimised and unoptimised monitor, which wastes more space for “book keeping”.

Figure 4.5: Difference in space consumption at runtime: optimised vs. unoptimised SA-based monitor.
can be obtained from the numbers given in Fig. 4.4. Sizes for the GBA look-up tables are shown as black coloured bars and have been discussed above, whereas the look-up tables of product automata are presented in grey. In contrast to former, the latter size is measured as following: $|P_{\varphi}| + \sum_{\forall x: \varphi \in \text{sf}(\varphi)} (|P_{\psi}|)$, where $|P_{\psi}|$ is the size of the product automaton obtained from the LTL3-Tools on input $\psi'$. We establish $|P_{\psi}|$ equal to $|A_{\psi}|$ with the only difference being that $P_{\psi}$ is a transition-based automaton, and therefore we count the number of propositional literals labelling transitions instead of states. On the other hand, for formulae $\varphi_1$, $\varphi_2$ and $\varphi_7$, the difference in size is positive, meaning that the optimised monitor is smaller at any point in time during monitoring.

As in our previous comparison, we get the best results when traces are longer; that is, the optimised monitor is considerably smaller on longer traces as the average size and the right markers almost coincide. This suggests that the unoptimised monitor accumulates a lot of data over time, which the optimised one does not. This is due to the fact that on the lowest level, the LTL3-Tools generate propositional and therefore minimal finite state machines of which the optimised monitor only needs to store a single state at runtime. This becomes even more apparent when looking at the average number of runs per monitor on the lowest level for the different formulae: for $\varphi_3 = 4$, $\varphi_4 = 5.6$, $\varphi_5 = 4$, $\varphi_6 = 13.8$ and $\varphi_7 = 52.7$.

Finally, let us consider results for $\varphi_1$ and $\varphi_2$, for which average and worst cases (left and right markers) are identical. We know for these formulae that no “book keeping” is required, and the results show that, indeed, none of the proposed algorithms in this thesis does. Recall, we have observed the same phenomena in Fig. 4.3 only for $\varphi_1$, as progression accumulates redundant information for $\varphi_2$.

### 4.5 Related monitoring approaches

This thesis is not the first work to discuss monitoring of traces with data nor to deal in particular with temporal first-order specifications. This section discusses monitoring approaches in the literature that all handle formalisms that are strictly beyond propositional expressiveness: Some offer forms of data quantification that is implicit [Stolz and Bodden, 2006; Chen and Rosu, 2009], explicit [Basin et al., 2015b; Hallé and Villemaire, 2008; Stolz, 2010], or restricted to only universal quantification [Chen and Rosu, 2009]. Some approaches are based on SMT solvers [Decker et al., 2016] or use different types of automata than SAs [Barringer et al., 2012; Stolz and Bodden, 2006; Havelund, 2014]. We focus on the major similarities and differences, including an informal comparison of expressiveness or efficiency.

**Past first-order temporal logic (Past FOTL).** Even before the existence of runtime verification as a scientific discipline8 the monitoring problem for different types of
first-order logic has been widely studied, e.g., in the database community. It was motivated by checking temporal triggers and temporal constraints. In that context, Chomicki [1995] presents a method to check for violations of temporal integrity constraints in databases, specified using (metric) past temporal operators. His logic differs from LTL$^{FO}$, in that it allows first-order quantification over a single countable and constant domain, whereas quantified variables in LTL$^{FO}$ range over elements that occur at the current position of the trace (see also Hallé and Villemaire [2008]; Bauer et al. [2009b]). Presumably, to achieve the same effect, Chomicki [1995] demands that policies are what is called “domain independent”, so that all statements refer to known (i.e., finitely many) objects, which therefore can be stored in materialised views. In other words, his proposed monitor construction for past FOTL extends every database state with auxiliary relations that contain the historical information necessary for checking constraints, and this way can be implemented by existing database technology. Domain independence is a property of the policy and shown to be undecidable [Paola, 1969]. In contrast, one could say that LTL$^{FO}$ has a similar notion of domain independence already built-in, because of its quantifier. Like LTL$^{FO}$, the logic of Chomicki [1995] is also undecidable [Harel, 1985]; no function symbols are allowed and relations are required to be finite. However, the logic is sometimes extended with an operator for equality and strict order, which have the expected infinite and rigid semantics.

Despite the fact that the prefix problem is not phrased as a decision problem, its basic idea is already denoted by Chomicki in terms of the potential constraint satisfaction problem. In particular, he shows that the set of prefixes of models for a given formula is not recursively enumerable. On the other hand, the monitor in Chomicki [1995] does not tackle this problem and instead solves what we have introduced as the word problem, which, unlike the prefix problem, is decidable.

Chomicki implicitly defines already what can be thought of as trace-length independence under the term bounded history encoding, to distinguish his method from naive methods that need to store the entire database (and therefore will theoretically require arbitrary space, as database changes over time are considered an infinite sequence). The term bounded history encoding captures the intuition that the amount of historical information stored does not depend on the length of the database history but only on the active domain (i.e., the different domain values that have appeared in the database history so far). However, in the worst case his method requires polynomially more space than a naive method that stores all the states in the history. What he did not take into account in his definition is that trace-length independence is not merely a property of a monitor, but also of the formula being checked. For example, he states a formula

$$\neg(\exists x, sal_1, sal_2)((F^{-1} Earns(x, sal_1)) \land Earns(x, sal_2) \land sal_2 < sal_1),$$

which could be written in LTL$^{FO}$ as $G \forall (x, sal_1) : earns. XG \forall (y, sal_2) : earns. (x = y \Rightarrow sal_1 \leq sal_2)$, and means “a person cannot earn less than at any time in the past”. We

http://www.runtime-verification.org/
term this kind of formula trace-length dependent (see hierarchy in §5), as its monitor
has no other choice than to grow with the trace and store all the past earnings.

Chomicki and Niwinski [1995] also closely studied a specific fragment of FOTL,
namely the biquantified formulas. This only allows future temporal operators and
quantification is restricted; that is, quantifier can either only appear outside the scope
of temporal operators (and are in this case universal), or inside their scope cannot be
a temporal operator. In other words, the arbitrary nesting of temporal and quantifiers
is not allowed. They show that already allowing one (existential or universal)
quantifier, whose scope is free of temporal operators makes the fragment undecidable.

**Metric first-order temporal logic (MFOTL).** Basin et al. [2010] extend Chomicki’s
monitor towards bounded future operators using the same logic (i.e., past metric
first-order temporal logic). This is a reasonable choice, as it is unknown whether the
past fragment of MFOTL has the same expressiveness as MFOTL with both past and
future operators. Note that for propositional LTL it is well-known that this is the case
[Chomicki and Niwinski, 1995] that even though LTL with past-only is exponentially more succinct than LTL [Laroussinie et al., 2002]. As MFOTL is like LTL\textsuperscript{FO}
undecidable (see Basin et al. [2013, Lemma B.4] and Theorem 3.2.2 respectively), the
authors restrict its monitor to formulae of the form $\mathcal{G}\phi$, where $\phi$ is bounded; that
is, a safety fragment where temporal operators refer only finitely far into the future.

While every propositional LTL formula representing a safety property is equivalent
to the above form [Pnueli, 1986], this is shown to be not the case in first-order temporal
logic. Chomicki and Niwinski [1995] provide an according biquantified formula
as a counter example. In other words, Basin et al. [2010] monitor a restricted safety
fragment, whereas monitoring of LTL\textsuperscript{FO} is not restricted to safety formulae.

Basin et al. [2010] allow infinite relations as long as these are representable by au-
tomatic structures, i.e., finite-state automata models. In this way, they show that the
restriction on formulae to be domain independent is no longer necessary, i.e., the un-
restricted use of negation and quantification becomes possible. LTL\textsuperscript{FO}, in comparison,
is more general, in that it allows computable relations and functions. On the other
hand, LTL\textsuperscript{FO} lacks syntax to directly specify metric constraints, and only has future
operators (but therefore unbounded ones). Nonetheless, the metric constraints of
operators can be defined by the following equivalent LTL\textsuperscript{FO} formulae (also shown by
Hallé and Villemaire [2008, §5.2.2]): We define a predicate $I(x, a, b) := x \leq b \land x \geq a$,
which checks if $x \in [a, b]$ for $x, a, b \in \mathbb{N}$. We then can express metric future temporal
operators of MFOTL in LTL\textsuperscript{FO} in the following way:

$$X_{[a,b]} \phi := \exists t_1 : \tau. X(\phi \land \exists t_2 : \tau. (I(t_2 - t_1, a, b) \land t_2 > t_1)),$$

$$\phi U_{[a,b]} \psi := \exists t_1 : \tau. \phi U(\psi \land \exists t_2 : \tau. (I(t_2 - t_1, a, b) \land t_2 > t_1)).$$

Note that in contrast to MFOTL, LTL\textsuperscript{FO} requires ground atoms wrt. a U-operator $\tau$ in
the trace to simulate the time-trace $\tau_0, \tau_1 \ldots$, which is defined in MFOTL as part of
the semantics.
Several practical extensions to MFOTL have been proposed, including operators for aggregation [Basin et al. 2015a]. These are useful when applying the logic and associated monitoring tool MonPoly to case studies such as Nokia’s Data-collection Campaign [Basin et al. 2013].

**Future (or past) temporal logic with an assignment quantifier.** Sistla and Wolfson [1995] also discuss a monitor for database triggers whose conditions are specified in a logic (with past or future operators), which uses an assignment quantifier (written as \([x \leftarrow t] \varphi(x)\)) that binds a single value or a relation instance \(t\) (i.e., the result of a database query in the current state) to a global, rigid variable \(x\). The assignment is similar to the freeze quantifier defined by [Henzinger 1990]. Their monitor is represented by a graph structure, which is extended by one level for each updated database state, and as such is proportional in size to the number of updates.

**Linear temporal logic with full first-order quantification (LTL-FO+).** The work of Hallé and Villemaire [2008] describes a monitoring algorithm for a logic with quantification identical to LTL\(^\text{FO}\), but without function symbols or arbitrary computable relations. The resulting monitors are generated “on the fly” by using syntax-directed tableaux, which is inspired by Gerth et al. [1995] adapted to first-order. Soundness and completeness proofs for the extension are not provided. Tool support is available and called BeepBeep. In our approach, however, it is possible to precompute the individual BAs for the respective subformulae of a policy (i.e., levels of the SA), and thereby bound the complexity of that part of our monitor at runtime by a constant factor.

**Parametric LTL (pLTL).** Stolz [2010] developed a monitoring approach for parametric LTL, i.e., LTL where propositions are enriched with variables, and where quantification is semantically identical to LTL\(^\text{FO}\). Consider for example the formula

\[
\begin{align*}
\mathsf{G} & \exists x : p(x) \rightarrow \varphi(x), \\
\end{align*}
\]

which is equivalent to the LTL\(^\text{FO}\) formula \(\mathsf{G} \exists x : p \cdot \varphi(x)\). This means \(\rightarrow\) is not the usual implication operator from Boolean logic, but a binding operator, where \(p(x)\) is called an existence predicate. It denotes that the values of the proposition \(p\) will be bound from the current event. The remainder of the formula, \(\varphi(x)\), only “uses” the variable \(x\). Semantically, for a current event \(\{p(1), p(2), q(3)\}\), in pLTL—and similarly in LTL\(^\text{FO}\)—we have to verify the formulae \(\mathsf{G}p(1) \rightarrow \varphi(1), \mathsf{G}p(2) \rightarrow \varphi(2)\) starting from this event, and \(\mathsf{G} \exists x : p(x) \rightarrow \varphi(x)\) again from the next event.

However, in contrast to LTL\(^\text{FO}\), pLTL does not allow interpreted relations, and is “next-free”. Another fundamental difference is that Stolz [2010] does not solve the
prefix problem. His monitor construction is based on a parameterised automaton, which essentially behaves like a normal alternating finite automaton [cf. Vardi, 1995; Finkbeiner and Sipma, 2004].

It is noteworthy that his work is an extension of one of the first approaches handling parameterised events, called Java Logical Observer (JLO) [Stolz and Bodden, 2006]. In JLO quantification is only implicit, depending on the “parent” temporal operator “shadowing” a proposition. For example, propositions inside the scope of a “finally”-operator are existentially quantified, and those inside the scope of a “globally”-operator are always universal. From an implementation point of view, parameterised LTL properties are specified over AspectJ pointcuts, which serve as propositions. A pointcut can for example refer to a method and triggers (i.e., its associated proposition becomes true) if the method gets called. As AspectJ works on the bytecode level, JLO can also instrument third-party Java programs whose source code is not accessible.

**Parametric monitoring and trace slicing.** There are works dealing with so-called parametric monitoring which, although not based on first-order logic, offer support for monitoring traces carrying data [cf. Allan et al., 2005; Chen and Rosu, 2009]. The approach followed by Chen and Rosu [2009] is to “slice” a trace according to the parameters occurring in it and then to forward the $n$ (effectively propositional) subtraces to $n$ monitor instances of the same specification; for example, one per logged-in user for the property

$$\Lambda u, ip. \ (\text{login}(u, ip) \ \text{send}(u, ip)^* \ \text{logout}(u))^*.$$  

This expresses that for any user $u$ who logs in from IP address $ip$, login and logout actions of that user should alternate, and data from that same $ip$ must be sent only in between. An equivalent formula in LTL$^{FO}$ was discussed in Example 4.1.1. The property above is of the form $\Lambda X. P$, denoting that $P$ is a so-called parametric property (here expressed as a regular expression). $\Lambda X$ further denotes an implicit universally quantifier over the set $X$, where actions are parametric with variables from $X$. For a trace $w = \text{login}(u \rightarrow u_1, ip \rightarrow ip_1) \ \text{send}(u \rightarrow u_1, ip \rightarrow ip_1) \ \text{send}(u \rightarrow u_1, ip \rightarrow ip_2)$ the according two slices are

$$\langle u \rightarrow u_1, ip \rightarrow ip_1 \rangle : \text{login, send,}$$

$$\langle u \rightarrow u_1, ip \rightarrow ip_2 \rangle : \text{send.}$$

On the left-hand side we see a unique list of bindings (i.e., variables mapped to parameters). These each refer to a subtrace on the right-hand side, which contains actions for that binding only, with parameters chopped off. For efficiency in practice, instead of storing the subtraces, a binding maps directly to a state of the propositional monitor for that binding (i.e., the state the monitor will be in wrt. observing the according subtrace). One major concern in trace slicing is indexing, i.e., to efficiently

locate all related monitor instances given the bindings for an incoming event. Note that the parametric property in the example above is violated by the trace $w$. This is because the second subtrace, which contains only the action “send”, violates the propositional property $(\text{login send} \quad \text{logout})^\ast$.

Besides the implicit universal quantification, there are further limitations to the slicing approach. For example, one cannot express that a user should not login from a different IP address at the same time, as this requires the same action name with different variables in one property (e.g., $\text{login}(u, ip)$ and $\text{login}(u, ip')$). More precisely, this does not allow creating a unique binding based on the action name [Chen and Rosu, 2009, Def. 3 and 6]. See further limitations below when discussing quantified event automata (QEAs).

A widely-used runtime verification system implementing the trace slicing idea is the monitoring oriented programming (MOP) framework [Meredith et al., 2012], with the instance JavaMOP for Java programs. It claims to be the most efficient state-of-the-art runtime verification tool. JavaMOP specifications are compiled into AspectJ [Kiczales et al., 2001] aspects, which can be “weaved” into a program that a user wishes to monitor, using any standard AspectJ compiler such as ajc. Since in the MOP framework, the indexing is decoupled from the actual property checking, it supports parameterisation for many propositional logic formalisms—provided as a plugin—such as those based on finite state machines, extended regular expressions (ERE), context free grammars [Meredith et al., 2010], and LTL with past and future operators.

What is more, JavaMOP supports fine-tuning of the monitor semantics: Events can be marked “creating” (i.e., for those only is created a new monitor instance), and different binding modes can be used, such as “full-binding” (which indicates that all variables must be instantiated, otherwise a monitor will not report a verdict). For the example above, in full-binding mode, a monitor that only observes a logout event would not report any violation.

It is worth noting that the MOP framework solves also the prefix problem for LTL; however, it uses a satisfiability checker applied to each derived formula.

Another slicing-based approach is Tracematches [Allan et al., 2005], which applies a similar technique as JavaMOP to match regular expressions with a program trace. It adds a history-based language feature to the AspectJ language; that is, the possibility to specify regular patterns (with free variables) over executions of a Java program. When the pattern matches, some code will be executed. After JavaMOP it is the second-most efficient Java monitoring system [Meredith et al., 2012]. Free variables in Tracematches are evaluated over all possible instantiations, i.e., there is assumed to be an implicit universal quantifier like in JavaMOP. Tracematch statements can be specified by JavaMOP through the ERE plugin [Meredith et al., 2012]. Note that JavaMOP can also capture the capabilities of JLO.

In summary, none of the slicing approaches above support arbitrary nesting of quantifiers and temporal operators, use of negation, or function symbols to name just some important restrictions. However, on the plus side, one is able to use optimised monitoring techniques, developed in the propositional domain, and apply them—with these restrictions in mind—to traces carrying data.
Rule-based runtime verification. Rule-based monitoring approaches, such as Eagle [Barringer et al., 2004] and RuleR [Barringer et al., 2010b], have been proposed as expressive general-purpose formalisms in which various temporal logics can be defined. In Eagle, new temporal operators are introduced as recursive equations (with a minimal/maximal fixpoint semantics), formed from propositional logic and three primitive temporal operators (“next”, “previously”, and “concatenation”). For example, the “globally”-operator in LTL can be defined as

\[
\text{max} \; \text{Always}(\text{Form } F) = F \land \Box \text{Always}(F),
\]

where \(\Box\) is the “next”-operator, and the keyword \text{max} specifies that \(\Box \text{Always}(F)\) is interpreted as true at the end of a trace. The fixpoint semantics is necessary as Eagle, and also RuleR, use a finite trace semantics, i.e., their monitors solve the word problem. What is more, rules can be parameterised with data, for example as seen by

\[
\text{min} \; R(\text{int } k) = \text{Eventually}(y = k) \quad \text{mon } M = \text{Always}(x > 0 \rightarrow R(x)),
\]

where the rule \(R\) is parameterised with an integer \(k\), which is bound whenever \(x > 0\) holds in property \(M\). The keyword \text{mon} indicates that property \(M\) is to be monitored. A similar statement would be expressed in LTL\textsuperscript{FO} as \(G \forall x' : x. (x' > 0) \Rightarrow F \exists y' : y. (y' = x')\). However, since we solve the prefix problem, we will see in §5 that this property is not monitorable in LTL\textsuperscript{FO} (see Def. 5.1.1). Regarding expressiveness, it was shown by examples that Eagle can define future and past time LTL properties, interval logic (e.g., metric temporal logic), ERE (using the concatenation operator), or even limited forms of quantification over possibly infinite data sets (as seen in property \(M\) above). The Eagle logic is like LTL\textsuperscript{FO}, undecidable, and its word problem decidable.

Hawk [d’Amorim and Havelund, 2005] introduces a programming-oriented extension to Eagle, dedicated to Java programs. As Hawk specifications are translated into Eagle logic, Hawk adds no expressiveness, and therefore is merely a practical and convenient way of specifying properties for Java programs. Like in JLO, specifications can directly refer to parameterised system events based on programs, which for this purpose are instrumented by AspectJ.

RuleR [Barringer et al., 2010b] was designed based on experiences gained from Eagle, and provides a similarly powerful but simpler “core” logic, which can be monitored more efficiently. A specification in RuleR is a set of rules of the form

\[
r : \text{condition}_1, \ldots, \text{condition}_n \rightarrow \text{action}_1, \ldots, \text{action}_m,
\]

where \(r\) is a rule’s name, and \text{condition}_i as well as \text{action}_i either some (possibly negated) occurrence of a rule or observation name. A condition holds if the according rule is active, or an according observation appears in the current state, respectively. If all conditions on the left-hand side hold, the rule triggers. In this case, an observation on the right hand side must hold in the next state, or a rule appearing on the right
side is made active (i.e., can trigger after the next observation if their conditions hold).
Note that the right side is a disjunction of conjunctions. For example, $G(a \lor b)$ could be expressed by the rule $r \rightarrow a, r|b, r$. As there is no condition on the left (which is equivalent to true), $r$ always triggers, and next must hold $r$ again, and $a$ or $b$. Like in Eagle, rules can be parameterised with data values, as shown in the following rule

$$r(k : \mathbb{N}) : \text{clock}(?t_1 : \mathbb{N}) \rightarrow \text{clock}(?t_2 : \mathbb{N}), p, t_2 - t_1 \leq k \mid \text{clock}(?t_2 : \mathbb{N}), \neg p, t_2 - t_1 \leq k, r(k - t_2 + t_1).$$

While $?t_1$ indicates that parameter $t_1$ binds the value of an action clock in the current observation, $?t_2$ indicates that in the next state the value of an observation clock will be bound to $t_2$. Assume that there is an initially active rule $r(5)$. Then, the above rule could be written in MFTOL as formula $F_{0,5}p$, assuming there is another rule that demands a ground atom clock$(t)$ with a strictly monotonic time value $t$ in each state.

Another rule-based system is LogFire [Havelund, 2015], which is implemented as a domain-specific language (DSL) in Scala. It is founded on an adaption of the Rete algorithm [Forgy, 1982], which is know from rule systems in the artificial intelligence community.

Quantified event automata (QEA). When exploring the spectrum between Java-MOP and more expressive systems, such as Eagle and RuleR, [Barringer et al., 2012] developed a monitoring approach that generalises parametric trace slicing. It is based on an extension of an event automaton (EA), which is essentially an NFA, where the alphabet is a set of parametric actions, and transitions can be labelled with guards and assignments. As an example, consider the following transition between two states

$$\langle 2, [\text{max} \mapsto 1] \rangle \xrightarrow{\text{bid}('hat',5)} \langle 3, [\text{max} \mapsto 5, \text{new} \mapsto 5] \rangle,$$

for which the following fraction defines a guard (top part) and an assignment (bottom part)

$$\frac{\text{new} > \text{max}}{\text{max} := \text{new}}.$$

In state 2 (with the value 1 bound to variable max), the transition above triggers on a new event bid('hat',5), with the consequence that the EA ends up in state 3 (with bindings max $\mapsto 5$, new $\mapsto 5$), since the guard $\text{new} > \text{max}$ holds and the “max” value is set to the “new” value.

A QEA extends an EA with quantification over variables appearing in parametric actions. For example, in the case of universal quantification, an accepting state must be reached for all bindings.

The approach from [Barringer et al., 2012] overcomes the following concrete limitations of trace slicing in JavaMOP: (1) events can be associated with multiple variable lists, (2) free variables can be rebound along the trace, and (3) nesting of universal and existential quantification is possible (recall that JavaMOP assumes universal quantification on all parameters). More precisely, the following properties, specified here in
LTL\textsuperscript{FO} for comparison, can be represented as QEAs, but not in JavaMOP:

$$\text{G} \forall x: \text{login. (}\forall y: \text{login. (}x = y\text{))}\text{U }\text{logout}(x),$$

means there should not be a login of another user $y$ before user $x$ logs out,

$$\text{G} \forall (\text{item, amount}): \text{bid. } \text{XG} \forall (\text{item}', \text{amount}') : \text{bid.}$$

$$(\text{item} = \text{item}') \implies (\text{amount}' > \text{amount}),$$

means that during an auction another bid on the same item must be have a higher amount (i.e., rebinding for the same event must happen), and

$$\text{G} \forall (v, p): \text{member. } \text{G} \forall (c, p') : \text{candidate.}$$

$$(p = p') \implies F \exists (v', c', r) : \text{rank. } (v = v') \land (c = c'),$$

means that all members of a party must rank all candidates of the same party (i.e., nested quantification is required).

The MarQ tool\textsuperscript{12} was developed to monitor properties expressed as QEAs [Reger et al., 2015], and took part in the 1st\textsuperscript{13} and 2nd [Falcone et al., 2015] international runtime verification competition.

Reger and Rydeheard [2015] recently studied the common aspects of first-order and trace slicing specifications, aiming towards unifying monitoring languages. They identified a fragment of first-order LTL that can be monitored with the QEA approach.

Havelund [2014] introduced another expressive monitoring formalism, called data automata, where states can be parameterised with data.

**Temporal data logic (TDL).** Decker et al. [2014] introduce an algorithm that supports monitoring of a propositional temporal logic in combination with a data logic, which together can express the timely behaviour of a system with respect to the data it processes. On the temporal side, they do not demand a particular temporal logic, but rather provide assumptions such a logic must fulfil so that it can be combined. More precisely, (1) as verdicts of multiple monitor instances need to be combined, truth values need to be from a complete lattice, (2) a propositional monitoring algorithm for the logic must exist that can be integrated in their framework, and (3) the temporal logic must be linear as well as having a propositional semantics (i.e., propositions can be substituted without effecting the temporal meaning of a formula).

Their algorithm supports monitoring of first-order logic formulae without restricting to a closed-world assumption as in LTL\textsuperscript{FO}. The authors argue that this allows for a more natural specification of properties in some cases. However, since their proposed logic differentiates between a foreground (temporal) and a background (data) part, their monitoring solution for a chosen background logic depends on the avail-

\textsuperscript{12}https://github.com/selig/qea
\textsuperscript{13}http://rv2014.imag.fr/monitoring-competition/results
ability of a corresponding theory solver. In practice, this means that this approach is targeting those domains, where so-called SMT-solvers have made an impact in recent years (e.g., using linear integer inequalities, or arrays), whereas the approach in this thesis is general in that regard. Also see Nieuwenhuis et al. [2006] for an overview on techniques and logics used in SMT-solving. Furthermore, the particular theory and universe is fixed on the data side and satisfiability for the first-order theory (such as arithmetic, arrays, lists, or uninterpreted functions) is decidable.

Tool support for TDL is implemented as part of jUnitRV [Decker et al., 2013], and currently uses the SMT solver Z3 [de Moura and Bjørner, 2008].

First-order RV-LTL with numerical constraints (LTL₄–FO₃). More recently Medhat et al. [2014] have presented an extension of the 4-valued RV-LTL semantics (see Def. 2.5.2) to first-order temporal logic, where quantifiers can have metric constraints; that is, quantifiers are of the form

$$\forall_{\sim} x : p(x) \Rightarrow \varphi \quad \text{or} \quad \exists_{\sim} x : p(x) \Rightarrow \varphi,$$

where $\sim \in \{<, >, \leq, \geq, =\}$, $k \in \mathbb{R}$, and $l \in \mathbb{Z}^+$. One can see that quantifiers are guarded in a similar way as in Stolz [2010] as well as in our logic. A metric constraint of the universal quantifier, for example $\forall_{>0.5}$, denotes that only more than 50% of property instances of $\varphi$ have to evaluate to true to satisfy the quantified formula (i.e., $\forall_{=1}$ denotes the standard semantics of the universal quantifier), whereas $\exists_{<3}$ means there have to be strictly less than 3 property instances of $\varphi$ evaluating to true (i.e., $\exists_{\geq1}$ denotes the standard semantics of the existential quantifier).

The logic further differs from LTL₄ in that nested quantifiers are restricted to appear leftmost in a formula without being mixed with temporal operators. Their monitoring algorithm, much like the one presented in this thesis, also constructs submonitors to evaluate quantifiers at runtime inspired by a divide and conquer algorithm. They then present an implementation using MapReduce [cf. Dean and Ghemawat, 2010], capable to run in parallel on GPUs, which leads to almost negligible monitoring overhead at runtime.

Temporal description logic (ALC-LTL). Baader et al. [2012] propose a monitoring approach for the logic ALC-LTL, which is a combination of LTL and the description logic ALC. More precisely, instead of propositional variables like in LTL, axioms of the description logic ALC are used to describe the state of a system. Their monitor construction essentially boils down to reusing the one for LTL based on $\text{BA}$, introduced by Bauer et al. [2011]; that is, they abstract ALC-formulae by propositions and ensure that $\omega$-words in the propositional case correspond to abstractions of ALC-LTL-structures, to then follow the known propositional $\text{BA}$ construction. Like in the LTL case, they solve the prefix problem. The size of the constructed monitors is like in the propositional case double-exponential and also optimal.
4.6 Summary

The first part of this chapter introduces general definitions of an [SA] and provides a sound translation from LTL^{FO} to an SA. In the second part, a monitor construction based on the new automaton model is provided, and optimisations for its implementation—to reduce the monitor’s states at runtime—are discussed. To the best of the author’s knowledge, the algorithm presented in this chapter is the first to devise impartial monitors, i.e., address the prefix problem instead of a (variant of the) word problem, for policies given in an undecidable first-order temporal logic. Furthermore, experiments put the size of the monitor at runtime in relation with its optimised version, and further with a naive rewriting-based monitor for LTL^{FO}. What is more, the monitor (and LTL^{FO}) are compared with other approaches (and their according languages defined in a broader sense also wrt. traces with data) in the literature.
Chapter 5

Towards a hierarchy of effectively monitorable languages

The experimental results of the previous chapter, although not comprehensive, have shown that it is possible to build an LTL\textsuperscript{FO} monitor for a large variety of formulae, including those on which the popular software specification patterns are based (see §2.4); i.e., practically useful ones. However, as both the satisfiability as well as the prefix problem are undecidable in LTL\textsuperscript{FO} (or any other extension of LTL towards full first-order logic), it is natural to ask what properties a language (or formula) needs to have, in order to be effectively monitorable; that is, by a monitor (the one presented in the previous chapter or otherwise) which does not have to store unbounded amounts of observed data, or by one which does not return ? until the end of time.

Due to the undecidability results, there exist obviously boundaries to what a monitor can achieve, irrespective of the algorithm it is based upon. In this chapter, we will outline and formalise some of these boundaries and this way provide a first classification towards a kind of hierarchy of monitorable languages or more precisely: languages, for which practically useful monitors can be built.

5.1 Monitorability

Pnueli and Zaks [2006] were the first to formalise a notion of monitorability in the setting of propositional LTL, which can be expressed in terms of good and bad prefixes as follows.

Definition 5.1.1 (Monitorability). A formula \( \varphi \in \text{LTL}(\text{AP}) \) is monitorable, if for all \( u \in (2^{\text{AP}})^* \) there exists a \( v \in (2^{\text{AP}})^* \) s.t. \( uv \in \text{good}(\varphi) \) or \( uv \in \text{bad}(\varphi) \) holds.

This definition asserts that a monitor (the one in this thesis or otherwise) cannot reasonably monitor a language that does not have a good or a bad prefix, because these are exactly what the monitor detects. Consider, for example, the propositional LTL formula \( \text{G}(p \Rightarrow \text{F}q) \): it describes a typical liveness property\(^1\), meaning that

\(^1\)According to Alpern and Schneider [1985], a language \( \mathcal{L} \subseteq \Sigma^\omega \) is called a liveness language, if for all prefixes \( u \in \Sigma^\omega \), it holds that \( \exists v \in \Sigma^\omega : uv \in \mathcal{L} \). As from the definition follows that infinite traces are
always every request is eventually answered at some point in time in the future. This property clearly has no good or bad prefix. Therefore, a monitor being coherent with a prefix semantics, such as the LTL₃ semantics (see Def. 2.5.1), could only return ? ad infinitum (recall the RV-LTL semantics in Def. 2.5.2 which was invented to return somewhat more meaningful information).

For LTL, monitorability can be decided in ExpSpace [Bauer, 2010], but the undecidability of the satisfiability (and therefore prefix) problem of LTLRO means that for LTLRO monitorability is also undecidable by way of a similar reduction already used in the proof of Theorem 3.2.5 or Bauer [2010, Theorem 2].

**Theorem 5.1.1.** Monitorability for LTLRO is undecidable.

**Proof.** We will reduce the undecidable satisfiability problem to the monitorability problem of LTLRO, i.e., we show for a given ϕ ∈ LTLRO and first-order structure ℵ that $L(ϕ)_{ℵ} = \emptyset$ iff ϕ is monitorable, where ϕ is constructed in the following way:

$$ϕ := Gp(c) \lor GF(q(c) \land ϕ),$$

(⇒:) If $L(ϕ)_{ℵ} = \emptyset$, then $ϕ \equiv Gp(c)$ is clearly monitorable.

(⇐:) Assume that ψ is monitorable, but that $L(ϕ)_{ℵ} \neq \emptyset$. Let ($ℵ$, u) be a trace, where $u \in (2^{Act})^*$ and $Act^* := \bigcup_{p' \in V \setminus \{p\}} \{(p', d) \mid d \in D_{p'}\}$ and $ℵ \sim ℵ$. It is easy to see that $\{(ℵ, uw) \mid (ℵ, w) \in Ev^u\} \cap L(Gp(c))_{ℵ} = \emptyset$, but $\{(ℵ, uw) \mid w \in Ev^u\} \cap L(ϕ)_{ℵ} \neq \emptyset$. Hence, for ϕ to be monitorable, u has to be extensible with some ($ℵ$, v), where $v \in Ev^*$, such that either $\{(ℵ, uwv) \mid w \in Ev^{uv}\} \cap L(GF(q(c) \land ϕ))_{ℵ} = \emptyset$, or such that $\{(ℵ, uwv) \mid w \in Ev^{uv}\} \subseteq L(GF(q(c) \land ϕ))_{ℵ}$. Now, observe that irrespective of our choice of ϕ (including the case $ϕ = true$), as long as $L(ϕ)_{ℵ} \neq \emptyset$, the set $L(GF(q(c) \land ϕ))_{ℵ}$ neither has a good nor bad prefix. This means that $∃u \in Ev^*, ∀v \in Ev^*, \{(ℵ, uwv) \mid w \in Ev^{uv}\} \cap L(ϕ)_{ℵ} \neq \emptyset \land \{(ℵ, uwv) \mid w \in Ev^{uv}\} \subseteq L(ϕ)_{ℵ}$; that is, ϕ is not monitorable. Contradiction.

## 5.2 Trace-length independence

However, assuming we have a monitorable LTLRO specification in the above sense (e.g., in form of a safety formula for which monitorability is obvious), it does not automatically mean, there exists an efficient monitor for it. Unlike in the setting of propositional LTL, a first-order monitor may have to store parts of or even the entire trace—depending on the formula. For example, an arithmetic LTLRO formula such as $G∃x : p. XG∃y : q. x ≠ y$ requires storing all the different ground atoms $p(d)$ appearing in the trace over time. Note that storing of all the $p(d)$s can be necessary even if we restrict ourselves to structures without interpreted predicate symbols, which becomes clear when looking at the formula $G∃x : p. XG(¬p(x) \land ∃y : p. p(y))$. On the other hand, a monitor for a formula, such as $G∀x : p. x ≥ k$, where $k$ is a

required for detecting satisfaction (or violations) of liveness properties, these are generally inadequate for runtime monitoring.
5.3 Strong trace-length dependence

constant, does not need to accumulate any information over time at all. The reason is that for every new occurring event added to the trace, it can be directly decided whether for all contained $p(d)$ it holds that $d \geq k$. Then, if they are, the monitor will continue, and if at least one of them is not, the monitor has detected a bad prefix and can stop. While we have already informally stated what trace-length independent means in Property 2, §1.2.2, we formalise its notions in the following.

By $|M_\varphi|$, we refer to the size (i.e., space) consumption of the monitor (see, for example, §4.1 for the definition of the size of the monitor in this thesis).

**Definition 5.2.1** (Trace-length independence). A formula $\varphi \in \text{LTL}^{\text{FO}}$ is trace-length independent, if it is monitorable and there are constants $k, c \in \mathbb{N}$ and a sound and anticipatory monitor for $\varphi$, $M_\varphi$, s.t. for any trace with no event $\sigma$, s.t. $|\sigma| > c$, it holds $|M_\varphi| \leq k$.

Defining trace-length independence without a bound $c$ on individual events of all traces would exclude certain formulae, which one might obviously want to consider as trace-length independent; that is, the definition would be too tight. Consider for example the formula $G \forall x : p. Xp(x)$, where two consecutive events have to contain a common subset of actions. This formula obviously requires a monitor to only store parts of a constant length of the trace (and therefore never depends on the complete trace), as only actions of the previously event need to be kept to compare with the ones in next. Without considering $c$, however, one could always construct a trace which contains individual events whose cardinality is sufficiently large, s.t. the corresponding monitor’s size has to exceed $k$, i.e., $|\sigma| > k$ (this even the case if the monitor does not store, but just processes events, like for the formula $G \forall x : p. x \geq k$ above). Hence, we require a $k$ to exist always wrt. a given $c$.

The reason for demanding a sound (i.e., impartial) monitor for a given $\varphi$ is obviously due to the fact that one could always come up with a monitor that completely ignores the LTL$^{\text{FO}}$ semantics, but whose space consumption is bounded by 1, for example. But assuming that we are, in fact, dealing with a sound monitor, it could still return ? until the end of time and thus render the monitoring process useless. Hence, we demand that $M_\varphi$ also be anticipatory, although this constraint (unlike soundness) could also have been weakened: all we really need here is that the monitor whenever it reads a good (resp. bad) prefix for $\varphi$ informs the user about it as soon as possible. In other words, a sound monitor that does not return the correct result until some exponential time in the future, could be as practically useless as an incorrect one.

Finally, note that the monitor construction for propositional LTL formulae given by Bauer et al. [2011] provides a formal argument to our intuition, namely that every monitorable propositional LTL formula is also trace-length independent. Therefore, we have not defined this notion first for LTL and then lifted it to first-order logic, as we have done it in §3.2, but stated it immediately in the context of LTL$^{\text{FO}}$.
Towards a hierarchy of effectively monitorable languages

For example, the LTL$^\forall$ formula $G \exists x : p. (r(x)U \exists y : q. x = y)$, which helps to illustrate the following point: the amount of information a corresponding monitor needs to accumulate over time is not a priori clear, because it depends on the actual data inside the trace; that is, if every new event that is added to the trace contains a $p(d)$, $r(d)$, and a $q(d')$ with $d = d'$, then it is clearly not necessary to keep any of this information around. If, on the other hand, a trace contains a $p(d)$ and $r(d)$, in each new addition, but no corresponding $q(d')$ (or, at least, not for a very long time), then the monitor needs to memorise the individual $p(d)$ until the corresponding $q(d')$ occurs. If the $q(d')s$ never occur, then the monitor will have to remember an unbounded amount of trace information. And if they do occur, its memory consumption increases until that point (after which “garbage collection” will clean up again), which we cannot determine in advance. Another such formula with similar characteristic was already introduced in Example 4.2.1, where logins and according IP addresses need to be kept until the logout of a user. These kind of formulae can be arguably effectively monitored, as the system under scrutiny will grow comparatively and need to keep the logins around as well. In other words, monitors for these formulae grow and shrink linearly in relation to the memory required by the system, which is practically acceptable.

This is different from the trace-length dependent example given in §5.2, where the corresponding monitor’s space consumption was bound to grow ad infinitum, irrespective of the data in the trace (unless, of course, the monitor finds a bad prefix, in which case it can stop altogether).

**Definition 5.3.1** (Strong trace-length dependency). Let $|M_\varphi(t)|$ denote the space consumption of some monitor $M_\varphi$ after processing prefix $t$. Then, $\varphi$ is called strongly trace-length dependent, if it is monitorable and for any such $t$, there exists a suffix $t'$, s.t. $|M_\varphi(tt')| > |M_\varphi(t)|$, but no suffix $t''$, s.t. $|M_\varphi(tt'')| < |M_\varphi(t)|$.

---

![Figure 5.1: The hierarchy of effectively monitorable languages.](image-url)
To be absolutely clear: the formula $G \exists x : p \land (r(x)) U q : q \land x = y$ is merely trace-length dependent, but not strongly trace-length dependent, whereas $G \exists x : p \land XG \exists y : q \land x \neq y$ is strongly trace-length dependent. Clearly, one can monitor the former, but not the latter, or at least, not for a very long time, because then the monitor will necessarily run out of space. Both formulae, however, satisfy the conditions of monitorability (see Definition 5.1.1). If a formula is both monitorable and not strongly trace-length dependent, we say it is effectively monitorable. Consequently, Fig. 5.1 depicts the different categories of languages defined in this chapter, and how these are related in the hierarchy. While the outer rectangle indicates all definable LTL$^{FO}$ languages, the grey area marks the effectively monitorable subset of languages.

Table 5.1 shows example formulae in each of the categories, without using any interpreted predicate symbols; thus, these categories are even relevant independent of the concrete computable structure chosen for monitoring (e.g., predefined arithmetic operators). Note that all formulae in Table 5.1 are monitorable. This is because a monitor does not need any storage if it is known that no conclusive result can be ever given; and therefore a further characterisation of non-monitorable formulae is by definition meaningless.

### Table 5.1: Trace-length (in)dependent example formulae.

<table>
<thead>
<tr>
<th>independent</th>
<th>dependent, but not strongly</th>
<th>strongly dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \forall x : p \land r(x)$</td>
<td>$G \exists x : p \land X \neg p(x) U q(x)$</td>
<td>$G \exists x : p \land XG (\neg p(x) \land \exists y : p(y))$</td>
</tr>
</tbody>
</table>

5.4 Summary

This chapter proposes definitions towards a classification of effectively monitorable languages; that is, a language must be monitorable and not strongly trace-length dependent to fulfil this property. In a nutshell, monitorability ensures that a monitor is able to return meaningful results for a formula, and not an inconclusive verdict until the end of time. Strongly trace-length dependent means that a monitor cannot check such kind of formula, unless growing continuously without ever shrinking. Trace-length independent means that a monitor’s required memory is bound by a constant. Note that every LTL formula can be monitored trace-length independently.

While definitions in this chapter allow for a classification of languages (and formulae) according to how efficient a monitor is that we can build for them, it is currently not known whether or not (strong) trace-length dependence is decidable. Having said that, in many practical scenarios the user knows whether or not a formula is monitorable and trace-length independent, so that this is not really a showstopper for first-order monitoring in itself, but more of theoretical interest.
Towards a hierarchy of effectively monitorable languages
Proof of concept: Android malware detection

The landmark work undertaken by the AMGP, initialised by Zhou and Jiang [2012], is the first in the security community that comprehensively collected and systematically analysed more than 1,200 Android malware samples. Despite the high total amount of unique samples, their study reveals that those can be divided into only 49 families and described by even fewer recurring attack patterns, which fall into the following categories: personal information stealing, financial charges, privilege escalation, and malicious payload activation. Based on these patterns and the actual malware files from the AMGP, a proof of concept has been conducted: First, §6.3 formalises common malicious behaviour in the specification language LTL$_K$. Sec-

Figure 6.1: Process of malware detection via runtime verification.
ond, §6.4 discusses experimental results from identifying real malwares on a user’s Android device—by checking their runtime footprint against the set of specifications. False positives are analysed for a set of 61 popular benign apps in the Android Play Store. Suspicious behaviour was detected for 46 out of 49 malware families (93.9%), while generating 28% false positive alerts. Fig. 6.1 depicts the idea of using runtime verification for malware detection.

Section §6.2 introduces the developed monitoring app, MonitorMe, which was built to undertake the experiments. More precisely, this section details on the technical means of the DroidTracer library for Android event interception. DroidTracer is used by MonitorMe to retrieve relevant events about app behaviour.

Section §6.5 discusses related work and DroidTracer’s major advantage compared to other tools which are usually either app- or platform-centric (i.e., approaches that modify either apps or the Android OS, respectively).

### 6.1 Android security concepts in a nutshell

Let us discuss the important details of the Android architecture and security concepts that are relevant to this chapter. Firstly, Android apps and most of the Android stack are written in Java (two topmost layers in Fig. B.3), whereas a modified Linux kernel serves as the platform’s low-level OS (bottom layer in Fig. B.3). Apps on Android are “sandboxed”, meaning that each executes within its own virtual machine (VM), and, from an OS point of view, as unique user; unlike standard Linux processes, which inherit the user identifier (UID) of the user who started them, Android apps all have a unique UID. In other words, each app is treated as an individual user from the low-level OS’s point of view. This strict “sandboxing” basically ensures that one app cannot modify (or even read it, unless dedicated inter process communication (IPC) takes place, which is discussed on a technical level in more detail in §6.2) the data of another installed app. For the same reason, apps cannot access sensitive information from system resources or built-in sensors unauthorised, including contact book entries (which are stored by the Android OS in an SQLite database) or Global Positioning System (GPS) coordinates. The “sandboxing” also encompasses system calls. Apps can invoke system calls directly (or more commonly through ways of the Java application programming interface (API)), in order to access files or connect to the internet; however, the underlying Linux kernel has been modified to check the permission of such calls based on group identifiers (GIDs). These are assigned to apps at installation time. Unfortunately, the harm caused by a malicious apps, is therefore not restricted to its “sandbox.” In fact, there are several ways in which a malicious app could exploit the device’s capabilities, or spy on its users. One way is to exploit vulnerabilities of some system library or the Linux OS, in order to gain superuser...

---

2. For a more comprehensive overview see Felt et al. [2011a], http://developer.android.com/ or http://developer.android.com/guide/topics/security/security.html
3. There is an exception to this rule, but this is not relevant here: apps which share a developer’s signature may run under the same UID.
rights on the device [Davi et al., 2010]. Another way occurs through users unintentionally granting malware access during installation. This is due to Android’s static permission system, which we discuss in the following.

Whether or not an app is allowed to use a certain functionality that an Android device offers is primarily determined at install-time, when the standard Android installer presents to the user a list of required app permissions. Users cannot revoke individual permissions that they may not feel comfortable with or that they do not understand, rather they need to grant all permissions or cannot install the app. Consequently, many users do not review the permissions at install-time [Felt et al., 2011b]. In fact, Android’s permission system is predominantly static, meaning that once an app is installed, users have basically no means of controlling that app’s runtime behaviour. For example, once an app has been granted permission to send SMS, it may do so in the background without requesting further user confirmation. According to the official documentation[4] the lack of dynamic security mechanisms is a design principle: “Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user experience to the detriment of security.”

Note that the situation on other mobile platforms, like the outdated Nokia’s Symbian OS, is similar [cf. Bose et al., 2008].

Since Android 6.0, which has been released at the time of writing this thesis, users are able to revoke permissions individually from apps at anytime. Therefore, apps developed for an older Android version often crash, as they usually do not catch exceptions when access to a resource is suddenly forbidden. Apps for Android 6.0 can be developed in such a way that they request permissions from the user at run-time[5]. These kind of apps will be installed right away without review of permissions. The first time they use a feature that uses a new permission, the user can allow or deny the access of requested sensitive information or system capabilities. Note that this new feature, unlike MonitorMe, does not let the user analyse the behaviour of apps nor provide patterns for identifying whether apps are potential malware.

6.2 A monitoring tool for the Android platform

To enable the modular way of malware detection on a user’s device, the author of this thesis has developed a monitoring app, called MonitorMe [Küster and Bauer, 2015].[6] This has two main components, which are depicted in Fig. 6.2: (1) a framework for collecting system events on Android (grey area, called DroidTracer)[7]and (2) an analysis part running on top of DroidTracer, which receives those events in chronological order and incrementally “feeds” them into monitors generated by Ltlfo2mon[8]. For details of the monitor construction recall §4.2 MonitorMe creates a monitor for each policy that specifies a certain malware behaviour and runs a copy of them per app.
Proof of concept: Android malware detection

Figure 6.2: Architecture of MonitorMe.

under inspection. In other words, verdicts are returned individually per attack pattern and app.

Recall that Ltlfo2mon is written in Scala, but is compatible to run as part of an Android app in Java. It is worth pointing out that DroidTracer works without polling for events, i.e., a Java callback method is triggered whenever a new system event occurs.

DroidTracer. In the following are explained the inner workings of DroidTracer (three sub-components marked in white inside the grey area); meaning, the novel way on how interactions between apps and the Android platform are intercepted without requiring platform or app modifications. As there is no public API for this task—not even on a rooted device—nor any complete documentation about Android’s internal communication mechanism, this approach is mainly based on insights gained from reverse engineering.

6.2.1 System call interception

We exploit the Android security design, which ensures that the control flow of all apps’ actions that require permission, such as requesting sensitive information (GPS coordinates, device ID, etc.) or connecting with the outside world (via SMS, internet, etc.), must eventually pass by one of the system calls in the Linux kernel. This way, requests are delegated to a more privileged system process that handles the request. In other words, intercepting the control flow at a central point in kernel space does not allow apps to bypass the approach in this thesis. Furthermore, system calls are unlikely to change so that hooking into them is robust against implementation details
on different Android versions.

Hence, the idea is to use kprobes\(^9\) (the kernel’s internal debugging mechanism) to intercept system calls in a non-intrusive way. More specifically, a custom kernel module was built (bottom sub-component in Fig. 6.2), which contains handler methods that get invoked by small bits of trampoline code (so-called probes). Those are added by kprobes dynamically to the following system calls:

- \texttt{sys\_open(const char __user *filename, ...)}
  - opens files for read or write that are identifiable by the path filename.

- \texttt{sys\_connect(int sockfd, const struct sockaddr *addr, ...)}
  - establishes an internet connection, where addr indicates the according IPv4 or IPv6 address.

- \texttt{do\_execve(char *filename, char __user *argv, ...)}
  - executes a program or shell script with name filename and with arguments argv.

- \texttt{ioctl(...)}
  - is generally used to control kernel drivers, such as Android’s Binder driver.

From the function arguments of \texttt{ioctl} we cannot directly retrieve relevant information (unlike for the other system calls shown above, which provide to us IP addresses, opened files, or executed program names). The reason is that information is compactly \textit{encoded} (for efficiency reasons), when sent through \texttt{ioctl} by Android’s own IPC mechanism, called Binder\(^{10}\). As Binder handles the majority of interesting interactions between apps and the Android platform, its decoding is crucial for the malware analysis. Hence, for a deeper understanding of Binder’s control flow, we look at the following Java code snippet of a method call that an app developer might write to send an SMS.

\begin{verbatim}
SmsManager sms = SmsManager.getDefault();
            sms.sendTextMessage("12345", null, "Hello!", null, null);
\end{verbatim}

Fig. 6.3 illustrates the control flow of the Binder communication when this code is executed. All Java code of an app is compiled into a single Dalvik executable \texttt{classes.dex} (upper left box), which runs in its own Dalvik VM. The called method \texttt{sendTextMessage} is part of the Android API (lower left box), which is linked into every app as a \texttt{JAR} file. But instead of implementing the functionality of sending an SMS itself, it rather hides away the technical details of a \textit{remote procedure call} (RPC), that is, a call of a Java method that lives in another Dalvik VM (right box). What further happens is that \texttt{SmsManager} calls the method \texttt{sendText} of the class \texttt{Proxy}, which has been automatically generated for the \texttt{ISms} interface. The \texttt{Proxy} then uses the class \texttt{Parcel} to \textit{marshal} the method arguments of \texttt{sendText} into a byte stream.

\footnotesize\begin{itemize}
  \item \begin{verbatim}
  \end{verbatim}\footnotesize
\end{itemize}
which is sent (together with other method call details) via the Binder driver to the matching Stub of ISms (lower right box). There, the arguments are unmarshalled and the final implementation of sendText in the SMS service is executed. As the SMS service is running on a Dalvik VM privileged to talk to the radio hardware, it can send the SMS. The Proxy and Stub are marked in grey to denote the interface endpoints of the RPC calls on both sides of the client and server, respectively.

6.2.2 Unmarshalling

The main challenge in reconstructing method calls was to reverse engineer how Binder encodes them to send them through the kernel, so that the task of unmarshalling for the malware analysis can be automated. Like for the code snippet of sending an SMS, we aim at reconstructing every method call in its original human readable format (including its Java method arguments and types). In what follows, it is described how it was achieved and what the implementation of this feature looks like on a technical level.

All we can intercept in the kernel is the following C structure, which wraps the information copied by Binder driver from the sender into the address space of the receiving process.

```c
struct binder_transaction_data {
    unsigned int code; // contains for the SMS example value 5, which
    // encodes the method name of sendText
    uid_t sender_euid; // UID of app initiating the request
    const void *buffer; // Fig. 6.6 shows its content of sendText
};
```

A comprehensive technical report contains more details on the implementation [Küster and Bauer, 2014] and shows how binder_transaction_data is intercepted during
6.2 A monitoring tool for the Android platform

A certain stage of the Binder driver communication, which follows a strict protocol. The integer `sender_euid` provides us with the UID to unambiguously identify the sender app of a request (i.e., for the SMS example in Fig. 6.3 the UID of the app in the left box). However, the method name the integer `code` translates to, and which arguments are encoded in the byte stream of `buffer`, is not transmitted. This is mainly for efficiency reasons. We have a closer look at some code of the Proxy in Fig. 6.4 and Stub in Fig. 6.5 which are automatically generated for the interface ISms, to better understand why there is no need for Binder to send this information.

```java
01: public void sendText(String destAddr, ..., String text, ...) ...
02: {
03:     android.os.Parcel _data =
04:         android.os.Parcel.obtain();
05:     ...
06:     _data.writeInterfaceToken(DESCRIPTOR);
07:     _data.writeString(destAddr);
08:     ...
09:     _data.writeString(text);
10:     ...
11:     mRemote.transact(Stub.TRANSACTION_sendText,
12:         _data, ...);
13:     ...
14: }

Figure 6.4: Auto-generated Proxy for the ISms interface.
```

```java
01: private ... String DESCRIPTOR =
02:     "com.android.internal.telephony.ISms";
03: ...
04:     switch (code) { ...
05:         case TRANSACTION_sendText: {
06:             data.enforceInterface(DESCRIPTOR);
07:             ...
08:             _arg0 = data.readString();
09:             ...
10:             _arg2 = data.readString(); ...
11:             this.sendText(_arg0,..., _arg2,...);
12:         }
13: ...
14: static final int TRANSACTION_sendText =
15:     (IBinder.FIRST_CALL_TRANSACTION + 4);

Figure 6.5: Auto-generated Stub class for the ISms interface.
```
When the Proxy makes the actual RPC for sendText via Binder (Fig. 6.4 line 12), it includes the integer TRANSACTION_sendText defined in its corresponding Stub (Fig. 6.5 lines 14-15). It was discovered that this is the value of code that can be found in binder_transaction_data. The second argument _data is an instance of the class Parcel and relates on a lower level to the buffer that is intercepted. More specifically, the Proxy takes a Parcel object (reused from a pool for efficiency) and then writes the DESCRIPTOR (Fig. 6.4 line 7), which is the name of the interface ISms (Fig. 6.5 lines 1-2), followed by the method arguments into it (Fig. 6.4 lines 8-10). It can be observed that this is done in the order of the arguments appearing in the method signature. Furthermore, dedicated write methods are used provided by Parcel, such as writeString. When the Stub receives the call, it executes the TRANSACTION_sendText part of a switch construct (Fig. 6.5 line 5), which reconstructs the arguments from the byte stream of the Parcel object. It uses thereby the equivalent read methods in the exact same order as the write methods have been used. Based on those key observations was designed the following three-step algorithm, in order to automate unmarshalling for arbitrary method calls with Droid-Tracer (top sub-component in Fig. 6.2):

1) **Unmarshal interface name (ISms)**

   (a) Take a Parcel object and fill it with the byte stream buffer. This is possible, as the class Parcel is public and provides an according method.

   (b) Read the DESCRIPTOR from the Parcel object via method readString, as it is always the first argument in buffer (see Fig. 6.6).

2) **Unmarshal method name (sendText)**

   (a) Use Java reflection to find the variable name with prefix TRANSACTION_ and code assigned to in the Stub of the unmarshalled interface name. This works, as every app, including MonitorMe, has access to the dynamically linked JAR of the Android API.

3) **Unmarshal method arguments (“12345”, null, “Hello!”, null, null)**

   (a) Determine the order and types of method arguments by accessing the signature of the unmarshalled method name via reflection.

   (b) Apply Parcel’s read methods according to the type and order of arguments appearing in the method signature. This works for Java primitives, but it was also reverse engineered how complex objects are composed into Java primitives.

It is worth pointing out that the unmarshalling algorithm does not rely on low-level Binder implementations. Even if these vary for different Android versions, it is always possible to use the correctly functioning Java read methods of the class Parcel, which are also used by the Android framework itself on a specific device.
6.3 Specifying malware behaviour

Section §6.3.1 describes the underlying system model of app behaviour, which is based on DroidTracer results, i.e., DroidTracer determines the different U-operators that can appear in a trace. Section §6.3.2 provides a specification manual on how unwanted app behaviour can be described and §6.3.3 discusses formulae derived from common malware behaviour of the AMGP. From a formal point of view, we merely use safety formulae and MonitorMe to detect finite counterexamples (i.e., bad prefixes, as defined in Def. §3.2.2).

6.3.1 System model of app behaviour

The observable behaviour of apps on a technical level equals the system events that can be collected with DroidTracer. Hence, whenever an app causes a native system event on the Android platform that can be intercepted with DroidTracer, we capture

\[ \text{Figure 6.6: The buffer sent via Binder containing the arguments of the method } \text{sendTextMessage.} \]

6.2.3 Netlink communication

As event interception takes place solely inside the kernel space and unmarshalling relies on access to the Android API, we need a mechanism that allows to pass data from inside the kernel module up to DroidTracer in user space. Moreover, we need to send data also in the other direction so that the user can control the kernel module for even the most basic tasks, for example, to switch event interception on and off. Android has no built-in way to serve as a solution, but it was possible to use netlink (a socket based mechanism of the Linux kernel) to bidirectionally communicate with user space. As only the kernel but not the Android API offers netlink support, a custom endpoint was built (middle sub-component in Fig. 6.2), using the Netlink Protocol Library Suite (libnl). This means, the middle component of DroidTracer is a shared C++ library. It contains extracted core functionality for netlink from libnl and was recompiled for Android using Android’s Native Developer Kit (NDK). Note that netlink implements a callback principle so that rather than polling the kernel module for new occurring system events, DroidTracer can push them all the way to the analysis component.

it as an action. We represent actions in the internal, logical model by ground terms 
\( p(d_1, \ldots, d_n) \), where \( p \) is an uninterpreted predicate symbol (i.e., \( U \)-operator) and \( d_i \) is a domain value. Recall that LTL\(^{30} \) is sorted, so that we can distinguish the \( d_i \)s coming from different sub-domains (e.g., the set of all integers or strings). Typically, \( p \) denotes the method and interface name of an intercepted method call, and \( d_i \) (if the method has any) its \( i \)th unmarshalled method argument. For example, we write

\[
\text{sendText}@\text{ISms}("12345", \text{null}, "Hello!", \text{null}, \text{null})
\]

for a ground term representing the sending of an SMS with text “Hello!” to phone number 12345, where we conventionally delimit method and interface name by the @-symbol. See Table B.4–B.6 in the appendix for the more than 100 interface names that DroidTracer has intercepted on real phones. For sorts of the arity of \( p \), we use the Java types of the actual method’s arguments involved in the RPC. Note that the phone number “12345” is here of type \text{String} (and not \text{int}), as the Java type is defined this way for the method signature of \text{sendText}. This allows for characters such as “+” to account for factors like country codes.

Recall that we refer to finite sets of actions as events. An app’s observed behaviour over time is therefore a finite trace of events. In the undertaken analysis, events contain only one action, and are ordered by the position at which the corresponding system event has been sent from DroidTracer’s kernel module via netlink. In other words, there is no predefined delay between events as the trace is only extended by one event whenever a new system event occurs. Table 6.1 shows a selection of events from a trace that was collected for a sample of the malware family Walkinwat. Distinct malware samples (i.e., with different hash values) are usually grouped under the same family name if they share the same attack patterns. Note that Table 6.1 resembles the table of an SQLite database that DroidTracer uses to persist events on the phone. Each row represents a system event. The column ID indicates its position in the trace and the remaining columns show the outcome of the three steps in the unmarshalling algorithm (see §6.2.2). For all other system calls that are intercepted additionally to Binder—and which require no unmarshalling—DroidTracer returns the generic interface name “syscall”, the actual function name of the system call in the kernel, as well as the function’s intercepted arguments. Let us look for example at the third row, which can be written as a ground term

\[
\text{sys_connect}@\text{syscall}("wringe.ispgateway.de")
\]

This means an internet connection has been established to \text{wringe.ispgateway.de} via the system call \text{sys_connect}. An action at position \( i \in \mathbb{N} \) in some trace means that at time \( i \) this action holds (or, from a practical point of view for the 397th event in Table 6.1 that Walkinwat has requested the Android framework to send an SMS to number 451-518-646 with text “Hey, just ...”).

The row with interface name \text{ISurfaceComposer} contains for the method name value “N/A, code: 10”. This denotes that DroidTracer was unable to decode the real Java method name; and therefore, it returns at least the value of \text{code} found in
### §6.3 Specifying malware behaviour

#### Table 6.1: Trace of system events for malware `Walkinwat` collected by DroidTracer.

<table>
<thead>
<tr>
<th>ID</th>
<th>Interface</th>
<th>Method</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>334</td>
<td>IPhoneSubInfo</td>
<td>getDeviceId</td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>IContentProvider</td>
<td>QUERY</td>
<td>content://contacts/phones, null, null, null, display_name ASC</td>
</tr>
<tr>
<td>392</td>
<td>syscall</td>
<td>sys_connect</td>
<td>wringe.ispgateway.de</td>
</tr>
<tr>
<td>397</td>
<td>ISms</td>
<td>sendText</td>
<td>451-518-646, null, “Hey, just downloaded a pirated App off the Internet, Walk and Text for Android. I’m stupid and cheap, it costed only 1 buck. Don’t steal like I did!”, null, null</td>
</tr>
<tr>
<td>407</td>
<td>IActivityManager</td>
<td>getIntentSender</td>
<td>1, com.incorporateapps.walktext, null, null, 0, Intent { act=SMS_SENT }, null, 0</td>
</tr>
<tr>
<td>408</td>
<td>IActivityManager</td>
<td>getIntentSender</td>
<td>1, com.incorporateapps.walktext, null, null, 0, Intent { act=SMS_DELIVERED }, null, 0</td>
</tr>
<tr>
<td>414</td>
<td>ISurfaceComposer</td>
<td>N/A, code: 10</td>
<td></td>
</tr>
<tr>
<td>492</td>
<td>syscall</td>
<td>sys_connect</td>
<td>wringe.ispgateway.de</td>
</tr>
<tr>
<td>578</td>
<td>IActivityManager</td>
<td>startActivity</td>
<td>null, Intent { act=android.intent.action.VIEW dat=market://details?id=com.incorporateapps.walktext }, null, N/A, N/A, N/A, N/A, N/A, N/A, N/A</td>
</tr>
</tbody>
</table>

*binder transaction data*. Recall that without knowing the method name, DroidTracer cannot unmarshal any method arguments, unlike in the last row of Table 6.1. Here, arguments could at least be partially unmarshalled, wherefore the last seven arguments are represented by the placeholder “N/A”.

#### 6.3.2 A user manual: How to specify malware behaviour in LTL$^F_0$

Let us now look at some common Android malware behaviour and explain how to model this behaviour in terms of LTL$^F_0$ formulae. The intention of this section is to ease the understanding of specified attack patterns in the following section—or for the curious reader, to be able to write their own policies for protection against future
or otherwise unwanted behaviour occurring on Android phones.

Lots of malware is “spyware”, i.e., it sends private user data or sensitive device details “home” to remote locations. In order to access any of this information, “sandboxed” malware has to request a system service; thus, a Binder call takes place and we can refer to its according action when writing a policy. In case of the device ID, we can write

\[ G \neg \text{getDeviceId} \oplus \text{IPhoneSubInfo}, \]

meaning that an app should never query the device ID. As the method getDeviceId has no arguments, the according \( U \)-operator is merely a proposition and therefore no quantifiers are needed. In case we do not want an app to read any names, addresses, etc. from the contact book of the phone, we write

\[ G \neg \exists (uri, \_): \text{QUERY} @ \text{IContentProvider. regex}(uri, \text{". } * \text{ contacts.}"), \]

meaning and app should not query a content provider, where the Uniform Resource Identifier (URI) matches the regular expression "*.contacts.*". A content provider is a wrapper (here of an SQLite database that stores the contacts), which can be queried from other apps via RPC if having the right permissions. In other words, we check if the first argument of the method QUERY contains the URI “content://contacts/phones”, as this unambiguously identifies a query to the contact database. Note that the method QUERY has more than one argument. For readability we use the “\_”-symbol as a placeholder for all remaining variables that do not appear in the formula. Similarly, access to most other data stores on Android can be monitored, such as the one containing all SMS messages, the call history, etc. The predicate regex is an \( I \)-operator, which returns true if a string—here the one bound to uri—matches a regular expression. Note that in both cases above, the formulae resemble the “absence” pattern from the LTL specification patterns discussed in §2.4.

The examples above are generic and therefore likely generate false positives. Therefore, one usually wants to describe behaviour in a more specific way; for example, by expressing that after accessing sensitive information, an app should not forward the information to some “sink” on the device (e.g., the internet). How to specify this is discussed in the following.

All apps of type Android/Actrack.A send GPS location, battery and radio status to a central internet server controlled by the vendor at regular intervals. A policy we may want to monitor in regards to that, more generally, could be “no app should request the GPS location, and later connect to the internet (possibly to transmit said location)”, which is captured by the following formula, where sys_connect@syscall appears in a trace whenever the app under scrutiny triggers the Linux system call sys_connect to some IP address y, and getLastLocation@ILocationManager whenever it requests the device’s current location:

\[
G(\exists x: \text{getLastLocation} @ \text{ILocationManager. regex}(x, \text{". } * \text{ g.ps. } * \text{"}) \rightarrow \\
G \neg \exists y: \text{sys_connect} @ \text{syscall. true}).
\]
An example for the parameter of `getLastLocation` is “Request[ ACCURACY_FINE
gps requested=0 fastest=0 num=1 ]; <app-package-name>”. This is an object of
class `LocationRequest` printed as string, which tells Android how an app wants
to request the location. For instance, “fastest=0” means that location updates are
requested within the fastest interval possible, which is usually only necessary for
navigation apps. Based on this information we could adapt the policy above to re-
veal specifically tracking behaviour. The formula’s pattern is of type “absence” with
scope “after”.

Android/NickiSpy, for example, represents a family of apps which secretly record
a user’s phone conversation on SD card in the compressed .amr format (adaptive
multi-rate). We can detect this family of malware via a simple policy,

\[
G \forall x : \text{sd\_write}@\text{syscall}. \neg \text{regex}(x, ".*\:\amr")
\]

However, should there be legitimate recording of .amr files to SD card, the user
is always able to ignore any reported violations of this policy.

As another example, consider the first ever Android Trojan (`Trojan-SMS.AndroidOS.FakePlayer.a`), disguised as media player, which secretly sent SMS mes-
sages to expensive premium numbers [Leyden 2010]. This could be monitored by a
more general behaviour, i.e., to be notified if any app sends an SMS to a number not
in the contacts:

\[
G \forall x : \text{sendText}@\text{ISms}. \text{contact}(x).
\]

While there may be legitimate violations of this policy, its monitoring at least lets
users keep track of which apps exhibit this type of behaviour. It’s then up to them
to decide to remove an app, if they feel it is not justified. It was predefined a special
I-operator, `contact`, which is non-rigid, i.e., its return value can change for the same
\(x\) depending if the number at the time queried is in the contact book or not. Recall
that we discussed this example already earlier in §3.1. The framework Ltlfo2mon lets
the user define further predicates for the specific need to check for certain behaviour,
which is usually not supported by other specification languages. In other words,
the feature of arbitrary computable predicates is needed for the following Android
analysis.

### 6.3.3 Common, recurring malware patterns

Based on the patterns from four different categories identified by Zhou and Jiang
[2012] in the AMGP, Küster and Bauer [2015] have formally specified various key
characteristics of malware behaviour in LTL\(^{\text{FO}}\). The outcome are the policies listed in
Table 6.2. For readability, we write \(\psi_{i\in[a,b]}\), grouping policies of the same pattern
together, where \(\psi_{i}, \psi'_{i} \) or \(\psi''_{i}\) are auxiliary formulae listed in Table 6.3. We surround a
policy \(\psi_{i}\) with \(n\) square brackets (calling it the \(n\)th refinement of \(\psi_{i}\)), if its bad prefixes
are a strict subset of the bad prefixes of \(\psi_{i}\) surrounded with \(n-1\) square brackets.
For example, a bad prefix of \([\psi_{1}]\)—the first refinement of \([\psi_{1}]\)—has to contain as well
an event of establishing a connection to the internet, after accessing the device ID.
### Table 6.2: Key characteristics of Android malware behaviour specified in LTL<sup>FO</sup>.

#### Personal information stealing

<table>
<thead>
<tr>
<th>ϕ &lt;sub&gt;i∈[1,14]&lt;/sub&gt;</th>
<th>G¬ψ&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ϕ &lt;sub&gt;i∈[1,14]&lt;/sub&gt;]</td>
<td>G(ψ&lt;sub&gt;i&lt;/sub&gt; ⇒ ¬Fψ′)</td>
</tr>
<tr>
<td>[[ϕ &lt;sub&gt;i∈[1,14]&lt;/sub&gt;]]</td>
<td>G(ψ&lt;sub&gt;i&lt;/sub&gt; ∧ ¬ψ′ ⇒ (¬ψ′W(N/A@ISurfaceComposer ∧ ¬ψ′)))</td>
</tr>
<tr>
<td>[[ϕ &lt;sub&gt;i∈[1,4]&lt;/sub&gt;]]′</td>
<td>G(ψ&lt;sub&gt;i&lt;/sub&gt; ⇒ ¬Fψ″)</td>
</tr>
</tbody>
</table>

#### Privilege escalation

ϕ<sub>15</sub> | G¬∃(args) : do_execv@syscall. regex(args,".* su.* | pm(un)?install | amstart.*/") |

#### Launching malicious payloads

<table>
<thead>
<tr>
<th>ϕ &lt;sub&gt;i∈[16,17]&lt;/sub&gt;</th>
<th>G¬ψ&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ϕ &lt;sub&gt;17&lt;/sub&gt;]</td>
<td>G(ψ&lt;sub&gt;17&lt;/sub&gt; ⇒ ¬Fψ′)</td>
</tr>
<tr>
<td>[[ϕ &lt;sub&gt;17&lt;/sub&gt;]]</td>
<td>G(ψ&lt;sub&gt;17&lt;/sub&gt; ⇒ ¬Fψ″)</td>
</tr>
</tbody>
</table>

#### Financial charges

| ϕ<sub>18</sub> | G∀(dest,_) : sendText@ISms. inContactBook(dest) |
| ϕ<sub>19</sub> | G(ψ<sub>17</sub> ⇒ ¬F∃(w, x, y, z, abort) : finishReceiver@IActivityManager. regex(abort, “true”)) |

This describes from a practical point of view a more severe malware behaviour for the user.

**Category 1: Information stealing.** The AMGP discovered that malware is often actively harvesting various sensitive information on infected devices. Thus, the policies ϕ<sub>i∈[1,11]</sub> specify that an app should neither request any permission secured sensitive data, such as the device or subscriber ID, subscriber identity module (SIM) serial or telephone number, or device software version, nor should it query any of the content providers that contain the call history, contact list, phone numbers, browser bookmarks, carrier settings or SMS messages. Policy ϕ<sub>12</sub> covers the harvesting of installed app or package names on a device. Policy ϕ<sub>13</sub> covers the reading of system logs via the Android logging system, called logcat. Note that before Android 4.1, an app could read other apps’ logcat logs, which might contain sensitive messages. Policy ϕ<sub>14</sub> specifies that neither the coarse grain location based on cell towers nor the more precise GPS location should be accessed.

The policies [ϕ <sub>i∈[1,14]</sub>] refine the policies above towards the more suspicious behaviour that an app should not, after requesting the sensitive information, eventu-
Table 6.3: Auxiliary formulae for Table 6.2

| $\psi_1$ | getDeviceId@IPhoneSubInfo |
| $\psi_2$ | getSubscriberId@IPhoneSubInfo |
| $\psi_3$ | getIccSerialNumber@IPhoneSubInfo |
| $\psi_4$ | getLine1Number@IPhoneSubInfo |
| $\psi_5$ | getDeviceSvn@IPhoneSubInfo |
| $\psi_6$ | $\exists (uri,_.): \text{QUERY}@IContentProvider. \text{regex}(\text{uri},\text{".* calls.*"})$ |
| $\psi_7$ | $\exists (uri,_.): \text{QUERY}@IContentProvider. \text{regex}(\text{uri},\text{".* contacts.*"})$ |
| $\psi_8$ | $\exists (uri,_.): \text{QUERY}@IContentProvider. \text{regex}(\text{uri},\text{".* phones.*"})$ |
| $\psi_9$ | $\exists (uri,_.): \text{QUERY}@IContentProvider. \text{regex}(\text{uri},\text{".* bookmarks.*"})$ |
| $\psi_{10}$ | $\exists (uri,_.): \text{QUERY}@IContentProvider. \text{regex}(\text{uri},\text{".* preferapn.*"})$ |
| $\psi_{11}$ | $\exists (uri,_.): \text{QUERY}@IContentProvider. \text{regex}(\text{uri},\text{".* sms.*"})$ |
| $\psi_{12}$ | $(\exists (_.): \text{getInstalledPackages}@IPackageManager. \text{true}) \lor$ |
| | $(\exists (_.): \text{getInstalledApplications}@IPackageManager. \text{true})$ |
| $\psi_{13}$ | $\exists (args): \text{do_execv}@syscall. \text{regex}(\text{args},\text{".* logcat.*"})$ |
| $\psi_{14}$ | $(\exists (_.): \text{notifyCellLocation}@ITelephonyRegistry. \text{true}) \lor$ |
| | $(\exists x: \text{getLastLocation}@ILocationManager. \text{regex}(x,\text{".* gps.*"})$ |
| $\psi'_c$ | $\text{", *. BOOFT_COMPLETED.*"}$ |
| $\psi''_c$ | $\text{", *. SMS_RECEIVED.*"}$ |
| $\psi_{16,17}$ | $\exists (intent,txt,_.): \text{system#scheduleReceiver}@IApplicationThread.$ |
| | $(\text{regex}(intent,c_i) \land \text{regex}(txt,\text{".* < pkg > .*"}))$ |
| $\psi'''$ | $(\exists (_.): \text{sys_connect}@syscall. \text{true}) \lor (\exists (_.): \text{sendText}@ISms. \text{true}) \lor$ |
| | $(\exists (x,intent,_.): \text{startActivity}@IActivityManager.$ |
| | $\text{regex}(\text{intent,}\text{"action.SEND"})$ |
| $\psi''''$ | $(\exists (dest,x,msg,_.): \text{sendText}@ISms. \text{regex}(msg,\text{".* < sensitiveInfo > .*"}) \lor$ |
| | $(\exists (x,intent,_.): \text{startActivity}@IActivityManager.$ |
| | $\text{regex}(\text{intent,}\text{".* < sensitiveInfo > .*"})$ |

ally connect to the internet, send an SMS or exchange data with another app. Even though a detected bad prefix for those policies does not guarantee that information has been leaked, the usage of above “sinks” bears at least its potential. As we treat apps as “black boxes” and sent data could have been encrypted or in other ways obscured by malware, we generally cannot prove leakage—unlike approaches for information flow analysis [cf. Enck et al., 2010]—based on traces we collect. Furthermore, $[[\varphi_{i\in[1,14]}]]$ expresses that there should exist some screen rendering (via
Proof of concept: Android malware detection

N/A@SurfaceComposer) in between information request and potential leakage. A detected bad prefix would mean in this case that the sending of data could not have been caused by some normal user interaction with the app, but rather by some app’s malicious background service. Note that we represent with N/A method names which could not be decoded with DroidTracer. Further note that the first and second refinements are based on the “absence after” and “exists between” specification patterns, respectively.

\[ \phi_i \in [1,4] \]' are further refinements, which only trigger if we find the device ID, etc., as cleartext (represented by the placeholder “< sensitiveInfo >”) in the trace.

Category 2: Privilege escalation. The attack of exploiting bugs or design flaws in software to gain elevated access to resources that are normally protected from an application is called privilege escalation. From the samples in Zhou and Jiang [2012], 36.7% exploit a version-specific known software vulnerability of either the Linux kernel or some open-source libraries running on the Android platform (such as WebKit or SQLite) to gain root privileges (e.g., to replace real banking apps with a fake one, for phishing attacks). Therefore, policy \( \phi_{15} \) lets us for example detect when an app opens a root shell, secretly starts, installs, or removes other packages via the activity manager (am) or the package manager (pm). Monitoring of this behaviour is possible, because the system call do_execve is exclusively used for the execution of any binary, shell command, or shell script on the underlying Linux OS.

However, not all root exploits need to use the su command to be executed (see, e.g., the “RageAgainstTheCage” exploit [cf. Drake et al., 2014, p.75], which leverages on the root shell obtained via the Android Debug Bridge (adb)). Therefore, it is not possible to detect root exploits in general with this formula. Once an app has gained root privileges, it can in principle not just unload MonitorMe’s kernel module, but also uninstall the app itself. In other words, malware on a compromised device would be able to successfully evade the detection from MonitorMe as well as any other mobile anti-virus software.

Category 3: Launching malicious payloads. Apps’ background services—which do not have any UI—cannot only be actively started when clicking on an app’s launch icon, but also by registering for Android’s system-wide events, called broadcasts. The AMGP discovered that 29 of the 49 malware families contain a malicious service that is launched after the system has been booted, or for 21 families when an SMS was received (i.e., they registered for the BOOT_COMPLETED or SMS_RECEIVED broadcast, respectively). Therefore, we consider it as suspicious if services are activated by the broadcasts mentioned above, which we specify in form of \( \phi_{i} \in [16,17] \). We replace “< pkg >” for each app individually with its package name. Note that to monitor this behaviour, we need to intercept system events of the Android system—which has UID 1000—as it starts the services that have registered for a certain broadcast via scheduleReceiver@ApplicationThread. We add the prefix “system#” to those actions; that is, to distinguish them from an app’s action in a trace. Since malware has access to the sender and content of an incoming SMS after registering for SMS_RECEIVED,
we check with the refinements \([\varphi_{17}]\) and \([[\varphi_{17}]]\) for information stealing. This means, similar as specified by the refinements of \(\varphi_{i\in[1,14]}\), the internet should not be accessed, and so on, after the broadcast was received.

**Category 4: Financial charges.** The AMGP discovered apps, such as the Android malware FakePlayer, which secretly calls or register for premium services via an SMS. As this behaviour can result in high financial charges for the user, Google labels the permissions that allow to call or send an SMS with “services that cost you money”. This policy is used as an example throughout this thesis. Recall that instead of defining policies that check outgoing messages against a fixed list of potential premium numbers, \(\varphi_{18}\), more generically specifies that an SMS should not be sent to a number not in the user’s contact book. Since Android 4.2, Google added a similar security check, where a notification is provided to the user if an app attempts to send messages to short codes as those could be premium numbers. We could have specified that apps should not make phone calls to numbers not in the phone book as well, but as this behaviour has not been observed during experiments we neglected to do so.

Before Android 4.4, apps were allowed to block incoming SMS messages. This was used by malware to suppress received confirmations from premium services or mobile banking transaction authentication numbers (TAN). The latter were then forwarded to a malicious user. Thus, policy \(\varphi_{19}\) checks if apps *abort* a broadcast after receiving *SMS_RECEIVED*, in which case the SMS would not be delivered further to appear in the usual messaging app on a device. In other words, if an SMS is blocked, malware can silently register users for premium services.

### 6.4 Identifying malware behaviour

MonitorMe was installed and provided with the policies introduced in §6.3.3 on the test device Nexus S, which was running Android 2.3.6. This old Android version was used, as most malware from the AMGP was built for version 2.3.6. During the experiments, one malware sample for each of the AMGP-families was monitored; that is, first its application package (APK) was installed, and before starting it (i.e., by clicking on its launch icon if it had any), test using and finally uninstalling it, potential background services have been tried to activate by sending the broadcast *BOOT_COMPLETE* via the `adb` and an SMS to the Nexus S. Even though MonitorMe performed *online* monitoring, the trace for each malware was also persisted in an SQLite database on the phone both for repeatability of the experiments and to provide them to other researchers.\[^{13}\]

\[^{13}\]Traces are available at [http://kuester.multics.org/DroidTracer/malware/traces](http://kuester.multics.org/DroidTracer/malware/traces)
6.4.1 Experimental results

Table 6.4 and Table 6.5 summarise for which malware families (49 in total) and policies MonitorMe detected bad prefixes during the experiments. The second column indicates, whether a malware or one of its services crashed during the experiments (e.g., due to incompatibility with the Android version on the test device). Accordingly, observing some critical behaviour might have been missed. The third column tells us if a malware had no launch icon, which is usually intended to stay hidden and spy on the user. SMSReplicator, for example, is used by parents to secretly forward all SMS messages received on their children’s phones. As in general all UIDs above 10000 were monitored\textsuperscript{14} apps without an icon could not bypass the analysis unnoticed. The fourth column shows the number of system events that has been recorded for each malware. The next four columns contain the individual monitor results for the categories introduced in Table 6.2. Note that for readability, monitor results for the first category (information stealing) are aggregated; that is, the fifth column contains the number of all bad prefixes discovered in that category per app. The complete results for this category can be found in the appendix in Table B.1. The cell containing $\phi_{18}$ in the row for Walkinwat denotes that the monitor for $\phi_{18}$ found a bad prefix for the Walkinwat sample after 397 events. As this implies that the same prefix is also a bad one for lower refinements of $\phi_{18}$, we neglect showing this information. The last column shows the number of bad prefixes found in total per malware.

In summary, for 46 out of 49 families (93.9%), suspicious behaviour was detected. This is under the assumption that we consider bad prefixes for policy $\phi_{16}$ alone as not critical. Note that the results take into account that, according to Zhou and Jiang \cite{zhou2012}, it could have been observed additional malicious behaviours guarded by $\phi_{15}$, $\phi_{18}$ and $\phi_{19}$ (indicated by an \textbullet~ in Table 6.4 and Table 6.5). The reason that this was not the case for $\phi_{15}$ is that the nine marked families targeted Android versions below 2.3.6. Thus, their exploits were not attempted in the first place or were unsuccessful. Bad prefixes for $\phi_{18}$ were missed, as malware was often waiting to receive instructions from a remote server, which was not active anymore. The servers are needed for malware to send an SMS, as they provide premium numbers dynamically. Regarding $\phi_{19}$, the blocking of incoming SMS messages could rarely be observed, as most malware was designed to only suppress the received confirmation from specific premium numbers. Out of 46 detected families, 34 can be associated with potential information stealing, as they use the internet or other sinks after accessing sensitive information. For NickySpy and SMSReplicator has been discovered that the device ID is leaked cleartext via an SMS, and an SMS received was forwarded to a malicious user, respectively.

To discuss limits of the malware detection, which are by no means unique to the approach in this thesis, consider the FakeNetflix family. It uses a phishing attack for which is no observable behaviour in the trace; that is, it shows a fake login screen to the user and then sends the entered credentials to a malicious server.

\textsuperscript{14}UIDs below 10000 are reserved for system apps with higher privileges.
### Table 6.4: Monitor results for malware of the AMGP—Part 1.

<table>
<thead>
<tr>
<th>Malware family</th>
<th>Crash</th>
<th>Hide</th>
<th>Events</th>
<th>Cat. 1 %</th>
<th>Cat. 2 %</th>
<th>Cat. 3 %</th>
<th>Cat. 4 %</th>
<th>Total %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRD</td>
<td>983</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>AnserverBot</td>
<td>2201</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Aeroot</td>
<td>192</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>BaseBridge</td>
<td>2030</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>BeanBot</td>
<td>2248</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CoinPirate</td>
<td>895</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CruiseWin</td>
<td>120</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DogWars</td>
<td>550</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidCoupon</td>
<td>841</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidDeluxe</td>
<td>1167</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidDream</td>
<td>480</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidDreamLight</td>
<td>536</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFu1</td>
<td>892</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFu2</td>
<td>2012</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFu3</td>
<td>370</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFu4</td>
<td>974</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFuSapp</td>
<td>1392</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFuUpdate</td>
<td>506</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DroidKungFuUpdate</td>
<td>1392</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Endofday</td>
<td>731</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>FakeNetflix</td>
<td>564</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>FakePlayer</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>GamblerSMS</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Geinimi</td>
<td>659</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>GGTacker</td>
<td>508</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

### 6.4.2 False positives analysis

Finally, it was checked if policies are suitable to distinguish malware from benign apps. Therefore, MonitorMe ran on a Nexus 5 with Android 5.0.1 that had more than 60 apps from common app categories in the Google Play Store installed: social
Table 6.5: Monitor results for malware of the AMGP—Part 2.

<table>
<thead>
<tr>
<th>Malware family</th>
<th>Crash</th>
<th>Hide</th>
<th>Events</th>
<th>Cat. 1</th>
<th>Cat. 2</th>
<th>Cat. 3</th>
<th>Cat. 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>26. GingerMaster</td>
<td>627</td>
<td>5</td>
<td>124</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>27. GoldDream</td>
<td>1517</td>
<td>1</td>
<td></td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>28. Gone60</td>
<td>555</td>
<td>5</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>29. GPSSMSSpy</td>
<td>✓</td>
<td>23</td>
<td></td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30. HippoSMS</td>
<td>888</td>
<td>2</td>
<td>129</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>31. Jifake</td>
<td>584</td>
<td></td>
<td>329</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32. jSMSHider</td>
<td>1283</td>
<td>3</td>
<td>250</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>33. KMin</td>
<td>850</td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>34. LoveTrap</td>
<td>✓</td>
<td>1791</td>
<td>1</td>
<td>119</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>35. NickyBot</td>
<td>619</td>
<td>3</td>
<td>135</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>36. NickySpy</td>
<td>✓</td>
<td>441</td>
<td>1</td>
<td>135</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>37. Pjapps</td>
<td>1808</td>
<td>1</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>38. Plankton</td>
<td>561</td>
<td>1</td>
<td>119</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>39. RogueLemon</td>
<td>2077</td>
<td>3</td>
<td>135</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>40. RogueSPPush</td>
<td>1653</td>
<td>3</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>41. SMSReplicator</td>
<td>✓</td>
<td>316</td>
<td>1</td>
<td>119</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>42. SndApps</td>
<td>559</td>
<td>2</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>43. Spitzm</td>
<td>✓</td>
<td>771</td>
<td>1</td>
<td>152</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>44. Tapsnake</td>
<td>✓</td>
<td>355</td>
<td>202</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>45. Walkinwat</td>
<td>848</td>
<td>3</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>46. YZHC</td>
<td>631</td>
<td>4</td>
<td>150</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>47. zHash</td>
<td>865</td>
<td>7</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>48. Zitmo</td>
<td>2027</td>
<td>1</td>
<td></td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>49. Zsone</td>
<td>✓</td>
<td>553</td>
<td>344</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Total 2 13 9 8 1

The results are shown in the appendix in Table B.2–B.3. It was discovered suspicious behaviour for 17 out of 61 apps (28%), using the same assumption as above. The false positive rate seems high at first glance.
However, a closer look reveals that some benign apps bear unwanted behaviour for the user, so that a warning by MonitorMe seems reasonable. For example, eleven apps surprisingly requested the device or subscriber ID, which is explicitly not recommended by the Google developer guidelines. Amongst those apps were a soccer news and Yoga app, which in the author’s opinion both do not require this data for its functionality, but rather collect sensitive data from the user. Another app was the private taxi app Uber, which has been criticised in the past due to collecting personal data without the user’s permission. Only five apps started after boot, such as the Dropbox app, which we can consider as necessary regarding its purpose, and only two apps after an SMS was received. These were a secure SMS app and Twitter.

### 6.4.3 Performance and portability

Table 6.6: Execution of Android method calls (each up to 10,000 times) with and without kprobes. The margin of error (MoE) is given for the 95% confidence interval.

<table>
<thead>
<tr>
<th>Interface name / Method name</th>
<th>Android Execution (ms)</th>
<th>Kprobes Execution (ms)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPhoneSubInfo / getDeviceId</td>
<td>5309</td>
<td>5517</td>
<td>3.92</td>
</tr>
<tr>
<td>IPhoneSubInfo / getIccSerialNumber</td>
<td>5346</td>
<td>5524</td>
<td>2.51</td>
</tr>
<tr>
<td>LocationManager / getLastKnownLocation</td>
<td>3516</td>
<td>3562</td>
<td></td>
</tr>
<tr>
<td>ISms / sendText</td>
<td>9166</td>
<td>9396</td>
<td></td>
</tr>
<tr>
<td>IPackageManager / getInstalledApplications</td>
<td>15730</td>
<td>15514</td>
<td></td>
</tr>
<tr>
<td>IConnectivityManager / getAllNetworkInfo</td>
<td>5769</td>
<td>5841</td>
<td></td>
</tr>
<tr>
<td>syscall / sys_open</td>
<td>15360</td>
<td>15531</td>
<td></td>
</tr>
</tbody>
</table>

MoE (ms)

Android

<table>
<thead>
<tr>
<th>Execution (ms)</th>
<th>MoE (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5309</td>
<td>± 15</td>
</tr>
<tr>
<td>5346</td>
<td>± 16</td>
</tr>
<tr>
<td>3516</td>
<td>± 13</td>
</tr>
<tr>
<td>9166</td>
<td>± 13</td>
</tr>
<tr>
<td>15730</td>
<td>± 204</td>
</tr>
<tr>
<td>5769</td>
<td>± 53</td>
</tr>
<tr>
<td>15360</td>
<td>± 72</td>
</tr>
</tbody>
</table>

Kprobes

<table>
<thead>
<tr>
<th>Execution (ms)</th>
<th>MoE (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5517</td>
<td>± 18</td>
</tr>
<tr>
<td>5524</td>
<td>± 16</td>
</tr>
<tr>
<td>3562</td>
<td>± 13</td>
</tr>
<tr>
<td>9396</td>
<td>± 13</td>
</tr>
<tr>
<td>15514</td>
<td>± 202</td>
</tr>
<tr>
<td>5841</td>
<td>± 60</td>
</tr>
<tr>
<td>15531</td>
<td>± 67</td>
</tr>
</tbody>
</table>

DroidTracer

<table>
<thead>
<tr>
<th>Execution (ms)</th>
<th>MoE (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5811</td>
<td>± 11</td>
</tr>
<tr>
<td>5817</td>
<td>± 7</td>
</tr>
<tr>
<td>4126</td>
<td>± 5</td>
</tr>
<tr>
<td>10216</td>
<td>± 10</td>
</tr>
<tr>
<td>15422</td>
<td>± 172</td>
</tr>
<tr>
<td>5671</td>
<td>± 7</td>
</tr>
<tr>
<td>15455</td>
<td>± 38</td>
</tr>
</tbody>
</table>

Overhead (%)

15[http://developer.android.com/training/articles/security-tips.html#UserData](http://developer.android.com/training/articles/security-tips.html#UserData)

DroidTracer and MonitorMe’s (1) performance was evaluated, i.e., the bare system event interception including unmarshalling as well as when running the monitors on top. Moreover, we (2) demonstrate that the proposed automated approach to unmarshalling is portable to different Android devices and versions.

Performance. The performance tests are based on seven specifically built test apps, where each was designed to generate 100 runs of up to 10,000 system events named by the interface and method names in Table 6.6. When MonitorMe is being executed with the policies in §6.3.3 and monitors the test apps individually, the highest performance overhead is 38.6% for the system event `sendText@sms`. This was determined on a Nexus 7 (quad-core CPU, 1 GB RAM) with Android 4.3. Note that the overhead involves the monitor for $q_18$ checking the contact book each time an SMS was sent. However, in practice `sendText@sms` and other system events relevant to our policies rarely occur; and therefore the respective monitors rarely cause overhead. Even though, in practice MonitorMe runs several monitors at the same time, a significant overhead is only noticeable if a system event is intercepted that appears in the monitored formulae and, therefore, requires the respective monitors to change their inner state.

Furthermore, as the results of these test runs are specific only to the implementation of runtime verification, it was also measured the performance overhead of DroidTracer when no further analysis was undertaken. Table 6.6 shows the execution time when intercepting the method calls of the above seven system events in three different modes of operation: (1) without DroidTracer enabled to get a reference execution time for the unmodified system, (2) with only the event interception part of our kernel module enabled, and (3) with unmarshalling and netlink communication added. During the experiments, all four cores of the Nexus 7 ran on a fixed frequency rate. This allowed to reduce the margin of error dramatically. Note that cells are left empty, if overhead could not significantly be determined wrt. the t-test. The results show that the actual performance overhead of using just the kernel module with kprobes is only 2.51–3.92%, whereas the complete performance overhead of DroidTracer is 8.81–17.35%. What is noteworthy is that a method call `getDeviceId` and `getIccSerialNumber` have significant lower overhead than `getLastKnownLocation` and `sendText`, as both former method signatures have no arguments that require unmarshalling. The call of `getLastKnownLocation` has the highest overhead. Most likely, because its arguments contain several complex objects, for example, one of type `LocationRequest`, which unmarshalling involves additional reflection calls. As `sendText` contains only Java primitives as arguments, its unmarshalling overhead is slightly lower.

Portability. DroidTracer was executed on three different devices and Android versions, including 5.0.1. At the time of writing, this is the second most recent one. Table 6.7 demonstrates the success of unmarshalling events that were intercepted. While the interface name of all method calls could be unmarshalled, this was possible only for 45.77%-68.78% of unique method names; that is, it was possible to
6.4 Identifying malware behaviour

Table 6.7: Unmarshable parts of observed system events.

<table>
<thead>
<tr>
<th>Device</th>
<th>Nexus S</th>
<th>Nexus 7</th>
<th>Nexus 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android version</td>
<td>2.3.6</td>
<td>4.3</td>
<td>5.0.1</td>
</tr>
<tr>
<td>Events</td>
<td>102,545</td>
<td>107,977</td>
<td>449,429</td>
</tr>
<tr>
<td>Interfaces (Unique)</td>
<td>58</td>
<td>89</td>
<td>108</td>
</tr>
<tr>
<td>Methods</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unique</td>
<td>804</td>
<td>378</td>
<td>474</td>
</tr>
<tr>
<td>Unmarshalled</td>
<td>368</td>
<td>236</td>
<td>326</td>
</tr>
<tr>
<td>Success rate (%)</td>
<td>45.77</td>
<td>62.43</td>
<td>68.78</td>
</tr>
<tr>
<td>Events with arguments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>54,596</td>
<td>70,746</td>
<td>264,058</td>
</tr>
<tr>
<td>Totally unmarshalled</td>
<td>43,318</td>
<td>67,866</td>
<td>227,708</td>
</tr>
<tr>
<td>Success rate (%)</td>
<td>79.34</td>
<td>95.93</td>
<td>86.23</td>
</tr>
<tr>
<td>Partially unmarshalled</td>
<td>47,923</td>
<td>69,474</td>
<td>255,263</td>
</tr>
<tr>
<td>Success rate (%)</td>
<td>87.78</td>
<td>98.20</td>
<td>96.67</td>
</tr>
</tbody>
</table>

discover for an integer code its according method name in the Android API via reflection. The number of unmarshalled method names seem low, but missing ones are mainly specific to Android internals, for example to render the screen. As such, they have no Stub or Proxy in the Android API, but only in some native C library. This functionality is also not accessible to the developer and therefore usually contain no relevant events for the analysis. If method calls had arguments, all arguments for 79.34%-95.93% and at least some arguments for 87.78%-98.20% could be unmarshalled. Note that if we fail to unmarshal one argument of a call, we also fail to unmarshal all the remaining arguments of that call, as Parcel’s read methods have to be applied in the correct order.

This high unmarshalling rates for a wide range of different Android versions—from an early to a recent one—support the hypothesis that DroidTracer is robust against the Android fragmentation problem. Recall that this is possible, because of mainly two reasons: (1) DroidTracer hooks into the kernel’s system calls, which are unlikely to change, and (2) the unmarshalling algorithm does not rely on low-level Binder implementations, as the high-level Java methods of the class Parcel are used. However, if the data structure binder_transaction.data changed, which transmits data in the kernel between processes, the implementation would have to be adapted. Furthermore, the approach in this thesis would have to be modified if the class Parcel could not be filled with a byte stream directly anymore. Another weak point is the communication between kernel and app. Since Android 5.0, the
system is hardened by SELinux\footnote{https://source.android.com/security/selinux/}, which runs in enforcement mode, so that netlink communication is forbidden. As root, however, SELinux can be switched back to permissive mode. Another point is that with every new Android API version, not only a lot of new methods appear that can be monitored, but also names of existing methods or their signatures are subject to change; especially, if those are not exposed publicly in the Android API. As these kind of changes require the formulae to be adapted, it becomes necessary to introduce and maintain a layer of abstraction, which maps internal methods to a fixed set of predicates.

6.5 Related work

Since the initial release of the Android OS, there has been various research with focus on enhancing the security of this OS or identifying apps that are malware.

One major research direction statically \cite[cf. ]{Enck2011,Arzt2014} and dynamically \cite[cf. ]{Blasing2010,Spreitzenbarth2015} analyses apps before those get installed on a user’s device. For example, Mobile-Sandbox \cite{Spreitzenbarth2015} combines both techniques to detect malicious behaviour. Results from static analysis are used to guide dynamic analysis. This helps to get insights into function calls to native libraries outside the Dalvik VM.

There are works validating permission assignment at the installation time of apps, which refine for that purpose for example Android’s default permission system \cite[cf. ]{Enck2009,Ongtang2012,Nauman2015}. The central components of the Saint framework described in \cite{Ongtang2012} are a modified Android application installer and a so called AppPolicy Provider. The custom installer ensures that at install-time only apps which do not violate policies stored in the AppPolicy Provider can be installed. The authors of Saint have gone to great lengths to check existing apps’ permissions for suspicious permission requests and from that derived practically useful policies for that purpose.

Another static approach by Gunadi et al. \cite{Gunadi2015} provides a type-based method to formally certify non-interference properties of Android bytecode. The goal is to eventually have a compiler tool chain, which developers can use to check if their apps comply with a given policy. Furthermore, they can generate final non-interference certificates as a guarantee for the user’s of apps. FlowDroid \cite{Arzt2014} is a static taint-analysis system that analyses apps’ bytecode and configuration files to identify potential privacy leaks, either caused accidentally by developers or created with malicious intention.

Moreover, there are virtualisation techniques aiming at isolating the data and apps on different trust levels (e.g., to separate environments for private and corporate use) \cite{Bugiel2011,Andrus2011,Lange2011}.

As this thesis is concerned with online monitoring, let us also consider related work that is, like MonitorMe, running completely on the phone and that checks the behaviour of running apps. One corner stone of the approach around MonitorMe is
the ability to specify policies over traces with data. Other runtime verification works that have not been applied to Android, but which also allow parameterised monitoring, have been discussed in §4.5 Some could in principle be ported to monitor policies based on DroidTracer events as discussed in this chapter. However, they need to incorporate the semantics of the “regex”- or “contact”-operator, which are possible in LTL due to its capability of defining arbitrary computational predicates.

An aspect that has not been covered in this thesis, is the type of privilege escalation attack where an app gains access to certain resources or functionalities that are not explicitly granted to it by the user, through indirect control flow. To tackle this problem, Gunadi and Tiu [2014] have developed the framework LogicDroid. It uses a tailored logic for that problem, called recursive metric temporal logic (RMTL). This logic is essentially a past-fragment of metric linear temporal logic (MTL) that is extended with first-order quantifiers and recursive definitions. The latter are used to express call chains between apps. It is currently not known whether the recursive call chains are expressible in LTL. Their monitoring algorithm is based on the rewriting based procedure for past-time LTL introduced by Havelund and Rosu [2002]. It is noteworthy that the form of intervals of metric operators are restricted in such a way that the monitor is proven to be trace-length independent. Another restriction to achieve this is that only safety properties can be monitored. A similar approach to LogicDroid, but based on maintaining call graphs, is XManDroid [Bugiel et al., 2012].

From an instrumentation point of view, most monitoring approaches for Android can be divided into two categories:

**App-centric** ones [cf. Backes et al., 2013; Rasthofer et al., 2014; Xu et al., 2012; Falcone et al., 2012a; Jeon et al., 2012; Davis et al., 2012] intercept events inside the apps by rewriting and repackaging them, so that neither root access nor modifying the Android OS is necessary. As a consequence, they are portable to most phones and Android versions and are easy to install even for non-experts. Examples are AppGuard [Backes et al., 2013] and Aurasium [Xu et al., 2012], which is even able to enforce security policies for apps’ native code as it rewrites Android’s own libc.so, which is natively linked into every app. However, the ease in portability comes at the expense of inherent vulnerabilities, namely that security controls run inside the apps under scrutiny and thus could be bypassed (e.g., by dynamically loading code after rewriting). Also, since apps have to be actively selected for rewriting, hidden malware, such as the ones without launch icon that we came across in §6.4.1 might be overlooked.

**Platform-centric** approaches [cf. Enck et al., 2010; Bugiel et al., 2012; Hornyack et al., 2011; Gunadi and Tiu, 2014; Nauman et al., 2010] usually tie deep into the source code of the Android OS and are therefore less portable. TaintDroid [Enck et al., 2010], a pioneering platform-centric tool for taint flow analysis, requires modifications from the Linux kernel all the way up to the Dalvik VM. Although it is being actively ported, users have to be sufficiently experienced to not only compile their own version of Android, including the TaintDroid changes, but also to make it work on a hardware of their choice. The approach proposed in this thesis is, conceptually, a combination of the advantages of app- and platform-centric monitoring:
DroidTracer can be loaded even into a currently running Android system, yet is able to trace app (even preinstalled Google apps that cannot be rewritten) and Android system interactions all the way down to the OS kernel level.

Recently, there have been in-memory patching approaches, such as PatchDroid [Mulliner et al., 2013] and DeepDroid [Wang et al., 2015], whose advantages are similar to DroidTracer. They also run on a stock device, do not modify apps, but require a rooted phone. Another recent approach that requires neither root nor modifications is Boxify [Backes et al., 2015]. It is essentially a normal app that can be installed on a stock device where it acts as an app virtualisation environment. It proxies all system calls and Binder channels of the apps that it is running in a de-privileged, isolated process.

While policies describing malware behaviour were derived manually in this work, there are works trying to achieve this automatically. Chen et al. [2016a] automatically construct formal, temporal properties from Android malware instances in form of finite state machines. Other works try to classify malware based on machine learning techniques [cf. Gascon et al., 2013; Yerima et al., 2013]. These works can be complemented by the results of this thesis, in the way that the detected bad prefixes of policies can be used as a feature set to the machine learning models.

6.6 Summary

This chapter presents a proof of concept that was carried out on the Android OS, mainly to show that monitoring wrt. traces with data benefits real systems and is efficiently feasible. The first part of this chapter explains the means of the DroidTracer library (used by the MonitorMe app). It details on how it is able to intercept and unmarshal arbitrary Binder events as well as other system calls without changes to the Android OS. The second part discusses experimental results from malware detection using runtime verification. It is based on malware files from the AMGP and derived specifications of common malicious behaviour. A performance and portability study shows that MonitorMe runs efficiently on various Android devices with different versions, i.e., interception and unmarshalling can be sufficiently automated to a large degree.
Chapter 7

Conclusions

This chapter provides an outlook on possible future research directions (§7.1). It further discusses problems that remain open in this thesis, and which might be of interest to being solved for the runtime verification community (§7.2).

7.1 Future research

It has been pointed out that the categorisation of monitoring problems by means of classical computational complexity theory does not adequately reflect the difference in what has to be solved solely once (ahead of monitoring) and what has to be solved for each processed event (during monitoring). The “hard work” should preferably be carried out upfront, in order to minimise recurring overhead. However, the complexity classes in Table 3.1 provide at least a lower bound on the overall complexity of the actual monitoring problem. For this reason, one future research direction is to formally define specific notions of “runtime complexity” classes, which make the differences between runtime overhead and generation explicit. In offline monitoring, where a trace is not expanding, the according decision problems suitably express the worst case of what is required wrt. space or computation time.

For the introduced definitions of an SA, one aspect would be to study the expressiveness of this novel automaton model. Similarly, to the relation of a BA and LTL, an SA is more expressive than LTL\(^{FO}\)-definable languages. However, while a BA is known to agree with the \(\omega\)-regular expressions, it is interesting to investigate which language-defining notions SAs correspond to.

Section §4.5 has discussed the various formalisms for monitoring data. Each logic usually comes with their own monitoring technique. Therefore, it is worth investigating whether monitor constructions can be reused between formalisms. However, formally comparing the expressiveness of different approaches is usually difficult. It is even unknown if past operators add expressiveness compared to having only future operators in first-order temporal logic. [Reger and Rydeheard, 2015] have made a first step towards unifying first-order LTL and notions of parametric trace slicing. They propose a fragment of the former, which can be monitored by QEAs as well (discussed in §4.5).

Regarding the proof of concept carried out in this thesis, to the best of the author’s
knowledge, this work is the first runtime verification approach to comprehensively monitor the vast set of real malwares collected by the AMGP. Detection rates are promising and help substantiate the claim that methods developed in the area of runtime verification are suitable not only for safety-critical systems, but also when (mobile) security is critical. Indeed, at the time of writing, the samples of the AMGP are about three years old, which in the rapid development of new attacks seems like a long time. However, the database has grown over a number of years and the underlying patterns emerged as a result of that. While there are always innovative, hard to detect malwares, it is not unreasonable to expect the bulk of new malwares to also fall into the existing categories, identified by the AMGP, and therefore be detectable by the approach in this thesis. Validation of this hypothesis, however, must be subject to further work, for which more recent malware collections should be used.

Furthermore, the AMGP or other more recent malware collections could serve as a benchmark; that is, these could be used to compare monitors and policies in runtime verification in terms of their effectiveness towards mobile security. An idea is to extend the recently introduced runtime verification challenge in a way to discuss detection and false positive rates in its own category.

7.2 Open problems

We have discussed in §5 examples that show that LTL$^{FO}$ is clearly not trace-length independent, even if we restrict the logic to only uninterpreted predicates. A practically relevant problem is, therefore, to characterise a given formula as being trace-length independent or not. This way one can be warned upfront whether a monitor might steadily grow or whether its memory consumption can be bound by a constant. However, regarding the given notions of trace-length (in)dependence, it is unfortunately currently not known if these are decidable, although it is suspected that they are not. An intuitive argument for this can be given by the theorem of Rice [1953], which intuitively asserts that nontrivial properties of Turing machines are not decidable. For a formal proof, a monitor has to be considered as something as generic as a Turing machine, and an upper bound on its memory (i.e., tape) is a nontrivial property. If the notions of trace-length independence were proven decidable, it is also not clear if a proof could provide us with a meaningful decision procedure that takes into account more than just the syntax of a given formula.

Some insight about trace-length independence can be gained by looking at the proposed monitor construction. It is expected that the monitor is trace-length independent for formulae that contain no interpreted predicates (and, of course, satisfy trace-length independence). An intuition is that every spawned submonitor essentially stores information about its local language in terms of an ordinary GBA, which

\footnote{For example, the mobile malware dump provided by \url{http://contagionminidump.blogspot.de/}.}

\footnote{To evaluate this idea, the persisted events of malware behaviour can be found under \url{http://kuester.multics.org/DroidTracer/malware/traces}.}
is structurally equivalent to the optimal LTL$_3$ monitor by Bauer et al. [2011]. For example, for a quantifier-free formula, the monitor is exactly the optimal monitor for LTL$_3$ (in case of applying the optimisation in §4.3).

However, it seems that there is no interesting fragment that can be trace-length independently monitored. The nesting of a temporal “until”- or “globally”-operator with a quantifier in between leads to a trace-length dependent formula. It is likely that all quantified subformulas have to have bounded prefixes [Kupferman et al., 2006] (which is only the case for the “next”-operator in LTL), so that submonitors can return a verdict in constant time.

Furthermore, it can be easily seen that the monitor is not trace-length independent if arbitrary computational relations are allowed. Consider, for example, the formula $G(\forall x : \text{time}. \ G(\forall y : \text{time}. \ x > y))$, which expresses that all future time stamps must be greater or equal to any time stamp in the past. At first glance this formula looks like it would be trace-length dependent, but due to the transitivity of the order relation, a more succinctly and equivalent formula is $G(\forall x : \text{time}. \ X(\forall y : \text{time}. \ x > y))$. This is clearly trace-length independently monitorable. However, the monitor in this thesis currently does not take the semantics of interpreted predicate symbols into account, but merely the temporal structure of the formula. Hence, it is not, as requested, trace-length independent if arithmetic is involved. It is the question whether one can build a trace-length independent monitor at all in the case where arbitrary predicate symbols are allowed. In this context, it is noteworthy that Du et al. [2015] recently proposed a characterisation of trace-length independence as well as an according monitor for past-time LTL with a counting quantifier.

Last but not least, an interesting and still open decision problem—that has not been researched in this thesis, but is worth mentioning in this context—exists for the propositional setting of LTL: the complexity of monitorability. It is still unknown how tight the bounds really are; that is, the problem is currently only known to be in ExpSpace (can be shown by ways of using the construction of the LTL$_3$ monitor) and PSpace-hard. For LTL$^{FO}$, this thesis shows that monitorability is undecidable.
Bibliography


Hopcroft, J. E. and Ullman, J. D. (1979). Introduction to Automata Theory, Languages and Computation. Addison-Wesley. (cited on pages [20, 39] and [52]).


Leyden, J. (2010). First SMS Trojan for Android is in the wild. Web site, The Register. (cited on page 87)


Appendix A

Detailed proofs

This appendix provides additional proof details for the assertions made in §§3 and §4 of this thesis.

A.1 Complexity results

This section details on the proofs in §3.2.2, which are Lemma 3.2.1 and Lemma 3.2.2 (used to show the undecidability of \(\text{LTL}^\omega\) and the prefix problem, respectively). Furthermore, it contains the complexity proofs of the word problem (see Theorem 3.2.3) and model checking problem (see Theorem 3.2.4).

First of all, let us provide necessary definitions, which are the Quantified Boolean Formula Problem (QBF) for the proof of Theorem 3.2.3, the Post Correspondence Problem (PCP), on which the proof of Lemma 3.2.2 is based, and a deterministic Turing machine (DTM), needed in Lemma A.1.1.

**Definition A.1.1** (Quantified Boolean Formula Problem). A quantified Boolean formula has the form

\[
F = Q_1 x_1 \cdot Q_2 x_2 \cdot \ldots \cdot Q_n x_n \cdot E(x_1, x_2, \ldots, x_n),
\]

where \(Q \in \{\forall, \exists\}\) and \(E\) is a Boolean formula over variables \(x_1, x_2, \ldots, x_n\). The expression \(\forall x. E(x)\) evaluates to \(\top\) if \(E(x)\) evaluates to \(\top\) for both assigning \(1\) (meaning \(\top\)) to all occurrences of \(x\) as well as assigning \(0\) (meaning \(\bot\)) to all occurrences of \(x\). On the other hand, \(\exists x. E(x)\) evaluates to \(\top\) if \(E(x)\) evaluates to \(\top\) for either all \(x\) being \(0\) or \(1\), or both.

Given a formula \(F\) from above with no free variables, the \(\text{QBF}\) then asks: “Does \(F\) evaluates to \(\top\)?”

**Definition A.1.2** (Post Correspondence Problem). An instance of the \(\text{PCP}\) given an alphabet \(\Sigma\), consists of \(k\) corresponding pairs \((x_i, y_i)_{0 \leq i < k}\), where \(x_i, y_i \in \Sigma^*\).

The question is then: “Does this instance of the \(\text{PCP}\) has a solution?”, where we say it has a solution, if there is a sequence of one or more indices \(i_1 \ldots i_m\), when interpreted as indices for the strings of the pairs, \(x_{i_1} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}\) holds.

**Definition A.1.3** (Deterministic Turing machine). A \(\text{DTM}\) is a tuple \(\mathcal{M} = (Q, \Gamma, \Sigma, \delta, q_0, B, F)\), where
• Q is a finite set of states,
• Γ a finite set of tape symbols,
• Σ ⊆ Γ a finite set of input symbols,
• δ : Q × Γ → Q × Γ × {left, right, none} a transition function,
• q₀ ∈ Q is the initial state of the DTM,
• B ∈ Γ \ Σ the blank symbol, and
• F ⊆ Q a set of final states.

Lemma 3.2.1. Let ϕ be a sentence in first-order logic, then we can construct a corresponding ψ ∈ LTL
FO s.t. ϕ has a finite model iff ψ is satisfiable.

Proof. We construct the LTL
FO formula ψ as follows. We first introduce a new unary
U-operator d whose arity is τ and that does not appear in ϕ. We then replace every
subformula in ϕ, which is of the form ∀x. θ, with ∀x : d(θ). Next, we
encode some restrictions on the interpretation of function and predicate symbols by
the following rules:

• For each constant symbol c in ϕ, we conjoin the obtained ψ with d(c).
• For each function symbol f in ϕ of arity n, we conjoin the obtained ψ with
∀x₁ : d...∀xₙ : d(θ).
• For each predicate symbol p in ϕ of arity n, we conjoin the obtained ψ with
∀(x₁...xₙ) : p(θ)
• We conjoin ∃x : d(θ) to the obtained ψ to ensure that the domain is not
empty.

Finally, we fix the arities of symbols in ψ appropriately to one of the following τ,
τ ×...× τ, τ ×...× τ → τ. Obviously, the formula ψ, constructed by the procedure
above, is a syntactically correct LTL
FO formula.

(⇐:) Now, if ψ is satisfiable by some (A', σ), where A' = (|A'|, I') and σ ∈ (A')-Ev, it is easy to construct a finite model A = (|A|, I) s.t. A |= ϕ holds in the
classical sense of first-order logic: set |A| = |A'|, c' = c, f' = f |d' ×...× d', p' = p', respectively. By an inductive argument one can show that the LTL
FO semantics is preserved.

(⇒:) The other direction, if ϕ is finitely satisfiable, is trivial: set |A'| = τ', c' = c, f' = f, respectively, and σ = {(p, e) | e ∈ p} ∪ {(d, e) | e ∈ |A|}.

Lemma A.1.1. Let $\mathcal{M}$ be a deterministic Turing machine (DTM), then we can construct a
ϕ$_{\mathcal{M}}$ ∈ LTL
FO s.t. $\mathcal{M}$ halts on the empty input iff ϕ$_{\mathcal{M}}$ is satisfiable.
Proof. Let us first state some important notes on the type of DTM we use (see Def. A.1.3): It has an infinite tape (to the left and right direction) consisting of infinite many cells, where each cell is identifiable by a unique index $i \in \mathbb{Z}$. Each cell can contain only a single symbol from the set of tape symbols. In the beginning all cells contain the blank symbol (meaning the tape is empty). The input to the machine is written to the tape starting at index 0 (but as we consider only the empty input to the machine, initially the tape is empty). The machine has a head that is always pointing to exactly one cell (in the beginning to cell 0). The “program” of the machine is represented by finitely many states, and transitions between those.

The machine executes in steps: In one step it reads the symbol from the cell that the head is currently pointing to, and according to this symbol and its current state, then writes a new symbol (which might be the same that the cell contains already) to that cell. Furthermore, it moves the head one cell to the left, right, or leaves it pointing to the current cell, and is afterwards in a next, new state (which can be the same, as the program is allowed to have loops).

When the machine reaches a final state, it self-loops in that state forever, does not move its head and writes the same symbol to the tape every time. We say, the machine halts if it reaches a final state.

We know that the following problem for this type of a Turing machine described above is undecidable: “Given an instance $\mathcal{M}$ of the machine, does $\mathcal{M}$ halt?”

We now construct a formula

$$\varphi_{\mathcal{M}} := \psi_{\text{init}} \land G(\psi_{\text{well-formed}} \land \psi_{<} \land \psi_{\text{step}} \land \neg \psi_{\text{final}}),$$

which is based on a machine instance $\mathcal{M}$, and show that $\varphi_{\mathcal{M}}$ has a model iff $\mathcal{M}$ does not halt. Before we discuss the meaning of the different subformulæ of $\varphi_{\mathcal{M}}$, let us introduce the following U-operators tape$_s$, head, state$_q$, $<$, and $=$, of which we make use to describe a run of $\mathcal{M}$: An event of the trace contains ground atom tape$_s(i)$ if for cell $i$ the tape contains the symbol $s$, head$(i)$ if the head is at position $i$, and the proposition state$_q$ represents the current state of $\mathcal{M}$ being $q$. For readability we use infix notations and write $x = y$ and $x < y$, to denote predicates for equality and a strict order relation, respectively. Let us now describe $\psi_{\text{init}}$, $\psi_{\text{well-formed}}$, $\psi_{<}$, $\psi_{\text{step}}$ and $\psi_{\text{final}}$ in more detail:

- The formula

$$\psi_{\text{init}} := \text{head}(0) \land \text{state}_{q_0} \land \bigwedge_{s \in \Gamma \setminus \{B\}} \neg \exists i : \text{tape}_s(i). \text{true}$$

ensures that in the beginning the head is pointing to cell 0, the machine is in its initial state $q_0$ and the tape is empty. Note that events cannot and also do not have to contain infinitely many ground atoms tape$_B(i)$; that is, for every cell $i$. This is implicitly the case if an event contains no other tape symbol for a cell.

- The formula $\psi_{\text{well-formed}}$ ensures that each configuration of the DTM is well-formed; that is, the head in exactly at one position, there is only one symbol
written on each cell of the tape (and different tape symbols cannot be equal), and the \text{DTM} is in exactly one state:

\[
\psi_{\text{well formed}} := (\forall i : \text{head}. \forall j : \text{head}. i = j) \land \\
(\forall i : \text{tape}_s. \forall j : \text{tape}_{s'}. i = j \Rightarrow s = s') \land \\
\left( \bigwedge_{s,s' \in \Gamma, s \neq s'} \neg(s = s') \right) \land \\
\neg \left( \bigvee_{p,q \in Q, p \neq q} (\text{state}_p \land \text{state}_q) \right)
\]

Note that there can be indistinguishable twins representing the current head. However, if an event contains for example \text{head}(0) and \text{head}(4), then \(0 = 4\) appears in the trace, and therefore tape symbols at position 0 and 4 must be the same (i.e., are indistinguishable).

- We ensure a strict order relation of the indices of the tape the following way:

\[
\psi_< := \left( \bigwedge_{s \in \Gamma} \forall x : \text{tape}_s. \neg(x < x) \right) \land \\
\left( \bigwedge_{s,s',s'' \in \Gamma} \forall x : \text{tape}_s. \forall y : \text{tape}_{s'}. \forall z : \text{tape}_{s''}. (x < y \land y < z) \Rightarrow (x < z) \right) \land \\
\left( \bigwedge_{s,s' \in \Gamma} \forall x : \text{tape}_s. \forall y : \text{tape}_{s'}. (x = y) \oplus (x < y) \oplus (y < x) \right)
\]

- We define a successor relation, which is needed for \(\psi_{\text{step}}\), the following way:

\[
\psi_{\text{succ}}(x, y) := x < y \land \bigwedge_{s \in \Gamma} (\forall k : \text{tape}_s. \neg(x < k \land k < y)).
\]

- The formula \(\psi_{\text{step}}\) ensures that every step that the DTM takes is correct; that is,

\[
\psi_{\text{step}} = \bigwedge_{q \in Q, s \in \Gamma, (q', q', d) \in \delta(q, s)} \psi_{\text{step}}(q, s, q', s', d).
\]

Note that this conjunction is finite as \(Q, \Gamma\) and \(\delta\) are finite. The \(\psi_{\text{step}}(q, s, q', s', d)\)s, where \(d \in \{\text{none, left, right}\}\) are defined as follows:

\[
\psi_{\text{step}}(q, s, q', s', \text{none}) := \text{state}_q \land \exists i : \text{tape}_s. \text{head}(i) \Rightarrow \\
\text{X}((\text{state}_{q'} \land \text{head}(i) \land \text{tape}_{s'}(i)))
\]

\[
\psi_{\text{step}}(q, s, q', s', \text{right}) := \text{state}_q \land \exists i : \text{tape}_s. \text{head}(i) \Rightarrow \\
\text{X}((\text{state}_{q'} \land \exists j : \text{tape}_{s'}. (\text{head}(j) \land \psi_{\text{succ}}(i, j))))
\]
§A.1 Complexity results

$\psi_{\text{step}}(q,s,q',s',\text{left}) := \text{state}(q) \land \exists i : \text{tape}_s. \text{head}(i) \Rightarrow \text{X} (\text{state}_{q'} \land \exists j : \text{tape}_{s'}. (\text{head}(j) \land \psi_{\text{succ}}(j,i)))$

- We conjoin $\psi_{\text{step}}$ with the formula

$$\bigwedge_{s \in \Gamma} \forall i : \text{tape}_s. \neg \text{head}(i) \Rightarrow \text{X} \text{tape}_s(i),$$

which expresses the fact that the tape can only change at the position where the head is currently pointing to.

- The formula $\psi_{\text{final}}$ describes the machine entering a final state, i.e.,

$$\psi_{\text{final}} := \bigvee_{q \in F} \text{state}_q$$

An instance $\mathcal{M}$ of a DTM does not halt if $\varphi_{\mathcal{M}}$ is satisfiable; that is, we have constructed $\varphi_{\mathcal{M}}$ in such a way that its models are infinite runs of a Turing machine that never reach a final state. Since $\varphi_{\mathcal{M}}$ can be constructed effectively for every formal description of an instance $\mathcal{M}$, the undecidability of the halting problem for Turing machines with empty input implies the undecidability of the satisfiability problem of LTL$^{\text{FO}}$.

**Theorem 3.2.3** The word problem for LTL$^{\text{FO}}$ is PSpace-complete.

*Proof.* To evaluate a formula $\varphi \in \text{LTL}^{\text{FO}}$ over some linear Kripke structure, $\mathcal{K}_{\text{FO}}$, we can basically use the inductive definition of the semantics of LTL$^{\text{FO}}$: If used as a function, starting in the initial state of $\mathcal{K}_{\text{FO}}$, $s_0$, it evaluates $\varphi$ in a depth-first manner with the maximal depth bounded by $|\varphi|$ (see a similar proof by [Bauer et al., 2009a, Theorem 4]).

To show hardness, we reduce the QBF (see Def. A.1.1), which is known to be PSpace-complete. The reduction of this problem proceeds as follows. We first construct a formula $\varphi \in \text{LTL}^{\text{FO}}$ in prenex normal form,

$$\varphi = Q_1 x_1 : d. Q_2 x_2 : d. \ldots Q_n x_n : d. E(p_{x_1}(x_1), p_{x_2}(x_2), \ldots, p_{x_n}(x_n)).$$

Then, using an $U$-operator $p_{x_i}$ for every variable $x_i$, we construct a singleton Kripke structure, $\mathcal{K}_{\text{FO}}$, s.t. $\lambda(s_0) = (\mathfrak{A}, \{(d,0), (d,1), (p_{x_1}, 1), (p_{x_2}, 1), \ldots, (p_{x_n}, 1)\})$, where $|\mathfrak{A}| = \{0, 1\}$ and $I$ defined accordingly. It can easily be seen that $F$ evaluates to $\top$ iff $\mathcal{K}_{\text{FO}}$ is a model for $\varphi$. Moreover, this construction can be obtained in no more than a polynomial number of steps wrt. the size of the input. \hfill $\Box$

**Theorem 3.2.4** The model checking problem for LTL$^{\text{FO}}$ is in ExpSpace.

*Proof.* For a given $\varphi \in \text{LTL}^{\text{FO}}$ and $(\mathfrak{A})$-Kripke structure $\mathcal{K}_{\text{FO}}$ defined as usual, where $\mathfrak{A} = (|\mathfrak{A}|, I)$, we construct a propositional Kripke structure $\mathcal{K}'$ and $\varphi' \in \text{LTL}$, s.t.
\( \mathcal{L}(\mathcal{K}_{FO}) \subseteq \mathcal{L}(\varphi) \) iff \( \mathcal{L}(\mathcal{K}') \subseteq \mathcal{L}(\varphi') \) holds. Assuming variable names in \( \varphi \) have been adjusted so that each has a unique name, the construction of \( \varphi' \) proceeds as follows.

Wlog. we can assume \(|\mathfrak{A}| \) to be a finite set \( \{d_0, \ldots, d_n\} \). We first set \( \varphi' \) to \( \varphi \) and extend the corresponding \( \Gamma \) by the constant symbols \( c_{d_0}, \ldots, c_{d_n} \), s.t. \( c_{d_i} = d_i \), respectively; that is, we add the respective interpretations of each \( c_i \) to \( I \). This step obviously does not require more than polynomial space. We then replace all subformulae in \( \varphi' \) of the form \( \nu = Q x : p. \psi(x) \) exhaustively with the following constructed \( \psi' \):

- Set \( \psi' = \top \).
- For each state \( s \in S \) do the following:
  - Let \( T = \{ d \mid \lambda(s) = (\mathfrak{A}', \sigma'), \mathfrak{A} \sim \mathfrak{A} \text{ and } (p, d) \in \sigma \} \).
  - If \( Q = \lor \), then
    \[
    \psi' = \psi' \land (\bar{s} \Rightarrow \bigwedge_{d \in T} \psi(x)[c / x]), \text{ where } c \text{ is s.t. } c^I = d,
    \]
  - otherwise
    \[
    \psi' = \psi' \land (\bar{s} \Rightarrow \bigvee_{d \in T} \psi(x)[c / x]), \text{ where } c \text{ is s.t. } c^I = d,
    \]
where \( \bar{s} \) is a fresh, unique predicate symbol meant to represent state \( s \).

Then, for all subformulae in \( \varphi' \) of the form \( \nu = Q x : p. \psi(x) \) do the following:

- For each \( r(t) \) occurring in \( \psi \), where \( r \in R \) and \( t \) are terms, let \( d = t^I \), and replace \( r(t) \) by a fresh, unique predicate symbol \( r_d \).

It is easy to see that, indeed, \( \varphi' \) is a syntactically correct standard LTL formula, where all quantifiers have been eliminated. In terms of space complexity, note that in the first loop, we replace each quantified formula by an expression at least \(|\mathcal{K}| \) times longer than the original quantified formula. In the worst case, the final formula’s length will be exponential in the number of quantifiers.

We now define the propositional Kripke structure \( \mathcal{K}' = (S', s_0', \lambda', \rightarrow') \) as follows.

Let \( S' = S \), \( s_0' = s_0 \), and \( \rightarrow' = \rightarrow \). In what follows, let \( s \) be a state and \( \lambda(s) = (|\mathfrak{A}|, I, \sigma) \). (Note, this is the labelling function of \( \mathcal{K} \).) The alphabet of \( \mathcal{K}' \) is given by \( 2^{AP} \), where \( AP = \{ r_d \mid r \in R \text{ and } d \in |\mathfrak{A}| \} \cup \{ \bar{s} \mid s \in S \} \). Finally, we define the labelling function of \( \mathcal{K}' \) as \( \lambda'(s) = \{ \bar{s} \} \cup \{ r_d \mid r \in R \text{ and } r^d(d) \text{ is true} \} \). It is easy to see that, indeed, \( \mathcal{K}' \) preserves all the runs possible through \( \mathcal{K}_{FO} \).

One can show by an easy induction on the structure of \( \varphi' \) that, indeed, \( \mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi) \) iff \( \mathcal{L}(\mathcal{K}') \subseteq \mathcal{L}(\varphi') \) holds.

**Lemma 3.2.2** Let \( \mathfrak{A} \) be a first-order structure and \( \varphi \in \text{LTL}^{FO} \), then \( \mathcal{L}(\varphi)_\mathfrak{A} = \{ (\bar{\mathfrak{A}}, w) \mid \mathfrak{A} \sim \mathfrak{A}, w \in \text{Ev}^\omega, \text{ and } (\bar{\mathfrak{A}}, w) \models \varphi \} \). Testing if \( \mathcal{L}(\varphi)_\mathfrak{A} \neq \emptyset \) is generally undecidable.
\[\textbf{Theorem 3.2.5} \quad \text{The prefix problem for LTL}^0 \text{ is undecidable.}\]

\[\text{Proof.} \quad (1): \text{By way of a similar reduction used in Theorem 3.2.1 already, i.e., for any } \varphi, \mathfrak{A}, \text{ and } \sigma \in \text{Ev we have that } (\mathfrak{A}, \sigma) \in \text{bad}(\text{X} \varphi) \text{ iff } L(\varphi)_\mathfrak{A} = \emptyset. \text{ The } \Leftarrow \text{-direction is obvious. For the other direction:}
\]

\[\begin{align*}
(\mathfrak{A}, \sigma) &\in \text{bad}(\text{X} \varphi) \\
\Rightarrow & \text{ for all } \mathfrak{A} \sim \mathfrak{A} \text{ and } w \in \text{Ev}^w, \text{ we have that } (\mathfrak{A} \mathfrak{A}, \sigma w) \not\models \text{X} \varphi \\
\Rightarrow & \text{ for all } \mathfrak{A} \sim \mathfrak{A} \text{ and } w \in \text{Ev}^w, \text{ we have that } (\mathfrak{A}, w) \not\models \varphi \\
\Rightarrow & L(\varphi)_\mathfrak{A} = \emptyset \text{ (which is generally undecidable by Lemma 3.2.2).}
\end{align*}\]

\[\square\]

\section*{A.2 Spawning automaton}

This section contains the proof of Theorem 4.1.1 showing the correctness of the translation of an LTL$^0$ formula to an SA, as well as the Lemma 4.1.1 on which the theorem is based.

\[\textbf{Lemma 4.1.1} \quad \text{Let } \varphi \in \text{LTL}^0 \text{ (not necessarily a sentence) and } v \text{ be a valuation. For each accepting run } \rho \text{ in } A_{\varphi,v} \text{ over input } (\mathfrak{A}, w), \varphi \in \text{cl}(\rho), \text{ and } i \geq 0, \text{ we have that } \psi \in \rho(i) \text{ iff } (\mathfrak{A}, w, v, i) \models \psi.
\]

\[\text{Proof.} \quad \text{We proceed by a nested induction on depth}(\varphi) \text{ and the structure of } \psi \in \text{cl}(\varphi). \text{ For the base case let } \text{depth}(\varphi) = 0: \text{ We fix } \rho \text{ to be an accepting run in } A_{\varphi,v} \text{ over } (\mathfrak{A}, w), \text{ and proceed by induction over those formulae } \psi \in \text{cl}(\varphi) \text{ which are of depth zero (i.e., without quantifiers) since } \text{depth}(\varphi) = 0. \text{ Therefore, this case basically resembles the correctness argument of BA}_k \text{ for propositional LTL (cf. [Baier and Katoen] 2008 §5). For an arbitrary } i \geq 0, \text{ we have}
\]
\[ \psi = r(t): \]
\[ r(t) \in \rho(i) \iff t^i \in r^i (\text{by the definition of } \delta-) \]
where, as before, for any variable \( x \) in \( t \), by \( x^i \) we mean \( v(x) \)
\[ \iff (\vec{A}, w, v, i) \models r(t) \text{ (by the semantics of LTL)} \]

\[ \psi = p(t): \text{ analogous to the above.} \]

\[ \psi = \neg \psi': \]
\[ \neg \psi' \in \rho(i) \iff \psi' \not\in \rho(i) \text{ (by the completeness assumption of all } q \in Q) \]
\[ \iff (\vec{A}, w, v, i) \not\models \psi' \text{ (by induction hypothesis)} \]
\[ \iff (\vec{A}, w, v, i) \models \neg \psi' \text{ (by the semantics of LTL)} \]

\[ \psi = \psi_1 \land \psi_2: \]
\[ \psi_1 \land \psi_2 \in \rho(i) \iff \{ \psi_1, \psi_2 \} \subseteq \rho(i) \text{ (by the completeness assumption of all } q \in Q) \]
\[ \iff (\vec{A}, w, v, i) \models \psi_1 \land \psi_2 \text{ (by induction hypothesis)} \]
\[ \iff (\vec{A}, w, v, i) \models \psi_1 \land \psi_2 \text{ (by the semantics of LTL)} \]

\[ \psi = X\psi': \]
\[ X\psi' \in \rho(i) \iff \psi' \in \rho(i+1) \text{ (by the definition of } \delta-) \]
\[ \iff (\vec{A}, w, v, i+1) \models \psi' \text{ (by induction hypothesis)} \]
\[ \iff (\vec{A}, w, v, i) \models X\psi' \text{ (by the semantics of LTL)} \]

\[ \psi = \psi_1 \psi_2: \text{ we first show the } \Rightarrow \text{-direction. For this, let us first show that there is a } j \geq i \text{, such that } (\vec{A}, w, v, j) \models \psi_2 \text{ holds. For suppose not, then for all } j \geq i, \text{ we have that } (\vec{A}, w, v, j) \not\models \psi_2 \text{ and, consequently, by induction hypothesis } \psi_2 \not\in \rho(j) \text{. By definition of } \delta-, \text{ since } \psi_1 \psi_2 \in \rho(i) \text{ and there is not a } j \text{ s.t. } \psi_2 \in \rho(i), \text{ we have that } \psi_1 \psi_2 \in \rho(i) \text{ for all } j \geq 0. \text{ On the other hand, } \rho \text{ is accepting in } A_{\psi_1}, \text{ thus there exist infinitely many } j \geq i, \text{ s.t. } \psi_1 \psi_2 \not\in \rho(j) \text{ or } \psi_2 \not\in \rho(i) \text{ by the definition of the generalised Büchi acceptance condition } F, \text{ which is a contradiction. Let us, in what follows, fix the smallest such } j. \text{ We still need to show that for all } i \leq k \leq j, (\vec{A}, w, v, k) \models \psi_1 \text{ holds. As } j \text{ is the smallest such } j, \text{ where } \psi_2 \not\in \rho(j) \text{ it follows that } \psi_2 \not\in \rho(k) \text{ for any such } k. \text{ As } \psi_1 \psi_2 \in \rho(i), \text{ it follows by definition of } \delta- \text{ that } \psi_1 \in \rho(i) \text{ and } \psi_1 \psi_2 \in \rho(i+1). \text{ We can then inductively apply this argument to all } i \leq k < j, \text{ such that } \psi_1 \in \rho(k) \text{ and } \psi_1 \psi_2 \in \rho(k+1) \text{ hold. The statement then follows from the induction hypothesis.} \]

Let us now focus on the } \Leftarrow \text{-direction, i.e., suppose } (\vec{A}, w, v, i) \models \psi_1 \psi_2 \text{ implies that } \psi_1 \psi_2 \in \rho(i). \text{ By assumption, there is a } j \geq i, \text{ such that } (\vec{A}, w, v, j) \models \psi_2 \text{ and for all } i \leq k < j, \text{ we have that } (\vec{A}, w, v, k) \models \psi_1 \text{. Therefore, by induction hypothesis, } \psi_2 \in \rho(j) \text{ and } \psi_1 \in \rho(k) \text{ for all such } k. \text{ Then, by the completeness assumption of all } q \in Q, \text{ we also get } \psi_1 \psi_2 \in p_j, \text{ and if } j = i, \text{ we are done.}
Otherwise with an inductive argument similar to the previous case on \( k = j - 1, k = j - 2, \ldots, k = i \), we can infer that \( \psi_1 \cup \psi_2 \in \rho(k) \).

Let \( \text{depth}(\varphi) = n > 0 \), i.e., we suppose that our claim holds for all formulae with quantifier depth less than \( n \). We continue our proof by structural induction, where the quantifier free cases are almost exactly as above. Therefore, we focus only on the following case.

- \( \psi = \forall x : p. \psi' \): for this case, as before with the \( \textbf{U} \)-operator, we will first show the \( \Rightarrow \)-direction, i.e., for all \( i \geq 0 \) we have \( \forall x : p. \psi' \in \rho(i) \) implies \( \forall x : p. \psi' \). By the semantics of \( \text{LTL}^{\infty} \), the latter is equivalent to for all \( (p, d) \in w_i \), \( (\mathfrak{A}, w, v \cup \{ x \rightarrow d \}, i) \models \psi' \). If there is no \( (p, d) \in w_i \) the statement is vacuously true. Otherwise, there are some actions \( (p, d) \in w_i \) and
\[
\delta_{\downarrow}(\rho(i), (\mathfrak{A}, w_i)) = B \land \bigwedge_{(p, d) \in w_i} A_{\psi', v \cup \{ x \rightarrow d \}},
\]

where \( B \) is a Boolean combination of SAs corresponding to the remaining elements in \( \rho(i) \). As \( \rho \) is accepting in \( A_{\psi', v} \), there exists a \( Y_i \) satisfying \( \delta_{\downarrow}(\rho(i), (\mathfrak{A}, w_i)) \), s.t. all \( A \in Y_i \) have an accepting run on input \( (\mathfrak{A}, w') \). It follows that \( Y_i \) contains an automaton \( A_{\psi', v \cup \{ x \rightarrow d \}} \) for each action \( (p, d) \in w_i \) that has an accepting run \( \rho' \). As the respective levels of these automata is \( n - 1 \), we can use the induction hypothesis and note that the following holds true for each of the \( A_{\psi', v \cup \{ x \rightarrow d \}} \in Y_i \):

\[
\text{for all: } v \in \text{cl}(\psi') \text{ and } l \geq 0, v \in \rho'(l) \text{ iff } (\mathfrak{A}, w, v \cup \{ x \rightarrow d \}, i + l) \models v,
\]

We can now set \( v = \psi' \), respectively, and \( l = 0 \), from which it follows that \( \psi' \in \rho'(0) \) iff \( (\mathfrak{A}, w, v \cup \{ x \rightarrow d \}, i) \models \psi' \), respectively. As by construction of an SA the initial states of runs contain the formula which the SA represents, we have \( \psi' \in \rho'(0) \) and hence \( (\mathfrak{A}, w, v \cup \{ x \rightarrow d \}, i) \models \psi' \), respectively. As this holds for all \( A_{\psi', v \cup \{ x \rightarrow d \}} \), where \( (p, d) \in w_i \), it follows by semantics of \( \text{LTL}^{\infty} \) that \( (\mathfrak{A}, w, v, i) \models \forall x : p. \psi' \).

Let us now consider the \( \Leftarrow \)-direction, i.e., \( (\mathfrak{A}, w, v, i) \models \forall x : p. \psi' \) implies \( \forall x : p. \psi' \in \rho(i) \), which we show by contradiction. Suppose \( \forall x : p. \psi' \notin \rho(i) \), which implies by the completeness assumption of all \( q \in Q \) that \( \neg \forall x : p. \psi' \in \rho(i) \) holds. If there is no \( (p, d) \in w_i \), then \( \delta_{\downarrow}(\rho(i), (\mathfrak{A}, w_i)) \) is equivalent to \( \bot \) and \( \rho \) could not be accepting. Therefore there must be some \( (p, d) \in w_i \), s.t.
\[
\delta_{\downarrow}(\rho(i), (\mathfrak{A}, w_i)) = B \land \bigvee_{(p, d) \in w_i} A_{\neg \psi', v \cup \{ x \rightarrow d \}},
\]

where \( B \) is a Boolean combination of SAs corresponding to the remaining elements in \( \rho(i) \). Because \( \rho \) is accepting in \( A_{\psi, v} \), there exists a \( Y_i \), such that \( Y_i \models \delta_{\downarrow}(\rho(i), (\mathfrak{A}, w_i)) \), and there is at least one SA, \( A' = A_{\neg \psi', v \cup \{ x \rightarrow d \}} \in Y_i \), with corresponding \( (p, d) \in w_i \), s.t. \( (\mathfrak{A}, w') \) is accepted by \( A' \) as input; that is,
\( \mathcal{A}' \) has an accepting run, \( \rho' \), on said input. As this automaton’s level is \( n - 1 \), we can apply the induction hypothesis and obtain

for all: \( v \in \text{cl}(\neg \psi') \) and \( l \geq 0, v \in \rho'(l) \) iff \( (\mathfrak{A}, w, v \cup \{ x \mapsto d \}, i + l) \models v \).

We can now set \( \nu = \neg \psi' \) and \( l = 0 \), and since \( v \) belongs to the initial states in accepting runs, we derive \( (\mathfrak{A}, w, v \cup \{ x \mapsto d \}, i) \models \neg \psi' \), which is a contradiction to our initial hypothesis.

\[\square\]

**Theorem 4.1.1** The constructed SA is correct in the sense that for any sentence \( \varphi \in \text{LTL}^\text{FO} \), we have that \( \mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi) \).

**Proof.** ⊆: Follows from Lemma 4.1.1 let \( \rho \) be an accepting run over \( (\mathfrak{A}, w) \) in \( \mathcal{A}_\varphi \). By definition of an (accepting) run, \( \varphi \in \rho(0) \), and therefore \( (\mathfrak{A}, w) \in \mathcal{L}(\varphi) \).

⊇: We show the more general statement: Given a (possibly not closed) formula \( \varphi \in \text{LTL}^\text{FO} \) and valuation \( \nu \). It holds that \( \{ (\mathfrak{A}, w) \mid (\mathfrak{A}, w, \nu, 0) \models \varphi \} \subseteq \mathcal{L}(\mathcal{A}_{\varphi, \nu}) \). We define for all \( i \geq 0 \) the set \( \rho(i) = \{ \psi \in \text{cl}(\varphi) \mid (\mathfrak{A}, w, \nu, i) \models \psi \} \) for some arbitrary but fixed formula \( \varphi \in \text{LTL}^\text{FO} \) and valuation \( \nu \), and arbitrary but fixed \( (\mathfrak{A}, w) \), where \( (\mathfrak{A}, w, \nu, 0) \models \varphi \). Let us now show that \( \rho = \rho(0) \rho(1) \ldots \) is a well-defined run in \( \mathcal{A}_{\varphi, \nu} \) over \( (\mathfrak{A}, w) \): Firstly, from the construction of \( Q \), it follows that for all \( i, \rho(i) \in Q \). Secondly, since \( \varphi \in \text{cl}(\varphi) \) and \( (\mathfrak{A}, w, \nu, 0) \models \varphi \), \( \rho(0) \) always contains \( \varphi \). Thirdly, \( \rho(i + 1) \in \delta_\to(\rho(i), (\mathfrak{A}, w)) \) holds for all \( i \). The latter is the case iff

- for all \( X \varphi \in \text{cl}(\varphi) \): \( X \varphi \in \rho(i) \) iff \( \varphi \in \rho(i + 1) \), and
- for all \( \psi_1 \psi_2 \in \text{cl}(\varphi) \): \( \psi_1 \psi_2 \in \rho(i) \) iff \( \psi_2 \in \rho(i) \) or \( (\psi_1 \in \rho(1) \) and \( \psi_1 \psi_2 \in \rho(i + 1) \).

The first condition can be shown as follows:

\[
X \varphi \in \rho(i) \iff (\mathfrak{A}, w, \nu, i) \models X \varphi \quad \text{(by definition of } \rho(i) \text{)}
\]
\[
\iff (\mathfrak{A}, w, \nu, i + 1) \models \psi \quad \text{(by the semantics of } \text{LTL}^\text{FO} \text{)}
\]
\[
\iff \psi \in \rho(i + 1) \quad \text{(by the definition of } \rho(i + 1) \text{)}.
\]

The second can be shown as follows:

\[
\psi_1 \psi_2 \in \rho(i) \iff (\mathfrak{A}, w, \nu, i) \models \psi_1 \psi_2 \quad \text{(by definition of } \rho(i) \text{)}
\]
\[
\iff (\mathfrak{A}, w, \nu, i) \models \psi_2 \lor (\psi_1 \land X(\psi_1 \psi_2))
\]
\[
\iff (\mathfrak{A}, w, \nu, i) \models \psi_2 \lor (\mathfrak{A}, w, \nu, i + 1) \models \psi_1 \psi_2
\]
\[
\iff \psi_2 \in \rho(i) \text{ or } (\psi_1 \in \rho(1) \text{ and } \psi_1 \psi_2 \in \rho(i + 1)) \quad \text{(by definition of } \rho \text{)}.
\]

It remains to show that \( \rho \) is also accepting in \( \mathcal{A}_{\varphi, \nu} \). We proceed by induction on \( \text{depth}(\varphi) \). In what follows, let \( \text{depth}(\varphi) = 0 \), i.e., we are showing local acceptance only. By the definition of acceptance we must have that for all \( \psi_1 \psi_2 \in \text{cl}(\varphi) \), there exist infinitely many \( i \geq 0 \), s.t. \( \rho(i) \in F_{\psi_1 \psi_2} \), where \( F_{\psi_1 \psi_2} \in \mathcal{F} \). For suppose not, i.e., there are only finitely many such \( i \), then there is a \( k \geq 0 \), s.t. for all \( j \geq k \) we have
Theorem 4.2.1. Constructed monitor is impartial.

This section contains the detailed proof of Theorem 4.2.1, which shows that the constructed monitor is impartial.

Let us now assume the statement holds for all formulae with depth strictly less than \( n \) and assume depth(\( \varphi \)) = \( n \), where \( n > 0 \). We don’t show local acceptance of \( \rho \) as it is virtually the same as in the base case, and instead go on to show that for all \( i \geq 0 \), there is a \( Y_i \) s.t. \( Y_i \models \delta_i(\rho(i), (\mathfrak{A}_i, w_i)) \) and all \( A \in Y_i \) are accepting \((\mathfrak{A}, w')\).

Let us define the following two sets:

\[
Y_i^\uparrow = \{ A_{\varphi, \nu \cup \{x \mapsto d\}} \mid \forall x : p. \ \psi \in \rho(i) \text{ and } (p, d) \in w_i \}
\]
and

\[
Y_i^\downarrow = \{ A_{\neg \varphi, \nu \cup \{x \mapsto d\}} \mid \neg \forall x : p. \ \psi \in \rho(i) \text{ and } (p, d) \in w_i \}
\]

Set \( Y_i = Y_i^\uparrow \cup Y_i^\downarrow \), which by construction satisfies \( \delta_i(\rho(i), (\mathfrak{A}_i, w_i)) \). We still need to show that every automaton in this set accepts \((\mathfrak{A}, w')\). Now for \( A_{\varphi, \nu \cup \{x \mapsto d\}} \in Y_i \) we have either \( v = \psi \) for some \( \forall x : p. \ \psi \in \rho(i) \) and \((p, d) \in w_i \), or \( v = \neg \psi \) for some \( \neg \forall x : p. \ \psi \in \rho(i) \) and \((p, d) \in w_i \). In either case by definition of \( \rho(i) \) and semantics of LTL\(^\Box\), it follows that \((\mathfrak{A}_i, w_i, \nu \cup \{x \mapsto d\}, i) \models v \). Since the level of \( A_{\varphi, \nu \cup \{x \mapsto d\}} \) is strictly less than \( n \), we can apply the induction hypothesis and construct an accepting run for \((\mathfrak{A}, w')\), where \((\mathfrak{A}_i, w_i, \nu \cup \{x \mapsto d\}, i) \models v \) in \( A_{\varphi, \nu \cup \{x \mapsto d\}} \). The statement follows. \( \square \)

### A.3 Monitor construction

This section contains the detailed proof of Theorem 4.2.1 which shows that the constructed monitor is impartial.

**Theorem 4.2.1** \( M_\varphi(\mathfrak{A}, u) = \top \Rightarrow (\mathfrak{A}, u) \in \text{good}(\varphi) \) (resp. for \( \bot \) and \( \text{bad}(\varphi) \)).

**Proof.** We prove the more general statement \( M_\varphi(\mathfrak{A}, u) = \top \Rightarrow (\mathfrak{A}, u) \in \text{good}(\varphi, v) \), where \( \varphi \) possibly has some free variables and \( v \) is a valuation, by a nested induction over depth(\( \varphi \)).

- For the base case let depth(\( \varphi \)) = \( 0 \), where \( \varphi \) possibly has free variables, \((\mathfrak{A}, u)\) be an arbitrary but fixed prefix and \( v \) a valuation. Suppose \( M_{\varphi, \nu}(\mathfrak{A}, u) \) returns \( \top \) after processing \((\mathfrak{A}, u)\), but \((\mathfrak{A}, u) \notin \text{good}(\varphi, v) \). By M3 and T10 the buffer of \( T_{\neg \varphi, \nu} \) is empty, i.e., \( B_{\neg \varphi, \nu} = \emptyset \). By T13 and because \( A_{\neg \varphi, \nu} \) has an accepting run \( \rho \) over \((\mathfrak{A}, u)\) with some suffix, \( B_{\neg \varphi, \nu} \) contains \((\rho(|u|), [\top]) \) after processing \((\mathfrak{A}, u)\). Furthermore, because \( \delta_i \) yields \( \top \) for any input iff depth(\( \neg \varphi \)) = \( 0 \), no run in the buffer is ever removed in T17. Contradiction.

- Let depth(\( \varphi \)) > \( 0 \), \((\mathfrak{A}, u)\) be an arbitrary but fixed prefix and \( v \) a valuation. Under the same assumptions as above, we will reach a contradiction showing
that after processing $(\overline{\mathcal{A}}_u)$, there is a sequence of obligations $(\rho(|u|),\{obl_0,\ldots,obl_n\})$ in buffer $B_{\overline{\mathcal{A}},\rho}$, which corresponds to an accepting run $\rho$ in $\mathcal{A}_{\overline{\mathcal{A}},\rho}$ over $(\overline{\mathcal{A}}_u)$ with some suffix $(\overline{\mathcal{A}}',w')$. That is, $M_{\overline{\mathcal{A}},\rho}$ cannot return $\top$, after $B_{\overline{\mathcal{A}},\rho}$ is empty, and $B_{\overline{\mathcal{A}},\rho}$ containing the above mentioned sequence at the same time. By T3, $B_{\overline{\mathcal{A}},\rho}$ contains a sequence $(\rho(|u|),\{obl_0,\ldots,obl_n\})$ that was incrementally created processing $(\overline{\mathcal{A}}_u)$ wrt. $\delta_{\rightarrow}$, eventually with some obligations removed if they were detected to be met by the input. We now show that this sequence is never removed from the buffer in T7. Suppose the run has been removed, then there was an $obl_j = \delta_{\leftarrow}(\rho(j),(\overline{\mathcal{A}}_u,u_j))$, that is

$$
\left(\bigwedge_{\forall x:p.\psi \in \rho(j)} \left(\bigwedge_{(p,d) \in u_j} A_{\psi,\rho'}\right)\right) \land \left(\bigwedge_{\neg \forall x:p.\psi \in \rho(j)} \left(\bigvee_{(p,d) \in u_j} A_{\neg \psi,\rho''}\right)\right),
$$

with $\rho' = v \cup \{x \mapsto d\}$ and $\rho'' = v \cup \{x \mapsto d\}$, evaluated to $\bot$ after $l$ steps, with $0 \leq j \leq l < |u|$. That is, at least one submonitor corresponding to an automaton in the second conjunction has returned $\bot$ (or all submonitors corresponding to automata in a disjunction, for which the following argument would be similar).

Wlog. let $\forall x:p.\psi \in \rho(j)$, $(p,d) \in u_j$, and $M_{\psi,\rho'}(\overline{\mathcal{A}}_j,\ldots,\overline{\mathcal{A}}_i,u_j,\ldots,u_l) = \bot$, i.e., $M_{\psi,\rho'}$ is the submonitor corresponding to $A_{\psi,\rho'}$. As $\text{level}(\psi) < \text{level}(\rho)$, from the induction hypothesis follows that $(\overline{\mathcal{A}}_j,\ldots,\overline{\mathcal{A}}_i,u_j,\ldots,u_l) \in \text{bad}(\psi,\rho')$, i.e., $(\overline{\mathcal{A}}_j,\ldots,\overline{\mathcal{A}}_i,\overline{\mathcal{A}}_j',u_j,\ldots,u_l\rho'') \not\models \psi$ with evaluation $\rho'$ for any $(\overline{\mathcal{A}}''',w''')$, and therefore $(\overline{\mathcal{A}}_j,\ldots,\overline{\mathcal{A}}_i,\overline{\mathcal{A}}_j',u_j,\ldots,u_l\rho'') \models \neg \forall x:p.\psi$ under valuation $v$. But as $\rho$ over $(\overline{\mathcal{A}}_i',\rho''')$ is an accepting run in $\mathcal{A}_{\overline{\mathcal{A}},\rho}$ and $\forall x:p.\psi \in \rho(j)$, it follows that $(\overline{\mathcal{A}}_i',\rho''') \models \forall x:p.\psi$. Now, we choose $(\overline{\mathcal{A}}''',w''')$ to be $(\overline{\mathcal{A}}_{l+1},\ldots,\overline{\mathcal{A}}_i|\overline{\mathcal{A}}_j',u_{l+1},\ldots,u_{l+1}|\rho''')$. Contradiction.

As for our second statement above, it can be shown similar as before.

$\square$
Proof of concept: Additional experiment data

Appendix B

Figure B.1: Global market share of the Android OS (Source: [http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/])

Figure B.3: Android architecture (Source: http://www.techotopia.com/index.php/File:Android_architecture.png).
### §B.1 Information stealing

#### Table B.1: Monitor results for the category information stealing of the AMGP

<table>
<thead>
<tr>
<th>Cat. 1 ($\varphi_1 - \varphi_{14}$)</th>
<th>$\varphi_1$</th>
<th>$\varphi_2$</th>
<th>$\varphi_3$</th>
<th>$\varphi_4$</th>
<th>$\varphi_5$</th>
<th>$\varphi_6$</th>
<th>$\varphi_7$</th>
<th>$\varphi_8$</th>
<th>$\varphi_9$</th>
<th>$\varphi_{10}$</th>
<th>$\varphi_{11}$</th>
<th>$\varphi_{12}$</th>
<th>$\varphi_{13}$</th>
<th>$\varphi_{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$3^{20}$</td>
<td>$6^{10}$</td>
<td>$2^{58}$</td>
<td>$5^{158}$</td>
<td>$10^{458}$</td>
<td>$15^{458}$</td>
<td>$20^{458}$</td>
<td>$25^{458}$</td>
<td>$30^{458}$</td>
<td>$35^{458}$</td>
<td>$40^{458}$</td>
<td>$45^{458}$</td>
<td>$50^{458}$</td>
<td>$55^{458}$</td>
</tr>
<tr>
<td>2</td>
<td>$2^{267}$</td>
<td>$3^{126}$</td>
<td>$5^{824}$</td>
<td>$7^{824}$</td>
<td>$11^{103}$</td>
<td>$13^{853}$</td>
<td>$17^{853}$</td>
<td>$21^{853}$</td>
<td>$25^{853}$</td>
<td>$29^{853}$</td>
<td>$33^{853}$</td>
<td>$37^{853}$</td>
<td>$41^{853}$</td>
<td>$45^{853}$</td>
</tr>
<tr>
<td>3</td>
<td>$1^{30}$</td>
<td>$2^{30}$</td>
<td>$4^{30}$</td>
<td>$5^{30}$</td>
<td>$7^{30}$</td>
<td>$9^{30}$</td>
<td>$11^{30}$</td>
<td>$13^{30}$</td>
<td>$15^{30}$</td>
<td>$17^{30}$</td>
<td>$19^{30}$</td>
<td>$21^{30}$</td>
<td>$23^{30}$</td>
<td>$25^{30}$</td>
</tr>
<tr>
<td>4</td>
<td>$3^{32}$</td>
<td>$5^{32}$</td>
<td>$7^{32}$</td>
<td>$11^{32}$</td>
<td>$13^{32}$</td>
<td>$17^{32}$</td>
<td>$19^{32}$</td>
<td>$23^{32}$</td>
<td>$29^{32}$</td>
<td>$31^{32}$</td>
<td>$37^{32}$</td>
<td>$41^{32}$</td>
<td>$43^{32}$</td>
<td>$47^{32}$</td>
</tr>
<tr>
<td>5</td>
<td>$1^{256}$</td>
<td>$2^{256}$</td>
<td>$4^{256}$</td>
<td>$5^{256}$</td>
<td>$7^{256}$</td>
<td>$9^{256}$</td>
<td>$11^{256}$</td>
<td>$13^{256}$</td>
<td>$17^{256}$</td>
<td>$19^{256}$</td>
<td>$23^{256}$</td>
<td>$25^{256}$</td>
<td>$29^{256}$</td>
<td>$31^{256}$</td>
</tr>
<tr>
<td>6</td>
<td>$1^{300}$</td>
<td>$2^{300}$</td>
<td>$4^{300}$</td>
<td>$5^{300}$</td>
<td>$7^{300}$</td>
<td>$9^{300}$</td>
<td>$11^{300}$</td>
<td>$13^{300}$</td>
<td>$17^{300}$</td>
<td>$19^{300}$</td>
<td>$23^{300}$</td>
<td>$25^{300}$</td>
<td>$29^{300}$</td>
<td>$31^{300}$</td>
</tr>
<tr>
<td>7</td>
<td>$1^{81}$</td>
<td>$2^{81}$</td>
<td>$4^{81}$</td>
<td>$5^{81}$</td>
<td>$7^{81}$</td>
<td>$9^{81}$</td>
<td>$11^{81}$</td>
<td>$13^{81}$</td>
<td>$17^{81}$</td>
<td>$19^{81}$</td>
<td>$23^{81}$</td>
<td>$25^{81}$</td>
<td>$29^{81}$</td>
<td>$31^{81}$</td>
</tr>
<tr>
<td>8</td>
<td>$1^{41}$</td>
<td>$2^{41}$</td>
<td>$4^{41}$</td>
<td>$5^{41}$</td>
<td>$7^{41}$</td>
<td>$9^{41}$</td>
<td>$11^{41}$</td>
<td>$13^{41}$</td>
<td>$17^{41}$</td>
<td>$19^{41}$</td>
<td>$23^{41}$</td>
<td>$25^{41}$</td>
<td>$29^{41}$</td>
<td>$31^{41}$</td>
</tr>
</tbody>
</table>

| 24 | 13 | 10 | 15 | 2 | 2 | 4 | 3 | 2 | 3 | 8 | 5 | 1 | 4 |
B.2 False positives

Table B.2 and B.3 show the results of MonitorMe for 61 benign apps from the Google Play Store. The left column contains the apps’ package name under which those appear on the test phone (Nexus 5 with version 5.0.1). Note that for Android version 5.0.1, aborting of received SMS as well as sending SMS messages to premium numbers is not possible anymore. Furthermore, there have not been observed any attempts to execute on the command line. For those reasons we leave out Category 2 and 4 in the tables.

Table B.2: False positives for 61 apps on the Nexus 5 running Android 5.0.1—Part 1.
Table B.3: False positives for 61 apps on the Nexus 5 running Android 5.0.1—Part 2.

<table>
<thead>
<tr>
<th>App package name</th>
<th>Cat. 1</th>
<th>Cat. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>de.spiegel.android.app.spon</td>
<td>31.</td>
<td></td>
</tr>
<tr>
<td>com.nianticproject.ingress</td>
<td>32.</td>
<td></td>
</tr>
<tr>
<td>com.google.android.apps.docs.editors.slides</td>
<td>33.</td>
<td></td>
</tr>
<tr>
<td>com.foursquare.robin</td>
<td>34.</td>
<td></td>
</tr>
<tr>
<td>com.meetup</td>
<td>35.</td>
<td></td>
</tr>
<tr>
<td>com.google.android.apps.fitness</td>
<td>36.</td>
<td></td>
</tr>
<tr>
<td>com.google.android.gms.exchange</td>
<td>37.</td>
<td></td>
</tr>
<tr>
<td>com.checkthis.frontback</td>
<td>38.</td>
<td></td>
</tr>
<tr>
<td>com.google.android.apps.docs.editors.docs</td>
<td>39.</td>
<td></td>
</tr>
<tr>
<td>com.mobilemotion.dubsmash</td>
<td>40.</td>
<td></td>
</tr>
<tr>
<td>com.fitnesskeeper.runkeeper.pro</td>
<td>41.</td>
<td></td>
</tr>
<tr>
<td>com.ulmon.android.citymaps2go</td>
<td>42.</td>
<td></td>
</tr>
<tr>
<td>com.andybotting.tramhunter</td>
<td>43.</td>
<td></td>
</tr>
<tr>
<td>netskyscanner.android.main</td>
<td>44.</td>
<td></td>
</tr>
<tr>
<td>com.skype.raider</td>
<td>45.</td>
<td></td>
</tr>
<tr>
<td>com.airbnb.android</td>
<td>46.</td>
<td></td>
</tr>
<tr>
<td>com.macropinch.swan</td>
<td>47.</td>
<td></td>
</tr>
<tr>
<td>com.devuni.compass</td>
<td>48.</td>
<td></td>
</tr>
<tr>
<td>com.netbiscuits.kicker</td>
<td>49.</td>
<td></td>
</tr>
<tr>
<td>de.bvb.android</td>
<td>50.</td>
<td></td>
</tr>
<tr>
<td>com.ubercab</td>
<td>51.</td>
<td></td>
</tr>
<tr>
<td>org.telegram.messenger</td>
<td>52.</td>
<td></td>
</tr>
<tr>
<td>org.thoughtcrime.securesms</td>
<td>53.</td>
<td></td>
</tr>
<tr>
<td>org.mozilla.firefox.sharedID:10142</td>
<td>54.</td>
<td></td>
</tr>
<tr>
<td>com.nytimes.android</td>
<td>55.</td>
<td></td>
</tr>
<tr>
<td>com.twitter.android</td>
<td>56.</td>
<td></td>
</tr>
<tr>
<td>com.tripadvisor.tripadvisor</td>
<td>57.</td>
<td></td>
</tr>
<tr>
<td>com.sticksports.stickcricket2</td>
<td>58.</td>
<td></td>
</tr>
<tr>
<td>com.zep tolab.ctr2.f2p.google</td>
<td>59.</td>
<td></td>
</tr>
<tr>
<td>com.yodo1.crossyroad</td>
<td>60.</td>
<td></td>
</tr>
<tr>
<td>bbc.mobile.news.ww</td>
<td>61.</td>
<td></td>
</tr>
</tbody>
</table>
B.3 Intercepted interface names

Table B.4—Table B.5 show in the left column the different interface names that have been discovered by DroidTracer on a real phone (Nexus 5 with Android version 5.0.1). The right column indicates how often method calls occurred for each interface. For example, for android.app.IActivityManager have been counted with 88,940 the most method calls. This is because the Activity Manager is continuously involved in managing the life cycle of apps, i.e., it starts, stops or resumes them. The data was collected during the false positive experiments; that is, the data is from monitoring the 61 benign apps. Method names are not shown here, as these would be too many. However, this tables provide an overview of the broad range of possible events that can be picked from when using DroidTracer for analysis.

Table B.4: Complete list of observed interface names on Nexus 5—Part 1.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMountService</td>
<td>1,164</td>
</tr>
<tr>
<td>android.accessibilityservice.IAccessibilityServiceConnection</td>
<td>223</td>
</tr>
<tr>
<td>android.accounts.IAccountManager</td>
<td>727</td>
</tr>
<tr>
<td>android.app.IActivityContainer</td>
<td>118</td>
</tr>
<tr>
<td>android.app.IActivityManager</td>
<td>88,940</td>
</tr>
<tr>
<td>android.app.IAlarmManager</td>
<td>1,974</td>
</tr>
<tr>
<td>android.app.IApplicationThread</td>
<td>60,211</td>
</tr>
<tr>
<td>android.app.INotificationManager</td>
<td>429</td>
</tr>
<tr>
<td>android.app.ISearchManager</td>
<td>8</td>
</tr>
<tr>
<td>android.app.IUiModeManager</td>
<td>117</td>
</tr>
<tr>
<td>android.app.admin.IDevicePolicyManager</td>
<td>14</td>
</tr>
<tr>
<td>android.app.backup.IBackupManager</td>
<td>8</td>
</tr>
<tr>
<td>android.bluetooth.IBluetooth</td>
<td>42</td>
</tr>
<tr>
<td>android.bluetooth.IBluetoothA2dp</td>
<td>1</td>
</tr>
<tr>
<td>android.bluetooth.IBluetoothHeadset</td>
<td>1</td>
</tr>
<tr>
<td>android.bluetooth.IBluetoothManager</td>
<td>17</td>
</tr>
<tr>
<td>android.content.IBulkCursor</td>
<td>8,632</td>
</tr>
<tr>
<td>android.content.IClipboard</td>
<td>2</td>
</tr>
<tr>
<td>android.content.IContentProvider</td>
<td>12,941</td>
</tr>
<tr>
<td>android.content.IContentService</td>
<td>10,483</td>
</tr>
<tr>
<td>android.content.ISyncContext</td>
<td>19</td>
</tr>
<tr>
<td>android.content.pm.IPackageManager</td>
<td>15,996</td>
</tr>
<tr>
<td>android.database.IContentObserver</td>
<td>87</td>
</tr>
<tr>
<td>android.drm.IDrm</td>
<td>42</td>
</tr>
</tbody>
</table>

1The complete SQLite database on which this aggregation is based, can be found under [http://kuester.multics.org/DroidTracer/malware/traces/](http://kuester.multics.org/DroidTracer/malware/traces/)
### §B.3 Intercepted interface names

Table B.5: Complete list of observed interface names on Nexus 5—Part 2.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.gui.DisplayEventConnection</td>
<td>38,724</td>
</tr>
<tr>
<td>android.gui.IProducerListener</td>
<td>156</td>
</tr>
<tr>
<td>android.gui.SensorEventConnection</td>
<td>1,151</td>
</tr>
<tr>
<td>android.gui.SensorServer</td>
<td>399</td>
</tr>
<tr>
<td>android.hardware.ICamera</td>
<td>50</td>
</tr>
<tr>
<td>android.hardware.ICameraService</td>
<td>48</td>
</tr>
<tr>
<td>android.hardware.display.IDisplayManager</td>
<td>11,369</td>
</tr>
<tr>
<td>android.hardware.input.IInputManager</td>
<td>248</td>
</tr>
<tr>
<td>android.location.ILocationManager</td>
<td>507</td>
</tr>
<tr>
<td>android.media.IAudioFlinger</td>
<td>478</td>
</tr>
<tr>
<td>android.media.IAudioPolicyService</td>
<td>34,837</td>
</tr>
<tr>
<td>android.media.IAudioSession</td>
<td>68</td>
</tr>
<tr>
<td>android.media.IAudioTrack</td>
<td>456</td>
</tr>
<tr>
<td>android.media.IMediaCodecList</td>
<td>1,463</td>
</tr>
<tr>
<td>android.media.IMediaMetadataRetriever</td>
<td>2,092</td>
</tr>
<tr>
<td>android.media.IMediaPlayer</td>
<td>850</td>
</tr>
<tr>
<td>android.media.IMediaPlayerService</td>
<td>417</td>
</tr>
<tr>
<td>android.media.IMediaRouterService</td>
<td>35</td>
</tr>
<tr>
<td>android.media.session.ISession</td>
<td>273</td>
</tr>
<tr>
<td>android.media.session.ISessionManager</td>
<td>36</td>
</tr>
<tr>
<td>android.net.INetworkStatsService</td>
<td>14,195</td>
</tr>
<tr>
<td>android.net.IWifiManager</td>
<td>91</td>
</tr>
<tr>
<td>android.net.wifi.IWifiManager</td>
<td>1,042</td>
</tr>
<tr>
<td>android.nfc.INfcAdapter</td>
<td>7</td>
</tr>
<tr>
<td>android.os.IMessenger</td>
<td>113</td>
</tr>
<tr>
<td>android.os.IPowerManager</td>
<td>2,620</td>
</tr>
<tr>
<td>android.os.IServiceManager</td>
<td>43,057</td>
</tr>
<tr>
<td>android.os.IUserManager</td>
<td>344</td>
</tr>
<tr>
<td>android.os.IVibratorService</td>
<td>50</td>
</tr>
<tr>
<td>android.ui.IGraphicBufferAlloc</td>
<td>57</td>
</tr>
<tr>
<td>android.ui.ISurfaceComposer</td>
<td>165</td>
</tr>
<tr>
<td>android.utils.IMemory</td>
<td>115</td>
</tr>
<tr>
<td>android.utils.IMemoryHeap</td>
<td>5</td>
</tr>
<tr>
<td>android.view.IAssetAtlas</td>
<td>198</td>
</tr>
<tr>
<td>android.view.IWindowManager</td>
<td>17,710</td>
</tr>
<tr>
<td>android.view.IWindowSession</td>
<td>2,930</td>
</tr>
<tr>
<td>android.view.accessibility.IAccessibilityInteractionConnectionCallback</td>
<td>1</td>
</tr>
<tr>
<td>android.view.accessibility.IAccessibilityManager</td>
<td>20,611</td>
</tr>
<tr>
<td>android.webkit.IWebViewUpdateService</td>
<td>71</td>
</tr>
<tr>
<td>com.android.internal.app.IAppOpsService</td>
<td>744</td>
</tr>
<tr>
<td>com.android.internal.appwidget.IAppWidgetService</td>
<td>365</td>
</tr>
<tr>
<td>com.android.internal.os.IResultReceiver</td>
<td>3</td>
</tr>
<tr>
<td>com.android.internal.telecom.ITelecomService</td>
<td>34,722</td>
</tr>
</tbody>
</table>
Table B.6: Complete list of observed interface names on Nexus 5—Part 3.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.android.internal.telephony.IPhoneSubInfo</td>
<td>225</td>
</tr>
<tr>
<td>com.android.internal.telephony.ISub</td>
<td>2,861</td>
</tr>
<tr>
<td>com.android.internal.telephony.ITelephony</td>
<td>480</td>
</tr>
<tr>
<td>com.android.internal.telephony.ITelephonyRegistry</td>
<td>22</td>
</tr>
<tr>
<td>com.android.internal.textservice.ISpellCheckerSession</td>
<td>15</td>
</tr>
<tr>
<td>com.android.internal.textservice.ISpellCheckerSessionListener</td>
<td>11</td>
</tr>
<tr>
<td>com.android.internal.textservice.ITextServicesManager</td>
<td>264</td>
</tr>
<tr>
<td>com.android.internal.view.IInputContext</td>
<td>2,121</td>
</tr>
<tr>
<td>com.android.internal.view.IInputContextCallback</td>
<td>262</td>
</tr>
<tr>
<td>com.android.internal.view.IInputMethodManager</td>
<td>1,419</td>
</tr>
<tr>
<td>com.android.internal.view.IInputMethodManager Session</td>
<td>686</td>
</tr>
<tr>
<td>com.android.internal.view.IInputSessionCallback</td>
<td>86</td>
</tr>
<tr>
<td>com.android.vending.billing.INAppBillingService</td>
<td>64</td>
</tr>
<tr>
<td>com.facebook.fbservice.observer.IBlueServiceObserver</td>
<td>2</td>
</tr>
<tr>
<td>com.facebook.fbservice.service.IBlueService</td>
<td>2</td>
</tr>
<tr>
<td>com.facebook.fbservice.service.ICompletionHandler</td>
<td>2</td>
</tr>
<tr>
<td>com.google.android.auth.IAuthManagerService</td>
<td>35</td>
</tr>
<tr>
<td>com.google.android.finsky.services.IMarketCatalogService</td>
<td>1</td>
</tr>
<tr>
<td>com.google.android.gms.ads.identifier.internal.IAdvertisingIdService</td>
<td>650</td>
</tr>
<tr>
<td>com.google.android.gms.ads.internal.gservice.IGservicesValueService</td>
<td>10</td>
</tr>
<tr>
<td>com.google.android.gms.ads.internal.request.IAdRequestService</td>
<td>23</td>
</tr>
<tr>
<td>com.google.android.gms.analytics.internal.IAnalyticsService</td>
<td>129</td>
</tr>
<tr>
<td>com.google.android.gms.appdatasearch.internal.IAppDataSearch</td>
<td>153</td>
</tr>
<tr>
<td>com.google.android.gms.clearcut.internal.IClearcutLoggerService</td>
<td>196</td>
</tr>
<tr>
<td>com.google.android.gms.common.internal.IGmsServiceBroker</td>
<td>373</td>
</tr>
<tr>
<td>com.google.android.gms.fitness.internal.IGoogleFitnessService</td>
<td>14</td>
</tr>
<tr>
<td>com.google.android.gms.http.IGoogleHttpService</td>
<td>466</td>
</tr>
<tr>
<td>com.google.android.gms.location.internal.IGoogleLocationManagerService</td>
<td>863</td>
</tr>
<tr>
<td>com.google.android.gms.location.reporting.internal.IReportingService</td>
<td>25</td>
</tr>
<tr>
<td>com.google.android.gms.maps.auth.IApiTokenService</td>
<td>2</td>
</tr>
<tr>
<td>com.google.android.gms.mdm.internal.INetworkQualityService</td>
<td>16</td>
</tr>
<tr>
<td>com.google.android.gms.people.internal.IPeopleService</td>
<td>17</td>
</tr>
<tr>
<td>com.google.android.music.net.IDownloadabilityChangeListener</td>
<td>1</td>
</tr>
<tr>
<td>com.google.android.music.net.INetworkChangeListener</td>
<td>5</td>
</tr>
<tr>
<td>com.google.android.music.net.INetworkMonitor</td>
<td>4</td>
</tr>
<tr>
<td>com.google.android.music.net.IStreamabilityChangeListener</td>
<td>1</td>
</tr>
<tr>
<td>com.google.android.music.playback.IMusicPlaybackService</td>
<td>87</td>
</tr>
<tr>
<td>com.google.android.music.preferences.IPreferenceChangeListener</td>
<td>3</td>
</tr>
<tr>
<td>com.google.android.music.preferences.IPreferenceService</td>
<td>5</td>
</tr>
<tr>
<td>com.google.android.music.store.IStoreService</td>
<td>798</td>
</tr>
<tr>
<td>com.google.android.now.INowAuthService</td>
<td>2</td>
</tr>
<tr>
<td>syscall</td>
<td>1,959</td>
</tr>
</tbody>
</table>
ω-word, 18
(first-order) structure, 32
(first-order) temporal structure, 33
(sorted first-order) signature, 31
acceptance set, 20
accepted language, 20, 22
accepting run, 20
accepting run of an SA, 44
acceptor, 19
action, 4, 32
alphabet, 17
Android OS, 9
Android app, 9
Android system, 9
anticipation, 8
app-centric, 12, 69
atomic propositions, 23
automaton size, 20
Büchi acceptance, 22
biquantified formulas, 60
bound variables, 33
closed formula, 33
closure, 18, 45
complete, 45
complete finite automaton, 20
complete trace, 6
computable structure, 32
concatenation, 18
coreact, 31
data quantification, 3
detection, 2
deterministic automaton, 20
domain, 32
domain independent, 59
effectively monitorable, 73
effectively monitorable, 69
empty word, 17
event, 4, 17, 32
finite run, 20
finite word, 17
formal specification language, 3
generalised Büchi acceptance, 22
generic subformula function, 45
good/bad prefix, 36
impartiality, 7
incomplete, 7
infinite run, 21
intersection, 18
Kripke language, 39
Kripke structure, 18, 88
labelling function, 18
language, 18, 23, 34
language of an SA, 44
left-total, 18
level, 44
linear Kripke structure, 19
linear time, 23
liveness language, 69
local language, 44
locally accepting run, 44
marshal, 79
met obligation, 48
metric first-order temporal logic, 60
minimal good/bad prefix, 36
model, 23
modular, 12
monitor, 4
monitorability, 69
monotonicity, 7
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-deterministic finite automaton</td>
<td>19</td>
</tr>
<tr>
<td>obligation</td>
<td>48</td>
</tr>
<tr>
<td>offline monitoring</td>
<td>6</td>
</tr>
<tr>
<td>online monitoring</td>
<td>6</td>
</tr>
<tr>
<td>parametric property</td>
<td>62</td>
</tr>
<tr>
<td>platform-centric</td>
<td>12</td>
</tr>
<tr>
<td>portable</td>
<td>12</td>
</tr>
<tr>
<td>potential constraint satisfaction</td>
<td>59</td>
</tr>
<tr>
<td>potentially locally accepting</td>
<td>48</td>
</tr>
<tr>
<td>prefix</td>
<td>18</td>
</tr>
<tr>
<td>prefix problem</td>
<td>36</td>
</tr>
<tr>
<td>proposition</td>
<td>31</td>
</tr>
<tr>
<td>propositional runtime verification</td>
<td>5</td>
</tr>
<tr>
<td>quantifier depth</td>
<td>46</td>
</tr>
<tr>
<td>regular expression</td>
<td>20</td>
</tr>
<tr>
<td>regular languages</td>
<td>19</td>
</tr>
<tr>
<td>restricted subformula function</td>
<td>45</td>
</tr>
<tr>
<td>run of an SA</td>
<td>44</td>
</tr>
<tr>
<td>runtime overhead</td>
<td>8</td>
</tr>
<tr>
<td>runtime verification</td>
<td>2</td>
</tr>
<tr>
<td>safety</td>
<td>1</td>
</tr>
<tr>
<td>safety property</td>
<td>36</td>
</tr>
<tr>
<td>satisfiable</td>
<td>23</td>
</tr>
<tr>
<td>satisfying a property</td>
<td>23</td>
</tr>
<tr>
<td>spawning automaton</td>
<td>44</td>
</tr>
<tr>
<td>spawning function</td>
<td>44</td>
</tr>
<tr>
<td>state formula</td>
<td>25</td>
</tr>
<tr>
<td>strong LTL semantics</td>
<td>27</td>
</tr>
<tr>
<td>strongly trace-length dependent</td>
<td>72</td>
</tr>
<tr>
<td>system event</td>
<td>4</td>
</tr>
<tr>
<td>temporal</td>
<td>23</td>
</tr>
<tr>
<td>temporal quantification</td>
<td>3</td>
</tr>
<tr>
<td>term</td>
<td>31</td>
</tr>
<tr>
<td>test oracle</td>
<td>35</td>
</tr>
<tr>
<td>trace</td>
<td>17</td>
</tr>
<tr>
<td>trace-length independent</td>
<td>71</td>
</tr>
<tr>
<td>traces with data</td>
<td>5</td>
</tr>
<tr>
<td>transducer</td>
<td>21</td>
</tr>
<tr>
<td>transition function</td>
<td>19</td>
</tr>
<tr>
<td>truncated trace</td>
<td>27</td>
</tr>
<tr>
<td>truth domain</td>
<td>28</td>
</tr>
<tr>
<td>union</td>
<td>18</td>
</tr>
<tr>
<td>unsatisfiable</td>
<td>23</td>
</tr>
<tr>
<td>untimed</td>
<td>3</td>
</tr>
<tr>
<td>verification</td>
<td>1</td>
</tr>
<tr>
<td>violated obligation</td>
<td>48</td>
</tr>
<tr>
<td>violating a property</td>
<td>23</td>
</tr>
<tr>
<td>well-formed LTL-formulae</td>
<td>23</td>
</tr>
</tbody>
</table>