UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
IECS International Doctoral School

Assumption-Based Runtime Verification of Finite- and Infinite-State Systems

Chun Tian

Advisor

Dr. Alessandro Cimatti

Fondazione Bruno Kessler, Italy

Co-Advisor

Dr. Stefano Tonetta

Fondazione Bruno Kessler, Italy

November 2022

Abstract

Runtime Verification (RV) is usually considered as a *lightweight* automatic verification technique for the dynamic analysis of systems, where a *monitor* observes executions produced by a system and analyzes its executions against a formal specification. If the monitor were synthesized, in addition to the monitoring specification, also from extra *assumptions* on the system behavior (typically described by a model as transition systems), then it may output more precise verdicts or even be predictive, meanwhile it may no longer be lightweight, since monitoring under assumptions has the same computation complexity with model checking.

When suitable assumptions come into play, the monitor may also support *partial observability*, where non-observable variables in the specification can be inferred from observables, either present or historical ones. Furthermore, the monitors are *resettable*, i.e. being able to evaluate the specification at non-initial time of the executions while keeping memories of the input history. This helps in breaking the monotonicity of monitors, which, after reaching conclusive verdicts, can still change its future outputs by resetting its reference time. The combination of the above three characteristics (assumptions, partial observability and resets) in the monitor synthesis is called the *Assumption-Based Runtime Verification*, or ABRV.

In this thesis, we give the formalism of the ABRV approach and a group of monitoring algorithms based on specifications expressed in Linear Temporal Logic with both future and past operators, involving Boolean and possibly other types of variables. When all involved variables have finite domain, the monitors can be synthesized as finite-state machines implemented by Binary Decision Diagrams. With infinite-domain variables, the infinite-state monitors are based on satisfiability modulo theories, first-order quantifier elimination and various model checking techniques. In particular, Bounded Model Checking is modified to do its work incrementally for efficiently obtaining inconclusive verdicts, before IC3-based model checkers get involved.

All the monitoring algorithms in this thesis are implemented in a tool called *NuRV*. NuRV support online and offline monitoring, and can also generate standalone monitor code in various programming languages. In particular, monitors can be synthesized as SMV models, whose behavior correctness and some other properties can be further verified by model checking.

Keywords
Formal Methods, Runtime Verification, Assumption-Based, Linear Temporal Logic, Bounded Model Checking

Contents

1 Introduction **1**

1.1 Motivation 2
1.2 Assumption-Based RV Approach 3
1.3 Innovative Aspects (aka Contributions) 5
1.4 Product (aka Result) 6
1.5 Structure of the Thesis 7

2 Background and Related Work **9**

2.1 Background 9
2.2 Taxonomy, Frontiers and Trends 11
2.3 Assumption-Related Work 12
2.4 SMT-Related RV Work 13

3 Preliminaries **15**

3.1 Finite and infinite words (or traces) 15
3.2 Satisfiability Modulo Theory 16
3.3 Linear Temporal Logic 16
3.4 Boolean formulae and functions 17
3.5 Binary Decision Diagrams 18
3.6 Fair Kripke Structure (Fair Transition System) 18
3.7 LTL to ω-automata Translation 19
3.8 Explicit-State Automata 21
3.9 First-Order Quantifier Elimination 23
3.10 LTL Model Checking 23

4 Runtime Verification **25**

4.1 LTL semantics for Runtime Verification 25
4.2 Runtime Verification Based on LTL_3 29
4.3 Assumptions and Partial Observability 30

4.4 Assumption-Based Runtime Verification . . . 31
4.5 Motivating Example . . . 33
4.6 Theoretical Results of ABRV . . . 34

5 Monitoring Finite-State Systems 43
5.1 Symbolic Monitoring Algorithm . . . 43
5.2 Explicit-State Monitor Construction . . . 47
5.3 From Offline to Online Monitoring . . . 52
5.4 Code Generation . . . 52

6 Monitoring Infinite-State Systems 55
6.1 Motivating Example . . . 56
6.2 ABRV Reduced to Model Checking . . . 57
6.3 ABRV Reduced to MC and QE . . . 60
6.4 Optimization to ABRV-MC Reduction . . . 62
6.5 Incremental Bounded Model Checking . . . 66
6.6 ABRV with Incremental BMC . . . 69
6.7 Unboundedness of Infinite-State Monitors . . . 75

7 Monitoring ptLTL (Past-Time LTL) 77
7.1 Introduction . . . 77
7.2 Connection with LTL_3 Semantics . . . 78
7.3 Contructing Explicit-state ptLTL Monitors . . . 83
7.4 Monitoring the original semantics of ptLTL . . . 83

8 NuRV: The Tool Implementation 85
8.1 Functionalities . . . 85
8.2 Architecture . . . 88
8.3 Use Case Scenario . . . 88
8.4 Online Monitoring . . . 90
8.5 Offline Monitoring . . . 91
8.6 API of Generated Code . . . 93
8.7 Code Generation – Backends . . . 94

9 Experimental Evaluation 107
9.1 Tests for Finite-State Monitors . . . 107
9.2 Tests for Infinite-State Monitors . . . 114

10 HOL Formalization **117**
10.1 Introduction . . . 117
10.2 Higher Order Logic (HOL) . . . 119
10.3 Linear Temporal Logic in HOL . . . 121
10.4 Partial Formalization of Main Theorem 5.1.2 . . . 126
10.5 LTL_3 and ptLTL (Alternative Semantics) . . . 127

11 Conclusions **129**
11.1 Future Directions . . . 130

Bibliography **141**

A Data and Tables **157**
A.1 SMV Models . . . 157
A.2 Dwyer's LTL Patterns . . . 158
A.3 Formal proofs . . . 161

B CORBA-Based Client-Server Monitoring **167**
B.1 About CORBA . . . 167
B.2 Client-Server Monitoring . . . 168
B.3 Additional Software Dependencies . . . 169
B.4 A Tutorial of CORBA-based Monitor . . . 170
B.5 The Simple Monitor Interface . . . 172
B.6 Monitor Client Programming . . . 173

Index **181**

List of Tables

4.1 Comparing LTL Semantics for Runtime Verification 29
4.2 Output table in ABRV-MC reduction 37
4.3 Alarm conditions as LTL [24] 41

6.1 Output Table of Fig. 6.1 and Algorithm 7 60

8.1 Classification of NuRV according to the taxonomy [66] 86
8.2 Distinguished features of three NuRV modes 87

9.1 Eight long formulae from Dwyer's patterns 110
9.2 The original QTL specification used in tests. $[p, q)$ is an abbreviation of $(\neg q)\,\mathbf{S}\,p$. 112
9.3 The ptLTL versions of QTL specifications of Table 9.2. 112
9.4 Evaluations of DejaVu and NuRV. 113

A.1 Dwyer's LTL patterns 158
A.2 Dwyer's LTL patterns (continued) 159
A.3 Size of monitors built from LTL patterns 160

List of Figures

1.1 Traditional RV (LTL_3-based, left) vs. ABRV (right) 4

4.1 Semantics of FLTL formulae over a trace $u = a_0 \dots a_{n-1} \in \Sigma^*$ 26
4.2 Semantics of $LTL^{\mp}$ formulae over a trace $u = a_0 \dots a_{n-1} \in \Sigma^*$ 27
4.3 ABRV Illustration 33
4.4 The Factory Model 34
4.5 LTL_3 (left) v.s. ABRV-LTL (right), the direction of arrows indicate the possible changes of monitor outputs after more inputs (without being reset) 35
4.6 ABRV reduced to MC 37

5.1 LTL monitors of $p \mathbf{U} q$ (level 1–3), assuming $p \neq q$ 50
5.2 LTL monitors of $\mathbf{Y}p \vee q$ (level 1 & 3), assuming $p \neq q$ 51

6.1 ABRV reduced to MC and QE 60

7.1 ptLTL (alternative) semantics vs. LTL_3 semantics 83
7.2 ptLTL monitors of *closed* $\wedge$ $\mathbf{Y}$ *open*' (level 3 and 4) (assuming *open* $\neq$ *closed*) . 84

8.1 The architecture of NuRV 87
8.2 Batch commands for generating monitors of $p\,\mathbf{U}q$ and $\mathbf{Y}p \vee q$ 89
8.3 `disjoint.smv` and `default.ord` 89
8.4 Offline monitoring in NuRV 89
8.5 The SMV file `disjoint.smv` for $p \mathbf{U} q$ (assuming $p \neq q$) 91
8.6 Explicit-state monitor of $\mathbf{G}(p \rightarrow \mathbf{X}\mathbf{X}q)$ 95

9.1 The number of observations before a conclusive verdict, with and without assumptions 109
9.2 Performance of generated Java monitors on 10^7 states. 111
9.3 Performance of five RV algorithms on Pattern 49 115
9.4 Performance of `bmc_monitor` and `monitor2_optimized` on all patterns . . . 116

10.1 HOL's type grammar 120
10.2 HOL's term grammar 120

Chapter 1

Introduction

Formal verification is the act of proving or disproving behavior correctness of systems, either mathematical, physical or cyber-physical, with respect to certain formal specifications described in logical or mathematical formulas. Traditional formal verification techniques mainly include *Model Checking* [53], *Testing* [29] and *Theorem Proving* [21].

Model Checking (MC) belongs to static formal verification techniques. It provides the ultimate guarantee that a system satisfies a specification, and is fully automated (thus easier to apply) in comparison with techniques like Theorem Proving. But applying MC to large complex systems is difficult, as it usually suffers from the so-called *state-explosion* problem [34], while the success of its application highly depends on the quality of the models; Testing, on the other hand, is effective in showing the defects of the system but is totally incapable of proving its correctness. Theorem Proving, either interactive (e.g. HOL [81] and Coq [21]) or semi-automatic (e.g. ACL2 [99]) ones, usually requires very expressive logics (undecidable, thus cannot be automatic in general) and can show the correctness of complex systems, but developing formal proofs in theorem provers is a time-consuming and highly non-trivial task even for domain experts (see, e.g., [55, 95]).

Runtime Verification (RV) is usually considered as a *lightweight* automatic formal verification technique for the dynamic analysis of systems, where a *monitor* observes executions produced by a system and analyzes its executions against a formal specification.[1] In other words, RV is "the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny (SUS) satisfies or violates a given correctness property" [111]. RV in the wide sense covers at least two areas:

1. *Instrumentation*: how to obtain the needed inputs from the system under scrutiny, to feed the monitor.

[1] Courtesy of Prof. César Sánchez for this definitional sentence occurred in his review texts of the first version of the present thesis.

2. *Monitor Synthesis*: how to build a monitor from a given specification (and perhaps also from other information).

3. *Execution Analysis*: how to actually run the monitor at runtime, analyze its outputs, and perhaps execute actions based on the monitor outputs.

The present thesis focuses on monitor synthesis under assumptions. Unified solution is the key contribution. For the monitoring specification language, we focus on Linear Temporal Logic with future and past operators, with Boolean and possibly other types of variables.[2]

1.1 Motivation

Traditional RV techniques have some severe limitations.

One classical RV research is to synthesize *Propositional Linear Temporal Logic* (PLTL or just LTL) formulas into finite-state machines by leveraging tableau translations from LTL to ω-automata, widely used by model checking community [19]. To fill the gaps between standard LTL semantics over infinite traces and the fact that runtime monitors can only have finite trace prefixes as inputs, the automata-based monitor synthesis must be adapted to slightly different LTL semantics over finite traces, e.g. LTL_3 [6]. This automata-based RV approach is also the starting point of the present thesis work.

But a wide range of practical monitoring properties are *non-monitorable* with respect to LTL_3, i.e. for these specifications the corresponding monitor cannot emit any verdict other than *unknown* (or *inconclusive*). (Bauer et al. report that 44% of the formulas they consider in their experiments are non-monitorable [118].)

Besides, most of existing RV work can only handle fully observed systems: all state variables of the SUS, at least those occurred in monitoring properties, must be fully observable from the input traces. But in practice, many systems under scrutiny are only partially observable: some state variables occurred in the monitoring specifications are non-observable from the input traces.

Another limitation is that, runtime monitors tend to be *monotonic*: once the monitor has reached any conclusion by outputting a conclusive verdict, it tends to keep the same outputs despite the new inputs. This behavior is correct with respect to the formal definition of monitors based on LTL semantics, but this means that the monitor becomes useless after its first conclusive answer. Of course, the monitoring program could be restarted, but then it will forget all existing inputs and the current internal states of the system, violating its formal definition.

[2]The tool implementation, however, is intended to be used in real projects (and have been actually used in several such projects), in which the instrumentation and execution analysis of the monitors were done by project developers, successfully.

1.2 Assumption-Based RV Approach

Many blackbox systems are not truly black boxes. In practice one almost always knows something about the SUS. This knowledge can be derived, for example, from models produced during the system design, or from the interaction with human operators.

If the monitor were synthesized, in addition to the monitoring specification, also from extra *assumptions* on the system behavior (typically described by a *model*), The benefits of incorporating extra assumptions during the monitor synthesis are manifold. For certain combinations of assumptions (models) and monitoring properties, assumption-based monitors are more precise (arriving at a conclusion based on the assumption while traditional monitors would be inconclusive), or even *predictive* (arriving at a conclusion before the input trace actually says so); in particular, if the monitor would never have reached a conclusive verdict, it might do so because of the assumption. (Such assumptions always exist, see [91].)

However, under assumptions the monitor may no longer be lightweight, since monitoring under assumptions has, in general, the same computation complexity with Model Checking, as we will show in this thesis.

When suitable assumptions come into play, the monitor may support *partial observability*, where the value of non-observable variables in the specification can be inferred from observables, either present or historical. Essentially, a partially observable trace can be viewed as a set of traces whose *projection* to the observable set of variables is the observed one. In this case, the monitor should emit a *conclusive* verdict if and only if all these traces compatible with the observations led to the same verdict. Thus, partial observability also makes a connection between RV and another field such as *Fault Detection, Identification (FDI)* [25], sometimes also called *FDIR* ("R" means "Recovery"), where the central task is the generation of diagnoser which, under certain conditions, can be simulated by a sequence of monitors with partial observability.

Furthermore, in this framework the monitors are *resettable monitors*, i.e. being able to evaluate the specification at arbitrary time of the executions while keeping memories of historical inputs. Roughly speaking, for any LTL formula φ, its semantics $[\![u, i \models \varphi]\!]$ can be evaluated (by a resettable monitor) for any trace u and position i of the trace, with the underlying assumptions taken into account. Consider the case where the monitor is evaluating an LTL property φ from the initial position (as in the traditional monitors), denoted by $[\![u, 0 \models \varphi]\!]$ (or just $[\![u \models \varphi]\!]$). Upon a sequence u of observations, receiving a new observation a as the next input, together with a reset signal, the monitor will evaluate φ from the last index of u: $[\![u \cdot a, |u| \models \varphi]\!]$. Taking one more observation b and this time without resets, the monitor will still evaluate φ in the previous position: $[\![u \cdot a \cdot b, |u| \models \varphi]\!]$.

The combination of the above three characteristics (assumptions, partial observability and resets) in monitor synthesis is called *Assumption-Based Runtime Verification*, abbreviated as

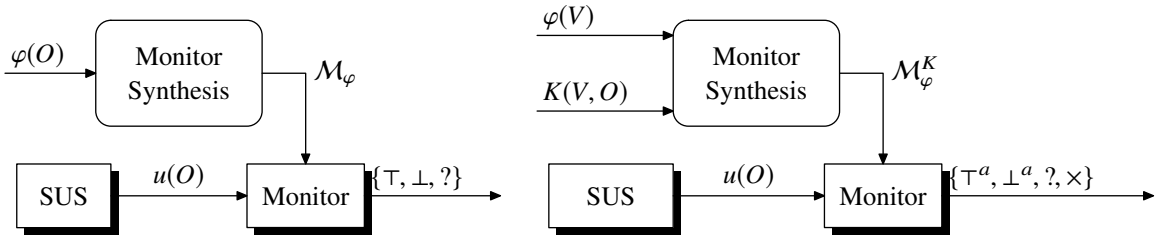


Figure 1.1: Traditional RV (LTL_3-based, left) vs. ABRV (right)

ABRV.

ABRV is a novel RV approach aiming at improving the quality and features of monitors by leveraging extra information and controls over the runtime monitors. The uses of extra assumptions, together with the support of partial observability and resets, have shown to be valuable as a unified solution to all limitations mentioned in previous section. To see the difference between RV and ABRV more clearly, in Fig. 1.1, the left part, there is a monitoring property $\varphi(O)$ built by only observable variables (from the set O), being synthesized into a monitor automaton $\mathcal{M}_\varphi$. The actual monitor program uses this automaton as its core, and takes at runtime an input trace $u(O)$ (i.e. a sequence of variable assignments of O) from the SUS outputting a 3-valued verdict corresponding to the entire trace (or a sequence of verdicts corresponding to each states of the trace). In Fig. 1.1, the right part, one can see the following differences of ABRV in comparison with traditional RV approaches:

1. The monitoring property may contain variables (in $V \setminus O$) which is unobservable from the trace;

2. The monitor synthesis process may leverage more information coming from a model (or partial model) K of the SUS;

3. The monitoring outputs can additional indicate (by outputting $\times$) that the input trace $u(O)$ does not follow K;

4. It is possible to reset the monitor on behave of a recovery strategy of the monitors. After the reset, the monitor still sees the same input trace, just using the resetting time as the reference time for the evaluation of monitoring specification.

In summary, ABRV proposes the following solutions to the limitations mentioned in the previous section:

1. Unlike other researchers who try to modify LTL_3 semantics to give "conclusive" verdicts to some non-monitorable properties, ABRV uses extra assumptions in the monitor synthesis task, so that non-monitorable properties may become monitorable.

2. With assumptions, the monitoring of partially observable systems become possible and practical.

3. By resetting the monitor, it evaluates the monitoring specification using a different reference time from the initial one. The monotonicity of monitors is thus broken and the monitors can still be useful after the initial conclusive answer while still follows its formal definition.

1.3 Innovative Aspects (aka Contributions)

Due to the complexity of cyber-physical systems, their implementations at runtime may behave differently from those given by their models. Thus it is not true that the actual traces of the systems always satisfy their models. This is why we call these models "assumptions". With assumptions in consideration, there are four possible verdicts in the ABRV framework, based on LTL_3 semantics: *conclusive true* ($\top^{\text{a}}$), *conclusive false* ($\bot^{\text{a}}$), *inconclusive* (?) and *out-of-model* ($\times$). The additional verdict *out-of-model* indicates that the monitor inputs have violated the assumptions under which the monitor is synthesized. We consider this possibility always exists and has brought it into our monitoring framework. Note that, besides LTL_3, other monitoring framework based on more refined LTL semantics can also incorporate this additional *out-of-model* verdict, once assumptions were introduced. Even for monitors not based on LTL, as long as assumptions are introduced, the possibility of monitors not following assumptions always exists. This amendment of monitoring verdicts forms the first contribution in this thesis.

The second contribution is the concept of *resets* of runtime monitors and an simple, elegant way to actually reset the monitors. Before our work, all LTL-based monitors essentially can only evaluate LTL formula φ over a finite trace at the fixed time 0, i.e. roughly speaking, the monitor is formally defined as $\mathcal{M}^{\varphi}(u) := [\![u, 0 \models \varphi]\!]$. The monotonicity (and the uselessness of monitors after the initial conclusive answer) essentially comes from the fact that the evaluation time cannot be easily changed by doing automata-based LTL translations. With resets, now it is possible to synthesize monitors doing $[\![u, i \models \varphi]\!]$ where i as the trace position (which represents the discrete time) can be arbitrary. The only limitation is that, each time when the monitor is reset, the new value of i will be the discrete time when the reset happens (which is also the length of current trace prefix, minus one), thus is always mono-increasing. (Therefore it is not possible, for example, to first have $i = 5$ and then to have $i = 3$.)

We remark that this reset (also called *soft reset*) is different from simply restarting the monitors (which can be called *hard reset* to distinguish). If the monitoring properties were evaluated under assumptions, or if the properties contained past temporal operators, the observations received before resets may contribute to the semantics of the monitoring property at the new position in time. For example, consider the LTL property $\varphi = \mathbf{G}\,\neg p$ ("p never occurs"), with an assumption K stating that "p occurs at most once." (see Dwyer's patterns for the precise LTL formula.) For

every input sequence u containing p, the monitor should report a violation of the property, with or without the assumption. After a violation, if the monitor is reset, given the assumption K on the occurrence of p, the monitor should predict that the property is satisfied by any continuation, because the monitor does not forget that a violation has already occurred in the past. Should the SUS produce a trace violating the assumption, where p occurs twice at time i and at $j > i$, the assumption-based monitor will output "×" (out-of-model) at time j.

We also need to remark that, in some RV work, the LTL formula being considered has only past operators. This fragment of LTL is usually called ptLTL, and its semantics over finite traces, if described in standard LTL semantics, has the reference time always at the end of the trace prefix. Existing work on ptLTL monitoring are all based on rewriting of ptLTL formulas into evaluation of it sub-formulas until the verdicts become trivial. With resets, we get the first automata-based approach for monitoring ptLTL. Note that this "first" is in the sense that, LTL-to-automata translation of ptLTL can be directly used for constructing the monitor. We think this was impossible without using the concept of resets. (See Chapter 7 for more details and citations.)

The third contribution is the SMT-based algorithm for runtime monitoring of infinite-state systems based on *Incremental Bounded Model Checking*. (Of course, the same technique can also be used for finite-state systems with SAT solvers in place of SMT solvers.) We start by showing that ABRV of LTL formulas can be reduced to LTL model checking. The theory domain is actually irrelevant with the monitoring algorithm: if the LTL formulas involve infinite-state variables, then infinite-state model checkers will do the job. With help of first-order quantifier elimination procedures, the monitor is online and incremental. To further optimize the use of complete model checking procedures, we can use Bounded Model Checking (BMC) first for detecting counter-examples only. Luckily, we found that, before a monitor reaches its first conclusive verdicts, it only needs BMC and the BMC procedure can be modified to do the job incrementally (no need to restart BMP loop from 0).

1.4 Product (aka Result)

We present a series of symbolic monitoring algorithms for all the above mentioned features in ABRV. The algorithm takes as input the monitoring property, the assumption and a finite partially observed trace representing the input sequence from SUS, mixed with possible reset signals (thus the trace contains reset signals). The monitoring output is a sequence of four-valued verdicts as mentioned above. The algorithm is based on an existing *symbolic* translation from LTL to ω-automata with minor modifications. Assumptions are supported by (symbolically) composing the ω-automata with a symbolic (fair) transition system representing the assumptions. The algorithm explores the space of *belief states*, i.e. the set of SUS states compatible with the current

observation from traces, and the symbolic computation of forward images naturally supports partial observability. The support of resettable monitors exploits some deep characteristics of the symbolic translation from LTL to ω-automata.

The monitoring algorithm is represented as an *offline* monitor but essentially this is an *online* monitoring algorithm: the input sequence can be fed into the algorithm incrementally and the memory consumption of the algorithm is independent with the length of the input sequence. In addition, we can pre-compute the states of the monitor for generating standalone *direct* monitors as finite-state automata. Then, the automata can be transformed into runtime monitors in various programming languages.

The software engineering part of this thesis is mostly about the implementation of our ABRV approach into a working software called *NuRV*, which is extended from the nuXmv model checker [37]. Besides online and offline monitoring directly using the software, NuRV can generate implicit (interpreted) or explicit (direct) monitors which can work standalone and can be deployed in online or offline modes. For the explicit monitors, NuRV generates embedded standalone monitor code in various programming languages. In addition, the monitor can be generated as SMV models, whose behaviors can be further verified in nuXmv.

A lot of efforts were spent on monitor code generation, in several programming languages and the *SMV model language*. The correctness and other properties of a monitor generated in SMV can be further verified by nuXmv model checker.

The proof engineering part of this thesis (Chapter 10) is a formally verified equivalence theorem between ptLTL (alternative semantics) and LTL_3 (Theorem 7.2.2). For this purposes, we have also discussed several existing LTL formalizations in HOL4 (Higher Order Logic), which may lead to a full formalization of RV algorithms in thesis, in the future.

We have evaluated NuRV on a number of benchmarks including Dwyer's LTL patterns, showing the feasibility, applicability and usefulness of assumptions. In addition to the correctness proof of the monitoring algorithm, it is possible to use nuXmv to verify the correctness and the effectiveness of each individual monitor generated in SMV language.

1.5 Structure of the Thesis

The rest of this thesis is organized as follows: In Chapter 2 we briefly introduce the research background and some related works, before going into technical details. Chapter 3 give some preliminaries for understanding the rest of the thesis, with all dependent concepts and theoretical utilities recalled with fair amount of details. The definition of Runtime Verification and the assumption-based extension are formulated in Chapter 4, which essentially gives the problems addressed by this thesis. The solutions are subsequently given in Chapter 5 and Chapter 6 for finite- and infinite-state systems, respectively. Additionally, in Chapter 7 we discuss how

Past-time LTL can be monitoring with our ABRV framework in a unified approach.

Starting from Chapter 8, this thesis switches from scientific to engineering contexts where the NuRV tool, which implements all previous mentioned RV algorithms, is introduced. This chapter also includes more details on monitor code generation. (Besides, a CORBA-based client-server monitoring facility implemented by the tool can be found in Appendix B.) In Chapter 9 we use NuRV to evaluate the performance and other aspects of the RV algorithms given in this thesis, with a basic comparison with some other RV tools.

Chapter 10 discusses the proof engineering part of the thesis. In this chapter, we present a partial formalization on the correctness of the finite-state ABRV monitors in Chapter 5, together with the formal version of the equivalence between ptLTL and LTL_3 semantics given in Chapter 7. Interactive Theorem Proving based on Higher Order Logic is involved here.

Finally, in Chapter 11 we conclude the entire thesis, with some discussions on possible future work in Section 11.1.

Chapter 2

Background and Related Work

The recent 20 years have seen great developments and applications of RV techniques as a completion of traditional formal methods, especially model checking. Many RV tools [10, 12, 40, 86, 89, 114, 131] have been published. (See also [66] for a more complete list of RV tools with a taxonomy.) In this chapter, we discuss some of the major progress in the field, together with works related to our assumption-based RV approach.

2.1 Background

Let us briefly review the history of RV up to the *state of the art* of this field, following the published papers in the past RV conferences since 2001. Although some RV papers are published in other conferences, journals and monograph books, most of these works are just extended versions of their initial work published in RV conferences. (Papers of other sources are referenced whenever necessary.) See also [111] for a brief account of runtime verification up to 2009.

According to the preface of RV conference proceedings in recent years, the RV conferences started in 2001 as an annual workshop and turned into a dedicate conference in 2010. "The workshops were organized as satellite events of established forums, including the Conference on Computer-Aided Verification and ETAPS." In fact, the first workshop on Runtime Verification (RV 2001) was held in Paris, France on 23 July 2001, as a satellite event of Computer Aided Verification conference (CAV 2001).

The idea of runtime monitors, however, has a slightly longer history. For instance, commercial network management (and monitoring) software such as HP OpenView[1] has the Fault/Resource Status Monitoring functionalities. RV, however, is distinguished by generating runtime monitors *automatically* from formal specifications, e.g., in Linear Temporal Logic (LTL) [115, 128].

[1] https://en.wikipedia.org/wiki/HP_OpenView

Early RV attempts (2001-2005) mostly focus on adding runtime assertion features (which is more powerful than the builtin assertion features) into existing programming languages like Java [13, 78, 98, 103, 104, 125, 132]. Other programming languages ever involved in RV field include C, Scala, Haskell, Erlang and OCaml, etc. [10, 35, 36, 69, 74, 144] For the particular case of Java (in which many industrial applications are written), the related RV research soon split into two major approaches: One is the idea of monitoring at Java byte-code level, which is sometimes easier and *universal*, i.e., without the need of modifying the original Java code) [1, 73, 85, 87, 88, 139]. The other approach is called *Monitoring-Oriented Programming (MOP)* [39, 40, 42, 123], a new programming paradigm inspired by Aspect-Oriented Programming (AOP)[2]. When using MOP, the user needs to write the monitoring properties in a particular syntax given by the MOP framework, then a mini compiler is used to compile the properties into another piece of Java code to be used together with the Java application under scrutiny. Both approaches went beyond Java: some other programming languages like Scala are also based on Java virtual machine and byte-code, while MOP has a branch called *BusMOP* for monitoring PCI bus traffic (the usual MOP approach is then called *JavaMOP*.) MOP has been proven very successful and was well known for its efficiency. However, an important characteristic of this period is that each approach and tool involves their own API or specification language for the description of monitoring properties, making migrations of RV application from one approach to another a very hard task.

The RV community soon agreed on adopting various *Temporal Logics* as the common specification languages for monitor synthesis.[3] This also paves the way for RV competitions in which different RV tools are compared by synthesizing monitors from the same set of specifications (see [14] for a review of the first five years of the International Competition on Runtime Verification (CRV)). Although branching temporal logics like Hennessy-Milner Logic (HML) are sometimes involved [3, 7, 71], most researchers adopt variants of LTL [115] as the working basis. This is a natural and reasonable choice, because in typical RV application scenario the monitor is only given a single trace (or execution) of the system under scrutiny and the evaluation of monitoring properties w.r.t. the given trace naturally fits into linear-time semantics, i.e. the states of the trace form a linear order of time. See also [65] for a detailed description of MOP and other early RV techniques.

The early history of RV regarding LTL focused on filling the gaps between standard LTL semantics over *infinite* traces and the reality that runtime monitors can only see *finite* prefixes of the infinite traces. In the case of automata-based RV approaches, although various translations from LTL to ω-automata are available from the Model Checking community, one cannot just use the translated automata as monitors by simulating input traces in them. As a branch of formal methods, to synthesize LTL-based monitors, the formal definition of runtime monitors w.r.t. the

[2]See, e.g., `https://en.wikipedia.org/wiki/Aspect-oriented_programming`.

[3]The MOP framework also supports different "logic plugins" including LTL and ptLTL [123].

standard LTL semantics must be established first.

Among various attempts (see [18] for more details), the LTL_3 semantics [19] seems the most convincing one due to its clarity and simplicity. LTL_3 is based on the observation that, given a finite trace as the prefix of some possible infinite traces extending it, some extensions may satisfy property (according to the standard LTL semantics), while others may violate the property. An LTL_3 monitor only returns *conclusive true* (or *conclusive false*, respectively) if *all* extensions of the current prefix satisfy (or violate, respectively) the monitoring property, otherwise the monitor returns *unknown* (or *inconclusive*). Under LTL_3, the LTL-RV problem can be resolved by combining the two automata translated from the monitoring property and its negated version [19].

However, many practical LTL properties are *non-monitorable* under LTL_3. An LTL property is *monitorable* iff the monitor generated from it has the possibility to return verdicts other than *inclusive*. There are in general three possible solutions to this problem:

1. Replacing LTL_3 with four-valued LTL [110], in which the inconclusive verdict (?) is refined into $\top^{\mathrm{p}}$ and $\bot^{\mathrm{p}}$ (the superscript p may stand for *probable*). However, without extra information it is not meaningful saying a prefix having ?-verdict before now has more probability to lead to a $\top$- (or $\top$-verdict, resp.) in the future.

2. Predictive LTL semantics, e.g. [150]. The LTL_3 semantics is extended with bounded predictions on the inputs, usually expressed in a table (mapping the current input to the possible next inputs). And this may turn some non-monitorable properties monitorable.

3. Model-based LTL semantics, e.g. [109]. The LTL_3 semantics is extended with a model such that all inputs must be compatible with the model. With a suitable model, the resulting monitor can also be predictive. However, LTL_3 cannot capture the possibility that the actual inputs are not compatible with the model, which may happen at runtime. ABRV can be considered as a further extension of this idea.

2.2 Taxonomy, Frontiers and Trends

The research scope and basic approaches of RV is relatively "stabilized" since the publication of the "official lecture" in 2017 [15]. A taxonomy of RV tools first appears in 2012 [108] and later appears again as a more complete work in 2018 [66].

The assumption-based approach presented in this thesis or any other similar work has no position in the above taxonomy. (The first two ABRV papers were published in RV 2019.) Without considering assumptions, however, it is possible to put our tool implementation into the 2018 taxonomy. See Section 8.1 for more details.

2.3 Assumption-Related Work

The idea of leveraging partial knowledge of a system to improve monitorablity is not altogether new. Leucker [109] considers an LTL_3-based predictive semantics $\mathrm{LTL}_{\mathcal{P}}$, where, given a finite trace u, an LTL formula φ is evaluated on every extension of u that are paths of a model $\hat{\mathcal{P}}$ of the SUS $\mathcal{P}$. Our proposal is a proper conservative extension of this work: in case of full observability, no reset, if the system always satisfies the assumption, i.e. $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\hat{\mathcal{P}})$, our definition coincides with [109]. As $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\hat{\mathcal{P}})$ is a strong assumption there, if it is violated, the monitor output will be undefined, while we explicitly take that possibility into account. Partial observability is essential for extending traditional RV approaches such that assumptions are really needed to evaluate the property (not only for prediction). Due to the partial observability, ABRV-LTL monitors cannot be expressed in traditional RV approach (quantifiers over traces would be necessary). Under full observability, if the model $\hat{\mathcal{P}}$ is expressed in LTL, the monitor of [109] can be seen as an ABRV monitor for $\hat{\mathcal{P}} \wedge \varphi$ (without extra assumptions). Pinisetty et al. further extend this idea to support RV of timed properties [127], where *a priori knowledge* (also expressed in timed properties) about the behavior of the SUS allows for predictive monitors.

Similarly, Sistla et al. [141] construct monitors from a specification Φ w.r.t. another interface specification Φ_I which can be considered as a partial system model (as it comes from an off-the-shelf component). The resulting monitor is roughly equivalent to the ABRV monitor synthesized from $\Phi_I \rightarrow \Phi$ (without extra assumptions).

Three-valued LTL semantics tracks its roots back to Kleene [106] but was mostly applied to Model Checking (e.g. [30]) until LTL_3 is used in RV. In another three-valued predictive LTL semantics [150], the assumption is based on predictive words. Given a sequence u, a predictive word v of subsequent inputs is computed with static analysis of the monitored program and the monitor output evaluates $[\![u \cdot v \models \varphi]\!]_3$. The assumption used in our framework can be also used to predict the future inputs, but can associate to each u an infinite number of words. Thus our assumption-based RV framework is more general than [150], even without partial observability and resets. On the other side, while our assumptions can be violated by the system execution, the predictive word of [150] is assured by static analysis.

The research of partial observability in Discrete-Event Systems is usually connected with diagnosability [133] and predicability [79, 80]. The presence of system models plays a crucial role here, although technically speaking the support of partial observation is orthogonal with the use of system models (or assumptions) in the monitoring algorithm. Given a model of the system which includes faults (eventually leading the system to a failure) and which is partially-observable (observable only with a limited number of events or data variables), diagnosability studies the problem of checking if the faults can be detected within a finite amount of time. On the other hand, if we take an empty specification (true) and use the system model as assumptions,

then our monitors will be checking if the system implementation is always consistent with its model—the monitor only outputs $\top^a$ and $\times$ in this case. This is in spirit of Model-based Runtime Verification [8, 152], sometimes also combined with extra temporal specifications [145, 151].

The support of partial observability is found in the context of Model Checking by explicit support of partial knowledge in the description of (incomplete) models, involving concepts like Partial Kripke Structures, Model Transition Systems and Incomplete Büchi Automata (e.g. [20, 122]). In comparison with these work, the models (as assumptions) considered here are *complete*, while the support of partial observability in our work is mostly reflected by input traces (of monitors), which do not contain values of all the variables in the models.

Other work with partial observability appears in decentralized monitoring of distributed systems [17, 56], where an LTL formula describing the system's global behavior may be decomposed into a list (or tree) of sub-formulae according to the system components, whose local behaviors are fully observable.

To the best of our knowledge, the concept of resettable monitors was never published before our first RV papers in RV 2019 [47]. Note that, if assumptions and past operators are not considered, resetting monitors is not very meaningful (thus perhaps this is why resets were not considered by other researchers before.) However, there exists some prior work with similar keywords. For instance, in [140], the authors extend a runtime monitor for regular expressions with recovery. Their concept of monitor recovery is based on a specific pattern of the monitoring protocol and after an error, the monitor waits for a signal to restart the monitoring of the protocol. Comparing with our work, it is specific to the given pattern and considers neither past operators, nor the system model.

We note that it would be straightforward to extend the standard RV framework for LTL [19] with the capability of resetting monitors. However, the task becomes non-trivial when assumptions on partial-observable systems are taken into account. The challenge is to reset the monitor without losing the knowledge gathered during the previous observations (see also the intuition of the motivating example). We provide a practical solution to this novel problem.

Monitorability is an important topic in RV and other related field [2, 126, 142]. Under assumptions, some important results were discussed by T. Henzinger and N.E. Saraç in their RV 2020 paper [91].

2.4 SMT-Related RV Work

Despite the vast literature on SAT- and SMT-based symbolic model checking [22], currently there are only few works on applying SAT/SMT solvers to Runtime Verification. One of the prominent approaches in this direction is the one on Monitoring Modulo Theories (MMT) [59] for monitoring Temporal Data Logic (TDL): propositional LTL extended with first-order quantifiers

and theories. MMT is implemented on top of the Z3 SMT solver. The SMT solver in MMT is mainly to deal with first-order quantifiers of TDL. They observed that, without first-order quantifiers TDL can be monitored in the same way as propositional LTL by treating theory-specific atoms as atomic propositions (AP). The situation is the same here, but NuRV also supports assumptions. In general propositional LTL with infinite-state assumptions cannot be synthesized into finite-state monitors.

In [146], SMT solvers are used to monitor partially synchronous distributed systems. In this work, SMT solvers evaluate partially observable formulas that contain non-observable variables that can have any possible value. However, in this work the SMT formula is generated in highly domain-specific ways and is directly treated as the monitoring property, without temporal extensions.

The relationship between MC and RV has been explored in previous research. The value of models (as RV assumptions) in synthesizing better monitors was first reported in [109]. Adapting existing model checkers for RV purposes is a natural idea for reducing the costs of tool development from scratch. Similar with NuRV, the DIVINE model checker was adapted to perform runtime verification [100]. However, the model checker is used to check the property on a trace (considered as system model), not to consider a system model as an assumption as in ABRV.

We consider the predictive feature of ABRV monitors as a side effect of the assumption-based approach, but there exist dedicated work on predictive semantics of runtime monitors, e.g. [150].

Belief states have been used in planning under partial observability. See, e.g., the work of Hoffmann et al. [92], from which we borrow the idea of representing them with symbolic formulas. To the best of our knowledge, our approach is the first attempt to combine them with the evaluation of temporal properties for RV.

Chapter 3

Preliminaries

3.1 Finite and infinite words (or traces)

Let Σ be a finite alphabet, and $u, v, w, \ldots$ be finite or (countably) infinite words (i.e. sequences of letters) over Σ. (Thus we have $u \in \Sigma^*$ or $w \in \Sigma^\omega$, if u is finite and w is infinite.) Empty words are denoted by ϵ. Furthermore, u_i denotes the *zero-indexed* ith letter in u (hereafter $i \in \mathbb{N}$), while u^i denotes the *sub-word* of u starting from u_i. $|u|$ denotes the *length* of u. Finally, $u \cdot v$ is the *concatenation* of a finite word u with another finite or infinite word v. These notions are quite standard in Automata Theory. In RV scenario, however, infinite words are elements of languages of models (or assumptions) of SUS, while finite words are usually executions of systems. For this reason, we also call them finite or infinite *traces*.

Note also that, in RV of finite-state systems, Σ is nothing but a set of Boolean variables in which all variables used in the monitoring specification and assumptions must be included. For RV on infinite-state systems, with slight abuse of notions, we also use Σ to denote a first-order signature and the set of involved variables (whose types may be Boolean, finite- or infinite-domain) of Σ is denoted by V (see below). Hereafter, when talking about a set of possibly infinite-domain variables (in either LTL or FTS, see below), we mostly use V instead of Σ.

In other words, let V be a set of variables, a (partial) value assignment s is a function mapping each variable $x \in V$ to a value of its type, e.g. $\{x \mapsto 1, y \mapsto 2\}$. A finite trace is a finite sequence of such value assignments, $u = s_0 s_1 \ldots s_n$ (the subscript $0, 1, \ldots$ are indexes of the trace). An infinite trace is a countable infinite sequence of such value assignments. If any variable in V has infinite possible values, e.g. unbounded integers or real numbers, the corresponding trace is called *infinite-state*, either finite or infinite.

3.2 Satisfiability Modulo Theory

For RV on infinite-state systems, we work in the setting of Satisfiability Modulo Theory (SMT) [9] and LTL Modulo Theory (see, e.g., [49]). First-order formulas are built as usual by proposition logic connectives, a given set of variables V and a first-order signature Σ, and are interpreted according to a given Σ-theory $\mathcal{T}$. We assume to be given the definition of $M, s \models_{\mathcal{T}} \varphi$ where M is a Σ structure, s is a value assignment to the variables in V, and φ is a formula. Whenever M is clear from contexts we omit it and simply $s \models_{\mathcal{T}} \varphi$. With slight abuse of notations, we also use an assignment $s = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ to represent the corresponding formula, i.e. the conjunction $\bigwedge_i (x_i = v_i)$. We sometimes write $\phi(V)$ or $\phi(V_1, V_2)$ instead of ϕ to highlight that the free variables of formula ϕ belong to V or $V_1 \cup V_2$, respectively. Arbitrary first-order theories can be supported by our RV algorithm, as long as the underlying SMT solver and model checker support them. For illustrating purposes, we only consider $\mathcal{LRA}$, the theory of linear arithmetics with real numbers.

3.3 Linear Temporal Logic

Linear Temporal Logic (LTL) is adopted as the specification language for monitoring synthesis in this thesis. (In theory, the concepts of ABRV also work with other specification languages.) LTL is recalled here mostly because different authors adopt slightly different set of primitive operators, sometime also with different symbols.

Definition 3.3.1 (LTL Syntax [115])**.** *The set of Propositional Linear Temporal Logic (LTL) formulae is inductively defined by atomic propositions, Boolean combinations and temporal operators:*

$$\varphi ::= \text{true} \mid \alpha \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \mathbf{Y}\,\varphi \mid \varphi\,\mathbf{S}\,\varphi$$

Here the (quantifier-free) formula α as atomic propositions, is built by a set of variables V and a first-order signature Σ, and is interpreted according to a Σ-theory $\mathcal{T}$. (In the pure Boolean case, V consists of only Boolean variables.) The temporal operator $\mathbf{X}$ stands for *next*, $\mathbf{U}$ for *until*, $\mathbf{Y}$ for *previous*, and $\mathbf{S}$ for *since*. Other logical constants and operators such as false, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are used as syntactic sugar with their usual meanings in propositional logic. The following temporal operators are used as abbreviations: $\mathbf{F}\,\varphi \doteq \text{true}\,\mathbf{U}\,\varphi$ (*eventually*), $\mathbf{G}\,\varphi \doteq \neg\mathbf{F}\,\neg\varphi$ (*globally*), $\mathbf{O}\,\varphi \doteq \text{true}\ \mathbf{S}\ \varphi$ (*once*), $\mathbf{H}\,\varphi \doteq \neg\mathbf{O}\,\neg\varphi$ (*historically*). In addition, $\mathbf{X}^n\,p$ denotes a sequence of n nested next operators: $\mathbf{X}\mathbf{X}\cdots\mathbf{X}\,p$; Similar for $\mathbf{Y}^n\,p$. Sometimes we also need the *weak until* operator: $\varphi\,\mathbf{W}\,\psi \doteq (\mathbf{G}\,\varphi) \vee (\varphi\,\mathbf{U}\,\psi)$ or $\varphi\,\mathbf{U}\,(\psi \vee \mathbf{G}\,\varphi)$. Similarily, one can define *weak since*. [1]

[1] NuSMV supports a wide list of derived LTL temporal operators, including all operators mentioned here, more Boolean logic operators and bounded versions of **G**, **F**, etc. (See Section 2.4.3 (LTL Specifications) of [38] for more details.) The NuRV tool supports all these operators,

Definition 3.3.2 (Semantics of LTL over infinite words)**.** *Let $w = a_0 \ldots \in \Sigma^\omega$ denote an infinite trace. The truth value of LTL formulae φ w.r.t. w at position i, denoted with $[\![w, i \models \varphi]\!]$, is an element of $\mathbb{B} \doteq \{\top, \bot\}$ and is inductively defined as follows:*

$$
\begin{aligned}
&w, i \models \text{true} \\
&w, i \models p && \textit{iff } p \in w_i \\
&w, i \models \neg\varphi && \textit{iff } w, i \not\models \varphi \\
&w, i \models \varphi \vee \psi && \textit{iff } w, i \models \varphi \textit{ or } w, i \models \psi \\
&w, i \models \mathbf{X}\varphi && \textit{iff } w, i+1 \models \varphi \\
&w, i \models \varphi\,\mathbf{U}\,\psi && \textit{iff for some } k \geqslant i, w, k \models \psi \textit{ and for all } i \leqslant j < k, w, j \models \varphi \\
&w, i \models \mathbf{Y}\varphi && \textit{iff } i > 0 \textit{ and } w, i-1 \models \varphi \\
&w, i \models \varphi\,\mathbf{S}\,\psi && \textit{iff for some } k \leqslant i, w, k \models \psi \textit{ and for all } k < j \leqslant i, w, j \models \varphi
\end{aligned}
$$

We write $[\![w \models \varphi]\!]$ for $[\![w, 0 \models \varphi]\!]$ and $\mathcal{L}(\varphi) \doteq \{w \in \Sigma^\omega \mid [\![w \models \varphi]\!] = \top\}$ for the *language* (i.e. the set of traces) of φ. Two LTL formulae φ and ψ are *equivalent*, denoted by $\varphi \equiv \psi$, if $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$.

3.4 Boolean formulae and functions

Let $\mathbb{B} \doteq \{\top, \bot\}$ denote the type of Boolean values ($\top$ for *true*, $\bot$ for *false*). A set of *Boolean formulae* $\Psi(V)$ over a finite set of propositional variables $V \doteq \{v_1, \ldots, v_n\}$, is the set of all *well-formed formulae* (wff) [4] built from V, $\mathbb{B}$, logical connectives ($\neg$, $\wedge$, etc.) and parenthesises. As usually in symbolic model checking, a Boolean formula $\psi(V) \in \Psi(V)$ also denotes a set of truth assignments that makes $\psi(V)$ true. Following McMillan [119], a Boolean formula $\psi(V)$ can be considered as *same* thing (they have the same types when sets are regarded as predicates) as a function of type $\mathbb{B}^{|V|} \to \mathbb{B}$ taking a vector of Boolean values and returning another: $\lambda(v_1, \ldots, v_n).\ \psi(v_1, \ldots, v_n)$ or $\lambda V.\ \psi(V)$, assuming a fixed order of variables in V. Thus $\Psi(V)$ itself has the type $(\mathbb{B}^{|V|} \to \mathbb{B}) \to \mathbb{B}$. Whenever V is clear from contexts, we can omit the whole λ–prefix. Therefore, set-theoretic operations such as intersection and union are interchangeable with logical connectives on Boolean formulae. For example,

$$
\begin{aligned}
\varphi(V) \cap \psi(V) &\doteq (\lambda V.\ \varphi(V)) \cap (\lambda V.\ \psi(V)) \\
&= \{(v_1, \ldots, v_n) \mid \varphi(v_1, \ldots, v_n)\} \cap \{(v_1, \ldots, v_n) \mid \psi(v_1, \ldots, v_n)\} \\
&= \{(v_1, \ldots, v_n) \mid \varphi(v_1, \ldots, v_n) \wedge \psi(v_1, \ldots, v_n)\} \\
&= \lambda V.\ \varphi(V) \wedge \psi(V) \doteq \varphi(V) \wedge \psi(V).
\end{aligned}
$$

but from a theoretic point of view they are nothing but syntactic sugar. In some RV literature, LTL is extended with intervals but the boundaries can only be integers, NuRV naturally supports these extensions *without* extra coding efforts.

3.5 Binary Decision Diagrams

Binary decision diagrams (BDD) [32] provide a data structure for representing and manipulating Boolean functions in symbolic form. They have been especially effective as the algorithmic basis for symbolic model checkers. A binary decision diagram represents a Boolean function as a directed acyclic graph, corresponding to a compressed form of decision tree. Most commonly, an ordering constraint is imposed among the occurrences of decision variables in the graph, yielding ordered binary decision diagrams (OBDD). Representing all functions as OBDDs with a common variable ordering has the advantages that (1) there is a unique, reduced representation of any function (called *canonical form*) [31], (2) there is a simple algorithm to reduce any OBDD to the unique form for that function, and (3) there is an associated set of algorithms to implement a wide variety of operations on Boolean functions represented as OBDDs.

BDD representation of Boolean formulae and BDD operations do not directly appear in any algorithm presented in this thesis. In another words, BDDs are only implicitly used in the RV algorithm (for finite-state systems). However, some of the RV algorithms seriously rely on the canonical form of BDDs. For example, the explicit-state monitor synthesis algorithm can only terminate if each involved Boolean formula has an canonical form.

One notable BDD implementation is the CUDD (Colorado University Decision Diagram) package by Fabio Somenzi. (The NuRV tool implementation is derived from NuSMV which links with CUDD version 2.4.1.1.)

3.6 Fair Kripke Structure (Fair Transition System)

System models, assumptions and ω-automata in this thesis are described by a symbolic presentation of Kripke structures called *Fair Kripke Structure (FKS)* [101], also known as *Fair Transition System (FTS)* [116]:

Definition 3.6.1. *Let V be a set of Boolean variables, and $V' \doteq \{v' \mid v \in V\}$ be the set of* next state *variables (disjoint with V). A* Fair Kripke Structure *$K \doteq \langle V, \Theta, \rho, \mathcal{J} \rangle$ is given by the set of variables V, a set of initial states $\Theta(V) \in \Psi(V)$, a transition relation $\rho(V, V') \in \Psi(V \cup V')$, and a set of Boolean formulae $\mathcal{J} \doteq \{J_1(V), \ldots, J_k(V)\} \subseteq \Psi(V)$ called* justice requirements[2].

Given any $K \doteq \langle V, \Theta, \rho, \mathcal{J} \rangle$, a *state* $s(V)$ of K is an element in 2^V representing a full truth assignment over V, i.e., for every $v \in V$, $v \in s$ if and only if $s(v) = \top$. For example, if $V = \{p, q\}$, a state $\{p\}$ means $p = \top$ and $q = \bot$. Whenever V is clear from the context, we can omit V and write s instead of $s(V)$. The *transition relation* $\rho(V, V')$ relates a state $s \in 2^V$ to its successor $s' \in 2^{V'}$. We say that s' is a *successor* of s (and that s is a predecessor of s') if

[2]We omit compassion requirements as they are not used in our approach

$s(V) \cup s'(V') \models \rho(V, V')$. For instance, if $\rho(V, V') = (p \leftrightarrow q')$, $s'(V') = \{q'\}$ is a successor of $s(V) = \{p\}$, since $s(V) \cup s'(V') = \{p, q'\}$ and $\{p, q'\} \models (p \leftrightarrow q')$.

A *path* in K is an infinite sequence of states $s_0, s_1, \ldots$ where $s_0(V) \models \Theta$ and, for all $i \in \mathbb{N}$, $s_i(V) \cup s_{i+1}(V') \models \rho(V, V')$. A state s is *reachable* if there exists a path $s_0, s_1, \ldots, s_k$ such that $s = s_k$.

Definition 3.6.2 (forward image)**.** *The* forward image *of a set of states $\psi(V)$ on $\rho(V, V')$ is a formula*

$$\mathrm{fwd}(\psi(V), \rho(V, V'))(V) \doteq (\exists V.\ \rho(V, V') \wedge \psi(V))[V/V'] \tag{3.1}$$

where $[V/V']$ denotes the substitution of (free) variables in V' with the corresponding one in V.

For finite-state FTS, the forward image computation can be effectively done by the existential abstraction function provided by BDD library (in CUDD, it is `Cudd_bddExistAbstract`). For infinite-state FTS, The existential quantifiers in forward images can be eliminated by QE procedures (see Section 3.9 below). We assume that all $\mathrm{fwd}(\cdot, \cdot)$ used in other definitions and theorems in this thesis are quantifier-free formulae.

An infinite word $s_0 s_1 \ldots \in \Sigma^\omega$ is a path of K iff for *all* i we have $s_i \cup s'_{i+1} \models \rho$; it is a *fair path*, in addition, if for all $J \in \mathcal{J}$ and *infinitely many* i we have $s_i \models J$. We denote by $\mathrm{FP}^\rho_{\mathcal{J}}(\psi)$ the *set of fair paths* starting from ψ (such that $s_0 \models \psi$). The *language* $\mathcal{L}(K)$ is the set of fair paths, i.e. $\mathrm{FP}^\rho_{\mathcal{J}}(\Theta)$. $L(K)$ is the set of finite prefixes of paths in $\mathcal{L}(K)$. The *empty* FKS $\langle V, \mathrm{true}, \mathrm{true}, \emptyset\rangle$ is overloaded on $\emptyset$ for any given V. Then it is not hard to see that $\mathcal{L}(\emptyset) = (2^V)^\omega$ and $L(\emptyset) = (2^V)^*$, i.e. the empty FKS contains all possible paths.

A state s is *fair* if it occurs in a fair path. The *set of all fair states* of K, denoted by $\mathcal{F}_K$, can be computed by standard algorithms like Emerson-Lei [63]. Finally, let $K_1 = \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1\rangle$ and $K_2 = \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2\rangle$, the *synchronous product* of K_1 and K_2 is defined as $K_1 \otimes K_2 \doteq \langle V_1 \cup V_2, \Theta_1 \wedge \Theta_2, \rho_1 \wedge \rho_2, \mathcal{J}_1 \cup \mathcal{J}_2\rangle$. Roughly speaking, an execution of $K_1 \otimes K_2$ can be viewed as a *joint execution* of K_1 and K_2 [101].

3.7 LTL to ω-automata Translation

Our RV work heavily relies on an existing symbolic translation algorithm from LTL to ω-automata. The algorithm traces its root back to [34, 52] where only future temporal operators are supported, with additional support of past temporal operators [75]. Essentially it constructs a tableau FKS T_φ from any LTL formula φ. (See [136, 137] for its correctness and [149] for a formal verification in HOL theorem prover, see also Chapter 10 for this topic.)

A set of propositional *elementary variables* of φ, denoted by $\mathrm{el}(\varphi)$, is used for converting any LTL formula into an equivalent propositional formula. For any LTL formula $\varphi \in \mathrm{LTL}(AP)$, $\mathrm{el}(\varphi)$ can be defined recursively as follows (where $p \in AP$, ϕ and ψ are subformulae of φ):

$$\begin{aligned}
\mathrm{el}(\mathrm{true}) &= \emptyset, & \mathrm{el}(\mathbf{X}\phi) &= \{\mathrm{x}_\phi\} \cup \mathrm{el}(\phi),\\
\mathrm{el}(p) &= \{p\}, & \mathrm{el}(\phi \mathbf{U} \psi) &= \{\mathrm{x}_{\phi \mathbf{U} \psi}\} \cup \mathrm{el}(\phi) \cup \mathrm{el}(\psi),\\
\mathrm{el}(\neg\phi) &= \mathrm{el}(\phi), & \mathrm{el}(\mathbf{Y}\phi) &= \{\mathrm{y}_\phi\} \cup \mathrm{el}(\phi),\\
\mathrm{el}(\phi \vee \psi) &= \mathrm{el}(\phi) \cup \mathrm{el}(\psi), & \mathrm{el}(\phi \mathbf{S} \psi) &= \{\mathrm{y}_{\phi \mathbf{S} \psi}\} \cup \mathrm{el}(\phi) \cup \mathrm{el}(\psi).
\end{aligned}$$

By induction on temporal formula structure, it is not hard to see that, for any LTL formula φ, $\mathrm{el}(\varphi) = \mathrm{el}(\neg\varphi)$. And also φ can be rewritten into a Boolean formula $\chi(\varphi)$ using only variables in $\mathrm{el}(\varphi)$. Below is the full definition[3] of $\chi(\cdot)$:

$$\chi(\varphi) = \begin{cases}
\varphi & \text{for } \varphi \text{ an elementary variable in } \mathrm{el}(\cdot),\\
\neg\chi(\phi) & \text{for } \varphi = \neg\phi,\\
\chi(\phi) \vee \chi(\psi) & \text{for } \varphi = \phi \vee \psi,\\
\mathrm{x}_{\phi \mathbf{U} \psi} & \text{for } \varphi \text{ in forms of } \mathbf{X}(\phi \mathbf{U} \psi),\\
\mathrm{x}_\phi & \text{for } \varphi \text{ in forms of } \mathbf{X}\phi \text{ (except for } \mathbf{X}(\phi \mathbf{U} \psi)),\\
\mathrm{y}_{\phi \mathbf{S} \psi} & \text{for } \varphi \text{ in forms of } \mathbf{Y}(\phi \mathbf{S} \psi),\\
\mathrm{y}_\phi & \text{for } \varphi \text{ in forms of } \mathbf{Y}\phi \text{ (except for } \mathbf{Y}(\phi \mathbf{S} \psi)).
\end{cases} \tag{3.2}$$

Note that, to apply (3.2), all sub-formulas of φ leading by $\mathbf{U}$ and $\mathbf{S}$ must be wrapped within $\mathbf{X}$ and $\mathbf{Y}$, respectively. This can be done (if needed) by using the following *Expansion Laws* of LTL[4]

$$\psi \mathbf{U} \phi \equiv \phi \vee (\psi \wedge \mathbf{X}(\psi \mathbf{U} \phi)), \qquad \psi \mathbf{S} \phi \equiv \phi \vee (\psi \wedge \mathbf{Y}(\psi \mathbf{S} \phi)). \tag{3.3}$$

For instance, $\chi(p \mathbf{U} q) = q \vee (p \wedge \mathrm{x}_{p \mathbf{U} q})$, and thus $\chi'(p \mathbf{U} q) = q' \vee (p' \wedge \mathrm{x}'_{p \mathbf{U} q})$.

The FKS translated from φ is given by $T_\varphi \doteq \langle V_\varphi, \Theta_\varphi, \rho_\varphi, \mathcal{J}_\varphi \rangle$, where $V_\varphi \doteq \mathrm{el}(\varphi)$. The initial condition Θ_φ is given by

$$\Theta_\varphi \doteq \chi(\varphi) \wedge \bigwedge_{\mathbf{Y}_\psi \in \mathrm{el}(\varphi)} \neg \mathrm{y}_\psi. \tag{3.4}$$

Here each $\mathrm{y}_\psi \in \mathrm{el}(\varphi)$ has an initial false assignment in Θ_φ. This is essentially a consequence of LTL semantics for the $\mathbf{Y}$ operator, i.e. for any word w and LTL formula ψ, $w, 0 \not\models \mathbf{Y}\psi$.

The transition relation ρ_φ (as a formula of variables in $\mathrm{el}(\varphi) \cup \mathrm{el}'(\varphi)$) is given by

$$\rho_\varphi \doteq \bigwedge_{\mathbf{X}_\psi \in \mathrm{el}(\varphi)} \left(\mathrm{x}_\psi \leftrightarrow \chi'(\psi)\right) \wedge \bigwedge_{\mathbf{Y}_\psi \in \mathrm{el}(\varphi)} \left(\chi(\psi) \leftrightarrow \mathrm{y}'_\psi\right). \tag{3.5}$$

[3] Strictly speaking, logical constants such as $\neg$ and $\wedge$ in LTL syntax are not the same constants as those in propositional logic. $\chi(\cdot)$ will translate these constants to their counterparts in propositional logic.

[4] There is a subtle but important matter here: the expansion laws, also called *unwinding laws*, holds also for LTL_3 (and also ABRV-LTL): the set of finite traces satisfying (or violating) any LTL formula coincides with the set of finite traces satisfying the same LTL formula after applying the expansion laws. Thus applying the expansion laws as the first step of ABRV monitor synthesis on the input LTL property does not change the monitoring outputs. The same property, however, may not hold for other LTL semantics over finite traces, e.g. FLTL. See [18] for more details and a comparison on several such LTL semantics for RV purposes.

Intuitively, the purpose of ρ_φ is to relate the values of elementary variables to their corresponding temporal variables: for any $\psi \in \mathrm{el}(\varphi)$, the current value of ψ is *memorized* by the value of Y_ψ in next state; and the next value of ψ is *guessed* by the current value of X_ψ.

The justice set $\mathcal{J}_\varphi$ is given by

$$\mathcal{J}_\varphi \doteq \left\{ \chi(\psi \,\mathbf{U}\, \phi) \rightarrow \chi(\phi) \;\middle|\; \mathrm{X}_{\psi \mathbf{U} \phi} \in \mathrm{el}(\varphi) \right\}. \tag{3.6}$$

It guarantees that, whenever a sub-formula $\psi \,\mathbf{U}\, \phi$ is satisfied, ϕ is also satisfied. (Thus an infinite sequence of q cannot be accepted by the FKS translated from $p \,\mathbf{U}\, q$.)

Notice that T_φ and $T_{\neg\varphi}$ only differ at their initial conditions Θ_φ and $\Theta_{\neg\varphi}$. In practice, if one needs both T^φ and $T^{\neg\varphi}$, it is possible to store the two parts of Θ_φ separately and get $\Theta_{\neg\varphi}$ by an inverse of $\chi(\varphi)$.

Remark 3.7.1. *The ABRV approach represented in this thesis, together with the actual monitoring algorithms for finite- and infinite-state systems, can be applied to any LTL properties and assumptions. (It is the responsibility of users to choose appropriate assumptions, however, to obtain useful monitors in which the properties are monitorable.) The ability of monitoring any LTL property is because ABRV leverages existing symbolic translations from LTL to ω-automata (though with some inspections on the translation internals for the realization of resets.) In theory, any other specification language that can be translated into ω-automata, can be also used in the existing ABRV framework, only changing the LTL translation calls to the procedures of the other specification language.* [5]

3.8 Explicit-State Automata

Various kinds of explicit-state automata are intermediate outputs of explicit-state monitoring synthesis algorithms to be presented in Chapter 5. However, we do not use DFA, for example, for actually accepting any word. Instead, all kinds of explicit-state automata are only intermediate results for building the final monitor finite-state machine (FSM), which is essentially a transducer tranducing monitoring inputs to outputs. Furthermore, we need two output functions λ and λ' instead of one in the standard setting. Below is the formal definition of all automata used in this thesis.

Definition 3.8.1 (Büchi Automata)**.** *A non-deterministic Büchi automaton (NBA) [33] is a tuple $\mathcal{A} = \langle \Sigma, Q, Q_0, \delta, F \rangle$, where*

- *Σ is a finite alphabet,*
- *Q is a finite nonempty set of states,*

[5] Not to mention, LTL variants that can be first translated to standard LTL, can be easily supported too. For instance, metric temporal operators with integer-valued bounds.

- $Q_0 \subseteq Q$ *is a set of initial states,*
- $\delta\colon Q \times \Sigma \to 2^Q$ *is a transition function, and*
- $F \subseteq Q$ *is a set of accepting states.*

We extend the transition function $\delta\colon Q \times \Sigma \to 2^Q$, as usual, to $\delta^*\colon 2^Q \times \Sigma^* \to 2^Q$ by

$$\delta^*(Q', \varepsilon) = Q' \quad \text{and} \quad \delta^*(Q', u\,a) = \bigcup_{q' \in \delta^*(Q', u)} \delta(q', a)$$

for $Q' \subseteq Q, u \in \Sigma^*$, and $a \in \Sigma$. To simplify the notation, we use δ for both δ and δ^*.

A *run* of an automaton $\mathcal{A}$ on an infinite word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a sequence of states $\rho = q_0, q_1, q_2, \ldots$ such that $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, a_i)$ for all $i \in \mathbb{N}$. For a run ρ, let $\mathrm{Inf}(\rho)$ denote the states visited infinitely often. A run ρ of an NBA $\mathcal{A}$ is called *accepting* iff $\mathrm{Inf}(\rho) \cap F \neq \emptyset$.

Definition 3.8.2 (Nondeterministic Finite-State Automata)**.** *A non-deterministic* finite-state automaton *(NFA)* $A \doteq \langle \Sigma, Q, Q_0, \delta, F \rangle$ *is an automaton where* Σ*,* Q*,* Q_0*,* δ *and* F *are defined as for NBA but which operates on finite words. A* run *of NFA on a finite word* $u = a_0 \ldots a_n \in \Sigma^*$ *is a sequence of states* $\rho = q_0, q_1, \ldots, q_{n+1}$*, where* $q_0 \in Q_0$ *and* $q_{i+1} \in \delta(q_i, a_i)$ *for all* $i = 0, 1, \ldots, n$*. The run is called* accepting *if* $q_{n+1} \in F$*.*

Definition 3.8.3 (Deterministic Finite Automata)**.** *An NFA is called* deterministic *iff for all* $q \in Q, a \in \Sigma, |\delta(q, a)| = 1$ *and* $|Q_0| = 1$*. We use DFA to denote a deterministic finite automaton.*

Definition 3.8.4. *A Finite-State Machine (FSM)* $M \doteq \langle \Sigma, Q, Q_0, \delta, \Delta, \lambda, \Delta', \lambda', \ldots \rangle$ *is a finite state automaton enriched with output, where* Σ, Q, Q_0*, and* δ *are defined as before and where* Δ *is the output alphabet used in the output function* $\lambda\colon Q \to \Delta$*. The (main) output of an FSM, defined by the function* λ*, is thus determined by the current state* $q \in Q$ *alone, rather than by input symbols. For a deterministic FSM, we denote with* λ *also the function that yields for a given word* u *the output in the state reached by* u *rather than the sequence of outputs. Extra outputs are provided by* $\lambda', \ldots$ *etc. with different output alphabets* $\Delta', \ldots$*. We also denote* $M(u) ::= \lambda(u)$ *when* λ *is the output function of* M*.*

Definition 3.8.5. *A Finite-State Transducer (FST)* $\mathcal{F} \doteq (\Sigma, Q, Q_0, \delta, \Delta, \rho)$*, is a finite state automaton enriched with outputs, where* Σ, Q, Q_0, δ*, and* Δ *are defined as before. The output of an FST, defined by the function* $\rho\colon Q \times \Sigma \to \Delta$*, is determined by the current state* $q \in Q$ *and input symbol. As before,* ρ *extends to the domain of words as expected. For a deterministic FST, we denote with* ρ *also the function that yields for a given word* u *the sequence of outputs.*

3.9 First-Order Quantifier Elimination

First-Order Quantifier Elimination [117] methods, which convert formulas into $\mathcal{T}$-equivalent quantifier-free formulas, are parts of many SMT solvers (e.g., Z3, Yices and MathSAT) for checking the satisfiability of quantified formulas. (Hereafter we will omit the words "first-order" and only call it "quantifier elimination" or QE. cf. *Second-Order Quantifier Elimination (SOQE)* [77].)

As an notable example of QE, from high school mathematics there is a theorem saying that a univariate quadratic polynomial has a real root if and only if its discriminant is non-negative. This can be expressed as a quantifier elimination problem:

$$\exists x \in \mathbb{R}.\ (a \neq 0 \wedge ax^2 + bx + c = 0) \Longleftrightarrow a \neq 0 \wedge b^2 - 4ac \geqslant 0$$

where the formula on the left-hand side involves a quantifier $\exists x \in \mathbb{R}$, and the equivalent formula on the right does not.

Formally speaking, if $\alpha(V_1 \cup V_2)$ is quantifier-free formula (of the theory $\mathcal{T}$) built by variables from the set $V_1 \cup V_2$, the role of quantifier elimination is to convert the first-order formula $\exists V_1.\alpha(V_1 \cup V_2)$ into an $\mathcal{T}$-equivalent formula $\beta(V_2)$, where β is quantifier-free and is built by only variables from V_2.

Quantifier elimination is not possible for arbitrary theories. For the theory domain $\mathcal{LRA}$ (linear arithmetics of real numbers), most SMT solvers implement at least one of the following procedures: Fourier-Motzkin [102], Ferrante-and-Rackoff [68] and Loos-and-Weispfenning [113]. Note also that QE procedures do not guarantee any kind of boundedness of the resulting formulas.

3.10 LTL Model Checking

Roughly speaking, the core task of LTL Model Checking[6] [32] is to evaluate $[\![K \models \varphi]\!]$ (LTL semantics over infinite traces), where K is the model of a system given as a transition system, e.g. as a (Fair) Kripke Structure, while φ is a specification, which is usually a plain logic formula (invariant checking) or temporal logic formula [53]. If $[\![K \models \varphi]\!] = \top$, then for all (fair) paths in the model K, the specification φ holds. Otherwise, there must exist at least one (fair) path in K, which is called a *counter-example*, on which φ does not hold. A model checking algorithm is required to effectively detect the case when $[\![K \models \varphi]\!] = \top$ (by simply indicating this fact), and actually constructs and returns the counter-example when $[\![K \models \varphi]\!] = \bot$

From software engineering point of view, the infrastructure provided by an LTL model checker provides good working basis for creating an LTL runtime monitor. For example, NuRV is built

[6] In this thesis we do not consider CTL-based model checking and other MC techniques based on branching-time logics, despite branching time logics have some positions in RV research. Whenever MC is referred, it always means LTL model checking.

by directly extending an existing LTL model checker with core RV algorithms. Algorithmically speaking, however, Model checking is involved in two aspects in this thesis:

1. There exists bidirectional reductions between ABRV and MC, and thus we show that ABRV and MC have the same space and time complexities.

2. In RV of infinite-state systems, the RV algorithm is based on first-order quantifier elimination and LTL model checking.

For the purpose of infinite-state monitoring, we use the state-of-art LTL model checker nuXmv (version: 2.0 and later) [27, 37] for infinite-state runtime monitoring.

More details of these aspects and the uses of MC will be given in Chapter 6, especially in Section 6.5 (Incremental Bounded Model Checking).

Chapter 4

Runtime Verification

The RV approach presented in this thesis is based on an LTL semantics over finite traces called LTL_3. Finite traces can be also seen as truncated (infinite) traces.

As LTL_3 is not the only LTL semantics over finite traces, it is imporant to discuss, before other things, why our RV approach is based on LTL_3. In Section 4.1, we give an overview and comparison of existing LTL semantics over finite traces [18] with some further discussions. The purpose is to try to convince the audience that LTL_3 is a reasonable choice to be based on. In the rest sections, Runtime Verification based on LTL_3, with and without assumptions, are introduced formally. Finally, we give the ABRV problem setting, to be resolved by algorithms given in subsequent chapters.

4.1 LTL semantics for Runtime Verification

Following [18], we present and compare five LTL semantics over finite traces: FLTL, LTL^+, LTL^-, LTL_3 and RV-LTL. The key difference among various such semantics is the truth value of $\mathbf{X}\,\varphi$ w.r.t. a trace u having only one state (i.e. $|u| = 1$).

4.1.1 FLTL

FLTL traces its root back to [116], where Manna and Pnueli suggest to enrich the standard framework by adding a dual operator, the *weak next* $\tilde{\mathbf{X}}$, which allows to smoothly translate formulae into negation normal form. In FLTL, the truth value of $\mathbf{X}\,\varphi$ w.r.t. a trace u having only one state (i.e. $|u| = 1$) is forcely defined as $\bot$, while the dual formula $\tilde{\mathbf{X}}\,\varphi$ is forcely defined as $\top$. See Def. 4.1.1 for more details.

Definition 4.1.1 (Semantics of FLTL [116])**.** *Let $u = a_0 \dots a_{n-1} \in \Sigma^*$ denote a finite trace of length $n = |u|$, with $u \neq \epsilon$. The truth value of an FLTL formula φ w.r.t. u, denoted with $[\![u \models \varphi]\!]_{\text{F}}$, is an element of $\mathbb{B}_2$ and is inductively defined as follows: Boolean constants, Boolean*

(weak) next

$$[\![u \models \mathbf{X}\varphi]\!]_{\mathrm{F}} = \begin{cases} [\![u^1 \models \varphi]\!]_{\mathrm{F}} & \text{if } |u| > 1, \\ \bot & \text{otherwise,} \end{cases} \qquad [\![u \models \tilde{\mathbf{X}}\varphi]\!]_{\mathrm{F}} = \begin{cases} [\![u^1 \models \varphi]\!]_{\mathrm{F}} & \text{if } |u| > 1, \\ \top & \text{otherwise.} \end{cases}$$

until/release

$$[\![u \models \varphi \mathbf{U} \psi]\!]_{\mathrm{F}} = \begin{cases} \top & \text{there exists } k \in \{0 \ldots n-1\} \colon [\![u^k \models \psi]\!]_{\mathrm{F}} = \top \text{ and} \\ & \text{for all } l \text{ with } 0 \leqslant l < k \colon [\![u^l \models \varphi]\!]_{\mathrm{F}} = \top, \\ \bot & \text{otherwise,} \end{cases}$$

$$[\![u \models \varphi \mathbf{V} \psi]\!]_{\mathrm{F}} = \begin{cases} \top & \text{for all } k \in \{0 \ldots n-1\} \colon [\![u^k \models \psi]\!]_{\mathrm{F}} = \top \text{ or} \\ & [\text{there exists } k \in \{0 \ldots n-1\} \colon [\![u^k \models \varphi]\!]_{\mathrm{F}} = \top \text{ and} \\ & \text{for all } l \text{ with } 0 \leqslant l \leqslant k \colon [\![u^l \models \psi]\!]_{\mathrm{F}} = \top], \\ \bot & \text{otherwise.} \end{cases}$$

Figure 4.1: Semantics of FLTL formulae over a trace $u = a_0 \ldots a_{n-1} \in \Sigma^*$

combinations and atomic propositions are defined as in standard LTL. Until/release and (weak) next are defined as shown in Fig. 4.1 [1].

4.1.2 $\mathrm{LTL}^{\mp}$

Note that the semantics of FLTL over empty traces is not specified. An alternative approach is to maintain the standard semantics of **X** and define truth values for atomic propositions over empty traces. Obviously, there are two variants of such semantics, originally proposed by Eisner et al. [62]. In LTL^{+}, the truth value of p w.r.t. a_0 ($p \in a_0$) is $\bot$, while in LTL^{-} the truth value of p is $\top$. See Def. 4.1.2 for more details.

Definition 4.1.2 (Semantics of $\mathrm{LTL}^{\mp}$ [62])**.** *Let $u = a_0 \ldots a_{n-1} \in \Sigma^*$ denote a finite trace of length $n = |u|$. The truth value of an $\mathrm{LTL}^{\mp}$ formula φ w.r.t. u, denoted with $[u \models \varphi]_{\mp}$, is an element of $\mathbb{B}_2$ and is inductively defined as follows: Boolean constants, Boolean combinations, and atomic propositions are defined as shown in Fig. 4.2, while the semantics for the remaining formulae is as in standard LTL.*

4.1.3 LTL_3

One major problem of FLTL and $\mathrm{LTL}^{\mp}$ is that the semantics is defined regardless of the future continution (or extension) of the current finite trace. LTL_3, on the other hand, follows the idea

[1] In Fig. 4.1, the operator **V** is the *weak until* operator, which is **R** or **W** in some literals. Here we follow the symbols adopted by NuSMV.

$$[\![u \models p]\!]_- = \begin{cases} \top & \text{if } u = \epsilon \text{ or } p \in a_0, \\ \bot & \text{otherwise.} \end{cases} \qquad [\![u \models p]\!]_+ = \begin{cases} \top & \text{if } u \neq \epsilon \text{ or } p \in a_0, \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![u \models \neg p]\!]_- = \begin{cases} \top & \text{if } u = \epsilon \text{ or } p \notin a_0, \\ \bot & \text{otherwise.} \end{cases} \qquad [\![u \models \neg p]\!]_+ = \begin{cases} \top & \text{if } u \neq \epsilon \text{ or } p \notin a_0, \\ \bot & \text{otherwise.} \end{cases}$$

Figure 4.2: Semantics of $\text{LTL}^{\mp}$ formulae over a trace $u = a_0 \dots a_{n-1} \in \Sigma^*$

that a finite trace is a prefix of a so-far unknown infinite trace (which fits very well with the RV scenario). In another word, the semantics of LTL_3 for atomic propositions and each temporal operator are fully given by the standard LTL semantics (over infinite traces), which is always convincing. See Def. 4.1.3 for more details.

Definition 4.1.3 (Semantics of LTL_3 [6])**.** *Let $u = a_0 \dots a_{n-1} \in \Sigma^*$ denote a finite trace of length $n = |u|$. The* truth value *of a LTL_3 formula φ w.r.t. u at position i (or trace element a_i), denoted with $[\![u, i \models \varphi]\!]_3$, is an element of $\mathbb{B}_3 \doteq \{\top, \bot, ?\}$ and defined as follows:*

$$[\![u, i \models \varphi]\!]_3 = \begin{cases} \top & \text{if } \forall w \in \Sigma^\omega \colon [\![u \cdot w, i \models \varphi]\!] = \top, \\ \bot & \text{if } \forall w \in \Sigma^\omega \colon [\![u \cdot w, i \models \varphi]\!] = \bot, \\ ? & \text{otherwise.} \end{cases} \tag{4.1}$$

Furthermore[2]*, $[\![u \models \varphi]\!]_3$ denotes $[\![u, 0 \models \varphi]\!]_3$.*

4.1.4 RV-LTL

One major drawback of LTL_3 is that many practical LTL properties are non-monitorable, i.e. the truth value of these LTL formulae are always inconclusive (?) regardless of the trace, as mentioned in Section 1.1. To overcome this difficulty, Bauer et al. propose RV-LTL [18], which refines the inconclusive verdict ? into two variants $\top^{\text{p}}$ and $\bot^{\text{p}}$ based on both LTL_3 and FLTL. See Def. 4.1.4 for more details.

Definition 4.1.4 (Semantics of RV-LTL [18])**.** *Let $u = a_0 \dots a_{n-1} \in \Sigma^*$ denote a finite and non-empty trace of length $n = |u|$. The truth value of an RV-LTL formula φ w.r.t. u, denoted with $[\![u \models \varphi]\!]_{\text{RV}}$, is an element of $\mathbb{B}_4 \doteq \{\top, \bot, \top^{\text{p}}, \bot^{\text{p}}\}$ and is defined as follows:*

$$[\![u \models \varphi]\!]_{\text{RV}} = \begin{cases} \top & \text{if } [\![u \models \varphi]\!]_3 = \top, \\ \bot & \text{if } [\![u \models \varphi]\!]_3 = \bot, \\ \top^{\text{p}} & \text{if } [\![u \models \varphi]\!]_3 = ? \text{ and } [\![u \models \varphi]\!]_{\text{F}} = \top, \\ \bot^{\text{p}} & \text{if } [\![u \models \varphi]\!]_3 = ? \text{ and } [\![u \models \varphi]\!]_{\text{F}} = \bot. \end{cases}$$

[2]It is trivial to also modify Def. 4.1.1, 4.1.2 and 4.1.4 to support evaluation of LTL formulae on trace positions other than 0.

We do not recommend RV-LTL, because its division of LTL_3's inconclusive verdict (?) into $\top^{\text{p}}$ (*presumably true*) and $\bot^{\text{p}}$ (*presumably false*) is quite arbitrary and even counter-intuitive sometimes.

To see the "arbitrary" point, note that FLTL distinguishes between strong and weak next operators, but in practice it is hard for the end user to decide which one to use, because in RV scenario a finite trace is nothing but a truncated version of the potential infinite trace coming from the execution of the SUS.

To see the "distinguishes" point, let u be such a trace and φ be a monitor property such that $[\![u \models \varphi]\!]_{\text{RV}} = \top^{\text{p}}$. By Definition 4.1.4 we know that $[\![u \models \varphi]\!]_3 = ?$ for the same u and φ. By Definition 4.1.3, this means that there still exists different continutions of u such that the infinite trace w extending u may satisfy either $w \models \varphi$ or $w \not\models \varphi$. Let us take the second case ($w \not\models \varphi$) for the actual future inputs from the SUS, and assume that the SUS eventually arrives at another finite trace u' (extended from u) such that $[\![u' \models \varphi]\!]_3 = \bot$ while $[\![u \models \varphi]\!]_{\text{RV}} = \top^{\text{p}}$, i.e. *presumably true but actually conclusively false*. There is no evidence that such an unfortunate case has only small probabilities in practice.

4.1.5 Four Maxims of RV-LTL Semantics

The inventors of RV-LTL, on the other hand, consider the following four maxims that they consider essential for LTL semantics aimed for runtime verification (we agree on this part):

MAXIM 1 *Existential next* requires the inclusion of a strong next operator.

MAXIM 2 *Complementation by negation* requires that a negated formula evaluates to the complemented and different truth value.

MAXIM 3 *Impartiality* requires that a finite trace is not evaluated to $\top$ ($\bot$) if there still exists an infinite continuation leading to another verdict.

MAXIM 4 *Anticipation* requires that once every infinite continuation of a finite trace leads to the same verdict, then the finite trace evaluates to this very same verdict.

In Table 4.1 (cf. Fig. 5 of [18] for more details), it is shown that RV-LTL satisfies all above four maxims, while LTL_3 does not satisfies MAXIM 2 (Complementation by Negation). However, RV-LTL has two serious drawbacks:

1. RV-LTL is not defined on empty traces;
2. RV-LTL is not *LTL compliant* (A linear temporal logic $\mathcal{L}$ is LTL compliant iff $\varphi \equiv \psi$ implies $\varphi \equiv_{\mathcal{L}} \psi$).

For the applications and extensions to be presented in this thesis, both of them are fetal:

	FLTL	$LTL^{\mp}$	LTL_3	RV-LTL
Domain	$u \neq \emptyset, u \in \Sigma^*$	Σ^*	Σ^*	$u \neq \emptyset, u \in \Sigma^*$
Existential Next (MAXIM 1)	yes	yes (+) / no (-)	yes	yes
Complementation by Negation (MAXIM 2)	yes	no	no	yes
Impartiality (MAXIM 3)	no	no	yes	yes
Anticipation (MAXIM 4)	no	no	yes	yes
Boolean laws	yes	no	yes	yes
Equivalences	yes	yes	yes	yes
LTL compliant	no	no	yes	no
Negation Normal Form	yes	yes	yes	yes
Inductive definition	yes	yes	no	no

Table 4.1: Comparing LTL Semantics for Runtime Verification

1. The bidirectional reductions between ABRV and MC require LTL semantics over empty traces. (Section 4.6.2 and 4.6.3)

2. LTL compliance is needed for the initial rewriting of LTL formulae based on Expanding Law during LTL-to-Büchi translations. (Section 3.7)

On the other hand, we think that MAXIM 2 is too strict, as it requires that the negation of any verdict must be *different* with the verdict. For LTL_3, the negation of inconclusive (?) can be naturally considered as itself, i.e. $\overline{?} = ?$. (Think, for example, that the negation of zero is still zero: $XS - 0 = 0$.) Thus if we slightly modify MAXIM 2 to the following revised version, then even LTL_3 can satisfy:

MAXIM 2' *Complementation by negation (revised)* requires that a negated formula evaluates to the complemented and *negated* truth value (In particular, $\overline{?} = ?$).

Finally, the fact that LTL_3 (and also RV-LTL) cannot be inductively defined has no impact for automata-based RV algorithms where the present thesis finds its place. In rewriting-based RV techniques [130], the verdict of LTL formulae is calculated by its sub-formulae, and in this case neither LTL_3 nor RV-LTL is a good choice of LTL semantics in rewriting-based RV techniques.

The conclusion here is that LTL_3 is the perfect choice of an LTL semantics over finite traces. For previous mentioned issue that most LTL properties are non-monitorable, our solution is to use assumptions to further narrow down the possible continuations of trace prefixes.

4.2 Runtime Verification Based on LTL_3

The (traditional) LTL_3-based RV approach is depicted on the left of Fig. 1.1. The SUS exposes to the monitor a run over a set of (observable) variables. We assume that the connection between

the SUS and the monitor is *synchronous*, i.e. at each monitoring step some information about the SUS is conveyed to the monitor, typically with fixed sampling rates. (There are also practical situations where the information flows from the SUS to the monitor in *asynchronous* ways, e.g. in event-driven settings, but we consider them out of the scope of this thesis. We only remark that the asynchronous view can be obtained by extending the synchronous one [25].) The monitor synthesis process takes as input a specification $\phi(O)$, and returns the corresponding monitor. At each cycle, the SUS conveys to the monitor a total truth assignment, assigning Boolean values to each observable variable in O (the set of observables). Whenever the monitor takes as input a new letter of the run, which amounts to having a series of prefixes of increased length, it returns $\top$ (or $\bot$, resp.) if the observed prefix u is such that, for all suffixes v, $u \cdot v \models \phi$ (or $u \cdot v \models \neg\phi$, resp.), or ? otherwise. The monitor is clearly *monotonic*: once the output verdict is conclusive ($\top$ or $\bot$), future outputs remain the same.

Definition 4.2.1 (LTL_3 monitor [19])**.** *Given an LTL property $\varphi(O)$. The goal of LTL_3-based RV is to construct a monitor function $\mathcal{M}^\varphi \colon \Sigma^* \to \mathbb{B}_3$ such that for all $u \in \Sigma^*$, $\mathcal{M}^\varphi(u) = [\![u \models \varphi]\!]_3$.*

4.3 Assumptions and Partial Observability

Let $\varphi \in \mathrm{LTL}(AP)$ be a monitor specification, $K \doteq \langle V_K, \Theta_K, \rho_K, \mathcal{J}_K \rangle$ be an FTS representing the assumptions under which φ is monitored. K can be either a detailed model of the SUS or just a simple constraint over the variables in AP. In general, we do not have any specific restrictions on the overlapping of AP and V_K; although it is quite common that $AP \subseteq V_K$. Note that V_K can be empty if there is no assumption at all. Let $V \doteq V_K \cup AP$ be the set of all involved Boolean variables.

We say that the SUS is *partially observable* when the monitor can observe only a subset $O \subseteq V$ of variables (O is called the *observables*). Thus, the input trace of the monitor contains only variables from O. However, it is *not* required that all variables in O must be observable in each input state of the input trace. For instance, if $O = \{p, q\}$, an observation reads that the value of p is true but do not know anything about q. It is even possible that an observation does not know anything about p and q, except that the SUS has indeed moved to the next state (i.e. the descrete time increased by one). Thus in general an observation can be viewed as a set of possible assignments to O. In particular, if $O = V$ and the observation contains a single (full) assignment to V, then we are speaking of *full observability*.

A partial observation of SUS states can be represented by a Boolean formula over the set of observables O. Thus in our RV framework the monitoring algorithm takes as input a sequence of Boolean formulae over O. For example, suppose that $O = \{p, q\}$ and the input trace is $\mu = p \cdot q \cdot \top$. Then in the first state, the monitor observes that p is true but does not see the value of q; in the second state, q is true but the value of p is not observed; in the third state,

neither the value of p nor of q is available (all four assignments are possible). Thus, the actual behavior of the SUS may be any trace compatible with these constraints.

However, in practice an input state of $2^{2^{|O|}}$ possible values is too much. For instance, a Boolean formula like $(p \wedge \neg q) \vee (\neg p \wedge q)$ (or equivalently $\{\{p\}, \{q\}\}$, denoting either $p = \top \wedge q = \bot$ or $p = \bot \wedge q = \top$) may never appear in the application scenario, but this kind of inputs can indeed be handled by our symbolic monitoring algorithm (to be given) without extra efforts. A slightly restricted approach is to consider each variable in O with three possible values in each input state: true ($\top$), false ($\bot$) and *unknown* (?). For those variables whose value is unknown, we just assume that both $\top$ and $\bot$ is possible. Under this setting, the possible values in each input is reduced from $2^{2^{|O|}}$ to $3^{|O|}$, suitable for low-level monitor code generation.

4.4 Assumption-Based Runtime Verification

Now we formally present the generalized RV framework which extends the traditional RV with three new features: assumptions, partial observability and resets. The ABRV-LTL semantics is an extension of LTL_3:

Definition 4.4.1. *(Semantics of ABRV-LTL [47]) Let $K \doteq \langle V_K, \Theta_K, \rho_K, \mathcal{J}_K \rangle$ be an FTS, and $u = a_0 \dots a_{n-1} \in \Psi(O)^*$ be a finite sequence of Boolean formulae over $O \subseteq V_K \cup AP$ ($|u| = n$) and*

$$\mathcal{L}^K(u) \doteq \left\{ w \in \mathcal{L}(K) \;\middle|\; w_i(V_K \cup AP) \models a_i(O) \text{ for all } i < n \right\} \tag{4.2}$$

be the set of runs in K which are compatible with u. The truth value of an ABRV-LTL formula φ w.r.t. u (at index i) under the assumption K, denoted with $[\![u, i \models \varphi]\!]_4^K$, is an element of $\mathbb{B}_4 \doteq \{\top^{\mathrm{a}}, \bot^{\mathrm{a}}, ?, \times\}$ and defined as follows:

$$[\![u, i \models \varphi]\!]_4^K \doteq \begin{cases} \times & \text{if } \mathcal{L}^K(u) = \emptyset, \\ \top^{\mathrm{a}} & \text{if } \mathcal{L}^K(u) \neq \emptyset \text{ and } \forall w \in \mathcal{L}^K(u) \colon w, i \models \varphi, \\ \bot^{\mathrm{a}} & \text{if } \mathcal{L}^K(u) \neq \emptyset \text{ and } \forall w \in \mathcal{L}^K(u) \colon w, i \models \neg\varphi, \\ ? & \text{otherwise.} \end{cases} \tag{4.3}$$

We also write $[\![u \models \varphi]\!]_4^K$ for $[\![u, 0 \models \varphi]\!]_4^K$.

ABRV-LTL proposes the following four verdicts:

- *conclusive true* ($\top^{\mathrm{a}}$): the specification is satisfied under the RV assumptions;
- *conclusive false* ($\bot^{\mathrm{a}}$): the specification is violated under the RV assumptions;
- *inconclusive* (?): the satisfication/violation of specification is unknown, but the input trace is still compatible with the RV assumptions;

- *out-of-model* ($\times$): the input trace from the SUS *violates* the RV assumptions.

Note that, the last verdict *out-of-model* ($\times$) can be added to other LTL semantics over finite trace, e.g. RV-LTL, when RV assumptions are taken into account during monitor synthesis.

Remark 4.4.2. *We note that the assumption of ABRV cannot be simply included in the property as the premise of an implication. More specifically, even if $\mathcal{L}(K)$ could be expressed by an LTL formula ψ, the ABRV semantics $[\![u, i \models \varphi]\!]_4^K$ is in general different from $[\![u, i \models \psi \rightarrow \varphi]\!]_3$. For example, it is possible that $[\![u, i \models \varphi]\!]_4^K = \bot^{\mathrm{a}}$ (thus, $\forall w \in \mathcal{L}^K(u).\ w, i \models \neg\varphi$) and that there exists w such that $u \cdot w, i \not\models \psi$ (thus, $u \cdot w, i \models \psi \rightarrow \varphi$ and $[\![u, i \models \psi \rightarrow \varphi]\!]_3 = ?$).*

4.4.1 Partial Observability

Due to partial observability, the finite trace u is actually *a set* of finite traces over O, where each state a_i is a set of truth assignments over O. When $\mathcal{L}^K(u) = \emptyset$, the monitor is unable to "follow" the behaviour shown from the SUS, hence the fourth verdict *out-of-model* ($\times$) comes.

The sequence of observations can be paired with a sequence of Boolean reset signals. (Hereafter, traces paired with reset signals are presented by Greek letters like μ.) Intuitively, if the monitor receives a reset at the index i, then the monitor will start to evaluate the truth value of φ at i (and does so until the next reset). Formally, the monitor receives inputs in $\Psi(O) \times \mathbb{B}$, a cross-product between formulae over the observables and the reset values. Thus $\mu = (u_0, \mathrm{res}_0), (u_1, \mathrm{res}_1), \ldots, (u_n, \mathrm{res}_n)$. We denote, with $\textsc{res}(\mu)$ and $\textsc{obs}(\mu)$, the projection of μ on the reset and observation components, respectively, i.e. $\textsc{res}(\mu) = \mathrm{res}_0, \mathrm{res}_1, \ldots, \mathrm{res}_n$ and $\textsc{obs}(u) = u_0, u_1, \ldots, u_n$.

4.4.2 ABRV problems and monitor definition

Definition 4.4.3 (ABRV with Partial Observability and Resets)**.** *Let K, φ and O have the same meaning as in Def. 4.4.1, Let $\mu \in (\Psi(O) \times \mathbb{B})^*$ be a finite sequence of observations paired with resets. The problem of Assumption-based Runtime Verification (ABRV) w.r.t. assumptions K, monitoring property φ and observables O is to construct a function $\mathcal{M}_\varphi^K \colon (\Psi(O) \times \mathbb{B})^* \rightarrow \mathbb{B}_4$ such that*

$$\mathcal{M}_\varphi^K(\mu) = [\![\textsc{obs}(\mu), \textsc{mrr}(\mu) \models \varphi]\!]_4^K \tag{4.4}$$

where $\textsc{mrr}(\mu)$ (the most recent reset) is the maximal i such that $\textsc{res}(\mu_i) = \top$, or 0 if such a maximal i does not exist (e.g., when $\textsc{res}(\mu_i) = \bot$ for all i).

An illustration of ABRV monitors is shown in Fig. 4.3. It shows how the belief states inside the monitor are narrowed down by each input states, while the monitoring outputs depend on the emptiness of belief states.

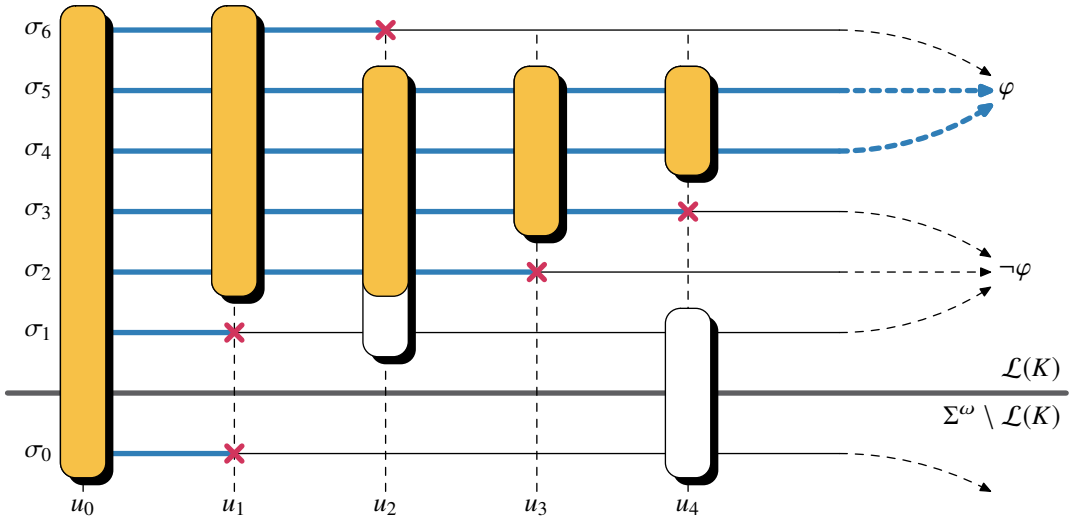


Figure 4.3: ABRV Illustration

4.5 Motivating Example

Here we present a motivating example to clarify the significance of the assumption-based RV approach. This is a real-world example taken from [67], which models a (simplified) assembly line in a factory, as shown in Fig. 4.4. Some empty bottles need to pass three positions in the assembly line to have two different ingredients filled in. The red ingredient is filled at position 0, while the green ingredient is filled at position 1. In case of faults, some ingredients may not be filled successfully. There is a transmission belt (as the grey bottom line in Fig. 4.4) moving all bottles to their next positions, and the filling operations can only be done when the belt stops (i.e. not moving). All variables in the model are Boolean: `bottle_present[]` (with index 0–2) denotes the existence of a bottle at a position. Similarly, `bottle_ingr1[]` denotes the existence of the red ingredient in the bottle at a position, and `bottle_ingr2[]` for the green ingredient. Besides, `move_belt` denotes if the belt is moving, and `new_bottle` denotes if there is a new bottle coming at position 0 before the belt starts to move. Finally, two unobservable variables `fault[0]` and `fault[1]` denote the faults: whenever it happens, the corresponding filling operations will definitely fail and the corresponding ingredients are not filled into the bottle at the positions. The belt moves *infinitely often* (i.e. it does not stop forever after certain time), and this is guaranteed by the fairness condition of the model.

The precise model definition in SMV language is given in the Appendix (Section A.1). But all the model does is to guarantee that *physically impossible things will not happen*, e.g. any bottle or the ingredients inside them does not suddenly appear or disappear, plus that the transmission belt does not stop forever (i.e. it infinitely often moves).

The monitoring specification is that, *whenever there is a bottle at position 2, both ingredients*

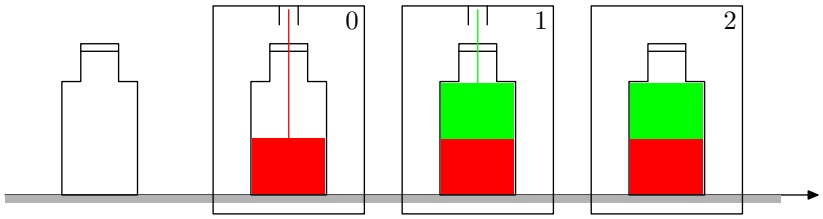


Figure 4.4: The Factory Model

are filled in that bottle. This can be expressed by LTL property $\mathbf{G}\,(\texttt{bottle\_present[2]} \rightarrow (\texttt{bottle\_ingr1[2]} \wedge \texttt{bottle\_ingr2[2]}))$. Partial observability of this factory model is the following: except for `fault[0]` and `fault[1]`, everything else is observable.

Consider an execution in which a fault happens and then a bottle is at position 0 but the red ingredient is not filled correctly, i.e. `bottle_present[0]` is true while `bottle_ingr1[0]` is false. It is not hard to imagine that the same bottle will eventually also present itself at position 2 without the red ingredient, i.e. `bottle_present[2]` is true while `bottle_ingr1[2]` is false. The ABRV monitor of the above specification and the model will *predict* this violation immediately after this bottle left position 0 but before arriving at position 2, while any monitor just synthesized from the specification itself will not be able to predict the violation so early.

Instead of writing the above monitoring specification explicitly by variables `bottle_present[2]`, `bottle_ingr1[2]`, `bottle_ingr2[2]`, another equivalent way is to monitor the LTL property $\mathbf{G}\,\neg(\texttt{fault[0]} \vee \texttt{fault[1]})$. Note that this new LTL property involves non-observable variables whose value cannot be directly obtained from the input traces. But anyway the corresponding ABRV monitor works and is also predictive. This is because, the value of `fault[0]` and `fault[1]` can be inferred from other observable variables in the model, and the actual input trace will provide enough information for deciding their values (and also the violation of the above new monitor specification.)

Finally, let us consider the case where two faults happen at different time for two different bottles. After detecting the first fault, we would like to reset the monitor to detect also the second fault. However, suppose the second fault occurs before the last reset: in this case, the observations before the reset may be necessary to detect the second fault. Resettable monitors allow us to change the monitor outputs back to those before faults happen, without losing track of the internal states of the SUS.

4.6 Theoretical Results of ABRV

Having the goal of ABRV monitor synthesis established (Def. 4.4.3, before we present monitor synthesis algorithms in next chapters, for both finite- and infinite-state systems, in this section we first discuss some theoretical results of ABRV monitors, to be used in later chapters.

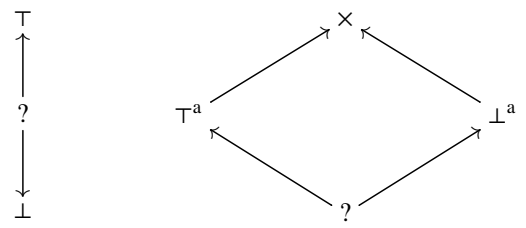


Figure 4.5: LTL_3 (left) v.s. ABRV-LTL (right), the direction of arrows indicate the possible changes of monitor outputs after more inputs (without being reset)

4.6.1 Basic properties of ABRV monitors

Here are some basic properties of the monitor defined in Def. 4.4.3. Let $(\mathbb{B}_4, \sqsubseteq)$ be a lattice with the partial order $? \sqsubseteq \top^{\mathrm{a}}/\bot^{\mathrm{a}} \sqsubseteq \times$, shown in Fig. 4.5 (with a comparison to the LTL_3 lattice). The proofs are similar with the monitonicity of LTL_3:

Lemma 4.6.1. *If there is no reset, the monitor $\mathcal{M}_\varphi^K$ is always mono-increasing:*

$$\forall u \in (\Psi(O) \times \{\bot\})^*, \psi \in \Psi(O).\ \mathcal{M}_\varphi^K(u) \sqsubseteq \mathcal{M}_\varphi^K(u \cdot (\psi, \bot)) \ .$$

Lemma 4.6.2. *ABRV-LTL monitor is* anti-monotonic *w.r.t. assumptions:*

$$\forall K_1, K_2.\ \mathcal{L}(K_2) \subseteq \mathcal{L}(K_1) \Rightarrow \forall u \in (\Psi(O) \times \mathbb{B})^*.\ \mathcal{M}_\varphi^{K_1}(u) \sqsubseteq \mathcal{M}_\varphi^{K_2}(u) \ .$$

We say that the assumption K is *valuable* (or *useful*) for φ if there exists $u \in (\Psi(O) \times \{\bot\})^*$ such that $\mathcal{M}_\varphi^\emptyset(u) = ?$ and $\mathcal{M}_\varphi^K(u) = \top^{\mathrm{a}}$ or $\bot^{\mathrm{a}}$. This can happen when the monitor $\mathcal{M}_\varphi^K$ is a *diagnostic monitor* which deduces non-observable values from the assumption and observations, or when the monitor $\mathcal{M}_\varphi^K$ is a *predictive monitor*, which deduces future facts from the assumption and observations.

Actually it is not hard to see that ABRV-LTL semantics satisfies all four maxims mentioned in Section 4.1, under the revised version of Maxim 2: the negation of *out-of-model* ($\times$) is itself, while $\top^{\mathrm{a}}$ and $\bot^{\mathrm{a}}$ are negations of each other. It is not very useful, on the other hand, to discuss if the four verdicts of ABRV-LTL form a lattice or not, because we never need to do any calculation on these verdicts (coming from sub-formulae, e.g.).

4.6.2 MC reduced to ABRV

From algorithmic point of view, having some extra assumptions during the monitor synthesis is nothing but an extra (Fair) Kripke Structure (a model as RV assumptions) synchronously

composed into the Kripke Structure translated from the monitoring property [109], thus RV assumptions have minor impacts in the shape of ABRV algorithms. On the other hand, although RV is usually considered as a lightweight verification technique, with the addition of assumptions we can show that any model checking problem can be reduced to ABRV monitoring with the same temporal logic. as the next theorem shows:

Theorem 4.6.3 (MC reduced to ABRV)**.** *Let K be a model (as an FTS), and φ be a property in temporal logics like LTL. The model checking problem $K \models \varphi$ can be done by ABRV monitoring on empty traces using the same model as RV assumptions.*

Proof. Let ϵ be an empty trace. From Definition 4.4.3 and 4.4.1 we have

$$\mathcal{M}_\varphi^K(\epsilon) = \begin{cases} \top^{\mathrm{a}}, & \text{if } [\![K \models \varphi]\!] = \top \text{ (and } [\![K \models \neg\varphi]\!] = \bot\text{)}, \\ \bot^{\mathrm{a}}, & \text{if } [\![K \models \varphi]\!] = \bot \text{ (and } [\![K \models \neg\varphi]\!] = \top\text{)}, \\ ?, & \text{if } [\![K \models \varphi]\!] = [\![K \models \neg\varphi]\!] = \bot \text{ (counterexamples exist on both sides)}, \\ \times, & \text{if } [\![K \models \varphi]\!] = [\![K \models \neg\varphi]\!] = \top \text{ (i.e. } \mathcal{L}(K) = \emptyset\text{, i.e. } K \text{ is an empty model)}. \end{cases}$$

Thus $[\![K \models \varphi]\!] = \top$ iff $\mathcal{M}_\varphi^K(\epsilon) = \top^{\mathrm{a}}$ (or $\times$ if the model K is empty). □

Thus ABRV cannot have lower time/space complexity than MC w.r.t. the combined size of the model K and the checking property φ. Also note that the benefits of choosing LTL_3 (over RV-LTL) as the basis of LTL semantics over finite trace now show up: both LTL_3 and ABRV-LTL are specified on empty traces.

4.6.3 ABRV reduced to MC

What is more interesting and useful, is that ABRV monitoring can be reduced to model checking, as the following theorem shows:

Theorem 4.6.4 (ABRV reduced to MC)**.** *Let K be RV assumptions (as FTS), and φ be a monitoring property in temporal logics like LTL. Let $u = a_0 \dots a_{n-1}$ be a finite trace ($|u| = n$). The ABRV monitoring problem $\mathcal{M}_\varphi^K(u)$ can be done by two calls of model checking on combined models from K and u.*

Proof. Let c be a fresh integer variable[3] taking finite domain values from 0 to $n-1$. Let $S_u = \langle V_k \cup \{c\}, \Theta, \rho, \emptyset\rangle$ be a Kripke Structure built from u, where

$$\Theta \doteq (c = 0) \wedge a_0, \qquad \rho \doteq \bigwedge_{i=0}^{n} \big((c = i) \rightarrow (c' = i + 1 \wedge a'_{i+1})\big)$$

Then, by Definition 4.4.3 and 4.4.1 $\mathcal{M}_\varphi^K(u)$ can be computed by two MC calls $[\![K \times S_u \models \varphi]\!]$ and $[\![K \times S_u \models \neg\varphi]\!]$, with the output verdict given by Table 4.2. □

$[\![K \times S_u \models \varphi]\!]$	$[\![K \times S_u \models \neg\varphi]\!]$	$\mathcal{M}_\varphi^K(u)$
$\top$	$\top$	$\times$
$\top$	$\bot$	$\top^{\mathrm{a}}$
$\bot$	$\top$	$\bot^{\mathrm{a}}$
$\bot$	$\bot$	?

Table 4.2: Output table in ABRV-MC reduction

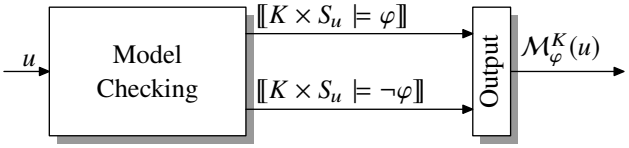


Figure 4.6: ABRV reduced to MC

Theorem 4.6.4 can be seen as a universal ABRV monitoring algorithm, as long as there exists model checkers supporting the temporal logics of the monitoring properties. Fig. fig:ABRVtoMC shows the workflow of such a trivial ABRV monitor. Note that this reduction-based ABRV monitor is not incremental, i.e. by taking one more input state from the SUS, new model checking calls cannot benefit from previous model checking results. To overcome this difficulity, in Chapter 6 (RV of infinite-state systems), an improved reduction-based RV algorithm based on incremental BMC model checkers will be given.

4.6.4 Undecidability of RV on infinite-state systems

Combining Theorem 4.6.3 and 4.6.4, one can see that ABRV and MC basically have the same space and time complexities, as there exist bidirectional reductions between them. Such a close relationship betwen ABRV and MC does not hold for traditional RV without assumptions.

In particular, it is known that MC on infinite-state systems is in general *undecidable* (mostly because the reachability problem is undecidable for infinite-state transition systems [44]), thus RV on infinite-state systems is also in general undecidable.[4]

4.6.5 ABRV extended with counter-examples

The MC-to-ABRV reduction given in Section 4.6.2 has one problem: when MC returns negative results, there is no way to have counter examples from ABRV monitors since monitors only report simple verdicts.

[3]Thus c can be encoded into at most $\log_2 n$ new Boolean variables if the underlying model checker supports only Boolean variables.

[4]Courtesy of Prof. César Sánchez, Here we also add his another comment which further illustrates our undecidability result in another (better) way: "Obviously, it is easy to prove that for arbitrary infinite-state systems the (monitoring) problem becomes undecidable (even for simple properties and complex systems - one could easily cover the Halting problem - or for trivial/no system and complex property - one could model PCP or counter machines.)

To overcome this difficulty (bringing also other benefits), ABRV monitors (Definition 4.4.3) can be modified to optionally provide extra informations together with $\mathbb{B}_4$ verdicts:

Definition 4.6.5 (Extended ABRV monitors)**.** *The ABRV monitor $\mathcal{M}_\varphi^K(\mu)$ may optionally return two (lasso-shaped, i.e. with ending loop) infinite traces $w_1, w_2 \in \mathcal{L}^K(\text{OBS}(u))$ (if they exist) such that $w_1, \text{MRR}(\mu) \models \varphi$ and $w_2, \text{MRR}(\mu) \models \neg\varphi$.*

Note that w_1 and w_2 do not always exist:

- when $\mathcal{M}_\varphi^K(\mu) = \top^{\text{a}}$, only w_1 exists;
- when $\mathcal{M}_\varphi^K(\mu) = \bot^{\text{a}}$, only w_2 exists;
- when $\mathcal{M}_\varphi^K(\mu) = ?$, both w_1 and w_2 exist;
- when $\mathcal{M}_\varphi^K(\mu) = \times$, none of w_1 and w_2 exists.

Now reconsider the ABRV-based model checker constructed in Theorem 4.6.3. When MC gives negative results, i.e. $[\![K \models \varphi]\!] = \bot$, in this case we have $\mathcal{M}_\varphi^K(\epsilon) = \bot^{\text{a}}$ or $?$. For both verdicts w_2 always exists and is the needed counter example such that $w_2 \in \mathcal{L}(K)$ and $w_2 \not\models \varphi$.

In practice, w_1 and w_2 should be as *short* as possible (i.e. as lasso-shaped traces they reach the ending loop as soon as possible). Meanwhile, since they are compatible with the current trace prefix μ, once can imagine that only the continuation parts of w_1 and w_2, after the end state of μ, provides some new information: a *prediction* of the SUS leading to conclusive verdicts. When $\mathcal{M}_\varphi^K(\mu) = ?$, instead of knowning nothing, w_1 and w_2 will provide some hints on how the SUS may evolve to different monitoring results. This is more precise than RV-LTL which refines LTL_3's inclusive verdict (?) into $\top^{\text{p}}$ and $\bot^{\text{p}}$ disregarding the future continuation of the current (finite) trace prefix.

4.6.6 Monitorability under assumptions

The assumption-based RV approach has received some attentions by other researchers since its initial publications in RV 2019 conference [47, 48]. The most profound one is the research of *monitorability under assumptions*. Here we briefly recall some key results (translated into symbols in this thesis), with comments.

A property is *monitorable* (classic, without assumptions) if every prefix of every trace has a finite extension that allows a verdict, positive or negative [129]. All safety and co-safety properties, and their boolean combinations, are monitorable [19, 64]. The classical monitorability has been extensively researched in scope of some variants of Hennessy-Milner Logic (beside LTL and CTL) [2, 3, 72]. The classic definition of monitorability assumes that the system may generate any trace. Under assumptions, Henzinger et al. [91] propose the following revised definition of monitorability:

Definition 4.6.6. *Let φ be a monitoring property, K be an assumption (Σ be its alphabet), and $u \in L(K)$ a finite trace. The property φ is* positively determined under *K by u iff, for all infinite traces $w \in \Sigma^\omega$, if $u \cdot w \in \mathcal{L}(K)$, then $u \cdot w \models \varphi$. Similarly, φ is* negatively determined under *K by u iff, for all $w \in \Sigma^\omega$, if $u \cdot w \in \mathcal{L}(K)$, then $u \cdot w \not\models \varphi$.*

Definition 4.6.7 (Monitorability under assumptions)**.** *The property φ is* u-monitorable under the assumption *K, where $u \in L(K)$ is a finite trace, iff there is a finite continuation v such that $u \cdot v \in L(K)$ positively or negatively determines φ under K. The property φ is* monitorable under *K iff it is u-monitorable under K for all finite traces $u \in L(K)$. The set of properties that are monitorable under K is denoted by* $\mathrm{Mon}(K)$.

Henzinger et al. proved the following interesting results regarding $\mathrm{Mon}(K)$:

Theorem 4.6.8. *For every assumption K, the set* $\mathrm{Mon}(K)$ *is closed under Boolean operations.*

Switching from properties to assumptions, monitorability, however, is not preserved under complementation, intersection, nor under union of assumptions. Among these Boolean operations, the union is arguably the most interesting one on assumptions. It is interesting that, under the supposition of *compatibility* (see below) with respect to a given property, the monitorability is preserved under the union of assumptions.

Definition 4.6.9 (Compatibility of assumptions)**.** *Let A and B be two assumptions, and φ be a property such that $\varphi \in \mathrm{Mon}(A)$ and $\varphi \in \mathrm{Mon}(B)$. The assumptions A and B are* compatible *with respect to φ iff for every finite trace $u \in L(A)$ that positively (resp. negatively) determines φ under A, there is no finite extension v such that $u \cdot v \in L(B)$ and $u \cdot v$ negatively (resp. positively) determines φ under B, and vice versa.*

Theorem 4.6.10 (Preservation of monitorability under union)**.** *Let A and B be assumptions, and φ be a property such that $\varphi \in \mathrm{Mon}(A)$ and $\varphi \in \mathrm{Mon}(B)$. If A and B are compatible with respect to φ, then $\varphi \in \mathrm{Mon}(A \cup B)$.*

The preservation of monitorability under the strengthening and weakening of assumptions is also explored: (Note that, in general, monitorability is neither downward nor upward preserved.)

Theorem 4.6.11. *Let A and B be assumptions, and φ be a property such that $B \subseteq A$[5] and $\varphi \cap A = \varphi \cap B$. If $\varphi \in \mathrm{Mon}(A)$ and $B \subseteq \mathrm{Mon}(A)$ such that every prefix that negatively determines B under A has a proper prefix that negatively determines φ under A, then $\varphi \in \mathrm{Mon}(B)$.*

Theorem 4.6.12. *Let A and B be assumptions, and φ be a property such that $B \subseteq A$ and $\varphi \cap A = \varphi \cap B$. If $\varphi \in \mathrm{Mon}(B)$ and $B \subseteq \mathrm{Mon}(A)$, then $P \in \mathrm{Mon}(A)$.*

[5]If A and B are both FTS, then $B \subseteq A$ is nothing but an abbreviation of $\mathcal{L}(B) \subseteq \mathcal{L}(A)$. Other set operations between assumptions, or between assumtions and properties, e.g. $\varphi \cap A$, should be understood similarly as set operations on their languages, i.e. set of infinite traces.

The above results have provided methodologies and useful guidelines for users of assumption-based monitor synthesis tools (including NuRV) to choose properly the RV assumptions for having monitorable properties (and the final monitors) as much as possible. The algorithms to be presented in this thesis, however, will only focus on the synthesis of correct monitors given the properties and assumptions.

4.6.7 ABRV and FDI (monitor vs. diagnoser)

Technically speaking, there is no much difference between ABRV monitors and the so-called *diagnosers* from the field of *Fault Detection, Identification (FDI)* [25], mostly because both monitors and diagnosers are synthesized with the knowledge of the system to be monitored or diagnosed.

FDI provides the ability to detect when and which faults occur during operation. Faults are often not directly observable. Their occurrence can only be inferred by observing the effects that they have on the observable parts of the system. An FDI component, also referred as a *diagnoser*, processes sequences of observations, made available by predefined sensors, and is required to trigger a set of predefined alarms in a timely and accurate manner [26]. Using RV terminologies, before a diagnoser actually triggers an alarm, it can be understood as a monitor returning inconclusive verdicts, while the alarm can be understood as a conclusive verdict (more precisely, a conclusive false $\bot^{\mathrm{a}}$). Roughly speaking, a plant is diagnosable iff its alarm conditions (as monitoring properties) are monitorable under the assumption as plant model. There exists extensive research on diagnosability of discrete-event systems [133].

Formally speaking, a diagnoser is a machine D that synchronizes with observable traces of the plant P. D has a set $\mathcal{A}$ of Boolean alarm variables that are activated in response to the monitoring of P. The first element for the specification of the FDI requirements is given by the conditions that must be monitored, denoted with β. In FDI, the *diagnosis condition* β to be monitored cannot be a temporal formula. Instead it is a Boolean combination of state variables of the plant, including the (non-observable) faults.

The temporal aspects of diagnosis conditions are provided by the second element of the specifiation of FDI requirements: the relation between a diagnosis condition and the raising of an alarm. An alarm condition is composed of two parts: the diagnosis condition and the delay. The delay relates the time between the occurrence of the diagnosis condition and the corresponding alarm; it might be the case that the occurrence of a fault can go undetected for a certain amount of time. By interaction with industrial experts four [6] initial patterns of alarm conditions are identified:

1. $\textsc{ExactDel}(A, \beta, n)$ specifies that whenever β is true, A must be triggered exactly n steps

[6]The first three alarm patterns are defined in [25, 26]. The fourth pattern $\textsc{BoundDel}_O(A, \beta, n)$ is added in [24].

Alarm Condition	LTL (Correctness)	LTL (Completeness)
$\textsc{ExactDel}(A, \beta, n)$	$\mathbf{G}(A \rightarrow \mathbf{Y}^n \beta)$	$\mathbf{G}(\beta \rightarrow \mathbf{X}^n A)$
$\textsc{BoundDel}(A, \beta, n)$	$\mathbf{G}(A \rightarrow \mathbf{O}^{\leqslant n} \beta)$	$\mathbf{G}(\beta \rightarrow \mathbf{F}^{\leqslant n} A)$
$\textsc{FiniteDel}(A, \beta)$	$\mathbf{G}(A \rightarrow \mathbf{O}\, \beta)$	$\mathbf{G}(\beta \rightarrow \mathbf{F} A)$
$\textsc{BoundDel}_O(A, \beta, n)$	$\mathbf{G}(A \rightarrow \mathbf{O}\, \beta)$	$\mathbf{G}(\beta \rightarrow \mathbf{F}^{\leqslant n} A)$

Table 4.3: Alarm conditions as LTL [24]

later and A can be triggered only if n steps earlier β was true;

2. $\textsc{BoundDel}(A, \beta, n)$ specifies that whenever β is true, A must be triggered within the next n steps and A can be triggered only if β was true within the previous n steps;

3. $\textsc{FiniteDel}(A, \beta)$ specifies that whenever β is true, A must be triggered in a later step and A can be triggered only if β was true in some previous steps;

4. $\textsc{BoundDel}_O(A, \beta, n)$ specifies that whenever β is true, A must be triggered within the next d steps and A can be triggered only if β was true in some previous steps.

The combination of diagnosis conditions and alarm patterns can be defined in uniformly in LTL with past operators, as shown in Table 4.3.

ABRV monitors can be used as diagnosers: one can synthesize monitors using plant models as RV assumptions. The non-monitorable faults appeared in LTL formulation of alarm patterns can be infered by other observable state variables with help of the plant models. Furthermore, note that the LTL formulae corresponding to alarm conditions must be evaluated at the last index of current input trace prefix. ABRV monitors can handle such LTL evaluations by resets. (See also Chapter 7 for an extensive application of resets when monitoring Past-Time LTL.)

Chapter 5

Monitoring Finite-State Systems

In finite-state systems, all state and input variables involved in system models (as RV assumptions) and monitoring properties are either Boolean variables or finite-domain scalar variables (enumerations, fixed-size machine words and arrays). The ABRV monitoring of finite-state systems can be done efficiently done by a symbolic monitoring algorithm implemented by BDD, and can be synthesized into explicit-state monitors generated into various programming languages.

5.1 Symbolic Monitoring Algorithm

The symbolic monitoring algorithm is presented in Algorithm 1 for the RV problem given in Def. 4.4.3, for finite-state systems. This algorithm leverages Boolean formulae and can be effectively implemented in BDD. A monitor is built from an assumption $K \doteq \langle V_K, \Theta_K, \rho_K, \mathcal{J}_K \rangle$ and an LTL property φ. The monitor can be used with any finite trace $u \in (\Psi(O) \times \mathbb{B})^*$, where $O \subseteq V_K \cup AP$ is the set of observables.

In the monitor building phase (L2–5), the LTL to ω-automata translation algorithm (Sect. 3.7) is called on φ and $\neg\varphi$ for constructing T_φ and $T_{\neg\varphi}$. The set of fair states of $K \otimes T_\varphi$ and of $K \otimes T_{\neg\varphi}$ are computed as $\mathcal{F}_\varphi^K$ and $\mathcal{F}_{\neg\varphi}^K$. Starting from L7, the purpose is to update two belief states r_φ and $r_{\neg\varphi}$ according to the input trace u. If we imagine $K \otimes T_\varphi$ and $K \otimes T_{\neg\varphi}$ as two NFAs, then r_φ and $r_{\neg\varphi}$ are the sets of current states in them. They are initialized with the initial conditions of $K \otimes T_\varphi$ and $K \otimes T_{\neg\varphi}$ (restricted to fair states). Indeed, their initial values are given by a chain of conjunctions at L6. They are then intersected with the first input state u_0 at L8. For the remaining inputs (if they exist), when there is no reset (L11–12), the purpose is to walk simultaneously in $K \otimes T_\varphi$ and $K \otimes T_{\neg\varphi}$ by computing the forward images of r_φ and $r_{\neg\varphi}$ with respect to the current input state and the set of fair states.

If any input state comes in with a reset signal, now the monitor needs to be reset (L14–15). Remarkably, this can be obtained simply by taking $r_\varphi \vee r_{\neg\varphi}$ at L14 (by calling `compute_-`

Algorithm 1: The symbolic (offline) monitor

1 **function** `symbolic_monitor`$(K \doteq \langle V_K, \Theta_K, \rho_K, \mathcal{J}_K\rangle, \varphi(AP), u \in (\Psi(O)\times\mathbb{B})^*)$
2 $\quad T_\varphi \doteq \langle V_\varphi, \Theta_\varphi, \rho_\varphi, \mathcal{J}_\varphi\rangle :=$ `ltl_to_automata`(φ);
3 $\quad T_{\neg\varphi} \doteq \langle V_\varphi, \Theta_{\neg\varphi}, \rho_\varphi, \mathcal{J}_\varphi\rangle :=$ `ltl_to_automata`$(\neg\varphi)$;
4 $\quad \mathcal{F}^K_\varphi :=$ `get_fair_states`$(K \otimes T_\varphi)$;
5 $\quad \mathcal{F}^K_{\neg\varphi} :=$ `get_fair_states`$(K \otimes T_{\neg\varphi})$;
6 $\quad \langle r_\varphi, r_{\neg\varphi}\rangle := \langle \Theta_K \wedge \Theta_\varphi \wedge \mathcal{F}^K_\varphi, \Theta_K \wedge \Theta_{\neg\varphi} \wedge \mathcal{F}^K_{\neg\varphi}\rangle$;
7 $\quad$ **if** $|u| > 0$ **then**
8 $\quad\quad \langle r_\varphi, r_{\neg\varphi}\rangle := \langle r_\varphi \wedge \textsc{obs}(u_0), r_{\neg\varphi} \wedge \textsc{obs}(u_0)\rangle$;
9 $\quad$ **for** $1 \leqslant i < |u|$ **do**
10 $\quad\quad$ **if** $\textsc{res}(u_i) = \bot$ **then** `/* without reset */`
11 $\quad\quad\quad r_\varphi := \mathrm{fwd}(r_\varphi, \rho_K \wedge \rho_\varphi)(V_K \cup V_\varphi) \wedge \mathcal{F}^K_\varphi \wedge \textsc{obs}(u_i)$;
12 $\quad\quad\quad r_{\neg\varphi} := \mathrm{fwd}(r_{\neg\varphi}, \rho_K \wedge \rho_\varphi)(V_K \cup V_\varphi) \wedge \mathcal{F}^K_{\neg\varphi} \wedge \textsc{obs}(u_i)$;
13 $\quad\quad$ **else** `/* with reset */`
14 $\quad\quad\quad \langle r'_\varphi, r'_{\neg\varphi}\rangle :=$ `compute_reset`$(r_\varphi, r_{\neg\varphi})$;
15 $\quad\quad\quad \langle r_\varphi, r_{\neg\varphi}\rangle := \langle r'_\varphi \wedge \textsc{obs}(u_i), r'_{\neg\varphi} \wedge \textsc{obs}(u_i)\rangle$;
16 $\quad$ **if** $r_\varphi = r_{\neg\varphi} = \bot$ **then return** $\times$;
17 $\quad$ **else if** $r_\varphi = \bot$ **then return** $\bot^{\mathrm{a}}$;
18 $\quad$ **else if** $r_{\neg\varphi} = \bot$ **then return** $\top^{\mathrm{a}}$;
19 $\quad$ **else return** ?;
20 **function** `compute_reset`$(r_\varphi, r_{\neg\varphi})$
21 $\quad r := r_\varphi \vee r_{\neg\varphi}$;
22 $\quad r'_\varphi := \mathrm{fwd}(r, \rho_K \wedge \rho_\varphi)(V_K \cup V_\varphi) \wedge \chi(\varphi) \wedge \mathcal{F}^K_\varphi$;
23 $\quad r'_{\neg\varphi} := \mathrm{fwd}(r, \rho_K \wedge \rho_\varphi)(V_K \cup V_\varphi) \wedge \chi(\neg\varphi) \wedge \mathcal{F}^K_{\neg\varphi}$;
24 $\quad$ **return** $\langle r'_\varphi, r'_{\neg\varphi}\rangle$;

`reset()`). Then the forward image computed at L15 is for shifting the current values of all elementary variables by one step into the past, then the conjunction of $\chi(\varphi)$ (or $\chi(\neg\varphi)$, resp.) makes sure that at next round the "new" automata will accept φ (or $\neg\varphi$, resp.). Note that we cannot use Θ_φ or $\Theta_{\neg\varphi}$ here, because they contain the initial all-false assignments of the past elementary variables, which may wrongly overwrite the history stored in r, as some of these variables may not be false any more. The whole reset process completes here, then the current input observation $\textsc{obs}(u_i)$ is finally considered and the new belief states must be restrict in fair states. Finally (L16–19) the monitor outputs a verdict in $\mathbb{B}_4$, depending on four possible cases on the emptiness of r_φ and $r_{\neg\varphi}$. This is in line with ABRV-LTL given in Def. 4.4.1.

Remark 5.1.1. *The Boolean formula r in the function* `compute_reset` *actually represents the* history *of the current input trace and the current "position" in the RV assumption. This is because, before taking the disjunction, r_φ as belief states represents 3 things: the history of the*

current input trace, the current position in the RV assumption, and the evaluation of φ*; while* $r_{\neg\varphi}$ *also represents 3 things: represents the history of the current input trace, the current position in the RV assumption,* and the evaluation of $\neg\varphi$*. If we take the disjunction, the third would become the evaluation of* $\varphi \vee \neg\varphi$*, or* true*, thus just disappeared, remaining the history of the current input trace and the current position in the RV assumption, which is used as the new initial condition for re-monitoring the same property* φ*.*

Running example Suppose we monitor $\varphi = p\,\mathbf{U}\,q$ assuming $p \neq q$, and both p and q are observable. Here we have:

- $O = \{p, q\}$,
- $V_\varphi = \{p, q, x \doteq \mathrm{x}_{p\mathbf{U}q}\}$,
- $\Theta_\varphi = q \vee (p \wedge x)$, $\Theta_{\neg\varphi} = \neg(q \vee (p \wedge x))$,
- $\rho_\varphi = x \leftrightarrow (q' \vee (p' \wedge x'))$,
- $K = \langle O, \top, p' \neq q', \emptyset\rangle$,
- $\mathcal{F}^K_\varphi = \mathcal{F}^K_{\neg\varphi} = \top$ (thus $\mathcal{J}_\varphi$ and $\mathcal{J}_{\neg\varphi}$ can be ignored since all states are fair).

Now let $u = \{p\}\{p\}\cdots\{q\}\{q\}\cdots$ (i.e., no resets). Initially (L7–6), $r_\varphi = \Theta_\varphi$, $r_{\neg\varphi} = \Theta_{\neg\varphi}$, Intersecting with the initial state $\{p\}$, they become (L8):

$$r_\varphi = \Theta_\varphi \wedge (p \wedge \neg q) = p \wedge \neg q \wedge x,$$
$$r_{\neg\varphi} = \Theta_{\neg\varphi} \wedge (p \wedge \neg q) = p \wedge \neg q \wedge \neg x \ .$$

Since both r_φ and $r_{\neg\varphi}$ are not empty, the monitor outputs ? (if u ends here). If the next state is still $\{p\}$, the values of r_φ and $r_{\neg\varphi}$ actually remain the same, because $\rho_\varphi \wedge (p' \wedge \neg q') \equiv x \leftrightarrow x'$ and L11–12 does not change anything. Thus the monitor still outputs ?, until it received $\{q\}$: in this case $\rho_\varphi \wedge (\neg p' \wedge q') \equiv x \leftrightarrow \top$, and $\mathrm{fwd}(r_{\neg\varphi}, \rho_\varphi)(V_\varphi) \wedge (\neg p' \wedge q')$ (L12) is unsatisfiable, i.e. $r_{\neg\varphi} = \bot$, while r_φ is still not empty, thus the output is $\top^{\mathrm{a}}$. Taking more $\{q\}$ does not change the output, unless the assumption $p \neq q$ is broken (then $r_\varphi = r_{\neg\varphi} = \bot$, the output is $\times$ and remains there, unless the monitor were reset).

Theorem 5.1.2. *The function* `symbolic_monitor` *given in Algorithm 1 correctly implements the monitor function* $\mathcal{M}^K_\varphi(\cdot)$ *given in Def. 4.4.3.*

Proof. Fix a trace $u \in (2^O \times \mathbb{B})^*$, we define the following abbreviations:

$$u \lesssim w \quad \Leftrightarrow \quad \forall i.\ i < |u| \Rightarrow w_i(V_k \cup AP) \models \textsc{obs}(u_i)(O), \tag{5.1}$$
$$\mathcal{L}^K_\varphi(u) \quad \doteq \quad \big\{w \in \mathcal{L}(K) \mid (w, \textsc{mrr}(u) \models \varphi) \wedge u \lesssim w\big\}, \tag{5.2}$$
$$L^K_\varphi(u) \quad \doteq \quad \big\{v \mid \exists w.\ v \cdot w \in \mathcal{L}^K_\varphi(u) \wedge |v| = |u|\big\} \ . \tag{5.3}$$

Intuitively, if $u \lesssim w$ holds, w is an (infinite) run of the FKS K *compatible* with the input trace u; $\mathcal{L}_\varphi^K(u)$ is the set of (infinite) *u-compatible* runs of K which satisfies φ with respect to the last reset position; And $L_\varphi^K(u)$ is the set of $|u|$-length prefixes from $\mathcal{L}_\varphi^K(u)$.

It is not hard to see that, Def. 4.4.3 can be rewritten in terms of $L_\varphi^K(u)$ and $L_{\neg\varphi}^K(u)$:

$$\mathcal{M}_\varphi^K(u) = [\![\textsc{obs}(u), \textsc{mrr}(u) \models \varphi]\!]_4^K = \begin{cases} \times, & \text{if } L_\varphi^K(u) = \emptyset \wedge L_{\neg\varphi}^K(u) = \emptyset, \\ \top^{\mathrm{a}}, & \text{if } L_\varphi^K(u) \neq \emptyset \wedge L_{\neg\varphi}^K(u) = \emptyset, \\ \bot^{\mathrm{a}}, & \text{if } L_\varphi^K(u) = \emptyset \wedge L_{\neg\varphi}^K(u) \neq \emptyset, \\ ?, & \text{if } L_\varphi^K(u) \neq \emptyset \wedge L_{\neg\varphi}^K(u) \neq \emptyset \ . \end{cases}$$

Now the proof of Theorem 5.1.2 can be reduced to the following sub-goals:

$$L_\varphi^K(u) = \emptyset \Rightarrow r_\varphi(u) = \emptyset \quad \text{and} \quad L_{\neg\varphi}^K(u) = \emptyset \Rightarrow r_{\neg\varphi}(u) = \emptyset \ . \tag{5.4}$$

(Notice that, however, $r_\varphi(u) \neq L_\varphi^K(u)$ (and $r_{\neg\varphi}(u) \neq L_{\neg\varphi}^K(u)$), because $r_\varphi(u)$ is a Boolean formula over $V_K \cup V_\varphi$ which may contain some extra elementary variables from the LTL translations, while $L_\varphi^K(u)$ is a Boolean formula over a smaller (or equal) set of variables $V_K \cup AP$.)

Equation (5.4) trivially holds when $u = \epsilon$, i.e. $|u| = 0$. Below we assume $|u| > 0$. We first prove the *invariant properties* of r_φ and $r_{\neg\varphi}$: (c.f. L10–15 of Algorithm 1)

$$\begin{aligned} r_\varphi(u) &= \{s \mid \exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, \textsc{mrr}(u) \models \varphi) \wedge u \lesssim w \wedge w_{|u|-1} = s\}, \\ r_{\neg\varphi}(u) &= \{s \mid \exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, \textsc{mrr}(u) \models \neg\varphi) \wedge u \lesssim w \wedge w_{|u|-1} = s\} \end{aligned} \tag{5.5}$$

where $T_\varphi^0 = \langle V_\varphi, \Theta_\varphi^0, \rho_\varphi, \mathcal{J}_\varphi \rangle$ and $\Theta_\varphi^0 = \bigwedge_{\mathrm{Y}_p \in \mathrm{el}(\varphi)} \neg \mathrm{Y}_p$, i.e. Θ_φ without $\chi(\varphi)$.

Intuitively, $r_\varphi(u)$ is the set of last states of u-compatible runs in K satisfying φ with respect to the last reset position $\textsc{mrr}(u)$. Now we prove (5.5) by induction:

- If $|u| = 1$, then $r_\varphi = \Theta_K \wedge \Theta_\varphi \wedge \mathcal{F}_{K,\varphi} \wedge \textsc{obs}(u_0)$. Thus, r_φ contains all states s such that $\exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, 0 \models \varphi) \wedge u_0 \lesssim w_0 \wedge w_0 = s$. The same for $r_{\neg\varphi}$, (5.5) is proven.

- If $|u| > 1$ and $\textsc{res}(u_n) = \bot$, let $|u| = n + 1$ and $u = v \cdot u_n$ with $|v| > 0$. Here $\textsc{mrr}(u) = \textsc{mrr}(v)$. By induction hypothesis, $r_\varphi(v) = \{s \mid \exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, \textsc{mrr}(v) \models \varphi) \wedge v \lesssim w \wedge w_{n-1} = s\}$. Thus $r_\varphi(u) = \mathrm{fwd}(r_\varphi(v), \rho_K \wedge \rho_\varphi) \wedge \textsc{obs}(u_n) = \{s \mid \exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, \textsc{mrr}(v) \models \varphi) \wedge v \cdot u_n \lesssim w \wedge w_n = s\}$. Same arguments for $r_{\neg\varphi}(u)$.

- If $|u| > 1$ and $\textsc{res}(u_n) = \top$, let $|u| = n + 1$ and $u = v \cdot u_n$ with $|v| > 0$. Here $\textsc{mrr}(u) = n$. By induction hypothesis, we have

$$\begin{aligned} r_\varphi(v) &= \{s \mid \exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, \textsc{mrr}(v) \models \varphi) \wedge v \lesssim w \wedge w_{n-1} = s\}, \\ r_{\neg\varphi}(v) &= \{s \mid \exists w \in \mathcal{L}(K \times T_\varphi^0).\ (w, \textsc{mrr}(v) \models \neg\varphi) \wedge v \lesssim w \wedge w_{n-1} = s\} \ . \end{aligned}$$

Here, if we take the *union* of $r_\varphi(v)$ and $r_{\neg\varphi}(v)$, the two conjugated terms $(w, \textsc{mrr}(v) \models \varphi)$ and $(w, \textsc{mrr}(v) \models \neg\varphi)$ will be just neutralized, i.e., $r_\varphi(v) \vee r_{\neg\varphi}(v) = \{s \mid \exists w \in \mathcal{L}(K \times T^0_\varphi).\ v \lesssim w \wedge w_{n-1} = s\}$. Thus $r_\varphi(u) = \text{fwd}(r_\varphi(v) \vee r_{\neg\varphi}(v), \rho_K \wedge \rho_\varphi) \wedge \textsc{obs}(u_n) \wedge \chi(\varphi) = \{s \mid \exists w \in \mathcal{L}(K \times T^0_\varphi).\ (w, n \models \varphi) \wedge v \cdot u_n \lesssim w \wedge w_n = s\}$. Similar steps for $r_{\neg\varphi}(u)$, and thus (5.5) is proven.

To finally prove (5.4), we first unfold (5.2) into (5.3) and get $L^K_\varphi(u) = \{v \mid \exists w.\ v \cdot w \in \mathcal{L}(K \times T^0_\varphi) \wedge (v \cdot w, \textsc{mrr}(u) \models \varphi) \wedge u \lesssim v \wedge |v| = |u|\}$. If $L^K_\varphi(u)$ is empty, then by (5.5) $r_\varphi(u)$ must be also empty, simply because $\mathcal{L}(K \otimes T^0_\varphi) \subseteq \mathcal{L}(K)$. This proves the first part of (5.4), the second part follows in the same manner. □

5.2 Explicit-State Monitor Construction

It is possible to remove the dependency of BDDs and generate an explicit-state monitor automaton, which can be further converted into standalone monitor programs. The whole process consists of three steps (suppose φ is the monitor specification):

1. Generate two DFAs corresponding to φ and $\neg\varphi$;
2. Combine the two DFAs into the monitor FSM;
3. Generate code from the monitor FSM to other programming languages.

Algorithm 2 is used for building two DFAs A^φ and $A^{\neg\varphi}$. For the actual work it calls the function `update_dfa`, which is also needed later in Algorithm 3 for updating DFAs with new initial (reset) states. Note that the set of observables O is used for picking transition actions in the DFAs.

The procedure `update_dfa()` uses a stack to recursively build an explicit-state automaton. Each state represents a set of belief states (of the SUS) after the current observed inputs. The transition labels are observation in each input state. The function `split_states`, whose definition is missing here, splits a set of belief states according to the set of observables O. It is implemented by low level functions of the BDD library which repeatedly picks minimal terms from q_0 with respect to O.

Algorithm 3 builds the final monitor FSM from A^φ and $A^{\neg\varphi}$. The idea is to walk simultaneously in both automata following the same input labels, thus each location in the monitor FSM represents a pair of locations in A^φ and $A^{\neg\varphi}$. For any input state which corresponds to a simultaneous transition in A^φ and $A^{\neg\varphi}$, if both automata are reachable, the current output is *inconclusive* (?); If only A^φ is reachable, the current output is *conclusively false* ($\bot^{\text{a}}$); If instead only $A^{\neg\varphi}$ is reachable, the current output is *conclusively true* ($\top^{\text{a}}$). In case both automata are not reachable, the current output is *out-of-model* (×), but there is no actual transition in the monitor

Algorithm 2: Construction of DFA $A_{K,O}$ from T, K and O

```
1  function compute_dfa(T ≐ ⟨V, Θ, ρ, J⟩, K ≐ ⟨V_K, Θ_K, ρ_K, J_K⟩, O)
2      F := get_fair_states(K ⊗ T);
3      q_0 := Θ_K ∧ Θ ∧ F;
4      A := ⟨V_K ∪ V, ∅, ∅, ∅, {q_0}⟩;
5      update_dfa(A, T, K, O, q_0);
6      return A;
7  procedure update_dfa(A ≐ ⟨_, Q, _, δ, _⟩, T, K, O, q_0)
8      Q := Q ∪ {q_0};
9      foreach (l, q) ∈ split_states(q_0, O) do
10         δ := δ ∪ {(q_0, l, q)};
11         if q ∉ Q then
12             Q := Q ∪ {q}; push(q, Stack)
13     while Stack is not empty do
14         q := pop(Stack);
15         q' := fwd(q, ρ_K ∧ ρ)(V_K ∪ V) ∧ F;
16         foreach (l, q'') ∈ split_states(q', O) do
17             δ := δ ∪ {(q, l, q'')};
18             if q'' ∉ Q then
19                 Q := Q ∪ q''; push(q'', Stack)
```

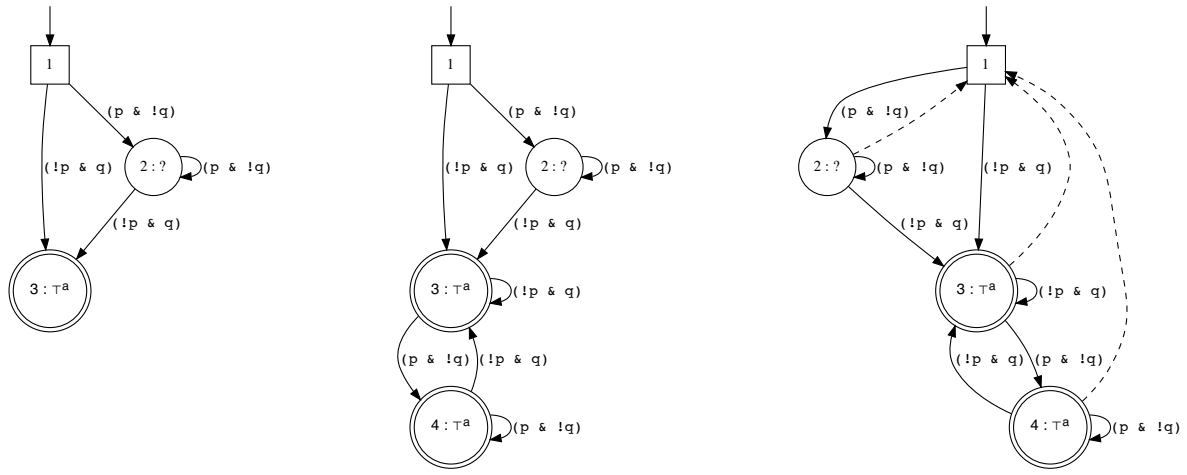
FSM, instead this case is handled directly in later phases when generating the monitor code. This process always terminate, as there are only finite number of locations in A^{φ} and $A^{\neg\varphi}$.

Whenever the monitor is reset, the current location "jumps" before processing the input states. Since for each current location there is also a corresponding "reset location", this information must be calculated for each location. However, sometimes the reset location goes outside of the original A^{φ} and $A^{\neg\varphi}$, and in this case we need to add new locations in A^{φ} and $A^{\neg\varphi}$, done in L25–26 of Algorithm 3.

Algorithm 3: Explicit-state monitor construction

```
1  function build_monitor(A^φ, A^¬φ, K, T^φ, T^¬φ, O, level)
2      q_0 := ⟨q_0^φ, q_0^¬φ, ⊤⟩; Q := {q_0}; push(q_0, Stack);
3      while Stack is not empty do
4          ⟨r_1, r_2, b⟩ := pop(Stack);
5          if level = 4 then                                  /* for ptLTL monitors */
6              ⟨q_1, q_2⟩ := compute_reset(r_1, r_2)
7          else ⟨q_1, q_2⟩ := ⟨r_1, r_2⟩;
8          for l ∈ {α | δ_φ(q_1, α) ≠ ∅ ∨ δ_¬φ(q_2, α) ≠ ∅} do
9              q'_1 := δ_φ(q_1, l), q'_2 := δ_¬φ(q_2, l);
10             if ⟨q'_1, q'_2, ⊥⟩ ∉ Q then
11                 Q := Q ∪ {⟨q'_1, q'_2, ⊥⟩}; δ(⟨r_1, r_2, b⟩, l) := ⟨q'_1, q'_2, ⊥⟩;
12                 if q'_1 ≠ ∅ ∧ q'_2 ≠ ∅ then
13                     λ(⟨q'_1, q'_2, ⊥⟩) := ? ; push(⟨q'_1, q'_2, ⊥⟩, Stack);
14                 else if q'_1 ≠ ∅ then
15                     λ(⟨q'_1, q'_2, ⊥⟩) := ⊤;
16                     if level > 1 then push(⟨q'_1, q'_2, ⊥⟩, Stack);
17                 else
18                     λ(⟨q'_1, q'_2, ⊥⟩) := ⊥;
19                     if level > 1 then push(⟨q'_1, q'_2, ⊥⟩, Stack);
20         if level = 3 then                                  /* for resettable monitors */
21             ⟨q'_1, q'_2, ⟩ := compute_reset(q_1, q_2);
22             λ'(⟨q_1, q_2, b⟩) := ⟨q'_1, q'_2, ⊤⟩;
23             if ⟨q'_1, q'_2, ⊤⟩ ∉ Q then
24                 Q := Q ∪ {⟨q'_1, q'_2, ⊤⟩}; push(⟨q'_1, q'_2, ⊤⟩, Stack);
25                 if q'_1 ∉ A^φ then update_dfa(A^φ, T^φ, K, O, q'_1);
26                 if q'_2 ∉ A^¬φ then update_dfa(A^¬φ, T^¬φ, K, O, q'_2);
27     return ⟨Q, {q_0}, δ, λ, λ'⟩;
```

Remark 5.2.1. *The intuition of Algorithm 3 is to "walk" in the belief states of the symbolic automaton as the monitor, with respect to the set of observables, just like it were explicit-state automaton. Each belief state which is already visited, is kept in a hash table so that later it will not be visited again when a later belief state has a transition back to it. Since the symbolic automaton has only finite states (when viewed as an explicit-state automaton), so must be the belief states (as sets of states), thus the walking process must terminate.*



Level 1 Level 2 Level 3

Figure 5.1: LTL monitors of $p \mathbf{U} q$ (level 1–3), assuming $p \neq q$

5.2.1 Levels of Explicit-state monitors

For the efficiency of generated monitors in different scenarios, Algorithm 3 supports four levels of explicit-state monitors (controlled by the parameter $level$ ranged from 1 to 4):

- Level 1: the monitor synthesis stops at all conclusive states;
- Level 2: the monitor synthesis explores all states;
- Level 3: the monitor synthesis explores all states and reset states;
- Level 4: the monitor always resets before taking next inputs.

Level 1 monitors are usually very small, but still fully functional when the monitor is never reset and the underlying assumption is never violated. (In this case the monitor is monotonic and never outputs $\times$.) Also, comparison tests to other RV tools are usually done by level 1 monitors, as they give exactly the same outputs in comparison with LTL_3 monitors. Level 2 monitors are full automata whose language is exactly the same as the language of the monitor specification (filtered or restricted by the assumption). Level 3 monitors are full-featured monitors supporting arbitrary resets, supporting all features of the ABRV framework. In addition, level 4 monitors consider the special use scenario in which the monitor is repeatedly reset before each new input state. This scenario is particularly useful for monitoring ptLTL, see Chapter 7 for more details.

A sample explicit-state monitor for LTL property $p \mathbf{U} q$ (level 1–3) is shown in Fig. 5.1. The monitor is generated under the assumption that p and q are always disjoint ($p \neq q$). The monitor

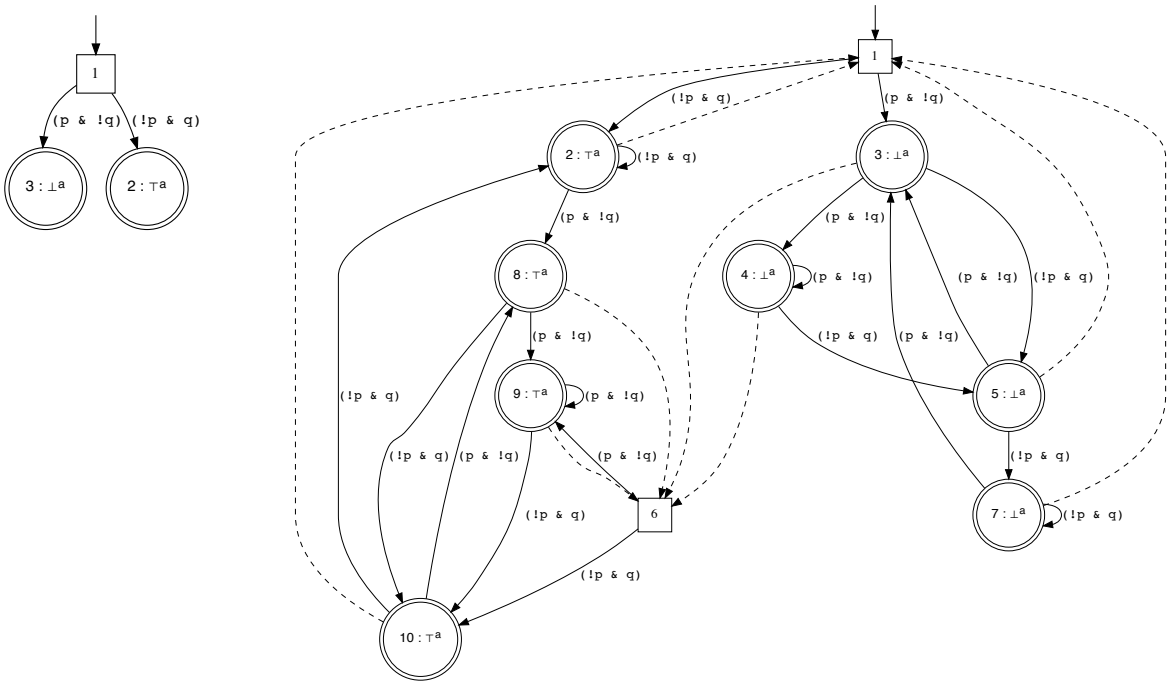


Figure 5.2: LTL monitors of $\mathbf{Y}p \vee q$ (level 1 & 3), assuming $p \neq q$

starts at location 1, and returns ? if the input is $p \wedge \neg q$ until it received $\neg p \wedge q$, which has the output $\top^{\mathrm{a}}$. The level 1 monitor has no further transition at locations associated with conclusive verdicts ($\top^{\mathrm{a}}$ or $\bot^{\mathrm{a}}$), since it can be easily proved that ABRV-LTL monitors are monotonic if the assumption is always respected by the input trace. The level 2 monitor contains all locations and transitions, thus it may return $\times$ even after the monitor reached conclusive verdicts. The level 3 monitor additionally contains information for the resets: in case the monitor is reset, the current location will first *jump* to the location following the dash lines, then goes to next location according to the input state. However, in the above monitor all reset locations are just the initial location (1), this is mostly because the assumption is an invariant property and the LTL property does not have any past operators.

Fig. 5.2 shows the level 1 and 3 monitors generated from LTL property $\mathbf{Y}p \vee q$ (the level 2 is monitor is omitted; it is like the level 3 monitor without dash lines and location 6). Fine structures of level 3 monitors are revealed here: depending on the last input, the monitor may be reset to location 1 or 6. In practice, the choice of monitor levels depends on the use scenarios, monitoring properties and assumptions, with the purpose of minimizing their size (while still fully functional).

5.3 From Offline to Online Monitoring

Algorithm 1 returns a single verdict after processing the entire input trace. This fits into Def. 4.4.3. However, runtime monitors are usually required to return verdicts for each input state and "should be designed to consider executions in an incremental fashion" [111]. Our algorithm can be easily modified for online monitoring, it outputs one verdict for each input state. It is indeed incremental since r_φ and $r_{\neg\varphi}$ are updated on each input state, and the time complexity of processing one input state is only in terms of the size of K and φ, thus *trace-length independent* [60]. Space complexity is also important, as a monitor may eventually blow up after storing enough inputs. Our algorithm is *trace non-storing* [130] with bounded memory consumption.

Algorithm 4 shows how the algorithm presented in Section 4 can be turned into an "online" monitor, that incrementally processes the observations received on an input stream. Functions `update` and `output` are defined as before.

Algorithm 4: The symbolic online monitor

```
1  program online_monitor(K = ⟨V, Θ, ρ, J⟩,φ(V),O,io)
2      T_φ ≐ ⟨V_φ, Θ_φ, ρ_φ, J_φ⟩ := ltl_to_automata(φ);
3      T_¬φ ≐ ⟨V_φ, Θ_¬φ, ρ_φ, J_φ⟩ := ltl_to_automata(¬φ);
4      F_K,φ = get_fair_states(K ⊗ T_φ);
5      r_φ := Θ_K ∧ Θ_φ ∧ F_K,φ;
6      r_¬φ := Θ_K ∧ Θ_¬φ ∧ F_K,φ;
7      write_stream(output(⟨r_φ, r_¬φ⟩))          /* before any observation */;
8      u_0 := read_stream(io);
9      r_φ := Θ_K ∧ Θ_φ ∧ F_K,φ ∧ OBS(u_0);
10     r_¬φ := Θ_K ∧ Θ_¬φ ∧ F_K,φ ∧ OBS(u_0);
11     write_stream(output(⟨r_φ, r_¬φ⟩))          /* after first observation */;
12     while io open do
13         u_i := read_stream(io);
14         r_φ, r_¬φ := update(r_φ, r_¬φ, RES(u_i), OBS(u_i));
15         write_stream(output(⟨r_φ, r_¬φ⟩))          /* more observations */;
```

5.4 Code Generation

The monitor FSM built in this way is deterministic. It is straightforward to generate program code which is equivalent to the monitor FSM. The idea is to update the current monitor location according to input state and possible reset signal, and return the monitor outputs stored at each location. Instead of representing FSM as special data structures, we can generate flat

code handling the transitions at each location, based on a direct comparison of input state and transition labels. The performance is also better due to the time saving on searching in the data structures.

Algorithm 5 shows the skeleton of the generated monitor code. The code skeleton can be easily adapted to different programming languages, without any third-party dependence. In case of small number of observables (less than the size of machine words), the cost of processing each input states is constant.

Algorithm 5: Skeleton of generated monitor code

```
1  function runtime_monitor_0(state, *current_loc, reset)
2      output := ?;
3      if reset = H then *current_loc := 1;          /* hard reset to init loc */
4      switch *current_loc do                        /* jump to current location */
5          case 1 do goto loc_1;
6          case 2 do
7              if reset = S then goto loc_j;
8              else goto loc_2;
9          ...
10     loc_1:
11     switch state do
12         case label_1 do *current_loc := k;        /* inconclusive state */
13         case label_2 do output := ⊤;              /* conclusive state */
14         ...
15         otherwise do output := ×;                 /* out of model */
16     goto exit;
17     loc_2:
18     switch state do
19         ...
20     goto exit;
21     ⋮
22     exit:
23     return output;
```

The engineering aspects of monitor code generation in various programming languages are presented in more details in Section 8.7.[1]

[1] When the number of states and transitions are too many, the generated monitor code may become too large in the code structure given in Algorithm 5, sometimes near one million lines of source code (GCC can still compile it, however). In later versions of our tool implementation, data tables and binary searching code are used instead of constant-time code jumping structures. These new design changes have greatly reduced the size of generated code while the performance lost is minor, but all these changes are merely engineering level work without much academic values.

Chapter 6

Monitoring Infinite-State Systems

In this chapter we present a general approach for ABRV of infinite-state systems (the related algorithms can also be applied to finite-state systems), originally published in RV 2021 [50]. There are some minor algorithmic corrections in this thesis.

Instead of relying on BDD, the idea of monitoring infinite-system systems is based on Satisfiability Modulo Theory (SMT) [9] and infinite-state LTL model checking. We show how ABRV problems can be directly reduced to LTL model checking problems, which is then solvable by SMT-based model checkers like nuXmv [37]. This solution is general because the theory domain is actually irrelevant with the core monitoring algorithm: any LTL MC (and QE) algorithms can be used, as long as they support the involved infinite-state variables occurred in the monitoring LTL properties and RV assumptions.

The ABRV definition and the related LTL semantics over finite traces is still the same, as described in Section 4.4. A basic reduction from ABRV to MC has also been described there (see Section 4.6.3). We will start from this basic reduction, gradually improving its performance by bringing new helper techniques into it, including First-Order Quantifier Elimination [117] and Bounded Model Checking (BMC) [23]. By modifying the original BMC algorithm to let it perform the model checking tasks *incrementally* (with respect to new monitor inputs), eventually we obtain a high performance SMT-based monitoring algorithm for infinite-state systems. All time-consuming underlying computations are essentially by SMT provers. If all involved variables are Boolean, then only SAT solvers can be used in place of SMT solvers, and the original IC3 model checker can be used in place of IC3-IA (stands for "IC3 Modulo Theories via Implicit Predicate Abstraction") model checker.

More importantly, we will show that, when the monitor outputs are inconclusive, it does not need to involve the complete (but slow) IC3-based model checkers [46], and thus can perform the verification quickly with performance comparable with BDD. The fact that when the monitor outputs are inconclusive, the BMC checker only needs to find counter-examples (thus the BMC loop stops before reaching a maximal bound), was the key to guarantee the SMT-based monitor

performance in practice, before any conclusive verdict were reached.

Essentially, the (online) infinite-state monitoring algorithm still works like the finite-state monitor based on BDD: it relies on the same LTL translation algorithm and keeps tracking two belief states which represent all the possible states in which the system may be at, after certain observations. The major difference is that now belief states have to be represented by raw formulas, which may be unbounded in worst case. However, we will show that this is inevitable in general for any ABRV algorithm over infinite-state systems: for certain RV assumptions the monitor just cannot use bounded resources (Section 6.7).

The following contents of this chapter are organized as following: First we give another motivating example to show the usefulness of infinite-state monitoring; Then we will construct a series of monitoring algorithms with gradually improved performance and complexities (so that they can be well understood one by one). The last one should be used in practice. This chapter ends by showing infinite-state ABRV monitors cannot use only bounded resources.

In general, there may be performance bottlenecks in MC- or SMT-based RV approaches in comparison with BDD-based approaches, because both model checking and quantifier elimination are computationally heavy. However, some real project use cases have shown that, for very complex RV assumptions and monitoring properties (all variables are Boolean), the SMT-based monitors are sometimes faster then BDD-based monitors.

6.1 Motivating Example

In this section, we describe another motivating example, where ABRV with infinite-state assumptions are used in a simple example of a temperature controller. Consider a system that heats the water in a tank until reaching the temperature of 100. The temperature is represented by a real variable t. The internal state of the system, which may be heating or not, is represented by the Boolean variable h. The command to switch on the heating system is represented by s, while f represents a fault that switches off the system permanently. Let us define a system model K with the following formulas:

- Initial condition: $t = 0$ (the temperature is initially 0)
- Transition conditions (implicitly conjoined):
  - $t' \geq 0 \wedge t' \leq 100$ (the temperature always remains between 0 and 100)
  - $h \rightarrow ((t = 100 \wedge t' = 100) \vee (10 \leq t' - t \leq 20))$ (if the system is heating, the temperature increases by a rate between 10 and 20 or remains 100 if it already reached that temperature)

- $\neg h \rightarrow ((t = 0 \wedge t' = 0) \vee (-20 \leq t' - t \leq -10))$ (if the system is not heating, the temperature decreases by a rate between -20 and -10 or remains 0 if it already reached that temperature)
- $h \rightarrow (h' \leftrightarrow \neg f)$ (if the system is heating, it remains so unless there is a fault)
- $(\neg h) \rightarrow (h' \leftrightarrow (s \wedge \neg f))$ (if the system is not heating and is not faulty, then it can be switched on with the command s)
- $f \rightarrow f'$ (the fault is permanent)

Suppose we can only observe the temperature and the switching command, and that we want to monitor the following property: $\varphi_1 = \mathbf{G}(s \rightarrow \mathbf{F}(t = 100))$ (whenever the heating system is switched on, the temperature will eventually reach the temperature of 100). The assumption that the system behaves according to K can be exploited by the ABRV monitor to deduce things like, whenever the temperature decreases there was a fault and so the temperature will never reach the desired level. Thus the monitor can detect the violation of a property which, without assumptions, would not be monitorable.

More specifically, consider the finite trace of observations $u = \{t \mapsto 0, s \mapsto \top\}, \{t \mapsto 20, s \mapsto \bot\}, \{t \mapsto 10, s \mapsto \top\}$. Since, without considering the assumption, there is a continuation of u satisfying φ_1 and one violating φ_1, a standard RV monitor is inconclusive (the output is ?). Considering K as assumption, all fair paths of K compatible with u violate φ. Thus, $[\![u, 0 \models \varphi_1]\!]_4^K = \bot^a$.

As an additional example, consider a stronger property $\mathbf{G}(s \rightarrow \mathbf{F}^{\leq 7}(t = 100))$, i.e., whenever the heating system is switched on, the temperature will reach 100 degree within 7 steps. In this case, from the assumption on the rates of the temperature, the ABRV monitor can deduce that after a number of steps, if the temperature is still low, it will not reach $t = 100$ in time. For example, if after 4 steps, the temperature is still less than 40, even with the maximum rate, it will not reach 100 in other 3 steps. Thus, at runtime the monitor can say that the property is violated 3 steps in advance.

6.2 ABRV Reduced to Model Checking

Given an FTS K as the RV assumption, a set of observable variables O, an LTL formula φ as the monitoring property, and a finite trace u over O, let S_u be an FTS whose fair paths are those compatible with u (formally, an FTS such that $\mathcal{L}(S_u) = \mathcal{L}^{\mathcal{U}}(u)$, where $\mathcal{U} = \langle V, \top, \top, \emptyset \rangle$ is the

FTS with an universal language). Then we have by (4.3),

$$\mathcal{M}_{\varphi}^{K}(u) = [\![u, i \models \varphi]\!]_4^K = \begin{cases} \times, & \text{if } K \times S_u \models \varphi \text{ and } K \times S_u \models \neg\varphi \\ \top^{\mathrm{a}}, & \text{if } K \times S_u \models \varphi \text{ and } K \times S_u \not\models \neg\varphi \\ \bot^{\mathrm{a}}, & \text{if } K \times S_u \not\models \varphi \text{ and } K \times S_u \models \neg\varphi \\ ?, & \text{otherwise } . \end{cases} \tag{6.1}$$

From this equation, we can derive a simple monitor called `monitor1`, which calls the model checker *twice* for each input state. It is already depicted in Fig. 4.6, where the output is defined as in (6.1) (cf. Table 4.2).

The Algorithm 6 shows the pseudo code for an implementation of `monitor1()`, a infinite-state monitor based on ABRV-MC reductions. The theoretical basis has been already discussed in Section 4.6.3.

Algorithm 6: ABRV reduced to model checking

```
1  function monitor1(K ≐ ⟨V_K, Θ_K, ρ_K, J_K⟩, φ, u ≐ a_0 ... a_{n-1})
2      Θ := ⊤, ρ := ⊤;
3      if |u| > 0 then
4          Θ := (c = 0) ∧ a_0;
5          if |u| > 1 then
6              ρ := ⋀_{i=0}^{|u|-1} ((c = i) → (c' = i + 1 ∧ a'_{i+1}));
7      S_u := ⟨V_k ∪ {c}, Θ, ρ, ∅⟩;
8      b_1 := model_checking(K × S_u, φ);
9      b_2 := model_checking(K × S_u, ¬φ);
10     if b_1 ∧ b_2 then return ×;                    // out of model
11     else if b_1 then return ⊤^a;                   // conditionally true
12     else if b_2 then return ⊥^a;                   // conditionally false
13     else return ?;                                 // inconclusive
```

The correctness of Algorithm 6 can be seen from the following theorem:

Theorem 6.2.1. *The function* `monitor1` *given in Algorithm 6 correctly implements the ABRV monitor* $\mathcal{M}_{\varphi}^{K}(\cdot)$.

sketch. The algorithm first constructs an FTS S_u from the input trace u (line 4–7). Here c is a fresh integer variable used as an internal counter. It is not hard to see that all paths of S, when projecting out c, are compatible with the input trace. For each $\sigma \in K \times S$, we have $\sigma_i \models_{\mathcal{T}} u_i$ when $i < |u|$. Thus roughly speaking we have $\mathcal{L}(K \times S_u) = \mathcal{L}^K(u)$ ($\mathcal{L}(K \times S_u)$ contains also value assignments of c).

On the other hand, by LTL semantics (over infinite traces), for any infinite trace w we have

$$[\![w \models \mathbf{X}^r \varphi]\!] = [\![w, r \models \varphi]\!] \tag{6.2}$$

The output of the monitor (line 10–13) is defined as in Eq. 6.1. There are four cases:

1. $[\![K \times S_u \models \varphi]\!] = [\![K \times S_u \models \neg\varphi]\!] = \top$. Thus,

$$\forall w \in \mathcal{L}(K \times S_u).\, w, 0 \models \varphi$$
$$\forall w \in \mathcal{L}(K \times S_u).\, w, 0 \models \neg\varphi$$

   But this is impossible, unless there is actually no fair path in $K \times S_u$. Thus we have $\mathcal{L}(K \times S_u) = \emptyset$, or $\mathcal{L}^K(u) = \emptyset$. By (4.3), $\mathcal{M}^K_\varphi(u) = [\![u, i \models \varphi]\!]^K_4 = \times$.

2. $[\![K \times S_u \models \varphi]\!] = \top$, $[\![K \times S_u \models \neg\varphi]\!] = \bot$. In this case we have

$$\forall w \in \mathcal{L}(K \times S_u).\, w, 0 \models \varphi$$
$$\exists w \in \mathcal{L}(K \times S_u).\, w, 0 \not\models \neg\varphi \quad (\text{or } w, 0 \models \varphi)$$

   The sole purpose of the above second formula is to guarantee $\mathcal{L}(K \times S_u) = \mathcal{L}^K(u) \neq \emptyset$, as there exists at least one fair path w for the universal quantifier of the first formula. Thus by (4.3), $\mathcal{M}^K_\varphi(u) = [\![u, i \models \varphi]\!]^K_4 = \top^{\mathrm{a}}$.

3. $[\![K \times S_u \models \varphi]\!] = \bot$, $[\![K \times S_u \models \neg\varphi]\!] = \top$. The proof of this case is simiar with the previous case, and we have $\mathcal{M}^K_\varphi(u) = [\![u, i \models \varphi]\!]^K_4 = \bot^{\mathrm{a}}$.

4. $[\![K \times S_u \models \varphi]\!] = [\![K \times S_u \models \neg\varphi]\!] = \bot$. In this case, nothing universal can be asserted, except for $\mathcal{L}^K(u) \neq \emptyset$. It falls into the "otherwise" case of (4.3) and we have $\mathcal{M}^K_\varphi(u) = [\![u, i \models \varphi]\!]^K_4 = ?$ (inconclusive).

□

If there is no reset nor assumption, the monitor becomes an LTL_3 monitor, as in this case $[\![S_u \models \varphi]\!] = [\![S_u \models \neg\varphi]\!] = \top$ can never happen (because $\mathcal{L}(S_u)$ is definitely not empty). We think this elegant connection and algorithm reduction from RV to MC suggests some advantages of the LTL_3 semantics (also the ABRV-LTL semantics) over other finite-trace LTL semantics. (See [18] for a detailed comparison.)

However, this algorithm is expensive and not incremental: the entire input trace must be used in constructing the model S_u. For long traces this results in big model checking problems, and the performance is far from linear to the size of input traces. The next algorithm greatly improves the situation.

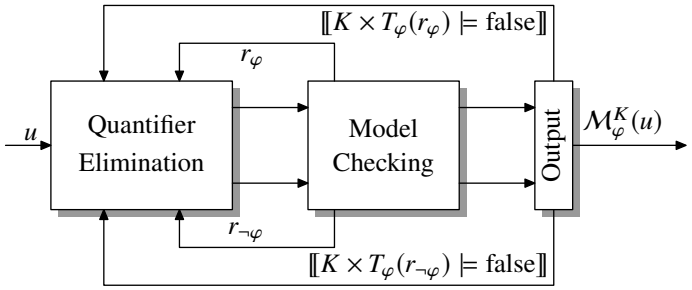


Figure 6.1: ABRV reduced to MC and QE

6.3 ABRV Reduced to MC and QE

In `monitor1` the entire input trace (the prefix received so far) must be encoded into a model (i.e. an FTS) S_u, and obviously the model checker is called on increasingly bigger problems linear to the length of the trace prefix. In practice, `monitor1()` is too slow after receiving even a small number of input states. The key for obtaining a better RV algorithm is to find a way to maintain some internal states (of the monitors), which is to be updated by each input state of the trace. For automata-based RV monitors, this internal state is the location of monitor automata. (For some rewriting-based RV approaches, the state is the current form(s) of the monitoring property after rewriting.)

Recall in the finite-state ABRV algorithm [47], the BDD-based symbolic monitor keeps track of two *belief states* r_φ and $r_{\neg\varphi}$ as the possible internal locations of automata $K \times T_\varphi$ and $K \times T_{\neg\varphi}$ (K is the RV assumption, T_φ and $T_{\neg\varphi}$ are LTL translations of φ and $\neg\varphi$, resp.), reachable with fair paths compatible with the input trace.These states are updated at each input. Since previous input states are not accessible by the algorithm, and the belief states as BDDs have bounded memory consumption, the RV algorithm given in [47] is trace-length independent [60], i.e. having bounded memory consumption (with also a time complexity linear to the length of the trace prefix).

$\neg[\![K \times T_\varphi(r_\varphi) \models \text{false}]\!]$	$\neg[\![K \times T_\varphi(r_{\neg\varphi}) \models \text{false}]\!]$	$\mathcal{M}_\varphi^K(\cdot)$
$\top$	$\top$	?
$\top$	$\bot$	$\top^{\text{a}}$
$\bot$	$\top$	$\bot^{\text{a}}$
$\bot$	$\bot$	$\times$

Table 6.1: Output Table of Fig. 6.1 and Algorithm 7

The monitor `monitor2` detailed in Algorithm 7 is very similar to the symbolic algorithm given [47]. Instead of representing formulas as BDDs, now we directly operate on raw formulas involving any type of variables. (However, in the worse case these formulas have unbounded

Algorithm 7: The RV monitor for infinite-state systems

```
1  function monitor2(K ≐ ⟨V_K, Θ_K, ρ_K, J_K⟩, φ, u)
2      T_φ  ≐ ⟨V_φ, Θ_φ  , ρ_φ, J_φ⟩ := ltl_translation(φ);          // χ(φ) is in Θ_φ
3      T_¬φ ≐ ⟨V_φ, Θ_¬φ, ρ_φ, J_φ⟩ := ltl_translation(¬φ);
4      V := V_K ∪ V_φ;
5      ⟨r_φ, r_¬φ⟩ := ⟨Θ_K ∧ Θ_φ, Θ_K ∧ Θ_¬φ⟩;
6      if |u| > 0 then
7          ⟨r_φ, r_¬φ⟩ := ⟨r_φ ∧ u_0, r_¬φ ∧ u_0⟩;
8      for 1 ⩽ i < |u| do
9          r_φ  := quantifier_elimination(V, ρ_K ∧ ρ_φ ∧ r_φ  ) ∧ u_i;
10         r_¬φ := quantifier_elimination(V, ρ_K ∧ ρ_φ ∧ r_¬φ) ∧ u_i;
11     b_1 := ¬model_checking(⟨V, r_φ, ρ_K ∧ ρ_φ, J_K ∪ J_φ⟩, false);
12     b_2 := ¬model_checking(⟨V, r_¬φ, ρ_K ∧ ρ_φ, J_K ∪ J_φ⟩, false);
13     if b_1 ∧ b_2 then return ? ;                                  // inconclusive
14     else if b_1 then return ⊤^a;                                  // conditionally true
15     else if b_2 then return ⊥^a;                                  // conditionally false
16     else return ×;                                                // out of model
```

sizes.)

The inputs of the algorithm are the RV assumption K (as an FTS), the monitoring property φ and a finite input trace u. See also Fig. 6.1, where $K \times T_\varphi(r_\varphi)$ is an abbreviation of $\langle V, r_\varphi, \rho_K \wedge \rho_\varphi, \mathcal{J}_K \cup \mathcal{J}_\varphi \rangle$. At first, φ and $\neg\varphi$ are translated into two FTS T_φ and $T_{\neg\varphi}$ (line 2–3). The initial conditions of T_φ and $T_{\neg\varphi}$, namely Θ_φ and $\Theta_{\neg\varphi}$ are respectively in the form $\chi(\varphi) \wedge \xi$ and $\neg\chi(\varphi) \wedge \xi$, where $\chi(\varphi)$ restricts the paths to satisfy φ and ξ initializes the encoding of past operators.

Initially, the belief states r_φ and $r_{\neg\varphi}$ are the initial conditions of T_φ and $T_{\neg\varphi}$, composed with the initial condition of K (line 5). The first input state u_0 is directly intersected with belief states (line 7). The *forward images* of current belief states are computed and then intersected with the current input state u_i (line 9–10).

The undefined function `quantifier_elimination` can be any (first-order) quantifier elimination procedure (for more details, see Section 3.9) such that

$$\texttt{quantifier\_elimination}(V,\ \alpha(V \cup V')) \doteq (\exists V.\alpha(V \cup V'))[V/V'] = \beta(V) \tag{6.3}$$

where $[V/V']$ substitutes the prefixed formula with all variables in V' to the corresponding variables in V. All variables in V must be eliminated from $\exists V.\alpha(V, V')$. $\beta(V)$ as the outcome of quantifier elimination is quantifier-free.

The main difference with the previous BDD-based algorithm (Algorithm 1 of [47]) is the treatment of fair states. For BDD-based FTS, the set of fair states can be computed a priori

(by algorithms like Emerson-Lei [63]) and intersected with the belief states whenever they are computed. However, for infinite-state FTS represented by raw formulas this is impossible. Thus r_φ and $r_{\neg\varphi}$ may have non-fair states in them. To check their (non)emptiness w.r.t. fair states, we leverage LTL model checking, by checking LTL formula false on the model $K \times T_\varphi$ (or $K \times T_{\neg\varphi}$, resp.) with r_φ (or $r_{\neg\varphi}$, resp.) as the initial condition (line 11–12). Here is the idea: if the model checking returned $\top$ saying for all fair paths in the input model the LTL property "false" holds (which is impossible), then the only possibility is that the input model actually does not have any fair path, i.e. the belief state is empty. The output of the monitor w.r.t. the model checking results (line 13–16) is summarized in Table 6.1.

The correctness of Algorithm 7 is given by the following theorem: (the proof is omitted due to page limits.)

Theorem 6.3.1. *The function* `monitor2` *given in Algorithm 7 correctly implements the ABRV monitor* $\mathcal{M}^K_\varphi(\cdot)$.

6.4 Optimization to ABRV-MC Reduction

In this section, we present few simple optimizations that reduce unnecessary (complete) MC calls, which are computationally expensive, or to replace them with relatively-cheap incomplete MC calls, which can only be used to detect counter-examples, e.g. the plain Bounded Model Checking (BMC). (Also note that, for infinite-state systems, the property may be violated but no lasso-shaped counter-example exists; in this case, neither BMC or the full IC3-IA algorithm can find it.)

The following four *basic* optimizations, namely *o1–o4*, are identified:

o1 If the monitor has already reached conclusive verdicts ($\top^{\mathrm{a}}$ or $\bot^{\mathrm{a}}$), then for the runtime verification of the next input state *at most one* MC call is need. In other words, one of b_1 and b_2 in 7 will not change its value if the monitor returns conclusive verdicts for the last input state. This is obvious. In Algorithm 6, if one of the two MC calls, say $[\![K \times S_u \models \varphi]\!]$, returns $\top$, then by the semantics of model checking we have φ is true for all paths in the model $K \times S_u$. Taking one more input state s, the new model $K \times S_{u\cdot s}$ is only more restrictive at its transition relation but all its (fair) paths are still paths of $K \times S_u$, thus for sure $[\![K \times S_{u\cdot s} \models \varphi]\!] = \top$.

   In fact, in this case, one of the belief states r_φ or $r_{\neg\varphi}$ becomes empty, while empty belief states can only lead to empty belief states by forward image computations. Furthermore, if the monitor has reached the verdict $\times$ (out-of-model), then it will maintain the same verdict, thus in this case no more MC (and QE) calls are necessary.

o2 Before calling model checkers to detect the emptiness of a belief state (w.r.t. fairness), an SMT checking can be done first, to check if the belief state formula can be satisfied or not. If the SMT solver returns UNSAT, then it means the formula is equivalent to $\bot$, then there is no need to further call model checkers to detect its emptiness. This is especially useful when combining with the [o4] optimization (to be explained shortly) where BMC is involved. In the plain BMC algorithm, the number of unrolling k in the BMC-encoded formula increases (until reaching a maximal given bound `max_k`) when SAT/SMT checking gives UNSAT. But if the UNSAT were actually caused by an unsatisfied initial condition, the MC procedure should immediately stop trying a bigger k and returns ($\top$), instead of unrolling once more (until reaching `max_k`).

o3 When `monitor2` is used as online monitor, the same LTL properties are sent to LTL model checkers with different models and are internally translated into equivalent FTS, in which either the initial condition (Algorithm 7) or the transition relation (Algorithm 6) is changed. Internally, the LTL properties are first translated into a tableau with respect to the fairness of the model, e.g. $(\bigwedge_{\psi \in \mathcal{J}_K \cup \mathcal{J}_\varphi} \mathbf{GF}\,\psi) \rightarrow$ false in the case of Algorithm 7, and

The translation can be done just once as part of the RV algorithm, if the involved model checkers can be modified to take pre-translated tableaux instead of LTL properties.

o4 Some model checking algorithms such as IC3-IA are more effective in proving properties, while others such as BMC can be used in practice to find counter-examples. This optimization is to call the incomplete plain BMC (or any other MC procedure which detects counter-examples) before calling a complete model checker such as IC3-IA. Note that the BMC bound parameter `max_k` can be chosen arbitrarily without hurting the correctness of the entire RV algorithm: if the counter-example does exist but BMC fails to find it due to a small `max_k`, the next complete MC call will still find it and lead to the same monitoring output as in the algorithm without this BMC optimization. On the other hand, if the plain BMC procedure reaches `max_k`, it could be that the initial condition of the input model is actually unsatisfiable. To prevent the full MC procedure being called as a fallback, the previously mentioned optimization [o2] must be also used.

The optimized versions of Algorithm 6 and 7 by the above ideas are given in Algorithm 8 and 9, respectively. (*o3* is not implemented for Algorithm 6 due to some technical limitations in our chosen model checkers but in theory the pre-translation is possible.) In the optimized algorithms, the Boolean flags o_1, o_2, o_3 and o_4 should be understood as global variables to control separately how each of the four optimizations are activated. The function `model_checking` is now defined explicitly and shared by both Algorithm 8 and 9, where `BMC` (the external bounded model checker, always assumed to be incomplete) is called before `IC3_IA` (the external IC3-IA model checker).

Algorithm 8: The optimized version of Algorithm 6

```
1  function monitor1_optimized(K ≐ ⟨V_K, Θ_K, ρ_K, J_K⟩, φ, u)
2      Θ := ⊤, ρ := ⊤;
3      if o_1 then b_1 := b_2 := ⊥ ;
4      if |u| > 0 then
5          Θ := (c = 0) ∧ u_0;
6          if |u| > 1 then
7              ρ := ⋀_{i=0}^{|u|−1} ((c = i) → (c′ = i + 1 ∧ u′_{i+1}));
8      S_u := ⟨V_k ∪ {c}, Θ, ρ, ∅⟩;
9      if o_1 → ¬b_1 then b_1 := model_checking(K × S_u, X^r φ) ;
10     if o_1 → ¬b_2 then b_2 := model_checking(K × S_u, X^r ¬φ) ;
11     if b_1 ∧ b_2 then return ×;                          // out of model
12     else if b_1 then return ⊤^a;                         // conditionally true
13     else if b_2 then return ⊥^a;                         // conditionally false
14     else return ?;                                       // inconclusive
```

Theorem 6.4.1. *Assuming* BMC *always find the counter-example whenever it exists,* IC3_IA *is called at most twice in the "online" version of Algorithm 8 with all optimizations.*

Proof. The "online" version of Algorithm 8 means that, for each input state u_{i+1} in the input trace u, the translation relation ρ is updated incrementally each time, i.e. $\rho := \rho \wedge (c = i) \rightarrow (c' = i + 1 \wedge u'_{i+1})$, and then the lines 8–14 are executed with a verdict returned, and then the same procedure happens again for the next input state.

Without loss of generality, assume that the monitor initially returns ? (inconclusive), then after some inputs changes to $\top^{\mathrm{a}}$, and finally $\times$. We analyze how the values of b_1 and b_2 changed for the entire trace:

1. Initially $b_1 = b_2 = \bot$ (so that the verdict is ?). This means that both model checking calls return $\bot$, i.e. BMC gives counter-examples.

2. If the monitor maintains the current verdict (?), for each successive input states two BMC calls are performed, returning $\bot$ (counter-example found).

3. After some inputs, at the moment when the monitor firstly returns $\top^{\mathrm{a}}$, we have $b_1 = \top$ and $b_2 = \bot$. IC3_IA is called once at line 9, BMC is called once (due to [o4]) at line 10.

4. If the monitor maintains the current verdict ($\top^{\mathrm{a}}$), IC3_IA will not be called again, because it is disabled by [o1] (at line 9) when $b_1 = \top$. (BMC is still called at line 10.)

5. After some inputs, at the moment when the monitor firstly returns $\times$, we have $b_1 = b_2 = \top$ (the value of b_2 changed). IC3_IA is called once again (at line 10).

6. From now on, no BMC nor IC3_IA is called, as they are all disabled by [o1], and the monitor maintains the verdict × (out of model).

Thus, in summary IC3_IA is called at most twice for any input trace. □

We found that IC3_IA is never called for $p\,\mathbf{U}\,q$, as in this case whenever the belief states are empty, they are literally unsatisfiable formulas. For LTL properties like $\mathbf{X}^2$false, the call to complete model checkers is inevitable, as in this case the involved belief states are only "empty" w.r.t. fairness.

Algorithm 9: The optimized version of Algorithm 7

```
1  function monitor2_optimized(K ≐ ⟨V_K, Θ_K, ρ_K, J_K⟩, φ, u)
2      T_φ  ≐ ⟨V_φ, Θ_φ , ρ_φ, J_φ⟩ := ltl_translation(φ);          // χ(φ) is in Θ_φ
3      T_¬φ ≐ ⟨V_φ, Θ_¬φ, ρ_φ, J_φ⟩ := ltl_translation(¬φ);
4      V := V_K ∪ V_φ;
5      ⟨r_φ, r_¬φ⟩ := ⟨Θ_K ∧ Θ_φ, Θ_K ∧ Θ_¬φ⟩;
6      if o_1 then b_1 := b_2 := ⊤ ;
7      if o_3 then F := ltl_translation((⋀_{ψ∈J_K∪J_φ} GF ψ) → false) ;
8      if |u| > 0 then
9          ⟨r_φ, r_¬φ⟩ := ⟨r_φ ∧ u_0, r_¬φ ∧ u_0⟩;
10     for 1 ⩽ i < |u| do
11         r_φ  := quantifier_elimination(V, ρ_K ∧ ρ_φ ∧ r_φ  ) ∧ u_i;
12         r_¬φ := quantifier_elimination(V, ρ_K ∧ ρ_φ ∧ r_¬φ) ∧ u_i;
13     if o_1 → b_1 then b_1 := check_nonemptiness(r_φ) ;
14     if o_1 → b_2 then b_2 := check_nonemptiness(r_¬φ) ;
15     if b_1 ∧ b_2 then return ? ;                                  // inconclusive
16     else if b_1 then return ⊤^a;                                  // conditionally true
17     else if b_2 then return ⊥^a;                                  // conditionally false
18     else return ×;                                                // out of model
19 function check_nonemptiness(r)
20     if o_2 ∧ (SMT(r) = unsat) then return ⊥ ;
21     else
22         return ¬model_checking(⟨V, r, ρ_K ∧ ρ_φ, J_K ∪ J_φ⟩, o_3 ? F : false)
23 function model_checking(M, ψ)
24     if o_4 then
25         if BMC(M, ψ) = ⊥ then return ⊥;                           // counter-example found
26         else                                                      // max_k reached
27             return IC3_IA(M, ψ)
28     else return IC3_IA(M, ψ);
```

One may think that the calls of complete model checkers (IC3_IA) are a bottleneck rendering

the whole idea infeasible. In fact, given all above optimizations we can prove that IC3_IA is called at most twice for each input trace:

Theorem 6.4.2. *Assuming* BMC *always find the counter-example whenever it exists,* IC3_IA *is called at most twice in the "online" version of Algorithm 9 with all optimizations.*

Proof. Without loss of generality, we analyze how the values of b_1 and b_2 change during the verification of a typical trace:

1. Initially $b_1 = b_2 = \top$ (so that the verdict is ?). This means that both calls of check_nonemptiness (at line 13–14) return $\top$, which further means that the underlying call to model_checking (line 22) returns $\bot$, i.e. BMC is involved returning $\bot$ (counter-examples found).

2. If the monitor maintains the current verdict (?), we have $b_1 = b_2 = \top$, and two BMC calls are performed, each returning $\bot$.

3. At the moment when the monitor firstly returns $\top^a$, we have $b_1 = \top$, $b_2 = \bot$, i.e. the call to check_nonemptiness at line 14 returns $\bot$. There are two possibilities:

   - The belief state $r_{\neg\varphi}$ is literally $\bot$ or unsatisfiable, detected by SMT (line 20) due to [o2]. No call to IC3_IA in this case.
   - The call to model_checking (line 22) returns $\top$, which means IC3_IA is called once (after BMC fails to find a counter-example.)

4. If the monitor maintains the current verdict ($\top^a$), IC3_IA will not be called again, because it is disabled by [o1] (at line 14) when $b_2 = \bot$.

5. At the moment when the monitor firstly returns $\times$, we have $b_1 = b_2 = \bot$ (the value of b_1 changed). check_nonemptiness returns $\bot$ is line 13. Either SMT is called (line 20) when r_φ is unsatisfiable (due to [o2]), or IC3_IA is called internally by model_checking (line 22) returning $\top$.

6. From now on, no BMC nor IC3_IA is called, as they are all disabled by [o1], and the monitor maintains the verdict $\times$ (out of model).

Thus, in summary IC3_IA is called at most twice for any input trace. □

6.5 Incremental Bounded Model Checking

Bounded Model Checking (BMC) [22, 23] is a SAT-based model checking technique. Given a FSM M and an LTL specification f, the idea is to look for counter-examples of maximum length

k, and to generate a Boolean formula which is satisfiable if and only if such counter-example exists. The Boolean formula is then given as input to a SAT solver (or SMT solver, if the theory domain is beyond propositional).

In theory, if the maximum length k has reached certain bound (depending on M), which is called the *diameter* of M [22], and there is still no counter-example found, then the whole model checking process can also be considered as completed with the conclusion that no counter-example exists. But such a bound is hard to compute and is ofter too large for practical problems. Thus, in practice BMC is usually used as a preliminary step before other unbounded MC procedures are involved.

With the above notations M and f, the BMC process can be formally described as a sequence of model checking problem $M \models_k \mathbf{E}f$, meaning "there exist an execution path of M of length k satisfying the temporal property f", where k ranges from 0 to a maximal given bound (whose value is given heuristically and is usually much smaller than the diameter of M). Roughly speaking, the checking of $M \models_k \mathbf{E}f$ is equivalent to the satisfiability problem of a Boolean formula $[[M, f]]_k$ defined as follows:

$$[[M, f]]_k := [[M]]_k \wedge [[f]]_k = I(V_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(V_i, V_{i+1}) \right) \wedge [[f]]_k \tag{6.4}$$

where $I(V_0)$ and $T(V_i, V_{i+1})$ represent the initial condition and transition relation of M, respectively, unrolled with state variables from certain discrete time. (Thus, for example, if the actual initial condition is $p \wedge q$ where p and q are state variables, then $I(V_0) := p_0 \wedge q_0$.) The term $[[f]]_k$ is called the *ending term*, which, besides representing the paths which satisfy f, also induces loops from the current step k back to all previous steps. The precise form of $[[f]]_k$ is not relevant here and has several versions [45].

Note that, if $[[M]]_k$ for certain k is already unsatisfiable (i.e. the input model M is actually empty), then there is no need to do SMT checking on $[[M]]_k \wedge [[f]]_k$, which must ba also unsatisfiable. Furthermore, there is also no need to do next rounds with bigger k values.

Now consider the following question of Incremental BMC:

Question 6.5.1 (Incremental BMC)**.** *If a previous BMC process (on M, f) has completed at certain value of k, say $k = k_0$ (assuming $k_0 > 0$), with a counter-example found, and then the model M is* updated *to M' by having one or more* observations *at certain time, does the new BMC process for M', f need to restart from $k = 0$?*

The answer is no. To explain this answer, first we must clarify what an *observation* (also called *step constraint*) is:

Definition 6.5.2 (Step constraints)**.** *A step constraint (aka* observation*) of the model*

$$M \doteq \langle V, I(V), T(V, V'), \mathcal{J} \rangle$$

is a pair $\langle t, s(V)\rangle$, where t is an integer indicating discrete time, and $s(V)$ is a formula of V. The model M updated with $\langle t, s(V)\rangle$, denoted by $M + \langle t, s(V)\rangle$, is a new model M' defined below:

$$\begin{aligned} M' \doteq \langle & V \cup \{c\}, \\ & I(V), \\ & T(V, V') \wedge (c' = \min\{c+1, t+1\}) \wedge ((c = t) \rightarrow s(V)), \\ & \mathcal{J} \rangle \end{aligned}$$

where $c \notin V$ is a fresh variable used as a counter, whose finite domain is from 0 to $t+1$. The initial value of c is 0, and is increased by one at each new step, until reaching $t+1$. The sole purpose of this counter is to make sure that, when its value is t, there is an extra step constraint $s(V)$ in addition to the existing transition relation $T(V, V')$ at time t.

With the above observation $\langle t, s(V)\rangle$, if $\mathcal{L}(M')$ is the language of M', then it is not hard to see that, for all infinite traces $\mu \in \mathcal{L}(M')$, the index t of the trace μ as a formula must be compatible with $s(V)$, i.e. $\mu_t \models s(V)$.

Furthermore, if the orignal model M is updated by multiple observations at different time, such as $\langle t_1, s_1(V)\rangle$, $\langle t_2, s_2(V)\rangle$, etc., then we also have $\mu_{t_1} \models s_1(V)$, $\mu_{t_2} \models s_2(V)$, etc., if μ is a trace of the updated model, no matter what the order of updates is.

To answer Question 6.5.1, let us first consider the case of just one observation: $M' = M + \langle t, s(V)\rangle$. Suppose the previous BMC process terminates at k_0 (less than the maximal bounds), then this means that a counter-example was found when doing SMT checking of $[[M, f]]_{k_0}$ (see Equation 6.4), which is *satisfiable*, while for all $k < k_0$, $[[M, f]]_k$ is *unsatisfiable*.

Now consider the form of $[[M', f]]_k$, where $M' = M + \langle t, s(V)\rangle$ and $k \leqslant k_0$. There are two possible cases:

1. If $t \leqslant k_0$, then $[[M']]_k$ and $[[M]]_k \wedge s(V_t)$ are equi-satisfiable, where $s(V_t)$ occurs either in the initial condition ($t = 0$) or the transition relation ($t > 0$) of M',

2. If $t > k_0$, then $[[M', f]]_k$ is unsatisfiable due to the domain of counter variable c.

For any $k < k_0$, if $[[M, f]]_k$ is unsatisfiable, then obviously $[[M', f]]_k$ is also unsatisfiable, thus there is no need to retry $k < k_0$ for the BMC process of M', f. On the other side, for $k = k_0$, as we know that $[[M, f]]_{k_0}$ is satisfiable, but after pushing one more term $s(V_t)$, the resulting new form $[[M', f]]_{k_0}$ may become unsatisfiable, thus the BMC process of M', f should retry $k = k_0$ and also $k+1$, $k+2$, until the maximal BMC bound.

Thus we have the following result for doing the BMC checking of $[[M', f]]_k$ incrementally:

Lemma 6.5.3 (Incremental BMC)**.** *The BMC process for M', f, if done incrementally from the BMC process of M, f, should start from $[[M', f]]_{k_0}$ (the accumulated formula left by previous*

BMC process) but may need to keep unrolling until reaching $[[M', f]]_{\max(k_0,t)}$*, which contains the current observation* $\langle t, s(V)\rangle$.

If there are more than one observations, it is not hard to see that, the incremental BMC process should keep unrolling until *all* observations are presented in the SMT formula, i.e. the minimal BMC starting position (i.e. where the SMT checking starts to happen) is the maximal time of all observation. On the other hand, if any observation falsifies the updated model, i.e. by making it an empty model. To skip the entire new BMC process, for each BMC round, an SMT checking can be done first without ending term, to trigger *early termination* of the entire BMC process.

In the next section, we show how this idea can be used to further optimize our ABRV monitoring algorithm of infinite-state systems.

6.6 ABRV with Incremental BMC

Continue with Section 6.4, further optimizations can be done by leveraging Incremental BMC, the idea has been discussed in the previous section, in Algorithm 9, where the function `BMC` has already been used as incomplete preliminary step before the full `IC3_IA` calls.

To see how the Incremental BMC is used in ABRV algorithms, first notice that, in Algorithm 9, all models used for model checking in `check_nonemptiness` (line 22) only differ at the initial condition r, while the transition relation $\rho_K \wedge \rho_\varphi$ and fairness components $\mathcal{J}_K \cup \mathcal{J}_\varphi$ never change. But even the initial condition does not change arbitrarily: for the (non)emptiness checking of each side (r_φ and $r_{\neg\varphi}$, see line 13 and 14) the next version of r_φ (or $r_{\neg\varphi}$) is nothing but the forward image of its current version, intersected with the next observation (from the input trace). This leave us the possibility to just assert the new observation on the BMC encoding (as unrolling of initial condition and transition relations) left by the previous BMC call, to get the *equisatisfiable* BMC encoding for the subsequent BMC calls.

We first define a BMC encoding of the belief states after a sequence of observations $u_0u_1\cdots u_n$, denoted by $\mathrm{bs}(u_0u_1\cdots u_n)$. These are inductively given by

$$\mathrm{bs}(u_0)(V) = I(V) \wedge u_0(V), \tag{6.5}$$

$$\mathrm{bs}(u_0u_1\cdots u_{i+1})(V) = \mathrm{fwd}\big(\mathrm{bs}(u_0u_1\cdots u_i)(V), T(V,V')\big)(V) \wedge u_{i+1}(V)\ . \tag{6.6}$$

The following theorem shows the relation between the belief states and a BMC encoding conjoined with the sequence of observations:

Theorem 6.6.1 (Equisatisfiability)**.** *When* $k > 1$*, the following two formulas*

$$I(V_0) \wedge u_0(V_0) \wedge \bigwedge_{j=0}^{k-1} \big[T(V_j, V_{j+1}) \wedge u_{j+1}(V_{j+1})\big], \tag{6.7}$$

and

$$\mathrm{bs}(u_0 u_1 \cdots u_k)(V) \tag{6.8}$$

are equisatisfiable.

The proof of Theorem 6.6.1 derives by repeatedly applying the following Lemma 6.6.2:

Lemma 6.6.2 (Equisatisfiability)**.** *The following three formulas*

$$I(V_0) \wedge u_0(V_0) \wedge \bigwedge_{j=0}^{k-1} \big[T(V_j, V_{j+1}) \wedge u_{j+1}(V_{j+1})\big], \tag{6.9}$$

and

$$\mathrm{bs}(u_0)(V_0) \wedge \bigwedge_{j=0}^{k-1} \big[T(V_j, V_{j+1}) \wedge u_{j+1}(V_{j+1})\big], \tag{6.10}$$

and (if $k > 1$)

$$\mathrm{bs}(u_0 u_1)(V_1) \wedge \bigwedge_{j=1}^{k-1} \big[T(V_j, V_{j+1}) \wedge u_{j+1}(V_{j+1})\big] \tag{6.11}$$

are equi-satisfiable.

Proof of Lemma 6.6.2. It is obvious that (6.9) and (6.10) are equivalent (using (6.5)) thus also equisatisfiable. Below we show that (6.10) and (6.11) are equisatisfiable. Actually it is sufficient to only show that the following two sub-formulas:

$$A(V_0, V_1) \doteq \mathrm{bs}(u_0)(V_0) \wedge T(V_0, V_1) \wedge u_1(V_1), \tag{6.12}$$

$$B(V_1) \doteq \mathrm{bs}(u_1)(V_1) \tag{6.13}$$

are equisatisfiable. (They cannot be just equivalent because A contains more free variables then B.)

Expand $B(V_1)$ by definition of bs (Eqs 6.5 and 6.6) and fwd (Eq 3.1):

$$\begin{aligned} B(V_1) &= \mathrm{fwd}\big(\mathrm{bs}(u_0)(V), T(V, V')\big)(V_1) \wedge u_1(V_1) \\ &= (\exists V.\ T(V, V') \wedge \mathrm{bs}(u_0)(V))[V_1/V'] \wedge u_1(V_1) \\ &= (\exists V.\ T(V, V_1) \wedge \mathrm{bs}(u_0)(V)) \wedge u_1(V_1) \\ &= (\exists V_0.\ T(V_0, V_1) \wedge \mathrm{bs}(u_0)(V_0)) \wedge u_1(V_1) \quad (\alpha\text{-conversion}) \end{aligned}$$

Now it is clear that A and B are equisatisfiable:

$$\begin{aligned} \exists V_1.\ B(V_1) &= \exists V_1.\ (\exists V_0.\ T(V_0, V_1) \wedge \mathrm{bs}(u_0)(V_0)) \wedge u_1(V_1) \\ &= \exists V_0, V_1.\ T(V_0, V_1) \wedge \mathrm{bs}(u_0)(V_0) \wedge u_1(V_1) \\ &= \exists V_0, V_1.\ \mathrm{bs}(u_0)(V_0) \wedge T(V_0, V_1) \wedge u_1(V_1) \\ &= \exists V_0, V_1.\ A(V_0, V_1)\ . \end{aligned}$$

Thus (6.9), (6.10) and (6.11) are all equisatisfiable. □

Now comes the second part of this idea: there is also no need to restart BMC inner loop from 0 (to the maximal bound k) after asserting a new observation. This is because, whenever the BMC inner loop stops at a value k in the previous call, all SMT formulas corresponding in steps $i < k$ are UNSAT, and they are still UNSAT after asserting anything new. In the ideal case (when BMC stopped by having found a counter-example, and the overall monitoring verdicts is conclusive), the monitor only needs to call SMT solver *once* to decide the next monitoring output.

In Algorithm 10 we gave the pseudo code of the optimized RV monitor based on incremental BMC. There are several undefined functions (methods) used here (to be given later in Algorithm 11 and 12):

- `init_nonemptiness` for creating a persistent SMT solver instance,
- `update_nonemptiness` for checking the non-emptiness of the belief states after a new observation,
- `reset_nonemptiness` for resetting the SMT solver, cleaning up all existing observations.

Here the code is given in object-oriented styles, with two instances of SMT solvers created by `init_nonemptiness`. Others methods operates on these instances, possibly with further arguments.

The correctness of Algorithm 10 (relative to the correctness of undefined methods) can be seen by a comparison with Algorithm 7. Now the computation of belief states from a sequence of observations is done in a new function `compute_belief_states()` on the object, which holds a sequence of observations asserted by each call of `update_nonemptiness`.

Algorithm 10: The optimized RV monitor based on Incremental BMC (toplevel)

```
1  function bmc_monitor(K ≐ ⟨V_K, Θ_K, ρ_K, J_K⟩, φ, u, max_k, window_size)
2      T_φ  ≐ ⟨V_φ, Θ_φ  , ρ_φ, J_φ⟩ := ltl_translation(φ);        // χ(φ) is in Θ_φ
3      T_¬φ ≐ ⟨V_φ, Θ_¬φ, ρ_φ, J_φ⟩ := ltl_translation(¬φ);
4      V := V_K ∪ V_φ;
5      e_1 := init_nonemptiness(Θ_K ∧ Θ_φ , ρ_K ∧ ρ_φ);
6      e_2 := init_nonemptiness(Θ_K ∧ Θ_¬φ, ρ_K ∧ ρ_φ);
7      if |u| > 0 then
8          b_1 := update_nonemptiness(e_1, u_0);
9          b_2 := update_nonemptiness(e_2, u_0);
10     for 1 ⩽ i < |u| do
11         b_1 := update_nonemptiness(e_1, u_i);
12         b_2 := update_nonemptiness(e_2, u_i);
13     if b_1 ∧ b_2 then return ? ;                          // inconclusive
14     else if b_1 then return ⊤^a;                          // conditionally true
15     else if b_2 then return ⊥^a;                          // conditionally false
16     else return ×;                                        // out of model
17 function compute_belief_states(e)
18     r := e.I(V);
19     for i ← 0 to e.n do
20         if i = 0 then  r := r ∧ e.observations[i](V);
21         else
22             r := quantifier_elimination(V, r ∧ T(V, V')) ∧ e.observations[i](V);
23     return r;
```

In Algorithm 11 the code of `init_nonemptiness` and `reset_nonemptiness` are given. Note that, although new BMC solver instances are created from just the initial condition and transition relation for simplification purposes, the actual code also needs the translation of LTL property $\left(\bigwedge_{\psi\in\mathcal{J}_K\cup\mathcal{J}_\varphi} \mathbf{GF}\,\psi\right) \rightarrow \text{false}$ as in Algorithm 9. The unrolling of this translated formula at time i, as the ending terms of BMC encodings, will be simply presented as $[[F]]_i$ in the related code (`update_nonemptiness`). The BMC solver object has some extra member variables, whose purposes are given in the comments of `reset_nonemptiness`. Whenever SMT solving is needed, it is done on the member variable $problem$.

Algorithm 11: Methods for Checking (Non)emptiness (Part 1)

```
1  function init_nonemptiness(I, T)
2      e := new BMC solver with initial formula I and transition relation T;
3      reset_nonemptiness(e, I);
4      return e;
5  procedure reset_nonemptiness(e, I)
6      e.problem := I(V_0);          // the initial formula unrolled at time 0
7      e.observations := [];                  // an array holding observations
8      e.n := 0;                                 // the number of observations
9      e.map := {};          // a hash map from time to (unused) observations
10     e.k := 0;               // the number of unrolled transition relations
11     e.min_k := 0;   // internal parameter, to be updated by e.observations
12     e.max_k := max_k;       // a local copy of external parameter max_k
```

The core of incremental BMC algorithm for RV, `update_nonemptiness`, is finally given in Algorithm 12. The overall structure of this algorithm is essentially the Incremental Bounded Model Checking algorithm described in Section 6.5, plus some further treatments when BMC search bound has been reached: in the latter case the belief states must be re-calculated from the last version accumulating all new observations, and is then sent to the IC3-based model checker, represented by the function `IC3_IA()`. In particular, note that there are both *min_k* and *max_k* being updated each time when the function `update_nonemptiness()` is called. The purposes of *min_k* is to guarantee all input observations are taken into consideration (otherwise the monitoring outputs may be wrong), while *max_k* is the BMC search bound, which must be enlarged by one for the next calls of `update_nonemptiness()` so that there be always enough times of unrolling (equals to the initial BMC bound) for each new monitor input when searching for counter-examples. Finally, an independent parameter *window_size* (usually set to twice of the initial value of *max_k*) sets the hard limit of the total number of unrolling of the SMT formula being tested in the BMC procedure. This is useful in practice when BMC becomes too slow due to the over long size of SMT formulas.

Remark 6.6.3. *Note some red-marked code in the pseudo code, they are the bugfix of this algorithm after the initial publication in RV 2021 [50]. The bug is due to lacking consideration of min_k, which caused some input observations beyond the current k in BMC loop not being considered in the found counter-examples, which leads to wrong monitor outputs.*

Algorithm 12: Methods for Checking (Non)emptiness (Part 2)

```
1  function update_nonemptiness(e, o)
2      e.map[e.n] := o;                          // store new observation in the map
3      e.n := e.n + 1;
4      e.observations[e.n] = o;                  // store new observation in the list
5      for (k, v) : e.map do
6          if k ⩽ e.k then
7              e.problem := e.problem ∧ v(V_i);
8              delete e.map[k];
9          if k > e.min_k then
10             e.min_k := k;                     // set to the maximal time of observations
11     result := ?;
12     while e.k ⩽ e.max_k and result = ? do
13         i := e.k;
14         if SMT(e.problem) = UNSAT then
15             result := ⊥;                      // literally empty believe states
16             break
17         if e.k ⩾ e.min_k and SMT(e.problem ∧ [[F]]_i) = SAT then
18             result = ⊤;                       // counter-example found (nonempty)
19             break
20         e.problem := e.problem ∧ e.T(V_i, V_{i+1});
21         if e.map[i + 1] exists then
22             e.problem := e.problem ∧ e.map[i + 1](V_{i+1});
23             delete e.map[i + 1];
24         e.k := e.k + 1;
25     e.max_k := e.max_k + 1;   // increase the search bound for next calls
26     if e.k > window_size or result = ? then
27         r := compute_belief_states(e);
28         reset_nonemptiness(e, r);   // prepare for complete model checking
29     if result = ⊤ or result = ⊥ then
30         return result;
31     else
32         return ¬IC3_IA(⟨V, r, e.T, J_K ∪ J_φ⟩, false);
```

6.7 Unboundedness of Infinite-State Monitors

Now we show the unboundedness of infinite-state monitors. This answers the question of constructing infinite-state monitors (in some paper submissions, nowhere to cite), which claims to be trace-length independent.

The symbolic ABRV monitors of finite-state systems constructed in Chapter 5 has bounded memory consumptions, for reasons below:

1. The monitor never needs to look back to the previous input states of the input trace. Instead, at each cycle the monitor only needs the current state of the trace, besides other internal states, to decide the current monitor verdict.

2. Those "other internal states" are fixed number of variables hold BDDs of fixed number of Boolean variables, where the number is a function of combined size of RV assumptions and the monitoring property. Thus the maximal memory consumptions of all such variables must be bounded.

Not every RV work can guarantee the above two points. And it would be nice if a monitor can run for a long period without blowing up the memories (or disk space) of the hosting computer. For ABRV monitors of infinite-state systems, believe states are now represented by raw formulas. After forward image computations by quantifier elimination procedure, the resulting new formulas may grow in size, and this size is in general unbounded.

In fact, it is hopeless to have a "better" monitoring algorithm which consumes only bounded memories in our general ABRV setting. Below we construct an counter-example to show that constant-memory monitor does not exist for a simple monitoring property under a special RV assumption, based on *Cantor's Ternary Set*[1].

The RV assumption (as a symbolic model) has two real variable a and b, initially $a = 0$, $b = 1$. Imagine they represent a closed interval $[a, b]$, which is to be divided into the union of 3 closed intervals with equal lengths:

$$[a, a + (b - a)/3], \quad [a + (b - a)/3, a + 2(b - a)/3], \quad [a + 2(b - a)/3, b]$$

At next discrete time, the new values of a, b, denoted by a' and b', are either the ends of 1st or the ends of 3rd intervals, i.e.

$$(a' = a \wedge b' = a + (b - a)/3) \vee (a' = a + 2(b - a)/3 \wedge b' = b)$$

Thus, we can observe the following patterns:

1. At time 0, there is only one interval $[0, 1]$;

[1] https://en.wikipedia.org/wiki/Cantor_set

2. At time 1, there are two intervals: $[0, \frac{1}{3}]$ and $[\frac{2}{3}, 1]$;
3. At time 2, there are four intervals: $[0, \frac{1}{9}]$, $[\frac{2}{9}, \frac{1}{3}]$, $[\frac{2}{3}, \frac{7}{9}]$, $[\frac{8}{9}, 1]$.
4. At time 3, there are 8 intervals: $[0, \frac{1}{27}]$, $[\frac{2}{27}, \frac{1}{9}]$, $[\frac{2}{9}, \frac{7}{27}]$, $[\frac{8}{27}, \frac{1}{3}]$, $[\frac{2}{3}, \frac{18}{27}]$, $[\frac{20}{27}, \frac{7}{9}]$, $[\frac{8}{9}, \frac{25}{27}]$, $[\frac{26}{27}, 1]$;
5. At time 4, there are 16 intervals: ...

The above mentioned intervals are actually possible intervals where we can find another particular value x, which is another input variable of the model. Here the monitoring property is simply this: $\mathbf{G}\,\neg(a \leqslant x \wedge x \leqslant b)$, i.e. x is NOT in any of the intervals.

If a monitor were built from the above property and assumptions, one can see that, for each new discrete time there must be doubled memories to hold a doubled amount of end points of all the possible (disjoint) intervals in forms of $[a, b]$, and none of them can be pruned or simplified away. Thus the monitor cannot have bounded or constant memory uses.

Chapter 7

Monitoring ptLTL (Past-Time LTL)

7.1 Introduction

Runtime Verification is commonly restricted to *Past-time LTL* (or *ptLTL*), i.e. LTL with only past operators [86]. The ptLTL is supported by many RV tools (some of these tools also support future-time LTL), such as Java PathExplorer [86, 87], RuleR [12], JavaMOP [40], RV-Monitor[1] [114], TraceContract [10], DejaVu [89] and R2U2 [131]. We show that ptLTL can also be monitored in our ABRV framework through a connection of ptLTL with LTL_3 semantics.

There are two slightly different ptLTL semantics found in the literature. The standard one, which is followed by most RV tools mentioned above, is neither a fragment of LTL nor LTL_3 semantics:

Definition 7.1.1 (ptLTL semantics)**.** *Let φ, ψ be ptLTL formulae and $p \in AP$ (atomic propositions), the semantics of φ with respect to a finite word $u = s_0 \cdots s_{n-1}$ (thus $|u| = n$), denoted by $[\![u \models_{\mathrm{p}} \varphi]\!]$ hereafter, is inductively defined as follows: ($u|_i = s_0 s_1 \cdots s_{i-1}$, $i \leqslant n$, denotes the prefix of u with the first i states.)*

$$
\begin{array}{ll}
u \models_{\mathrm{p}} \mathrm{true} & \\
u \models_{\mathrm{p}} p & \textit{iff } p \in s_{n-1} \\
u \models_{\mathrm{p}} \neg\varphi & \textit{iff } u \not\models_{\mathrm{p}} \varphi \\
u \models_{\mathrm{p}} \varphi \vee \psi & \textit{iff } u \models_{\mathrm{p}} \varphi \textit{ or } u \models_{\mathrm{p}} \psi \\
u \models_{\mathrm{p}} \mathbf{Y}\varphi & \textit{iff } u|_{n-1} \models_{\mathrm{p}} \varphi \textit{ if } n > 1 \textit{ or } u \models_{\mathrm{p}} \varphi \textit{ if } n = 1 \\
u \models_{\mathrm{p}} \varphi\, \mathbf{S}\, \psi & \textit{iff for some } j \leqslant n, u|_j \models_{\mathrm{p}} \psi \textit{ and for all } j < i \leqslant n, u|_i \models_{\mathrm{p}} \varphi
\end{array}
$$

The key is at the semantics of **Y**, which is "consistent with the view that a trace consisting of exactly one state s is considered like a *stationary* infinite trace containing only the state s." [86,

[1] The ptLTL plugin of RV-Monitor comes from JavaMOP.

p. 345] For instance, with the above definition we have $[\![\{p\} \models_{\mathrm{p}} \mathbf{Y}^2\, p\,]\!] = [\![\{p\} \models_{\mathrm{p}} \mathbf{Y}\, p\,]\!] = [\![\{p\} \models_{\mathrm{p}} p\,]\!] = \top$. In this case, we may understand $u = \{p\}$ as an *backward* infinite word $w = \cdots s_{-2}s_{-1}s_0 = \cdots\{p\}\{p\}\{p\}$, and therefore $w \models \mathbf{Y}\, p$ iff $p \in s_{-1}(= s_0)$, while $w \models \mathbf{Y}^2\, p$ iff $p \in s_{-2}(= s_0)$.

Alternatively, one could consider that $\mathbf{Y}\, p$ is false on a one-state trace for any atomic proposition p [84, p. 98], as follows:

Definition 7.1.2 (alternative ptLTL semantics)**.** *Following the same notation, the* alternative semantics *of ptLTL formula φ with respect to a finite word u, denoted by $[\![u \models_{\mathrm{p}'} \varphi]\!]$, is inductively defined as in Definition 7.1.1, except for* $\mathbf{Y}$ *operator:*

$$u \models_{\mathrm{p}'} \mathbf{Y}\varphi \quad \textit{iff } n > 1 \textit{ and } u|_{n-1} \models_{\mathrm{p}'} \varphi$$

With this alternative definition we have $[\![\{p\} \models_{\mathrm{p}} \mathbf{Y}^2\, p\,]\!] = [\![\{p\} \models_{\mathrm{p}} \mathbf{Y}\, p\,]\!] = \bot$. In this case, we may understand $u = \{p\}$ as an infinite word $w = \cdots s_{-2}s_{-1}s_0 = \cdots \emptyset \cdot \emptyset \cdot \{p\}$, and therefore $w \not\models \mathbf{Y}\, p$ as $p \notin s_{-1}(= \emptyset)$, while $w \not\models \mathbf{Y}^2\, p$ as $p \notin s_{-2}(= \emptyset)$.

7.2 Connection with LTL$_3$ Semantics

The corresponding RV problem (under full observability, without assumptions) for ptLTL can be resolved by rewriting-based [130], rule-based [11] approaches, or dynamic programming [86]. But none of the existing work is based on well-known tableau translations from LTL to Büchi automata.

ABRV framework provides a *unified* approach, which can handle ptLTL as a side effect. Now it is possible to generate an automaton - internally it is symbolic and is implemented by BDDs - monitoring ptLTL, just like the automata generated from normal LTL. The key is to convert ptLTL semantics to LTL$_3$ semantics, and then make sure that the monitor is reset before walking into next positions in the (finite-state) monitor automaton. The code generation techniques represented in Section 5.4 can be also used for generating monitor code synthesized from ptLTL.

The following lemma and theorem show the connection between (alternative) ptLTL and LTL$_3$ (without future temporal operators) semantics:

Lemma 7.2.1. *The LTL$_3$ semantics of any ptLTL formula φ with respect to any non-empty finite trace u (thus $|u| > 0$) is always conclusive:*

$$\forall i < |u|.\ [\![u, i \models \varphi\,]\!]_3 = \top \textit{ or } \bot.$$

Proof. The original proof goal can be reduced (by LTL$_3$ semantics) to the following one: the standard LTL semantics is the same for any two continuations of u, i.e., for any $w, w' \in (2^{AP})^\omega$

we have

$$\forall i < |u|.\ u \cdot w, i \models \varphi \Leftrightarrow u \cdot w', i \models \varphi \tag{7.1}$$

The above goal can be proven by an outer complete induction on i, i.e., by showing (7.1) holds for any $i < |u|$ assuming it holds for all $j < i$, with an inner induction on the structure of φ for each ptLTL operator. The complete proof of this lemma is trivial but tedious. It has been formalized in HOL Theorem Prover (See Section 10.5 for more details.) □

Theorem 7.2.2. *The alternative semantics of any ptLTL formula φ with respect to any non-empty finite trace u can be expressed by LTL_3 semantics, i.e.,*

$$[\![u \models_{p'} \varphi]\!] = [\![u, |u| - 1 \models \varphi]\!]_3 \tag{7.2}$$

Similarily with the proof of Lemma 7.2.1, the above theorem can be proven by an outer complete induction on the length of u, i.e., we can prove (7.2) assuming it holds for all v such that $|v| < |u|$ (but v does not need to be a subtrace of u), with an inner induction on the structure of φ for each ptLTL operator. For atomic propositions, $\neg$ and $\vee$, the outer induction hypothesis is not used. Non-trivial cases are for the temporal operators $\mathbf{Y}$ and $\mathbf{S}$. Lemma 7.2.1 is repeatedly used during the proof.

Proof of Theorem 7.2.2. We prove (7.2) by a complete induction on the length of u ($|u| > 0$). Fixing u (and let $n = |u|$ hereafter), the induction hypothesis is the following (ψ is any ptLTL formula):

$$\forall v, \psi.\ 0 < |v| < n \Rightarrow [\![v \models_{p'} \psi]\!] = [\![v, |v| - 1 \models \psi]\!]_3 \tag{7.3}$$

Now the only free variable of (7.2) is φ, we further do an induction on the structure of φ. In other words, if we see (7.2) as $P(\varphi)$, to prove it we need to prove the following subgoals instead (all involved LTL formulae are ptLTL formulae, i.e. having no future operator):

1. $P(\text{true})$ and $\forall p \in AP.\ P(p)$;
2. $\forall \psi.\ P(\psi) \Rightarrow P(\neg\psi)$;
3. $\forall \psi_1, \psi_2.\ P(\psi_1) \wedge P(\psi_2) \Rightarrow P(\psi_1 \vee \psi_2)$;
4. $\forall \psi.\ P(\psi) \Rightarrow P(\mathbf{Y}\,\psi)$;
5. $\forall \psi_1, \psi_2.\ P(\psi_1) \wedge P(\psi_2) \Rightarrow P(\psi_1\,\mathbf{S}\,\psi_2)$.

The proofs of the first three subgoals are straightforward, where the induction hypothesis (7.3) is not used. We focus on the last two subgoals about the past temporal operators.

Case 1 (Y operator) Here the goal is to prove

$$[\![u \models_{\mathrm{p}'} \mathbf{Y}\psi]\!] = [\![u, n-1 \models \mathbf{Y}\psi]\!]_3 \tag{7.4}$$

assuming (7.3), and $[\![u \models_{\mathrm{p}'} \psi]\!] = [\![u, n-1 \models \psi]\!]_3$ (but this latter fact is not used).

If $n = 1$, (7.4) can be directly proved, as both its left and right sides are $\bot$ by LTL, LTL$_3$ and (alternative) ptLTL semantics. If $n > 1$, by Definition 7.1.2 the goal (7.4) can be reduce to

$$[\![u|_{n-1} \models_{\mathrm{p}'} \psi]\!] = [\![u, n-1 \models \mathbf{Y}\psi]\!]_3$$

and further (by induction hypothesis (7.3), taking $v = u|_{n-1}$) to

$$[\![u|_{n-1}, n-2 \models \psi]\!]_3 = [\![u, n-1 \models \mathbf{Y}\psi]\!]_3 \tag{7.5}$$

By Lemma 7.2.1, $[\![u_{n-1}, n-2 \models \psi]\!]_3$ and $[\![u, n-1 \models \mathbf{Y}\psi]\!]_3$ can only be $\top$ or $\bot$, thus there are four combinations. When they have the same value, the proof is finished; It remains to show that the other two subcases where they have different values, are actually impossible. For instance, suppose $[\![u_{n-1}, n-2 \models \psi]\!]_3 = \top$ and $[\![u, n-1 \models \mathbf{Y}\psi]\!]_3 = \bot$, by LTL and LTL$_3$ semantics we have

$$\forall w.\ u|_{n-1} \cdot w, n-2 \models \psi \tag{7.6}$$

$$\forall w'.\ u \cdot w', n-1 \not\models \mathbf{Y}\psi \quad \text{or} \quad \forall w'.\ u \cdot w', n-2 \not\models \psi \tag{7.7}$$

Choosing $w = s_{n-1} \cdot w'$ (for any w', e.g. $\emptyset^\omega = \emptyset \cdots$) in (7.6), we have $u|_{n-1} \cdot w = (u|_{n-1} \cdot s_{n-1}) \cdot w' = u \cdot w'$ and thus $u \cdot w', n-2 \models \psi$ but this is conflict with (7.7). The other subcase is similar.

Case 2 (S operator) Here the goal is to prove

$$[\![u \models_{\mathrm{p}'} \psi_1 \mathbf{S} \psi_2]\!] = [\![u, n-1 \models \psi_1 \mathbf{S} \psi_2]\!]_3 \tag{7.8}$$

assuming the *outer* induction hypothesis (7.3) and the following *inner* induction hypothesises:

$$[\![u \models_{\mathrm{p}'} \psi_1]\!] = [\![u, n-1 \models \psi_1]\!]_3 \tag{7.9}$$

$$[\![u \models_{\mathrm{p}'} \psi_2]\!] = [\![u, n-1 \models \psi_2]\!]_3 \tag{7.10}$$

Again, by Lemma 7.2.1 all involved LTL$_3$ semantics are conclusive. By Definition 7.1.2 the goal (7.8) can be reduced to

$$\big(\exists j.\ j \leqslant n \ \wedge\ u|_j \models_{\mathrm{p}'} \psi_2 \ \wedge\ \forall i.\ j < i \leqslant n \Rightarrow u|_i \models_{\mathrm{p}'} \psi_1\big) = [\![u, n-1 \models \psi_1 \mathbf{S} \psi_2]\!]_3 \tag{7.11}$$

Depending on the value of $[\![u, n-1 \models \psi_1 \mathbf{S} \psi_2]\!]_3$, the above proof goal can be further reduced to one of the following two subgoals:

Case 2.1 ($[\![u, n-1 \models \psi_1 \mathbf{S} \psi_2]\!]_3 = \top$) In this case, we need to prove

$$\exists j.\ j \leqslant n \ \wedge\ u|_j \models_{\mathrm{p}'} \psi_2 \ \wedge\ \forall i.\ j < i \leqslant n \Rightarrow u|_i \models_{\mathrm{p}'} \psi_1 \tag{7.12}$$

assuming (7.3), (7.9), (7.10), and, by LTL$_3$ semantics, $\forall w.\ u \cdot w, n-1 \models \psi_1 \mathbf{S} \psi_2$, or equivalently

$$\begin{aligned} \forall w.\ \exists k.\ k \leqslant n-1 \ \wedge\ u \cdot w, k \models \psi_2 \ \wedge \\ \forall l.\ k < l \leqslant n-1 \Rightarrow u \cdot w, l \models \psi_1 \end{aligned} \tag{7.13}$$

Let $w = \emptyset^\omega$ in (7.13), then k is fixed, such that $k \leqslant n-1$ and

$$u \cdot \emptyset^\omega, k \models \psi_2 \tag{7.14}$$

$$\forall l.\ k < l \leqslant n-1 \Rightarrow u \cdot \emptyset^\omega, l \models \psi_1 \tag{7.15}$$

Now we choose $j = k+1$ in (7.12) Thus $j \leqslant n$, and it remains to prove $u|_{k+1} \models_{\mathrm{p}'} \psi_2$ and $\forall i.\ k+1 < i \leqslant n \Rightarrow u|_i \models_{\mathrm{p}'} \psi_1$, respectively (the order is not important):

- To prove $u|_{k+1} \models_{\mathrm{p}'} \psi_2$, the special case $k = n-1$ (thus $u|_{k+1} = u|_n = u$) must be eliminated first. This can be done by using (7.10), (7.14), Lemma 7.2.1 and LTL$_3$ semantics.

  If $k < n-1$ (thus $k+1 < n$), let $v = u|_{k+1}, \psi = \psi_2$ in (7.3) and we have

  $$[\![u|_{k+1} \models_{\mathrm{p}'} \psi_2]\!] = [\![u|_{k+1}, k \models \psi_2]\!]_3 \tag{7.16}$$

  Now we show that $[\![u|_{k+1}, k \models \psi_2]\!]_3$ cannot be $\bot$ (by Lemma 7.2.1 it also cannot be ?), because if it is, by LTL$_3$ semantics we have $\forall w.\ u|_{k+1} \cdot w, k \not\models \psi_2$. Taking $w = s_{k+1} \cdots s_{n-1} \cdot \emptyset^\omega$ (or $w = s_{n-1} \cdot \emptyset^\omega$ if $k+1 = n-1$), we have $u|_{k+1} \cdot w = u \cdot \emptyset^\omega$ and thus $u \cdot \emptyset^\omega, k \not\models \psi_2$, which conflicts with (7.14). Thus $[\![u|_{k+1}, k \models \psi_2]\!]_3 = \top$ and we have $u|_{k+1} \models_{\mathrm{p}'} \psi_2$ by (7.16).

- To prove $u|_i \models_{\mathrm{p}'} \psi_1$ for all i such that $k+1 < i \leqslant n$ (thus $k < n-1$), the special case $i = n$ (thus $u|_i = u|_n = u$) must be eliminated first. This can be done by using (7.9), (7.15) (taking $l = n-1$), Lemma 7.2.1 and LTL$_3$ semantics.

  If $k+1 < i < n$, let $v = u|_i, \psi = \psi_1$ in (7.3) and we have

  $$[\![u|_i \models_{\mathrm{p}'} \psi_1]\!] = [\![u|_i, i-1 \models \psi_1]\!]_3 \tag{7.17}$$

  Similarly with the above item, we can show that $[\![u|_i, i-1 \models \psi_1]\!]_3$ cannot be $\bot$, which conflicts with (7.15) (taking $l = i-1$). By Lemma 7.2.1 it also cannot be ?. Thus it must be $\top$ and we have $u|_i \models_{\mathrm{p}'} \psi_1$ by (7.17).

Case 2.2 ($[\![u, n-1 \models \psi_1 \mathbf{S} \psi_2]\!]_3 = \bot$) This case is slightly more tricky, as we need to prove (cf. (7.11))

$$\neg(\exists j.\ j \leqslant n \ \wedge\ u|_j \models_{\mathrm{p}'} \psi_2 \ \wedge\ \forall i.\ j < i \leqslant n \Rightarrow u|_i \models_{\mathrm{p}'} \psi_1)$$

or equivalently, for any $j \leqslant n$,

$$u|_j \models_{\mathrm{p}'} \psi_2 \Rightarrow \exists i.\ j < i \leqslant n \ \wedge\ u|_i \not\models_{\mathrm{p}'} \psi_1 \tag{7.18}$$

assuming (7.3), (7.9), (7.10), and, by LTL$_3$ semantics, $\forall w.\ u \cdot w, n-1 \not\models \psi_1 \mathbf{S} \psi_2$. Taking $w = \emptyset^\omega$ in the latter equation, by LTL semantics we have

$$\begin{array}{c} \forall k.\ k \leqslant n-1 \ \wedge\ u \cdot \emptyset^\omega, k \models \psi_2 \Rightarrow \\ \exists l.\ k < l \leqslant n-1 \ \wedge\ u \cdot \emptyset^\omega, l \not\models \psi_1 \end{array} \tag{7.19}$$

The special case $j = n$ (thus $u|_j = u|_n = u$) must be eliminated first. In this case, an i such that $n < i \leqslant$ does not exist in (7.18), thus the goal is to find a contradiction between the antecedent $u \models_{\mathrm{p}'} \psi_2$ and other assumptions. By (7.10) we have $[\![u, n-1 \models \psi_2]\!]_3$, or equivalently $\forall w'.\ u \cdot w', n-1 \models \psi_2$. But this is impossible: taking $k = n-1$ in (7.19) an l such that $n-1 < l \leqslant n-1$ does not exist, therefore we have $u \cdot \emptyset^\omega, n-1 \not\models \psi_2$ (an exception of w').

Below we assume $j < n$, and the new goal is to find that i in (7.18) assuming $u|_j \models_{\mathrm{p}'} \psi_2$. As $|u_j| < |u|$, by induction hypothesis (7.3) we have $[\![u|_j, j-1 \models \psi_2]\!]_3 = \top$ or $\forall w'.\ u|_j \cdot w', j-1 \models \psi_2$ by LTL$_3$ semantics. Taking $w' = s_j \cdots s_{n-1} \cdot \emptyset^\omega$ (or $w' = s_{n-1} \cdot \emptyset^\omega$ if $j = n-1$) we have $u \cdot \emptyset^\omega, j-1 \models \psi_2$, which could be the antecedent of (7.19) if we take $k = j-1$. So there exists l such that $j-1 < l \leqslant n-1$ and

$$u \cdot \emptyset^\omega, l \not\models \psi_1 \tag{7.20}$$

Again, the special case $l = n-1$ must be eliminated. This can be done by choosing $i = n$ in (7.19), then by (7.9) the goal can be reduced to $[\![u, n-1 \models \psi_1]\!]_3 = \bot$. By Lemma 7.2.1, this means $[\![u, n-1 \models \psi_1]\!]_3 \neq \top$ or $\neg\forall w''.\ u \cdot w'', n-1 \models \psi_1$ or $\exists w''.\ u \cdot w'', n-1 \not\models \psi_1$, and (7.20) just provided such an instance.

Finally, we have $j-1 < l < n-1$ or $j < l+1 < n$. By choosing $i = l+1$ in (7.19), the goal can be reduced to $u|_{l+1} \not\models_{\mathrm{p}'} \psi_1$. As $|u_{l+1}| < |u|$, by induction hypothesis (7.3) the goal can be further reduced to $[\![u_{l+1}, l \models \psi_1]\!]_3 = \bot$. By Lemma 7.2.1 and LTL$_3$ semantics, it suffices to find just one continuation w''' such that $u_{l+1} \cdot w''' \not\models \psi_1$. By (7.20) we can simply take $w''' = s_{l+1} \cdots s_{n-1} \cdot \emptyset^\omega$ (or $w''' = s_{n-1} \cdot \emptyset^\omega$ if $l+1 = n-1$). □

The above proof has been formalized in Higher Order Logic (HOL4) interactive theorem prover. See Chapter 10 for more details.

Fig. 7.1 demonstrates the equivalence of ptLTL (with alternative semantics) and LTL$_3$ monitors for the same future-free LTL property φ on the same trace $\sigma = u_0\, u_1 \cdots u_n \cdots$, where the size of trace prefix $|u_0 u_1 \cdots u_n| = n+1$ (cf. Equation (7.2) in Theorem 7.2.2). Note that, most existing LTL$_3$ monitors can only monitor LTL$_3$ semantics at time 0, i.e. $[\![u_0\, u_1 \cdots u_n, 0 \models \varphi]\!]_3$. ABRV resolves this problem by always resetting the monitors such that $\textsc{mrr}(u) \equiv |u| - 1$ for any trace prefix u (cf. Equation (4.4) in Definition 4.4.3). Besides, obviously ABRV supports monitoring ptLTL under assumptions.

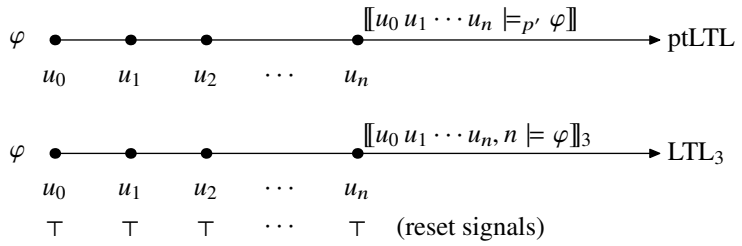


Figure 7.1: ptLTL (alternative) semantics vs. LTL_3 semantics

7.3 Contructing Explicit-state ptLTL Monitors

Using Algorithm 3 with $level = 4$, we can construct explicit-state ptLTL monitors, which always reset themselves before taking next inputs. See Fig. 7.2 for the sample monitors of ptLTL formula $closed \wedge \mathbf{Y}\, open$ taken from [84, p. 99]. Both level 3 and level 4 monitors are given. Note that, the level 3 monitor contains more information as it can be reset arbitrarily, while the level 4 monitor takes no explicit reset signal and it always resets itself before taking next inputs. Given the same trace, it is not hard to see that, both monitors give the same results (assuming the level 3 always jump to the reset locations before taking next inputs).

7.4 Monitoring the original semantics of ptLTL

We conjecture that, by modifying the initial condition in the LTL to ω-automata translation (Section 3.7), the original semantics of ptLTL (Definition 7.1.1) can also be monitored in NuRV. The modified Θ'_φ is given by

$$\Theta'_\varphi \doteq \chi(\varphi) \wedge \bigwedge_{\mathrm{Y}_\psi \in \mathrm{el}(\varphi)} \big(\mathrm{Y}_\psi \leftrightarrow \chi(\psi)\big). \tag{7.21}$$

Intuitionally, the initial value assignments of all past elementary variables are by their corresponding present value (instead of false). This idea is not implemented so far.

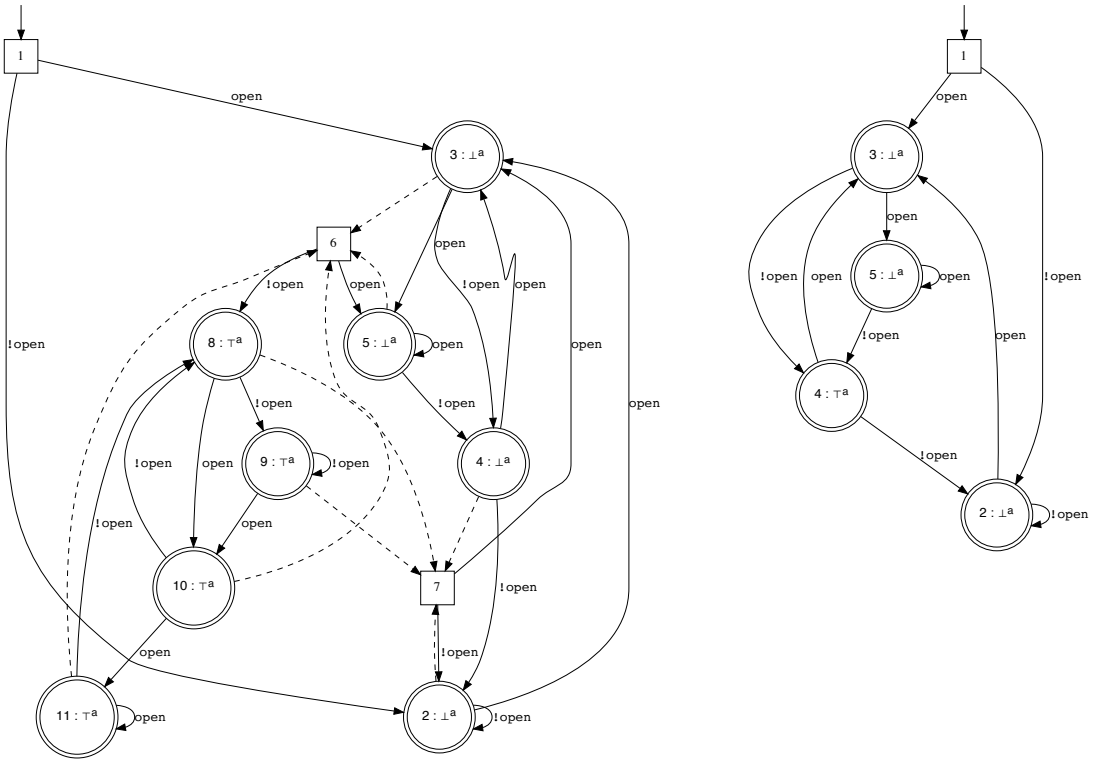


Figure 7.2: ptLTL monitors of *closed* ∧ **Y** *open*' (level 3 and 4) (assuming *open* ≠ *closed*)

Chapter 8

NuRV: The Tool Implementation

ABRV has been implemented in a tool called NuRV[1] since 2018. NuRV is built on top of nuXmv model checker [37] thus can access all functionalities of nuXmv and the underlying tools like BDD library and SAT/SMT checkers. In other words, NuRV implements the Assumption-based Runtime Verification (ABRV) with partial observability and resets. Monitoring properties are expressed in Propositional Linear Temporal Logic (LTL) [115] with both future and past temporal operators.

8.1 Functionalities

As a program, NuRV takes an assumption (as SMV model), some LTL properties and input traces, and output the verification results or some standalone monitor code, according to a batch of commands.

The detailed functionalities of NuRV can be described by the classification of RV tools according to the taxonomy proposed in [66], as shown in Table 8.1.

NuRV can generate embedded standalone monitor code in various programming languages, including C, C++, Java and Common Lisp. In addition, the monitor can be generated as SMV models, whose correctness and other properties can be further verified in NuSMV or nuXmv.

From the end-users' point of view, NuRV extends nuXmv with the following new commands:

1. `build_monitor`: build the symbolic monitor for a given LTL property;
2. `verify_property`: verify a currently loaded trace in the symbolic monitor;
3. `heartbeat`: verify one input state in the symbolic monitor (online monitoring);
4. `generate_monitor`: generate standalone monitors in a target language.

[1] The official web site is currently at `https://es-static.fbk.eu/tools/nurv/`.

Concepts	Branches	Classification of NuRV
Specification	data	*propositional*
	output	*verdict*, *stream*
	time (logical)	*total order (linear time)*
	time (physical)	$\mathbb{N}$ *(discrete time)*
	modality	*all (future, past, current)*
	paradigm	*all (declarative, operational)*
Monitor	decision procedure	*automata-based*
	generation	*all (implicit, explicit)*
	execution	*all (interpreted, direct)*
Deployment	stage	*all (online, offline)*
	synchronisation	*synchronous*
	architecture	*centralised*
	placement	*all (inline, outline)*
	instrumentation	*none*
Reaction	active	none
	passive	*specification output*
Trace	information	*all (events, states)*
	sampling	*all (event-triggered, time-triggered)*
	evaluation	*points*
	precision	*all (precise, imprecise)*
	model	*infinite (LTL), finite (LTL_3, ptLTL)*

Table 8.1: Classification of NuRV according to the taxonomy [66]

The commands `build_monitor` and `verify_property` together implemented the offline monitoring algorithm described in [47]. The command `generate_monitor` further generates explicit-state monitors in various languages from the symbolic monitor built by the command `build_monitor`. These commands must work with other nuXmv commands [27] to be useful.

4. `read_model`: reads a SMV file into nuXmv;
5. `flatten_hierarchy`: flattens the hierarchy of modules;
6. `encode_variables`: builds the BDD variables necessary to compile the model into a BDD;
7. `build_flat_model`: compiles the flattened hierarchy into a Scalar FSM;
8. `build_model`: compiles the flattened hierarchy into a BDD;
9. `add_property`: adds an (LTL) property to the list of properties;
10. `read_trace`: loads a previously saved trace (in XML format).

Concepts	BDD-based (offline)	BDD-based (online)	Code generation
Specification: output	*verdict*	*stream*	*both (verdict, stream)*
Monitor: generation	*implicit*	*implicit*	*explicit*
Monitor: executation	*interpreted*	*interpreted*	*direct*
Deployment: stage	*offline*	*online*	*both (online, offline)*
Deployment: placement	*outline*	*outline*	*inline*

Table 8.2: Distinguished features of three NuRV modes

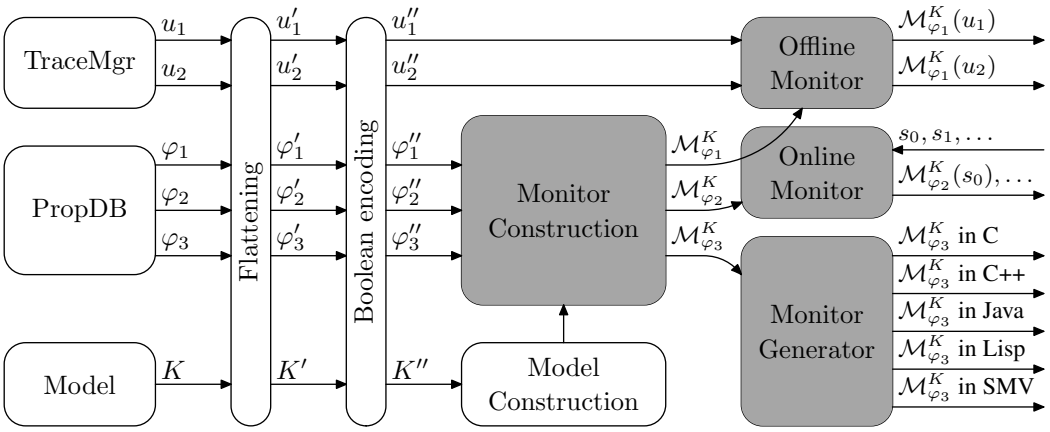


Figure 8.1: The architecture of NuRV

In model checking scenario, the commands 4–8 essentially initialize the system for the verification. If all these commands take default parameters (while the input model is given in other ways, e.g. by environment variable or command line argument), the user could instead use a single command `go`, which is equivalent to the command sequence 4–8. The command `add_property` can be used to add new LTL properties, with each of them a monitor can be built and associated by calling the command `build_monitor`. (An alternative way of adding properties is to put them into SMV files as LTL specifications, i.e. `LTLSPEC` [27].)

The command `read_trace` can be used for loading offline traces into nuXmv for offline monitoring. In nuXmv, a trace consists of an initial state, optionally followed by a sequence of state-inputs pairs corresponding to a possible execution of the model. However, in RV scenario we treat the model as an assumption which estimates the SUS, thus the trace may goes outside of the model if being simulated on the model. The only requirement for the successful loading of a trace, is that all variables used in the trace file (in nuXmv's XML format) must be defined in the model. If a variable is not mentioned in any state of the input trace, it is assumed that its value is not observed in that state. In this way, partial observed traces can still be monitored.

8.2 Architecture

The internal structure of NuRV is shown in Fig. 8.1. The monitor construction starts from the modular description of a model K (used as assumptions in ABRV) and a set of LTL properties $\varphi_1, \ldots, \varphi_n$. The model is used also to declare the variables (and their types) in which the LTL properties are expressed, thus the *alphabet* of the input words of the monitors. NuRV has inherited nuXmv's support of hierarchical models and rich variable types (such as bound integers and arrays), all input data (models, properties and traces) are flattened and boolean encoded before going to further steps. The *Model Construction* component generates (from the model) a BDD-based representation of the Finite State Machine (FSM), which is then used in the monitor construction step, together with the monitoring property, to produce another BDD-based FSM representing the symbolic monitor. The resulting monitor can be used in two ways: 1) as an online/offline monitor running inside nuXmv, accepting finite traces incrementally, outputting verification results for each input states. 2) as the input of the *Monitor Generator* component, resulting into standalone monitor code.

The *Monitor Generator* components internally generate monitor code in two steps: 1) generating explicit-state monitor automata from the symbolic monitor; 2) converting monitor automata into code in specific languages. NuRV can generate

Standalone monitor code are literally translated from these monitor automata (FSMs). The correctness of monitors in C, for instance, comes *indirectly* from the correctness of the symbolic algorithm and mode checking on SMV-based monitors.

8.3 Use Case Scenario

Now we briefly demonstrate the process of generating a monitor for LTL properties $\varphi_0 = p \,\mathbf{U}\, q$ and $\varphi_1 = \mathbf{Y}p \vee q$, assuming $p \neq q$. A batch of commands shown in Fig. 8.2 does the work (also c.f. Fig. 8.3 for the contents of two helper files).

The command `go` builds the model from the input file `disjoint.smv` which defines two Boolean variables p and q, together with the invariant $p \neq q$.

The generated monitors `M0.c` and `M1.c` (together with their C headers) are under the full observability of p and q. The variable ordering is given by the file `default.ord`, in which each line denotes one variable in the model.

The simplest way to use the generated monitor, `M0` for instance, is to declare an integer and call the monitor function like this: (e.g. when monitoring a C program linked with the generated monitor code, p and q may denote two assertions in the program)

```
int monitor_loc, out;
out = M0 (0b01 /* p & !q */, 1 /* hard */, &monitor_loc);
out = M0 (0b10 /* !p & q */, 0 /* none */, &monitor_loc);
```

```
set input_file "disjoint.smv"
set input_order_file "default.ord"
go
add_property -l -p "p␣U␣q"
add_property -l -p "Y␣p␣|␣q"
build_monitor -n 0
build_monitor -n 1
generate_monitor -n 0 -l 3 -L "c" -o "M0"
generate_monitor -n 1 -l 3 -L "c" -o "M1"
quit
```

Figure 8.2: Batch commands for generating monitors of $p\,\mathbf{U}q$ and $\mathbf{Y}p \vee q$

```
MODULE main
VAR   p : boolean; q : boolean;
INVAR p != q
```

```
p
q
```

Figure 8.3: `disjoint.smv` and `default.ord`

There is no need to initialize the integer `monitor_loc` as the first `M0` call with a value 1 will also do the monitor initialization. (Actually it just set `monitor_loc` to 1, we may call it a *hard reset*.) The first function call returns 0 indicating ABRV-LTL value ? (unknown); the second call returns 1 indicating $\top^{\mathrm{a}}$ (conclusive true).

For offline monitoring, there is no need to call `generate_monitor` in above batch command. Suppose a trace $u = p\,p\,p\,q\,q\,q$ has been loaded (by `read_trace`), the command `verify_property` verifies the trace against the symbolic monitor of φ_0, shown in Fig. 8.4 (here "`-n 0`" denotes the first monitor, and `1` denotes the first loaded trace).

It is also possible to verify just one input state by `heartbeat` (online monitoring). It has a

```
NuRV > verify_property -n 0 1
1, unknown
2, unknown
3, unknown
4, true
5, true
6, true
```

Figure 8.4: Offline monitoring in NuRV

similar interface with `verify_property`, just the trace ID is replaced by a single state expressed by a logical formula (as a string), e.g. `"p & !q"`.

8.4 Online Monitoring

In online monitoring, the synthesized runtime monitor takes a single input state and immediately output a verdict corresponding to the input state. The related NuRV command is `heartbeat`. (For a "real" online monitor which can be called from remote, see Appendix B.) The monitor maintains it internal states (aka belief states) for handling future inputs. The monitor can be softly or hardly reset when taking an input state.

For RV on finite-state systems, it is also possible to generate monitor code into various programming languages (see Chapter **??** for more details). Then it is up to the user to use such generated monitors in a online or offline manner. However, essentially the generated monitor are online monitors taking input states one by one.

As an example of online monitoring (using the same setting given in Fig. 8.5), the following batch command (saved as `online.cmd`) can be used for demo purposes:

```
go
build_monitor -n 0
heartbeat -n 0 -c "p"
heartbeat -n 0 -c "p"
heartbeat -n 0 -c "p"
heartbeat -n 0 -c "q"
heartbeat -n 0 -c "q"
heartbeat -n 0 -c "q"
quit
```

If one calls NuRV in the following way:

```
$ NuRV -quiet -source online.cmd disjoint.smv <RET>
```

It will output the following results indicating the command output of each `heartbeat` commands: (However, in practice the commands should be understood as receiving at runtime from the system under scrutiny.)

```
unknown
unknown
unknown
true
true
true
```

Currently the online monitoring support of NuRV can only be used for debugging or manual testing purposes (of the monitors), because interactively calling `heartbeat` commands is not

very useful when NuRV is actually used as online monitors. In future versions NuRV may provide a network-based monitor server so that external callers may call `heartbeat` commands, among other commands, remotely.

8.5 Offline Monitoring

In offline monitoring, one or more traces must be loaded into NuRV's trace manager by the command `read_trace`. Then user can use the command `verify_property` to check if a loaded trace is verified or violated against an LTL property.

For RV on finite-state systems, it is also possible to generate monitor code into various programming languages (see Chapter **??** for more details). Then it is up to the user to use such generated monitors in a online or offline manner. However, essentially the generated monitor are online monitors taking input states one by one.

```
MODULE main
VAR
        p : boolean;
        q : boolean;
INVAR
        p != q
LTLSPEC
        p U q
```

Figure 8.5: The SMV file `disjoint.smv` for p **U** q (assuming $p \neq q$)

The SMV model file including the LTL property is given in Fig. 8.5. Suppose we wanted to monitor a trace $u = \{p, \neg q\}\{p, \neg q\}\{p, \neg q\}\{\neg p, q\}\{\neg p, q\}\{\neg p, q\}$, that is, for the first 3 states p is true (and q is false), then q becomes true (and p becomes false). The following XML file (saved as `trace.xml`, for example) should be prepared (or generated by the user using other programs) as the trace:

```
<?xml version="1.0" encoding="UTF-8"?>
<counter-example type="0" id="1" desc="LTL␣Counterexample">
        <node>
                <state id="1">
                        <value variable="p">TRUE</value>
                        <value variable="q">FALSE</value>
                </state>
        </node>
        <node>
                <state id="2">
                        <value variable="p">TRUE</value>
                        <value variable="q">FALSE</value>
```

```
            </state>
        </node>
        <node>
            <state id="3">
                <value variable="p">TRUE</value>
                <value variable="q">FALSE</value>
            </state>
        </node>
        <node>
            <state id="4">
                <value variable="p">FALSE</value>
                <value variable="q">TRUE</value>
            </state>
        </node>
        <node>
            <state id="5">
                <value variable="p">FALSE</value>
                <value variable="q">TRUE</value>
            </state>
        </node>
        <node>
            <state id="6">
                <value variable="p">FALSE</value>
                <value variable="q">TRUE</value>
            </state>
        </node>
</counter-example>
```

The following batch command (saved as `offline.cmd`) will load the trace and call `verify_property` to verify the trace:

```
go
build_monitor -n 0
read_trace trace.xml
verify_property -n 0 1
quit
```

Now NuRV can be called in this way:

```
$ NuRV -quiet -source offline.cmd disjoint.smv <RET>
```

The above command will output the following results (beside a message saying the trace has been correctly loaded and stored) indicating the verdict ? for the first 3 states and $\top^a$ for the rest states: (To write monitoring results into a file, use `-o` command-line option with `verify_property`. Also note that the output is 0-indexed while the input trace XML is 1-indexed.)

```
1, unknown
2, unknown
```

```
3, unknown
4, true
5, true
6, true
```

Note that, starting from version 1.7.0, the first column of the output of command `verify_property` has been changed to be aligned with trace manager, i.e. the state indexes of loaded traces now starts from 1.

8.6 API of Generated Code

NuRV currently (version 1.9.1 by the time of thesis writing) supports monitor code generation into the following programming languages: C, C++, Java, Python, Common Lisp and Prolog. NuRV can also generate LLVM IR (Intermediate Representation) code [107] which is near assembly and can run on bare matel (i.e. no dependency on any external library). Besides, NuRV supports generating SMV models. The structure of generated monitor code for all these target languages are the same. It is simple but efficient: it simply mimics the simulations of deterministic FSMs.

The monitor code generated (in C, for example) has the following signature:

```
monitor
  (long /* state [in] */,
   int  /* reset [in] (0 = none, 1 = hard, 2 = soft) */,
   int* /* current_loc: [in/out] */);
```

The function name (*monitor* here) is given by the user. It takes three parameters: 1) `state`: an encoded long integer representing the current input state of the trace, 2) `reset`, an integer representing the possible reset signal, and 3) `current_loc`: a pointer of integer holding the internal state of the monitor. It is caller's responsibility to allocate an integer and provide the pointer to the monitor (otherwise the function returns -1 indicating *invalid locations*), and this is actually the only thing to identify a monitor instance. The sole purpose of the function is to update `*current_loc` (the value behind the pointer) according to `state` and `reset` and to return a monitoring output. NuRV supports two different encodings for `state`:

1. *static* partial observability: `state` denotes a full assignment of the observables, encoded in binary bits: 0 for *false* ($\bot$), 1 for *true* ($\top$);

2. *dynamic* partial observability: `state` denotes a ternary number, whose each ternary bit represents 3 possible values of an observable variable: 0 for *unknown* (?), 1 for *true* ($\top$) and 2 for *false* ($\bot$).

Note that the symbolic monitoring algorithm can take in general input states expressed in Boolean

formulae (e.g., if the observables are p and q, our monitor may take an input state "p xor q", either p or q is true but not both), but this is not supported by the generated code.

BDD operations are implemented by the BDD manager. Their performance strongly depends on the variable ordering used in the BDD construction. This can be controlled by setting an `input_order_file` in nuXmv. The input of generated monitor code requires an encoding of BDDs into long integers according to this file. This encoding is done from the least to the most significant bit. For instance, if the observables are p and q with the same order, an binary encoding for the state $\{p = \top, q = \bot\}$ would be $(01)_2 = 1$, and a ternary encoding for the same state would be $(21)_3 = 7$. The design purpose is to make sure that the comparison of two encoded states can be as fast as possible. The signatures of monitors in other languages are quite similar, except that the parameter `current_loc` can be put inside C++/Java classes as an member variable, and each monitor is an instance of the generated monitor class.

8.7 Code Generation – Backends

NuRV supports code generation of runtime monitors in Propositional LTL under finite-state assumptions. The generated monitor code, whenever in programming languages, can be regarded as online monitors which can also be used in offline manners. (Any online monitor is also an offline monitor, but not vice versa.)

The monitor is deterministic. It is straightforward to generate progarm code equivalent to the monitor FSM. The idea is to update the current monitor location according to input state and possible reset signal, and return the monitor outputs stored at each location.

8.7.1 Code generation in Prolog

Since NuRV 1.8.0. the monitor code can be generated in ISO Prolog (by using command-line option `-L "prolog"` when calling `generate_monitor`) as a module:

```
:- module(rvsynth, [monitor/5, monitor_old/5]).
```

where `monitor` is the monitor function name given by the caller, and `rvsynth` is the default module name.

Monitor code in Prolog is particular interesting because essentially the monitor is generated into relational data which can be directly put into relational database systems (RDBMS) like Oracle (and the SQL engine can be monitoring engine).

For example, the following Prolog code is generated from LTL property $\mathbf{G}\,(p \rightarrow \mathbf{X}\,\mathbf{X}\,q)$ under the invariant assumption that $p \neq q$. The monitor FSM has seven locations in total. The first part is the reset table. Since there is no past operators, all states reset to the initial state 1, including the initial state itself: (Note that code lines leading by % are Prolog comments.)

```
% monitor_reset_table(loc, next_loc)
monitor_reset_table(7, 1).
monitor_reset_table(6, 1).
monitor_reset_table(5, 1).
monitor_reset_table(4, 1).
monitor_reset_table(3, 1).
monitor_reset_table(2, 1).
monitor_reset_table(1, 1).
```

The second part is the transition table of the monitor FSM. The table has four columns: location, (input) states, output, and next location. Note that input states are single element list, in which the elements are bitwise encoding of two Boolean variables p and q:

```
% monitor_trans_table(loc, states, output, next_loc)
monitor_trans_table(7, [1], 2, 6).
monitor_trans_table(7, [2], 2, 7).
monitor_trans_table(6, [1], 2, 6).
monitor_trans_table(6, [2], 2, 7).
monitor_trans_table(5, [1], 2, 6).
monitor_trans_table(5, [2], 0, 3).
monitor_trans_table(4, [1], 2, 6).
monitor_trans_table(4, [2], 0, 5).
monitor_trans_table(3, [1], 0, 2).
monitor_trans_table(3, [2], 0, 3).
monitor_trans_table(2, [1], 0, 4).
monitor_trans_table(2, [2], 0, 5).
monitor_trans_table(1, [1], 0, 2).
monitor_trans_table(1, [2], 0, 3).
```

The third part is just a table translating the numerical outputs into symbolic verdicts: (Note that there is still another symbol `error` as *out-of-model*.)

```
% monitor_output_table(output, verdict)
monitor_output_table(0, unknown).
monitor_output_table(1, true).
monitor_output_table(2, false).
```

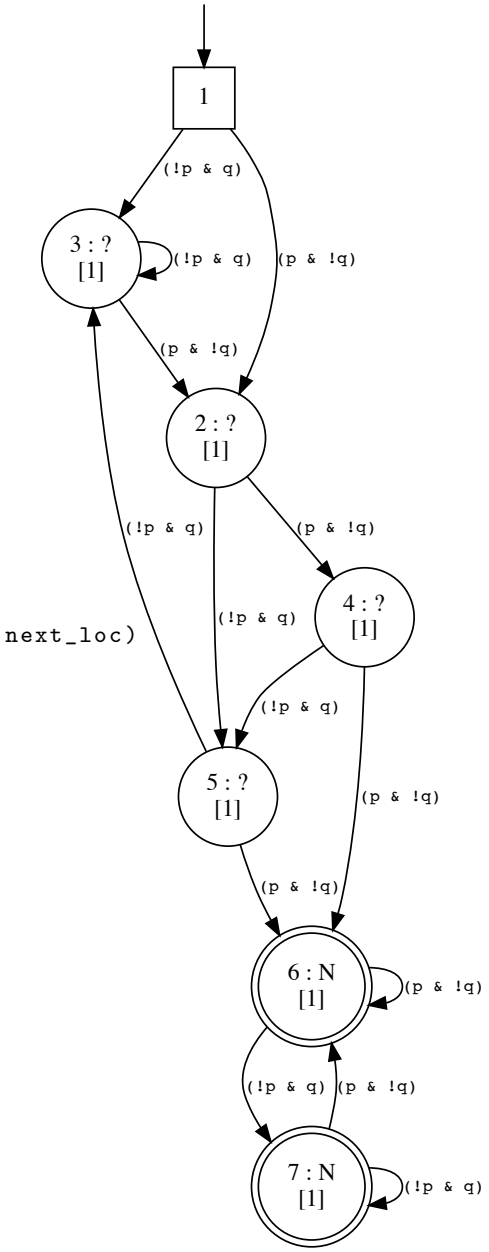


Figure 8.6: Explicit-state monitor of $\mathbf{G}\,(p \rightarrow \mathbf{X}\,\mathbf{X}\,q)$

The main Prolog predicate is called `monitor` with the following API of five parameters:

```
% new API: monitor(states, reset, loc, output, next_loc)
```

Of the five parameters: *states*, *reset* and *loc* are input parameters, while *output* and *next_loc* are output parameters. The actual definition of the monitor engine is divided by different combinations of input parameters:

1. On hard resets, the monitor sets whatever current location to the initial location and issues no reset for the rest of processing:

```
monitor(S, hard_reset, _, V, NL) :- monitor(S, no_reset, 1, V, NL), !.
```

2. On soft resets, the current location is changed according to the reset table, then the monitor is given by the location after resets, while other parameters are the same:

```
monitor(S, soft_reset, L, V, NL) :-
    monitor_reset_table(L, RL), !,
    monitor(S, no_reset, RL, V, NL).
```

3. If the current location match any location in the transition table, then the monitor's output can now be decided:

```
monitor([S0], no_reset, L, V, NL) :-
    monitor_trans_table(L, [S0], O, NL), !,
    monitor_output_table(O, V).
```

4. Finally, if the current location match any location in the transition table, which indicates that the monitor has gone outside of the model (assumptions):

```
monitor(_, no_reset, L, error, L).
```

Note that the above Prolog code logic can be easily implemented in SQL while the data tables can be real tables in any RDBMS. In the other code generation languages, essentially we are following the same structure of the above Prolog code.

8.7.2 Code generation in C

The monitor code generated in ANSI C (by using command-line option `-L "c"` in `generate_monitor`) has the following (old) signature:

```
int /* [out] (0 = unknown, 1 = true, 2 = false, 3 = out-of-model) */
  monitor
    (long /* state [in] */,
     int  /* reset [in] (0 = none, 1 = hard, 2 = soft) */,
     int* /* current_loc: [in/out] */);
```

Note that the maximum number of observable bits is limited by C `long` type, which has at most 31 valid bits in portable code. Starting from NuRV 1.8.0, the following *new* signature is also generated with first parameter `long states` extended to an array of `long`s which can support arbitrary number of underlying bits:

```
RV_value monitor_ex
  (long    *states,        /* [in] */
   size_t   width,         /* [in] */
   RV_reset reset,         /* [in] */
   int      *current_loc); /* [in/out] */
```

The two types `RV_value` and `RV_reset` involved in the above signature are enumerations with underlying values compatible with the old interface:

```
typedef enum {
  RV_UNKNOWN = 0, RV_TRUE = 1, RV_FALSE = 2, RV_ERROR = 3,
  RV_INVALID_ARG = 4, RV_INVALID_LOC = 5
} RV_value;

typedef enum {
  NO_RESET = 0, HARD_RESET = 1, SOFT_RESET = 2
} RV_reset;
```

The function name (*monitor* here) is given by the user when calling the NuRV command `generate_monitor`. (Note that, in the new signature, a suffix *_ex* is attached to the monitor function name.)

The generated monitor fucntions takes three (or four) parameters:

1. `state` (or `states`): an encoded long integer (or an array of long integers) representing the current input state of the trace,
2. `width`: in the new signature, this is the length of `states` supplied by the user.
3. `reset`, an integer representing the possible reset signal, and
4. `current_loc`: a pointer of integer holding the internal state of the monitor.

It is the caller's responsibility to allocate an integer and provide the pointer to the monitor (otherwise the function returns -1 or `RV_INVALID_LOC` indicating *invalid locations*).

NuRV supports two different encodings for `state` (or `states`) (depending on the presense of the command-line option `-p` when calling `generate_monitor`):

1. *Static* partial observability: `state` denotes a full assignment of the observables, encoded in binary bits: 0 for *false* ($\bot$), 1 for *true* ($\top$);
2. *Dynamic* partial observability: `state` denotes a ternary number, whose each ternary bit represents 3 possible values of an observable variable: 0 for *unknown* (?), 1 for *true* ($\top$) and 2 for *false* ($\bot$).

Note that the symbolic monitoring algorithm can take in general input states expressed in Boolean formulae (e.g., if the observables are p and q, our monitor may take an input state "p xor q", either p or q is true but not both), but this is not supported by the generated code.

BDD operations are implemented by the BDD manager. Their performance strongly depends on the variable ordering used in the BDD construction. This can be controlled by setting an `input_order_file` in nuXmv. The input of generated monitor code requires an encoding of BDDs into long intergers according to this file. This encoding is done from the least to the most significant bit. For instance, if the observables are p and q with the same order, an binary encoding for the state $\{p = \top, q = \bot\}$ would be $(01)_2 = 1$, and a ternary encoding for the same state would be $(21)_3 = 7$. The design purpose is to make sure that the comparison of two encoded states can be as fast as possible.

In the new signature, an array of long integers `states` with the length of array, `width`, are supplied by the caller. To correctly fill up this array, it is important for the caller to know the following information:

1. How many bits are encoded into a single `long` integer?

   This value is given by a per-monitor generated C macro *monitor*`_segment`, whose default value is 31 but may be changed to be flexible in the future.

2. How many `long` integers are necessary?

   The minimal value of `width` is given by a C macro *monitor*`_width`. Note that providing a bigger array will not cause any issue as the generated monitor code simply will not read those extra elements. On the other hand, an input array whose length is smaller than *monitor*`_width` will immediately cause the monitor to return `RV_INVALID_ARG` (invalid arguments).

8.7.3 Structure-based interface of monitors in C

Note that, besides Boolean variables, the SMV language also supports finite-domain integers and fixed-length machine words, which is called *scalar variables*. In the BDD-based symbolic monitoring, scalar variables can be freely used in the model and monitoring properties, because internally they are encoded as Boolean bits. (In the SMT-based monitoring, most of the scalar variables are directly handled by the underlying SMT solvers.)

However, when generating standalone monitor code, it is almost impossible for the end user to do the bit encodings for scalar variables, because the mapping between a scalar variable and its underlying bits is totally an internal matter of NuRV without clear patterns. Even with only Boolean variables in the model, the encoding of Boolean values into `state` (or `states`) should be a task that can be automated during the code generation process.

Suppose there are 4 Boolean variables p, q, r, s in the model, NuRV also generates the following *scalar* signature:

```
/* input data */
typedef struct {
```

```
  short p; /* 0: false, other: true */
  short q; /* 0: false, other: true */
  short r; /* 0: false, other: true */
  short s; /* 0: false, other: true */
} monitor_input_t;

/* input masks (0: not observable, 1: observable) */
typedef struct {
  unsigned int p : 1;
  unsigned int q : 1;
  unsigned int r : 1;
  unsigned int s : 1;
} monitor_mask_t;

/* 3. scalar API */
RV_value monitor_scalar
  (monitor_input_t *input,
   monitor_mask_t *masks,
   RV_reset reset,
   int *current_loc);
```

Now, instead of encoding the values of p, q, r, s into long integers, now the end user only need to allocate a structure `monitor_input_t` and set the Boolean values directly as slots of this structure. (Note that each Boolean variable is mapped to a C `short` value.)

Remark 8.7.1. *(The other structure `monitor_mask_t` is intended to support partial observability of values in the structure `monitor_input_t`, i.e. an input value is considered observable whenever the corresponding slot in `monitor_mask_t` is non-zero. Currently this is not implemented yet, and user can just supply `NULL` as the value of `masks`.)*

In another case, the model contains the following variables:

```
VAR
    i : 140 .. 160;
    k : {a, b, 0, 1};
    l : {b, c};
```

where `a`, `b` and `c` are constant symbols. NuRV may generate the following code where `m1` is the monitor name:

```
/* constants used in the model */
enum {
  b = 2,
  a = 3,
  c = 4
};

/* input data */
```

```
typedef struct {
  int i;
  int k;
  int l;
} m1_input_t;

/* input masks (0: not observable, 1: observable) */
typedef struct {
  unsigned int i : 1;
  unsigned int k : 1;
  unsigned int l : 1;
} m1_mask_t;

/* 3. scalar API */
RV_value m1_scalar
  (m1_input_t *input,
   m1_mask_t *masks,
   RV_reset reset,
   int *current_loc);
```

Then the caller may, for example, set $i = 150, k = b, l = b$ when calling the generated monitor code. The encoding of scalar varibles into underlying Boolean bits are done automatically by the generated monitor code. BDD input ordering file is not needed if the caller only uses the scalar API.

8.7.4 Observable expressions in generated monitor

One major problem of using scalar variables in generated monitor code is that there are usually *too many* underlying Boolean bits. For example, a finite domain integer having values from 0 to 1024 will involved 10 underlying Boolean bits, and in the worst case each single value of this integer may correspond to one possible monitor input (transition arc) in the generated explicit-state monitor. With several scalar variables it is easy to cause a blow up in the size of generated code.

To overcome the potential blow up in the size of generated monitor code. NuRV now supports setting *observable expressions* during the monitor code generation. For example, if the monitoring property is $(i < 150)\,\mathbf{U}\,(i >= 150)$, while i is a finite domain integer having a large range of values, without further constraints in the model the generated monitor could be already accurate if $(i < 150)$ and $(i >= 150)$ were considered as atomic propositions, and the generated monitor should have the same size as the one generated from $p\,\mathbf{U}\,q$ where p and q are Boolean variables. However, the generated monitor should still accept the original values of i, and the calculation of $(i < 150)$ and $(i >= 150)$ should be done in the generated code, to re-contruct the underlying internal bits. The propositions $(i < 150)$ and $(i >= 150)$ can

be considered as *observable expressions*. (It is user's responsibility to supply a good list of observable expressions to have the generated monitors accurate enough.)

To generate monitor with *observable expressions*, a new parameter `-C` is added into the command `generate_monitor`. `-C` takes a file name, in which each line is an observable expression. See NuRV User Manual [51] for more details.

8.7.5 Code generation in C++

The monitor code can be generated as classes in C++ (ISO/IEC 14882, aka C++98) (by using command-line option `-L "cpp"` when calling `generate_monitor`.) The generated monitor class headers are like this:

```
class monitor : base_monitor {
public:
  monitor() { current_loc = 1; }
  int run(long state, int reset); // old API
  RV_value run(vector<long> &states, RV_reset reset); // new API

private:
  ...
}
```

Both old and new APIs are supported. For the new API, a vector of long values are supplied to support arbitrary number of Boolean bits.

All generated monitor C++ classes inherit a common base class called `base_monitor`:

```
class base_monitor {
protected:
  int current_loc;
public:
  virtual RV_value run(vector<long> &states, RV_reset reset) {
     return RV_ERROR;
  };
};
```

The presense of a share base class allows calling different monitors from the (virtual) method calls of the base class (polymorphism). The involved types `RV_value` and `RV_reset` have the same definitions as the generated code in C.

The generated C++ code is compatible with C++98, C++11 and later C++ standards.

8.7.6 Code generation in Java

The monitor code can be generated in Java (by using command-line option `-L "java"` when calling `generate_monitor`.) Java code is always object oriented. The Java monitor class has the following structure:

```
package eu.fbk.rvsynth;

public class Monitor extends BaseMonitor {
  private int current_loc = 1;
  // old API
  public int // 0 = unknown, 1 = true, 2 = false, 3 = error
    run (long state,
         int reset) // 0 = none, 1 = hard, 2 = soft
  {
    ...
  }

  // new API
  public RV_value run(long[] states, RV_reset reset)
  {
    ...
  }
}
```

The default Java package name `eu.fbk.rvsymth` can be changed by `-m` parameter of the command `generate_monitor`.

A base class `BaseMonitor` is generated in a separate code file `BaseMonitor.java` in the same directory with the following contents:

```
package eu.fbk.rvsynth;

// monitor base class
public abstract class BaseMonitor {
  // old API
  public abstract
  int /* out (0 = unknown, 1 = true, 2 = false, 3 = error) */
    run (long state,
          int reset /* in (0 = none, 1 = hard, 2 = soft) */);

  // new API
  public abstract RV_value run(long[] states, RV_reset reset);
};
```

Two helper classes `RV_reset` and `RV_value` are also generated in separate code files. The generated Java code is tested on JDK 8 and should work in more recent JDK versions.

8.7.7 Code generation in Common Lisp

The monitor code can be generated in Common Lisp, a major dialect of LISP, the second (the first is FORTRAN) oldest advanced programming language in the history of computer science. This is supported in NyRV by using command-line option `-L "lisp"` when calling `generate_-`

monitor. The generated Common Lisp is Object-Oriented (in CLOS, aka the *Common Lisp Object System*), having the following structure:

```
;;; base monitor class
(defclass base-monitor ()
  ((current-loc :type fixnum :accessor current-loc :initform 1)
   (reset-table :type simple-vector :reader reset-table)
   (trans-table :type simple-vector :reader trans-table))
  (:documentation "NuRV␣base␣monitor␣class"))

;;; generic function
(defgeneric run (instance states reset)
  (:documentation "monitor␣entry␣function"))

;;; monitor class
(defclass monitor (base-monitor)
  ()
  (:documentation "NuRV␣monitor␣class"))

;;; old API
(defmethod run ((instance monitor) (state fixnum) (reset symbol))
  ...)

;;; new API
(defmethod run ((instance monitor) (states sequence) (reset symbol))
  ...)
```

8.7.8 Code generation in LLVM IR

The monitor code can be generated in LLVM Intermediate Representation (LLVM IR) [107], by using command-line option `-L "llvm"` when calling `generate_monitor`. This functionality is made by outputting LLVM IR code in plain text without linking any library from LLVM. In particular, the resulting LLVM IR code does not call any external function like functions in standard C library. For this purposes, when doing binary search, the searching algorithm is also emitted in LLVM IR as part of the embedded monitor engine.

Essentially the API is the same as the generated C code, except that all involved `long` values in the C code are fixed as signed 32-bit integers in LLVM IR code. Both old and new APIs are supported. Below are the code pieces showing the beginning part of external functions:

```
; main function (old API)
define external i32 @monitor
   (i32  %state_in,     ; [in]
    i32  %reset,        ; [in]
    i32* %current_loc) ; [in/out]
{
```

```
    ...
}
; main function (new API)
define external i32 @monitor_ex
   ([1 x i32]* %states,      ; [in]
    i32        %width,       ; [in]
    i32        %reset,       ; [in]
    i32*       %current_loc) ; [in/out]
{
    ...
}
```

There's no separate header generated. The headers generated for C code can be used in a compatible way.

LLVM IR is assembly-like, simplier to parse and analyze (by using libraries from the LLVM project, thus is considered as a good *intermediate* language for exchanging purposes. In theory, end users can write their own translators for the translation of generated monitor code from LLVM IR to other programming languages.

8.7.9 Code generation in SMV

The monitor code generated in SMV (by using command-line option `-L "smv"` when calling `generate_monitor`) are mainly for model checking purposes (to verify the correctness of the monitor itself). The following SMV file represents the monitor of $p \mathbf{U} q$:

```
MODULE monitor (p, q, _reset)

VAR
  _loc     : 0 .. 4;
  _rloc    : 0 .. 4;
  _out     : { true, false, unknown, error };

DEFINE
  _true    := ((_loc = 4) | (_loc = 3) | FALSE);
  _false   := (FALSE);
  _unknown := ((_loc = 2) | (_loc = 1) | FALSE);
  _error   := (_loc = 0);
  _valid   := _true | _false | _unknown;
  _concl   := _true | _false;

ASSIGN
  _out := case
    _true : true; _false : false; _unknown : unknown; TRUE : error;
  esac;

  _rloc := _reset ? case
```

```
    (_loc = 4) : 1; (_loc = 3) : 1; (_loc = 2) : 1; (_loc = 1) : 1; TRUE : 0;
  esac : _loc;

  init(_loc) := 1;
  next(_loc) := case
    ((_rloc = 4) & (!p & q)): 3;
    ((_rloc = 4) & (p & !q)): 4;
    ((_rloc = 3) & (!p & q)): 3;
    ((_rloc = 3) & (p & !q)): 4;
    ((_rloc = 2) & (!p & q)): 3;
    ((_rloc = 2) & (p & !q)): 2;
    ((_rloc = 1) & (!p & q)): 3;
    ((_rloc = 1) & (p & !q)): 2;
    TRUE : 0;
  esac;

INVARSPEC
  count(_true, _false, _unknown, _error) = 1;
```

In Section 9.1.2, we show how the generated SMV models are used for verifying some characteristics of the corresponding monitors.

Chapter 9

Experimental Evaluation

9.1 Tests for Finite-State Monitors

In this section, we report about the experiments performed to validate the approach and evaluate its performance. The correctness of generated explicit-state monitor code has been extensively tested by comparing the outputs with those from the symbolic monitors, on a large set of LTL properties and random traces.

9.1.1 Tests on the factory model

As a proof of concept, we evaluated the factory example described in Section 4.5. The related model files are part of the downloadable artifacts. The SMV model definition is also given in Appendix (Section A.1).

In particular, we found that the monitor of the property, generated with the factory model as assumption, is indeed predictive: it outputs $\bot^a$ *almost immediately* after the first fault happened.[1] To see such a possible trace, again we used model checking. In particular, we considered the LTL specification $\neg\mathbf{F}\ \texttt{AV}$ with $\texttt{AV} := (\texttt{M1.\_concl} \wedge \neg\, \texttt{M2.\_concl})$, where `M1` and `M2` are the monitor instances of the SMV modules generated from the above safety property built with (without) assumption, respectively. The counter-example shows such a trace: the fault happens at state 4, and the filling of the red ingredient at position 0 failed at position 1; the monitor with assumption outputs $\bot^a$ at state 6, before the bottle is moved to position 1, while the monitor without assumption can only output $\bot^a$ at state 10, after the bottle is moved to position 2. This is because any unfilled bottle at position 0 or 1 will remain unfilled at position 2 under the model, thus the monitor with assumption knows the faults before any unfilled bottle arrived at position 2, even if the fault itself is not directly observable. In practice, there may be more positions (and

[1] Actually in this case the monitor outputs $\bot^a$ one step after the fault happens. It did not immediately outputs $\bot^a$ at the moment when the fault happens because, according to the model, the bottle may get filled again before the belt moves.

more ingredients) in the assembly line, reporting the faults as early as possible may skip the rest of filling operations of the faulty bottle (e.g. the bottle can be removed from the assembly line by a separate recovery process) and potentially reduce the costs.

In another test (see `fault_monitor.cmd` in the artifacts), a monitor for LTL property $\mathbf{G}\,\neg(\texttt{fault[0]} \vee \texttt{fault[1]})$ is built for monitoring faults. Two sample traces are prepared, with a fault happens at position 1 and 2, respectively. With the factory model as assumptions, the monitor *immediately* reports $\bot^{\mathrm{a}}$ when the fault happens.

In the test of resets, we simulated a scenario in which two faults happens at different time[2], and we would like to see the monitor being reset after the first detection of violation and continue to work just like the fault did not happen, until the next fault. In the sample trace, observables are the following: `bottle_present[0]`, `bottle_ingr1[0]`, `bottle_ingr2[1]`, `bottle_present[2]`, `bottle_ingr1[2]`, and `bottle_ingr2[2]`.

The first fault happens at state 3 when the red ingredient was being filled into a new bottle at position 0, while the second fault happens at state 7 when the red ingredient was being filled into another new bottle at position 0. The first fault is detected by the monitor at state 4 (as early as possible, given the observations), and then the bottle moved out of the assembly line since state 8. Here the resetting strategy is to reset the monitor *immediately after each violation.* However, since the faulty bottle remains on the belt until state 7, the monitor will keep outputting $\bot^{\mathrm{a}}$ for these states (from 4 to 7). At state 8 the monitor verdicts would have been restored to inconclusive *if there were no other faulty bottles*, but actually the verdict is still $\bot^{\mathrm{a}}$, as the second fault is now detected, although the faulty bottle is still at position 1, *thanks to the observations since state 6.* (The monitor of $\mathbf{G}\,\neg(\texttt{fault[0]} \vee \texttt{fault[1]})$, on the other hand, reports the first fault immediately at state 3, and after resets the monitor verdicts restored to inconclusive; Then it reports the second fault again immediately (at state 7) and restored to inconclusive after resets.)

9.1.2 Tests on Dwyer's LTL patterns

To show the feasibility and effectiveness of our RV approach, we have generated monitors from a wide coverage of practical specifications, i.e. Dwyer's LTL patterns [61][3].

To show the impact of assumptions, we generated two groups of monitors, with and without assumption. The chosen assumption says that *the transitions to s-states occur at most 2 times*, which can be expressed in LTL as $((\neg s)\,\mathbf{W}\,(s\,\mathbf{W}\,((\neg s)\,\mathbf{W}\,(s\,\mathbf{W}\,(\mathbf{G}\,\neg s)))))$. Under this assumption we found that, non-monitorable properties like $\mathbf{G}(p \rightarrow \mathbf{F}s)$ now become monitorable, i.e. the monitor may output conclusive verdicts on certain inputs. This is because, if the transitions to s-state have already occurred 2 times, there should be no s any more in the

[2]See `factory_example/trace3.xml` in the artifact for the contents of this trace.

[3]The latest version (55 in total) is available at `https://matthewbdwyer.github.io/psp/patterns/ltl.html`. We call them Pattern 0, 1, . . . , 54 in the same order.

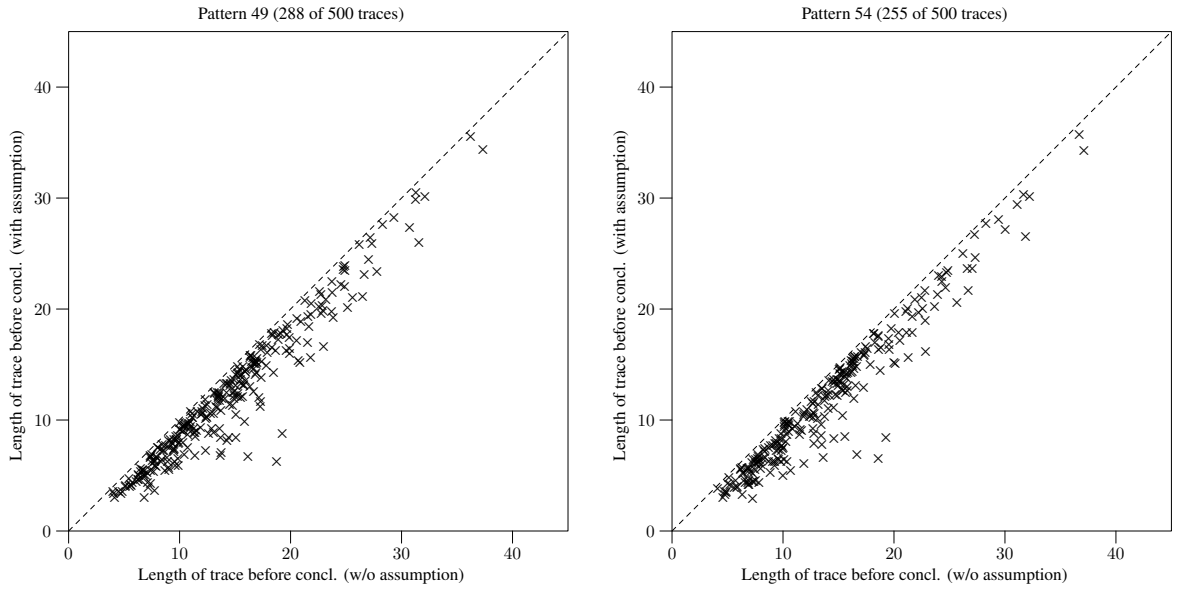


Figure 9.1: The number of observations before a conclusive verdict, with and without assumptions

remaining inputs. Thus whenever p occurs, for whatever future inputs it is impossible to satisfy $\mathbf{F}s$, thus the property is violated conclusively. Eight monitors (Pattern 25, 27, 40, 42, 43, 44, 45, 50) are found to be monitorable under this fairness assumption.

On the other hand, under this assumption some patterns result in predictive monitors, which output conclusive verdicts earlier than those without assumptions. For showing it, we generated 500 random traces (uniformly distributed), each with 50 states, under the assumption (thus the monitor outputs cannot be *out-of-model*). For each pair of monitors (with and without assumption), we record two numbers of states before reaching a conclusive verdict. Whenever the two numbers are the same, the related plot is omitted. In summary, fifteen monitors (Pattern 25, 27, 29, 37, 38, 39, 40, 41, 42, 43, 44, 45, 49, 50, 54) are predictive, and five of them (Pattern 29, 37, 41, 49, 54) have more than 50 traces showing the difference. Fig. 9.1 shows, for example, the tests of Pattern 29 (*s responds to p after q until r*) and 49 (s, t *responds to p after q until r*). The time needed to run the tests on all traces is almost negligible (less than one second) for each pattern.

The *interesting* traces (which show predictive verdicts) can be also obtained by model checking on monitors generated into SMV models. Suppose we have two monitors `M1` (with assumption) and `M2` (w/o assumption), and `AV` := (`M1._concl` $\wedge \neg$ `M2._concl`) (the assumption is valuable iff `M1` has reached conclusive verdicts ($\top^{\mathrm{a}}$, $\bot^{\mathrm{a}}$ or $\times$) while `M2` has not), then the counter-example of model-checking $\neg\mathbf{F}$ `AV` (`AV` *cannot* eventually be true) will be a trace showing that the monitor `M1` is predictive: $\emptyset, \{p, s\}, \emptyset, s, p, \emptyset, \ldots$. Furthermore, it is possible to find a trace such that the distance of conclusive outputs from the two mon-

Table 9.1: Eight long formulae from Dwyer's patterns

ID	Pattern	LTL
13	Trans to p occ. at most twice (betw. q and r)	$\mathbf{G}\,((q \wedge \mathbf{F}\,r) \rightarrow ((\neg p \wedge \neg r)\,\mathbf{U}\,(r \vee ((p \wedge \neg r)\,\mathbf{U}\,(r \vee ((\neg p \wedge \neg r)\,\mathbf{U}\,(r \vee ((p \wedge \neg r)\,\mathbf{U}\,(r \vee (\neg p\,\mathbf{U}\,r))))))))))$
14	Trans to p occ. at most twice (after q until r)	$\mathbf{G}\,(q \rightarrow ((\neg p \wedge \neg r)\,\mathbf{U}\,(r \vee ((p \wedge \neg r)\,\mathbf{U}\,(r \vee ((\neg p \wedge \neg r)\,\mathbf{U}\,(r \vee ((p \wedge \neg r)\,\mathbf{U}\,(r \vee (\neg p\,\mathbf{W}\,r) \vee \mathbf{G}\,p)))))))))$
39	p precedes s,t (after q until r)	$\mathbf{G}\,(q \rightarrow (\neg(s \wedge (\neg r) \wedge \mathbf{X}\,(\neg r\,\mathbf{U}\,(t \wedge \neg r)))\,\mathbf{U}\,(r \vee p) \vee \mathbf{G}\,(\neg(s \wedge \mathbf{X}\,\mathbf{F}\,t))))$
43	p responds to s,t (between q and r)	$\mathbf{G}\,((q \wedge \mathbf{F}\,r) \rightarrow (s \wedge \mathbf{X}\,(\neg r\,\mathbf{U}\,t) \rightarrow \mathbf{X}\,(\neg r\,\mathbf{U}\,(t \wedge \mathbf{F}\,p)))\,\mathbf{U}\,r)$
44	p responds to s,t (after q until r)	$\mathbf{G}\,(q \rightarrow (s \wedge \mathbf{X}\,(\neg r\,\mathbf{U}\,t) \rightarrow \mathbf{X}\,(\neg r\,\mathbf{U}\,(t \wedge \mathbf{F}\,p)))\,\mathbf{U}\,(r \vee \mathbf{G}\,(s \wedge \mathbf{X}\,(\neg r\,\mathbf{U}\,t) \rightarrow \mathbf{X}\,(\neg r\,\mathbf{U}\,(t \wedge \mathbf{F}\,p)))))$
49	s,t responds to p (after q until r)	$\mathbf{G}\,(q \rightarrow (p \rightarrow (\neg r\,\mathbf{U}\,(s \wedge \neg r \wedge \mathbf{X}\,(\neg r\,\mathbf{U}\,t))))\,\mathbf{U}\,(r \vee \mathbf{G}\,(p \rightarrow (s \wedge \mathbf{X}\,\mathbf{F}\,t))))$
53	s,t without z responds to p (betw. q and r)	$\mathbf{G}\,((q \wedge \mathbf{F}\,r) \rightarrow (p \rightarrow (\neg r\,\mathbf{U}\,(s \wedge \neg r \wedge \neg z \wedge \mathbf{X}\,((\neg r \wedge \neg z)\,\mathbf{U}\,t))))\,\mathbf{U}\,r)$
54	s,t without z responds to p (after q until r)	$\mathbf{G}\,(q \rightarrow (p \rightarrow (\neg r\,\mathbf{U}\,(s \wedge \neg r \wedge \neg z \wedge \mathbf{X}\,((\neg r \wedge \neg z)\,\mathbf{U}\,t))))\,\mathbf{U}\,(r \vee \mathbf{G}\,(p \rightarrow (s \wedge \neg z \wedge \mathbf{X}\,(\neg z\,\mathbf{U}\,t)))))$

itors is arbitrary large. For this purpose, we can setup a bounded counter c, whose value only increases when `AV` is true and then verify if c can reach a given maximum value, say, 10. By checking the *invariance* specification $c < 10$, the counter-example will be the desired trace. Similarly, the monotonicity ($\mathbf{G}\,$`M._unknown` $\vee$ (`M._unknown` $\mathbf{U}$ `M._concl`)), the correctness (($\mathbf{F}\,$`M._true`) $\rightarrow \varphi$ and ($\mathbf{F}\,$`M._false`) $\rightarrow \neg\varphi$), and the correctness of resets ($\mathbf{X}^n$(`M._reset` $\wedge\, \mathbf{X}(\neg\,$`M._reset` $\mathbf{U}\,$`M._true`)) $\rightarrow \mathbf{X}^n\varphi$) of any monitor `M` generated from φ can also be checked in nuXmv.

9.1.3 Comparisons with RV-Monitor

The purpose here is to generate the same monitors from NuRV and RV-Monitor (rvm) and compare their performances and other characteristics. All these patterns are expressed in six Boolean variables (p, q, r, s, t and z). RV-Monitor is event-based, i.e. the alphabet is the set of these variables instead of their power set. This means our monitors can be built under the assumption that all six variables are disjoint.

Unfortunately, RV-Monitor (rvm) fails in generating monitors from eight long formulae (Pattern 13, 14, 39, 43, 44, 49, 53 and 54), shown in Table 9.1. Also it does not generate[4] monitors from all ten safety properties (Pattern 5, 7, 22, 25, 27, 40, 41, 42, 45 and 50). Eventually we got only 37 monitors out of 55 LTL patterns, and we confirmed that, whenever rvm monitors report violations, our monitors behave the same. Our 55 monitors were quickly generated in 0.467s (MacBook Pro with Intel Core i7 2.6 GHz, 4 cores) using a single core,

[4]The error message is "violation is not a supported state in this logic, ltl."

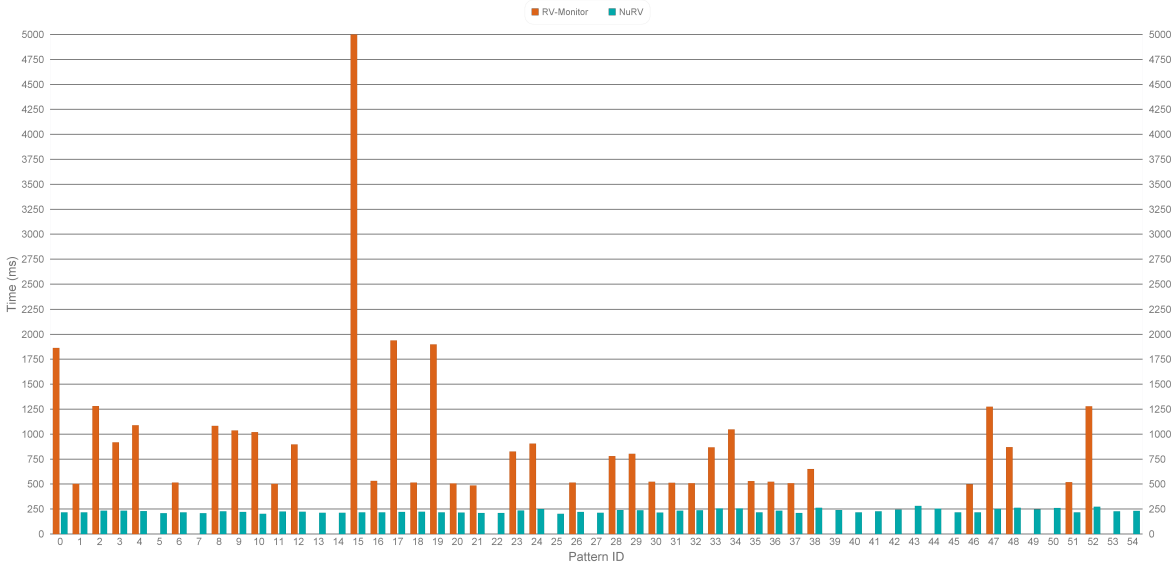


Figure 9.2: Performance of generated Java monitors on 10^7 states.

while the 37 rvm monitors were generated in 78.619s on the same machine using multiple cores.

We observed that rvm monitors does not report further violations once the first violation happens, and goes into terminal states. To get visible performance metrics we chose to reset all monitors once a violation is reported. Also, to prevent extra performance loss in rvm monitors by creating multiple monitor instances [41], we have used a single trace (stored in a vector) with 10^7 random states. For each of the 37 LTL patterns, we recorded the time (in ms) spent by both monitors (running in the same Java process), the result is shown in Fig. 9.2. Our monitors (in Java) have shown a constant-like time complexity (approx. 250ms), i.e. the time needed for processing one input trace is almost the same for all patterns. This reflects the spirit of automata-based approaches. Rvm monitors vary from 500ms to more than 6s, depending on the number of resets.

9.1.4 Comparisons with DejaVu (for ptLTL)

We compared the performance of NuRV with DejaVu on several typical ptLTL formulae. The monitoring specification of DejaVu is QTL (Quantifier Temporal Logic) with only past temporal operators [90]. Ignoring first-order quantifiers, the rest of the QTL semantics, especially for the past operator, is exactly the same as the alternative ptLTL semantics given in Definition 7.1.2. Thus the same (first-order) ptLTL property should result into equivalent monitors between DejaVu and NuRV.

However, as NuRV supports only propositional LTL, we have to simplify the original QTL formulae, shown in Table 9.2, to propositional ptLTL formulae with selective instantiations of

quantifiers, shown in Table 9.3. The QTL specification *access* requires that to access a file a user must first login (without logout) while the file must be opened before being closed. In its ptLTL version, we assume that there are just one user and one file. The QTL specification *file* says that to close a file it must be first opened without being closed yet. In its ptLTL version, we assume three such files, each has the same requirements independently. The QTL specification *fifo* describes a FIFO (First In First Out) scenario: whoever enters an area must also exit first. In its ptLTL version we consider only two such persons. DejaVu is event-based, i.e. each state of the input trace has just one (first-order) proposition. Thus NuRV must assume all involved propositions are djsjoint.

Name	QTL specification
access	$\forall user, file.\, access(user, file) \rightarrow$ $[login(user), logout(user)) \wedge [open(file), close(file))$
file	$\forall f.\, close(f) \rightarrow \exists m.\, [open(f, m), close(f))$
fifo	$\forall x.\, (enter(x) \rightarrow \neg \mathbf{Y}\, \mathbf{O}\, enter(x)) \wedge$ $(exit(x) \rightarrow \neg \mathbf{Y}\, \mathbf{O}\, exit(x)) \wedge (exit(x) \rightarrow \mathbf{Y}\, \mathbf{O}\, enter(x)) \wedge$ $(\forall y.\, (exit(y) \wedge \mathbf{O}\, (enter(y) \wedge \mathbf{Y}\, \mathbf{O}\, enter(x))) \rightarrow \mathbf{Y}\, \mathbf{O}\, exit(x))$

Table 9.2: The original QTL specification used in tests. $[p, q)$ is an abbreviation of $(\neg q)\, \mathbf{S}\, p$.

Name	ptLTL specification
access	$access \rightarrow \mathbf{Y}\, ((\neg\, logout\, \mathbf{S}\, login) \wedge (\neg\, close\, \mathbf{S}\, open))$
file	$(close[0] \rightarrow \mathbf{Y}\, (\neg close[0]\, \mathbf{S}\, open[0])) \wedge$ $(close[1] \rightarrow \mathbf{Y}\, (\neg close[1]\, \mathbf{S}\, open[1])) \wedge$ $(close[2] \rightarrow \mathbf{Y}\, (\neg close[2]\, \mathbf{S}\, open[2]))$
fifo	$(enter[0] \rightarrow \neg \mathbf{Y}\, \mathbf{O}\, enter[0]) \wedge$ $(exit[0] \rightarrow \neg \mathbf{Y}\, \mathbf{O}\, exit[0]) \wedge$ $(exit[0] \rightarrow \mathbf{Y}\, \mathbf{O}\, enter[0]) \wedge$ $((exit[1] \wedge \mathbf{O}\, (enter[1] \wedge \mathbf{Y}\, \mathbf{O}\, enter[0])) \rightarrow \mathbf{Y}\, \mathbf{O}\, exit[0]) \wedge$ $(enter[1] \rightarrow \neg \mathbf{Y}\, \mathbf{O}\, enter[1]) \wedge$ $(exit[1] \rightarrow \neg \mathbf{Y}\, \mathbf{O}\, exit[1]) \wedge$ $(exit[1] \rightarrow \mathbf{Y}\, \mathbf{O}\, enter[1]) \wedge$ $((exit[0] \wedge \mathbf{O}\, (enter[0] \wedge \mathbf{Y}\, \mathbf{O}\, enter[1])) \rightarrow \mathbf{Y}\, \mathbf{O}\, exit[1])$

Table 9.3: The ptLTL versions of QTL specifications of Table 9.2.

DejaVu first generates the corresponding monitor code (in Scala) in which a Java-based BDD library is used, then runs the generated code on CSV-based input traces. NuRV, on the other hand, runs in two modes: 1) it loads the specification and XML-based trace file into memory and then call the BDD-based `verify_property` command to get the monitoring outputs. 2) it first generates a standalone monitor funtion in C, then we write another small C program to read the trace (also in CSV) and get the monitor outputs. For the outputs, DejaVu only reports the

Property	Trace length	DejaVu		NuRV (BDD)		NuRV (C)
		compilation	trace analysis	trace loading	trace analysis	
access	100,000	4.51s	1.17s	1.35s	2.77s	0.069s
	1,000,000	4.40s	3.84s	13.5s	27.5s	0.411s
file	100,000	4.29s	1.43s	1.50s	3.04s	0.066s
	1,000,000	4.56s	5.05s	15.14s	28.91s	0.451s
fifo	100,000	5.04s	2.05s	1.64s	2.77s	0.055s
	1,000,000	4.29s	14.01s	14.75s	29.05s	0.431s

Table 9.4: Evaluations of DejaVu and NuRV.

trace indexes which corresponding to the violation of the monitoring specification, while NuRV outputs another CSV file of the same length of the input trace, each line contains the trace index and a string: `true` for $\top^a$ and `false` for $\bot^a$. For each pair of QTL and ptLTL specifications in above tables, the same random traces are generated in supported formats of DejaVu and NuRV. In all these tests, first of all, we confirmed that both tools give the same monitoring outputs which only differ in formats. Thus the only focus is on their performance and memory consumptions.

The evalution results are given in Table 9.4. For both DejaVu and NuRV (BDD) the time values consist of the time of monitor synthesis and the time for the verification of trace. For NuRV (C) the time values include only the time for loading offline traces into memory and the time for the actual verification. The test environment is a MacBook Pro running Mac OS X 10.11.6, with 2.6GHz Intel Core i7 CPU (4 cores) and 16GB memory. For each properties we generated two random traces of the length 100,000 and 1,000,000. The time spent by DejaVu consists of two parts: 1) the time of monitor synthesis, which is quite stable in all cases; 2) the time of actual verification. NuRV uses only one CPU core when executing a single batch command, the BDD-based offline monitoring also takes significant time to load the XML-based traces (the time for monitor synthesis is negligible), while the overall timing for BDD-based monitors are slightly slower than DejaVu, perphaps because both of them internally use BDD for the actual computation. On the other hand, the C-based standalone monitors generated by NuRV are much faster. In fact, these monitors are very small: all three monitor automata have less than 100 nodes, and the compiled C object files are less than 20KB. It turns out that, for both software the actual monitor synthesis time is negligible in comparison with other time-consuming work.[5]

[5]According to DejaVu developers, in DejaVu a processes are spawned to (a) compile the generated monitor in Scala, and (b) to execute it, since this is the only way to do this (compile and run another program Y) from within a program X. It is a purely operation systems technicality. There is no concurrent execution going on. Things are executed in sequential order: first compilation and the trace analysis. Also note that most of the monitor synthesis time is spent on compiling the generated Scala program (the Scala compiler is slow), and not on synthesizing it, which usually is very quick (no more than a second). Finally, DejaVu was designed for the first-order case, with the propositional case being a special case without quantifiers. The main challenge of DejaVu was how to handle quantifiers over data.

9.2 Tests for Infinite-State Monitors

The correctness of RV algorithms for infinite-state systems, beside the related theorems and proofs, lies also on the fact that, for each input trace (and RV assumptions) being tested, all five RV algorithms (`monitor1`, `monitor1_optimized`, `monitor2`, `monitor2_optimized`, and `bmc_monitor`) give the same results (except that `monitor1` and `monitor1_optimized` only give the verdicts for the last state of the input trace). Below we mainly focus on their (relative) performance.

On the other hand, the performance of RV algorithms presented in this thesis heavily depend on the performance of underlying model checkers, SMT solvers and QE procedures, as different choices of these underlying tools lead to not only monitors of different performance but also monitors handling different kind of properties and assumptions. (We have chosen MathSAT5 as the SMT solver, which provides also the QE APIs based on Fourier-Motzkin and Loos-and-Weispfenning methods (the two methods did not show any performance difference in our cases). For the purposes of model checking, we have used the "msatic3" library created by the same authors of MathSAT5. All IC3-IA calls are wrapped through the nuXmv model checker, which additionally provides the LTL translation interface. The plain BMC procedure is provided by the "msatic3" library.) Due to limitations of the underlying tools, the actual support of infinite-state systems is limited to those with linear constraints of variables, e.g. linear arithmetic on the rationals/reals, integers and mixed rational-integers, etc.

We mainly focus on the overall complexity behavior and relative performance of the algorithms with respect to optimizations. In particular, we want to show that, the performance of monitors changes *dramatically* after the optimizations described in Section 6.4.

9.2.1 Tests on the motivating example

The actual monitoring results on the motivating example in Section 6.1 are the same with those expected. The total execution time for the offline monitoring of the two sample properties on the three-state sample trace u is about: 2.3s (`monitor1_optimized`), 13s (`monitor2_optimized`) and 0.9s (`bmc_monitor`). Note that `monitor1_optimized` is faster than `monitor2_optimized` mostly because the input trace is very short and it only needs to output the verdict for the last input state. On the other hand, the BMC search bound (`max_k`) in `bmc_monitor` was set to 50, while the execution time can be shorten to 0.6s if `max_k` were set to 30.

9.2.2 Tests on Dwyer's LTL patterns

We use again Dwyer's LTL patterns [61] (55 in total[6]) as the main LTL benchmark, which comes from a wide coverage of practical specifications and has a good coverage on different kind of LTL properties. The original patterns involve six Boolean variables p, q, r, s, t, z, and to adapt them for infinite-state scenarios we have changed to use one integer variable i and one real variable x for the replacements of q and r: $q \leftrightarrow 0 \leqslant i$ and $r \leftrightarrow 0.0 \leqslant x$. Then we generated random traces where $i \in [-500, 500]$ and $x \in [-0.500, 0.500]$ are uniformly chosen, such that q and r become random in the original patterns. Furthermore, we choose a model with fairness as the RV assumptions, in which the p-transition (i.e., from $\neg p$ to p) happens at most 4 times. The purpose of this assumption is to force the monitor to arrive at $\times$ verdicts at certain moments, so that the related monitors could go through different verdicts as much as possible.

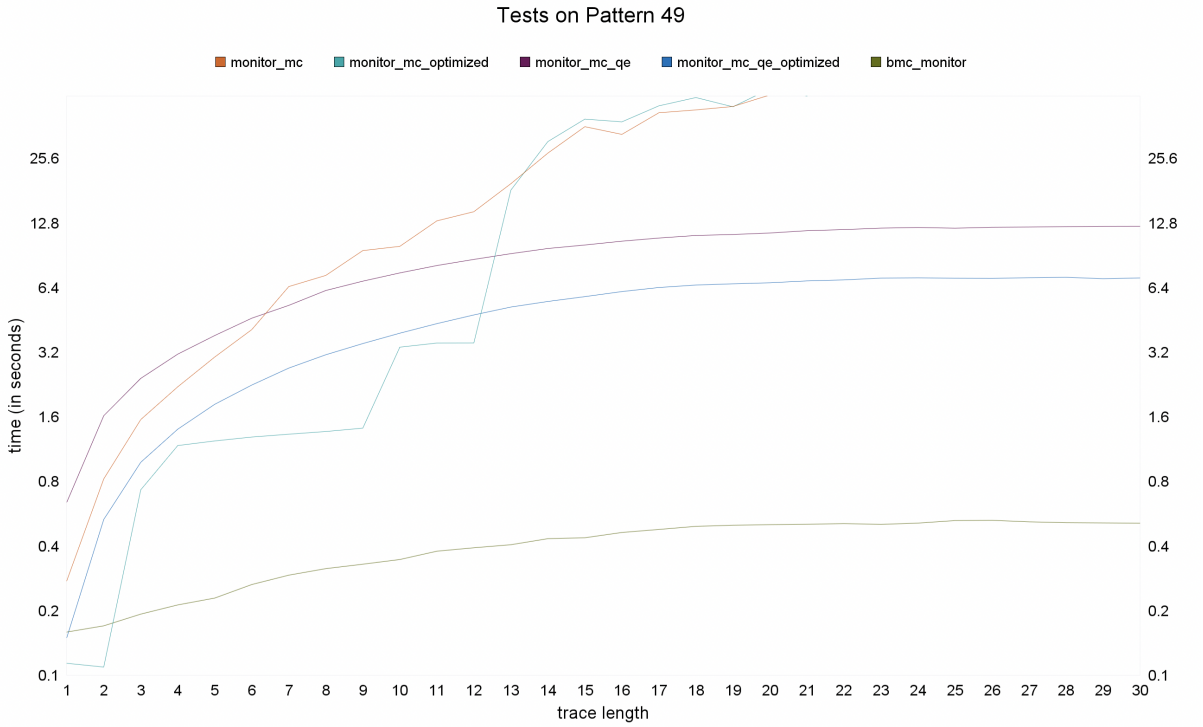


Figure 9.3: Performance of five RV algorithms on Pattern 49

Fig. 9.3 gives the relative performance of all five RV algorithms on Pattern 49 (s, t responds to p after q until r, results are similar for other patterns), a complex property for showing the performance of RV algorithms in practical. The monitors are generated under the above chosen assumptions, which is expressed as an infinite-state model. The length of input traces increases

[6]See also `https://matthewbdwyer.github.io/psp/patterns/ltl.html`.

from 1 to 30. Each plot represents the average time of a monitor spent on certain length of three random traces. We found that 1) the optimizations on `monitor1` and `monitor2` indeed work; 2) `bmc_monitor` is about 10x faster than `monitor2_optimized`, which is again about 10x faster than `monitor1_optimized`. Note that these relative performance ("10x faster") between different monitors is based middle-sized traces: if the trace is too short, usually `monitor1` is faster.

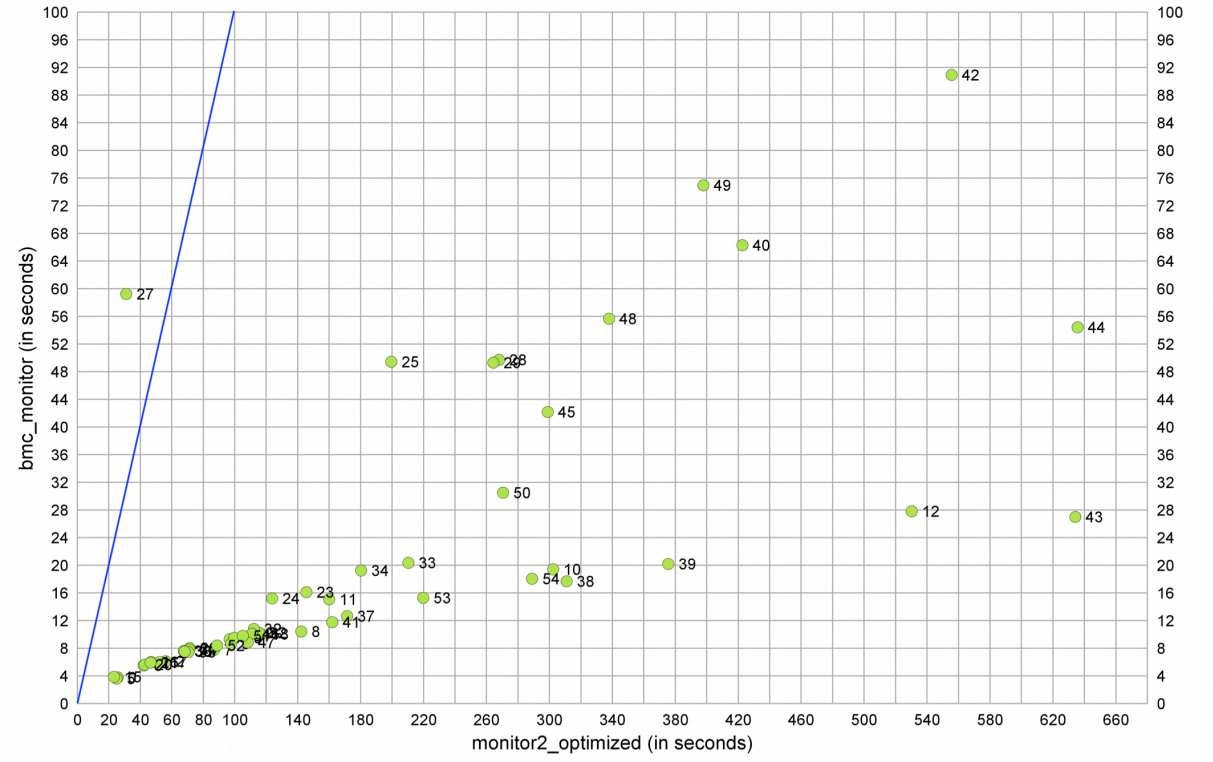


Figure 9.4: Performance of `bmc_monitor` and `monitor2_optimized` on all patterns

Fig. 9.4 additionally shows the relative performance between `bmc_monitor` and `monitor2_optimized`. For each LTL pattern, the two monitors with the fairness assumptions take 10 random traces as input, each with 50 states. The x- and y-axes of each plot (identified by pattern ID) corresponds to the overall time spent on the two monitors. For most patterns (and also on average), `bmc_monitor` is about 10x faster than `monitor2_optimized`.

Chapter 10

HOL Formalization

10.1 Introduction

In this chapter, we present a partial formalization on the correctness of the finite-state ABRV monitors (Chapter 5, together with the formal version of the equivalence between ptLTL and LTL_3 semantics given in Chapter 7. The main tool used here is Higher Order Logic theorem prover, also called *HOL4*[1]

Most of the RV tools developed so far, including NuRV, are not formally verified. Actually this is a common problem of most working software in nowadays. For things like model checkers, runtime monitors and even SAT/SMT solvers, the following chain of issues may cause their potential failures in applications:

1. The algorithms used by the software are usually described in academic papers with informal, paper-and-pencil proofs, and in rare cases the algorithms may give wrong outputs (at certain boundary inputs, if not all.)

2. The algorithms were mathematically correct, but the software engineering process (i.e. programming or coding) has introduced the so-called "bugs", which some times cannot be found by unit tests or other test methods.

3. Both algorithms and programming implementations were correct, but the compiler used in building the software has (usually very rare) bugs, causing wrong translations from the high-level programming language(s) to low-level machine assembly code.

4. Finally, even all above aspects were correct, the computer hardware where the software runs has design problems, typically at the CPU but nowadays has extended to GPU and other components due to their high complexities.

[1]The official site of HOL4 is at `https://hol-theorem-prover.org`.

The above points 1) can be resolved by (interactive) theorem provers, most if not all. The point 2) can also be resolved by some theorem provers but usually the best (or feasible) approach is to first develop formal proofs and then generate working program code from the proofs. The point 3) requires verified programming language compilers, and perhaps also a verified operating system, at least the kernel part. They are very rare, but see, e.g., the CakeML project and the seL4 verified microkernel. The point 4) can be either resolved by (interactive) theorem provers or automatic verification tools like model checkers, by CPU vendors like Intel, AMD and ARM.

A classic and famous example of the above point 4) is the Pentium `FDIV` bug. It is a hardware bug affecting the floating-point unit (FPU) of the early Intel Pentium processors. Because of the bug, the processor would return incorrect binary floating point results when dividing certain pairs of high-precision numbers. (See `https://en.wikipedia.org/wiki/Pentium_FDIV_bug` for more details.)

A quite new story related to the above point 2) is recently experienced by the author of this thesis. The old version of the famous MiniSAT SAT solver, version 1.14, obsoleted but still used inside HOL theorem prover, has a bug so far only observed on Linux/ARM64, where for certain inputs it should report UNSAT but actually SAT (with fake models). The root cause was that the author of MiniSAT uses C type `char` for holding signed integers (ranged from -127 to 128), but unfortunately `char` is `unsigned` on Linux/ARM64. (The root cause was found by another proof engineer using the *UndefinedBehaviorSanitizer* (UBSan)[2] from LLVM project, and the fix is quite simple, just replacing most `char` to `int8_t` in the code.)

Some classes of tools is immune to at least the above points 1) and 2), but only at certain classes of inputs, and the results thus can be trusted. For instance, if an SAT/SMT solver returns "SAT" (satisfied) with some models, in principle these outputs can be either manually verified or verified by third-party tools. Similarily, if a model checker returns "false" with counter examples, in principle one can easily verify the output by using the counter examples. On the other hand, if SAT/SMT solvers simply returns "UNSAT", or model checkers simply returns "true", it's possible that the results are actually wrong due to software bugs.

In the field of Interactive Theorem Proving (ITP), software tools called *Theorem Provers* or *Proof assistants* are used for the verification tasks.[3] When proving a theorem, a similar duality like the above cases can be observed: if the proof completes successfully, it usually can be fully trusted, because the core inference enginee of theorem provers is usually written in several hundreds lines of programming language code and has been well checked for its correctness in logical, algorithmic and software engineering levels. Potential bugs in the other part of the theorem provers can only cause the proof not completed, which coincides also with the case that the original theorem statements are actually wrong.

[2] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

[3] Note that modern theorem provers usually contain automatic decision procedures or semi-automatic provers, which greatly improves the productivities of proof engineers on many practical logic fragments (usually decidable ones).

Back to the RV area, Joshua Schneider at al. has done the first (at least the first one that the author is aware of) formalization of their MFOTL-based monitoring algorithm [134], in Isabelle proof assistant[4]. Using Isabelle, they managed to generate from the formal proofs to working monitor software, which is slower than previously hand-coded programs but now is formally verified (thus the output can be fully trusted.)

For us, one possible research goal is to formally verify its core monitoring algorithms (presented in this thesis and related papers) in a theorem prover. However, it would be infeasible to directly verify the programming code of NuRV, which is written in C and C++. By choosing Higher Order Logic (HOL4), which is essentially a general programming language platform (Standard ML) enhanced with theorem proving kernal and libraries, it is in theory possible to re-implement the RV algorithms to produce another monitor synthesis tool compatible with NuRV. Beside the partial formalization itself, a key message is to convince the audience that, there are already good formalizations of LTL, automata, etc., which can be used as the working basis for a possible, full formalization of RV algorithms or even a runtime monitor (or monitor synthesis tool.)

10.2 Higher Order Logic (HOL)

Higher Order Logic (HOL) [94, 143] traces its roots back to the *Logic of Computable Functions (LCF)* [82, 124] by Robin Milner and others since 1972. It is a variant of Church's Simple Theory of Types (STT) [43], plus a higher order version of Hilbert's choice operator ε, Axiom of Infinity, and Rank-1 (prenex) polymorphism. HOL4 has implemented the original HOL, while some other theorem provers in the HOL family (e.g. Isabelle/HOL) have certain extensions. Indeed, HOL has considerably simpler logical foundations than most other theorem provers. As a consequence, theories and proofs verified in HOL are easier to understand for people who are not familiar with more advanced dependent type theories, e.g. the *Calculus of Inductive and Co-inductive Constructions* implemented by Coq.

The word "HOL" is both the name of a logic system and the software which implements this logic. HOL4 is the latest version of the software, which is implemented in Standard ML (SML). SML as a general programming language plays three different roles:

- The underlying implementation language for the core HOL engine;
- The language in which proof tactics are implemented;
- The interface language of the HOL proof scripts and interactive shell.

Moreover, using the same language HOL4 users can write complex automatic verification tools by calling HOL's theorem proving facilities.

[4] https://isabelle.in.tum.de

The HOL logic is a formal system of typed logical terms. The types are expressions that denote sets (in the universe $\mathcal{U}$). HOL type system is much simpler than those based on dependent types and other type theories. There are four kinds of types in the HOL logic, as illustrated in Fig. 10.1 for its BNF grammar. In HOL, the standard atomic types *bool* and *ind* denote, respectively, the distinguished two-element set **2** and the distinguished infinite set **I**.

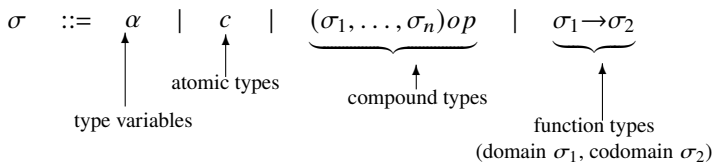


Figure 10.1: HOL's type grammar

HOL terms represent elements of the sets denoted by their types. There are four kinds of HOL terms, which can be described (in simplified forms) by the BNF grammar in Fig. 10.2. (See [94] for a complete description of HOL, including the primitive derivative rules to be mentioned below.)

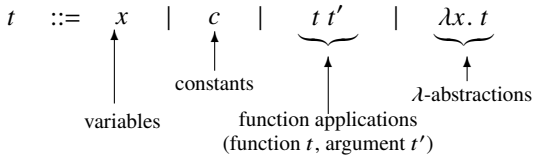


Figure 10.2: HOL's term grammar

The deductive system of HOL is specified by eight primitive derivative rules:

1. Assumption introduction (`ASSUME`);
2. Reflexivity (`REFL`);
3. β-conversion (`BETA_CONV`);
4. Substitution (`SUBST`);
5. Abstraction (`ABS`);
6. Type instantiation (`INST_TYPE`);
7. Discharging an assumption (`DISCH`);
8. Modus Ponens (`MP`).

All proofs are eventually reduced to applications of the above rules, which also give the semantics of two fundamental logical connectives, equality (=) and implication ($\Rightarrow$). The remaining logical connectives and first-order quantifiers, including the logical true (T) and false (F), are further defined as λ-functions:

$\vdash$ T $\stackrel{\text{def}}{=}$ $((\lambda x_{bool}.\ x) = (\lambda x_{bool}.\ x))$

$\vdash$ $\forall \stackrel{\text{def}}{=} \lambda P_{\alpha\to bool}.\ P = (\lambda x.\ \mathrm{T})$

$\vdash$ $\exists \stackrel{\text{def}}{=} \lambda P_{\alpha\to bool}.\ P(\varepsilon\ P)$

$\vdash$ F $\stackrel{\text{def}}{=} \forall b_{bool}.\ b$

$\vdash$ $\neg \stackrel{\text{def}}{=} \lambda b.\ b \Rightarrow \mathrm{F}$

$\vdash$ $\wedge \stackrel{\text{def}}{=} \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$

$\vdash$ $\vee \stackrel{\text{def}}{=} \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)$

$\vdash$ One_One $\stackrel{\text{def}}{=} \lambda f_{\alpha\to\beta}.\ \forall x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2)$

$\vdash$ Onto $\stackrel{\text{def}}{=} \lambda f_{\alpha\to\beta}.\ \forall y.\ \exists x.\ y = f\ x$

$\vdash$ Type_Definition $\stackrel{\text{def}}{=} \lambda P_{\alpha\to bool}\ rep_{\beta\to\alpha}.$ One_One $rep\ \wedge\ (\forall x.\ Px = (\exists y.\ x = rep\ y))$

The last logical constant, Type_Definition, can be used to define new HOL types as bijections of subsets of existing types [120]. HOL `Datatype` package [93, 121] automates this tedious process. Finally, the whole HOL *standard* theory is based on the following four axioms:[5]

`BOOL_CASES_AX`	$\vdash \forall b.\ (b = \mathrm{T})\ \vee\ (b = \mathrm{F})$
`ETA_AX`	$\vdash \forall f_{\alpha\to\beta}.\ (\lambda x.\ f\ x) = f$
`SELECT_AX`	$\vdash \forall P_{\alpha\to bool}\ x.\ P\ x \Rightarrow P(\varepsilon\ P)$
`INFINITY_AX`	$\vdash \exists f_{ind\to ind}.$ One_One $f\ \wedge\ \neg$(Onto f)

Usually the above four axioms are the only axioms allowed in conventional formalisation projects in HOL4: adding new axioms manually may break logical consistency.

Remark 10.2.1. *HOL4 has been used by the author in a formalization of Milner's Calculus of Concurrent Systems with some new results obtained [147], during the PhD period. In comparision with other theorem provers, HOL4 is a good choice for its rich core theories (especially in formal mathematics like Probability Theory), good community support and its simple and elegant logical foundation, which is easy to understand while strong enough for most applications.*

10.3 Linear Temporal Logic in HOL

The most difficult part of a possible formalization of RV algorithms in this thesis, in any theorem prover, is to choose a way to formalize LTL and the translation from LTL to ω-automata, which plays a core role in LTL model checking and several work are currently available in HOL.

[5]HOL is strictly weaker than ZFC (the Zermelo-Frankel set theory with the Axiom of Choice), thus not all theorems valid in ZFC can be formalised in HOL. (See [94] for more details.)

There are in general two approaches to formalize another logic system in theorem provers like HOL: *deep* and *shallow embeddings*, each has pros and cons.

In shallow embeddings of LTL, each "atomic" proposition is a function of the type `num -> bool`, i.e. a total function mapping all natural numbers to Boolean values. By evaluating this function on each natural numbers, one can get its truth values along the whole infinite trace. Each temporal operator is defined separately, thus there is no distinguish between primitive temporal operators (like **X** and **U**) and derived operators (like **G** and **F**) at all. For example, in HOL's official core theory `Temporal_LogicTheory` (contributed by Klaus Schneider et al. [135, 138]) the *next* (**X**) and *always* (**G**) operator can be defined below:

```
[NEXT]
⊢ NEXT (P :num -> bool) = (λ(t :num). P (SUC t))
[ALWAYS]
⊢ ALWAYS (P :num -> bool) (t0 :num) ⟺ ∀(t :num). P (t + t0)
```

Another character of shallow embeddings is that Boolean connectives in LTL are nothing but the original Boolean connectives in the host logic systems (HOL itself, in this case). The following simple lemma shows that the logical *and* can be safely moved out from "atomic" propositions to the temporal formulae levels:

```
[AND_NEXT]
⊢ ∀Q P. NEXT (λt. P t ∧ Q t) = (λt. NEXT P t ∧ NEXT Q t)
```

One can imagine that, each temporal formula represented in this way can be satisfied by a set of infinite traces. ω-automata, on the other hand, are nothing but the same kind of Boolean functions, only with existential quantified variables as input symbols. The "translation" from LTL to ω-automata (in particular, Büchi automata) is nothing but a process of translating temporal formulae into existential quantified variables:

```
[BUECHI_TRANSLATION]
⊢ (Phi (NEXT phi) ⟺ ∃q0 q1. T ∧ (∀t. (q0 t ⟺ phi t) ∧ (q1 t ⟺ q0 (t + 1))) ∧ Phi q1) ∧
  (Phi (ALWAYS a) ⟺
   ∃q. T ∧ (∀t. q t ⟺ a t ∧ q (t + 1)) ∧ (∀t1. ∃t2. a (t1 + t2) ⇒ q (t1 + t2)) ∧ Phi q) ∧
  (Phi (EVENTUAL a) ⟺
   ∃q. T ∧ (∀t. q t ⟺ a t ∨ q (t + 1)) ∧ (∀t1. ∃t2. q (t1 + t2) ⇒ a (t1 + t2)) ∧ Phi q) ∧
  (Phi (a SUNTIL b) ⟺
   ∃q. T ∧ (∀t. q t ⟺ b t ∨ a t ∧ q (t + 1)) ∧
       (∀t1. ∃t2. q (t1 + t2) ⇒ ¬a (t1 + t2) ∨ b (t1 + t2)) ∧ Phi q) ∧
  (Phi (a UNTIL b) ⟺
   ∃q. T ∧ (∀t. q t ⟺ b t ∨ a t ∧ q (t + 1)) ∧
       (∀t1. ∃t2. ¬q (t1 + t2) ⇒ ¬a (t1 + t2) ∨ b (t1 + t2)) ∧ Phi q) ∧
  (Phi (a SWHEN b) ⟺
   ∃q. T ∧ (∀t. q t ⟺ if b t then a t else q (t + 1)) ∧
       (∀t1. ∃t2. q (t1 + t2) ⇒ b (t1 + t2)) ∧ Phi q) ∧
  (Phi (a WHEN b) ⟺
   ∃q. T ∧ (∀t. q t ⟺ if b t then a t else q (t + 1)) ∧
       (∀t1. ∃t2. q (t1 + t2) ∨ b (t1 + t2)) ∧ Phi q) ∧
  (Phi (a SBEFORE b) ⟺
   ∃q. T ∧ (∀t. q t ⟺ ¬b t ∧ (a t ∨ q (t + 1))) ∧
```

$$
\begin{array}{l}
\quad (\forall t_1.\ \exists t_2.\ q\ (t_1 + t_2) \Rightarrow a\ (t_1 + t_2) \vee b\ (t_1 + t_2)) \wedge Phi\ q) \wedge \\
(Phi\ (a\ \texttt{BEFORE}\ b) \iff \\
\ \exists q.\ \texttt{T} \wedge (\forall t.\ q\ t \iff \neg b\ t \wedge (a\ t \vee q\ (t + 1))) \wedge \\
\quad (\forall t_1.\ \exists t_2.\ \neg q\ (t_1 + t_2) \Rightarrow a\ (t_1 + t_2) \vee b\ (t_1 + t_2)) \wedge Phi\ q)
\end{array}
$$

The pros of shallow embeddings is at their elegance: they uses things from the host logic systems as much as possible. The cons is that shallow embeddings cannot easily support finite traces, and the lacking of fixed alphabet (i.e. the set of involved Boolean variables) may create extra difficulties in support partial observability found in ABRV.

Deep embeddings, on the other side, defines LTL as an inductive algebraic structure: a system of logical terms inductively built from atomic propositions, (fake) logical connectives and temporal operators. Here, all primitive temporal operators must be included as the very definition, then other temporal operators can be defined as syntax sugars of primitive temporal operators. For example, in HOL's official `temporal_deep` example (contributed by Thomas Tüerk et al. [148, 149]), Full LTL (i.e. with both future and past operators) is defined in the following code as an algebraic datatype in HOL [121]:

```
Datatype :
  ltl = LTL_PROP    ('prop prop_logic)
      | LTL_NOT      ltl
      | LTL_AND     (ltl # ltl)
      | LTL_NEXT     ltl          (* X in NuSMV *)
      | LTL_SUNTIL  (ltl # ltl)   (* U in NuSMV *)
      | LTL_PSNEXT   ltl          (* Y in NuSMV *)
      | LTL_PSUNTIL (ltl # ltl)   (* S in NuSMV *)
End
```

The standard semantics (over infinite traces), Definition 3.3.2, must be explicitly and inductively formalized: (Note that in shallow embeddings the semantics is directly associated with the definition of each temporal operator)

```
[LTL_SEM_TIME_def]
⊢ (∀v t f. LTL_SEM_TIME t v (LTL_NOT f) ⟺ ¬LTL_SEM_TIME t v f) ∧
  (∀v t f2 f1.
     LTL_SEM_TIME t v (LTL_AND (f1,f2)) ⟺ LTL_SEM_TIME t v f1 ∧ LTL_SEM_TIME t v f2) ∧
  (∀v t b. LTL_SEM_TIME t v (LTL_PROP b) ⟺ P_SEM (v t) b) ∧
  (∀v t f. LTL_SEM_TIME t v (LTL_NEXT f) ⟺ LTL_SEM_TIME (SUC t) v f) ∧
  (∀v t f2 f1.
     LTL_SEM_TIME t v (LTL_SUNTIL (f1,f2)) ⟺
     ∃k. k ≥ t ∧ LTL_SEM_TIME k v f2 ∧ ∀j. t ≤ j ∧ j < k ⇒ LTL_SEM_TIME j v f1) ∧
  (∀v t f. LTL_SEM_TIME t v (LTL_PREV f) ⟺ t > 0 ∧ LTL_SEM_TIME (PRE t) v f) ∧
  ∀v t f2 f1.
    LTL_SEM_TIME t v (LTL_SINCE (f1,f2)) ⟺
```

$\exists k.\ k \leq t \wedge$ LTL_SEM_TIME $k\ v\ f_2 \wedge \forall j.\ k < j \wedge j \leq t \Rightarrow$ LTL_SEM_TIME $j\ v\ f_1$

[LTL_SEM_def]
$\vdash$ LTL_SEM $v\ f \iff$ LTL_SEM_TIME $0\ v\ f$

Furthermore, for LTL variants like ptLTL it is possible to define special predicate to limit the use of temporal operators:[6]

```
[IS_PAST_LTL_def]
⊢ (∀b. IS_PAST_LTL (LTL_PROP b) ⟺ T) ∧
  (∀f. IS_PAST_LTL (LTL_NOT f) ⟺ IS_PAST_LTL f) ∧
  (∀f2 f1. IS_PAST_LTL (LTL_AND (f1,f2)) ⟺ IS_PAST_LTL f1 ∧ IS_PAST_LTL f2) ∧
  (∀f. IS_PAST_LTL (LTL_PREV f) ⟺ IS_PAST_LTL f) ∧
  (∀f2 f1. IS_PAST_LTL (LTL_SINCE (f1,f2)) ⟺ IS_PAST_LTL f1 ∧ IS_PAST_LTL f2) ∧
  (∀f. IS_PAST_LTL (LTL_NEXT f) ⟺ F) ∧ ∀f2 f1. IS_PAST_LTL (LTL_SUNTIL (f1,f2)) ⟺ F
```

The translation of LTL deep embeddings into ω-automata (more precisely, the generalized Büchi automata) involves many formal definitions which we do not use so far. Here we only show the main translation theorem without explaination in details (roughly speaking, what LTL_TO_GEN_BUECHI___EXTEND_def has for each temporal operator is equivalent with LTL translation algorithm given in Section 3.7.

```
[LTL_TO_GEN_BUECHI_def]
⊢ LTL_TO_GEN_BUECHI l b1 b2 =
  LTL_TO_GEN_BUECHI___EXTEND l b1 b2 EMPTY_LTL_TO_GEN_BUECHI_DS

[LTL_TO_GEN_BUECHI___EXTEND_def]
⊢ (∀p b2 b1 DS.
     LTL_TO_GEN_BUECHI___EXTEND (LTL_PROP p) b1 b2 DS =
     ((λsv. p),
      EXTEND_IV_BINDING_LTL_TO_GEN_BUECHI_DS DS {(LTL_PROP p,b1,b2,(λsv. p))}
        (P_USED_VARS p))) ∧
  (∀l b2 b1 DS.
     LTL_TO_GEN_BUECHI___EXTEND (LTL_NOT l) b1 b2 DS =
     (let
        (pf'1,DS'1) = LTL_TO_GEN_BUECHI___EXTEND l b2 b1 DS
      in
        ((λsv. P_NOT (pf'1 sv)),
         EXTEND_IV_BINDING_LTL_TO_GEN_BUECHI_DS DS'1
           {(LTL_NOT l,b1,b2,(λsv. P_NOT (pf'1 sv)))} ∅))) ∧
  (∀l2 l1 b2 b1 DS.
     LTL_TO_GEN_BUECHI___EXTEND (LTL_AND (l1,l2)) b1 b2 DS =
     (let
        (pf'1,DS'1) = LTL_TO_GEN_BUECHI___EXTEND l1 b1 b2 DS;
```

[6] Another way is to define a *subtype* as a bijection from the set of full LTL formulae to a subset satisfying the same predicate. However, in HOL tradition proof engineers tend to simply add more antecedents into theorems to limit the applicability of theorems to a conceptual subset of involved types.

```
      (pf'_2,DS'_2) = LTL_TO_GEN_BUECHI___EXTEND l_2 b_1 b_2 DS'_1
    in
      ((λ sv. P_AND (pf'_1 sv,pf'_2 sv)),
       EXTEND_IV_BINDING_LTL_TO_GEN_BUECHI_DS DS'_2
         {(LTL_AND (l_1,l_2),b_1,b_2,(λ sv. P_AND (pf'_1 sv,pf'_2 sv)))} ∅))) ∧
(∀ l b_2 b_1 DS.
   LTL_TO_GEN_BUECHI___EXTEND (LTL_NEXT l) b_1 b_2 DS =
   (let
      (pf'_1,DS'_1) = LTL_TO_GEN_BUECHI___EXTEND l b_1 b_2 DS
    in
      ((λ sv. P_PROP (sv DS'_1.SN)),
       EXTEND_LTL_TO_GEN_BUECHI_DS DS'_1 1 [] ∅
         [(λ sv. XP_EQUIV (XP_PROP (sv DS'_1.SN),XP_NEXT (pf'_1 sv)))] []
         {(LTL_NEXT l,b_1,b_2,(λ sv. P_PROP (sv DS'_1.SN)))}))) ∧
(∀ l b_2 b_1 DS.
   LTL_TO_GEN_BUECHI___EXTEND (LTL_PREV l) b_1 b_2 DS =
   (let
      (pf'_1,DS'_1) = LTL_TO_GEN_BUECHI___EXTEND l b_1 b_2 DS
    in
      ((λ sv. P_PROP (sv DS'_1.SN)),
       EXTEND_LTL_TO_GEN_BUECHI_DS DS'_1 1 [(DS'_1.SN,F)] ∅
         [(λ sv. XP_EQUIV (XP_NEXT_PROP (sv DS'_1.SN),XP_CURRENT (pf'_1 sv)))] []
         {(LTL_PREV l,b_1,b_2,(λ sv. P_PROP (sv DS'_1.SN)))}))) ∧
(∀ l_2 l_1 b_2 b_1 DS.
   LTL_TO_GEN_BUECHI___EXTEND (LTL_SUNTIL (l_1,l_2)) b_1 b_2 DS =
   (let
      (pf'_1,DS'_1) = LTL_TO_GEN_BUECHI___EXTEND l_1 b_1 b_2 DS;
      (pf'_2,DS'_2) = LTL_TO_GEN_BUECHI___EXTEND l_2 b_1 b_2 DS'_1
    in
      ((λ sv. P_PROP (sv DS'_2.SN)),
       EXTEND_LTL_TO_GEN_BUECHI_DS DS'_2 1 [] ∅
         [(λ sv.
               XP_EQUIV
                 (XP_PROP (sv DS'_2.SN),
                  XP_OR
                    (XP_CURRENT (pf'_2 sv),
                     XP_AND (XP_CURRENT (pf'_1 sv),XP_NEXT_PROP (sv DS'_2.SN)))))]
         (if b_1 then [(λ sv. P_IMPL (P_PROP (sv DS'_2.SN),pf'_2 sv))] else [])
         {(LTL_SUNTIL (l_1,l_2),b_1,b_2,(λ sv. P_PROP (sv DS'_2.SN)))}))) ∧
∀ l_2 l_1 b_2 b_1 DS.
  LTL_TO_GEN_BUECHI___EXTEND (LTL_SINCE (l_1,l_2)) b_1 b_2 DS =
  (let
     (pf'_1,DS'_1) = LTL_TO_GEN_BUECHI___EXTEND l_1 b_1 b_2 DS;
     (pf'_2,DS'_2) = LTL_TO_GEN_BUECHI___EXTEND l_2 b_1 b_2 DS'_1
   in
     ((λ sv. P_OR (pf'_2 sv,P_AND (pf'_1 sv,P_PROP (sv DS'_2.SN)))),
      EXTEND_LTL_TO_GEN_BUECHI_DS DS'_2 1 [(DS'_2.SN,F)] ∅
```

```
[(λ sv.
      XP_EQUIV
        (XP_NEXT_PROP (sv DS'_2.SN),
         XP_OR
           (XP_CURRENT (pf'_2 sv),
            XP_AND (XP_CURRENT (pf'_1 sv),XP_PROP (sv DS'_2.SN)))))] []
{(LTL_SINCE (l_1,l_2),b_1,b_2,
  (λ sv. P_OR (pf'_2 sv,P_AND (pf'_1 sv,P_PROP (sv DS'_2.SN))))})

[LTL_TO_GEN_BUECHI_THM]
⊢ LTL_TO_GEN_BUECHI_DS___SEM (SND (LTL_TO_GEN_BUECHI l b_1 b_2)) ∧
  (l,b_1,b_2,FST (LTL_TO_GEN_BUECHI l b_1 b_2)) ∈ (SND (LTL_TO_GEN_BUECHI l b_1 b_2)).B
```

10.4 Partial Formalization of Main Theorem 5.1.2

We provide a partial formalization of the main RV theorem, Theorem 5.1.2, without support of partial observability and resets. Full proof scripts can be found in Appendix, Section A.3.

For any finite trace u, an infinite traces i is said to *extend* u if there exists another infinite trace c such that $i = u \cdot c$:[7]

```
[extends_def]
i extends u ≝ ∃c. i = u + c
```

Here we formalize the RV assumptions as a (possibly infinite) set of infinite traces, i.e. the language of an FTS. An finite trace u is said to be *compatible* with a model K, if after extending u into an infinite trace, this infinite trace is in the language of K:

```
[compatible_def]
compatible u K ≝ ∃c. u + c ∈ K
```

Next, the ABRV monitor output table is defined by mapping two Boolean parameters (they will correspond to the emptiness of r_φ and $r_{\neg\varphi}$ in the informal proofs of Theorem 5.1.2) to LTL_3 verdicts:

```
[LTL4_output_def]
LTL4_output T T ≝ unknown
LTL4_output T F ≝ true
LTL4_output F T ≝ false
LTL4_output F F ≝ error
```

The "belief states" here (or we can call it a "belief run") is defined as a set of infinite traces *compatible* with the current input trace prefix w.r.t. an LTL property (and the model):

```
[GEN_LTL4_belief_run_def]
GEN_LTL4_belief_run K phi u t ≝ {i | i extends u ∧ LTL_SEM_TIME t i phi ∧ i ∈ K}
```

Note that `GEN_LTL4_belief_run` is the reason that we call this formal proof an "abstract" one: it is a correct math definition but from it one does not directly get a working algorithm: to

[7] In the formalization, trace concatenation $\cdot$ is overloaded by the sum operator +.

actually compute the set of infinite traces, or a set of automata locations, one has to first translate LTL to ω-automata and compute forward images w.r.t. to each state in the input trace prefixes.

Now, the ABRV monitor can be defined below, using `LTL4_output` and `GEN_LTL4_belief_run`:

```
[ABRV_monitor_def]
ABRV_monitor K phi u t ≝
  LTL4_output (GEN_LTL4_belief_run K phi u t ≠ ∅)
    (GEN_LTL4_belief_run K (LTL_NOT phi) u t ≠ ∅)
```

On the other hand, the formal definition of ABRV-LTL semantics precisely follows Definition 4.4.1:

```
[LTL4_SEM_TIME_def]
⊢ LTL4_SEM_TIME K phi u t =
   if ¬compatible u K then error
   else if compatible u K ∧ ∀w. u + w ∈ K ⇒ LTL_SEM_TIME t (u + w) phi then true
   else if compatible u K ∧ ∀w. u + w ∈ K ⇒ ¬LTL_SEM_TIME t (u + w) phi then false
   else unknown
```

Now we can prove that the ABRV monitor is correct, because for all LTL properties, the monitoring verdict is the same as the ABRV-LTL semantics of the same LTL properties (at the same time of the same trace prefix, w.r.t. the assumptions):

```
[ABRV_monitor_thm]
⊢ ABRV_monitor K phi u t = LTL4_SEM_TIME K phi u t
```

Remark 10.4.1. *The formal proof of the above theorem has indeed followed a part of the original informal proof of Theorem 5.1.2, though the monitor defined in this way is not represented by symbolic automata, thus no hope to further synthesize explicit-state automata-based monitors. In fact, this proof looks very similar as an alternative definition of LTL_3 semantics based on standard LTL semantics. It remains to work out a full formalization of Theorem 5.1.2 based on automata translations and other first-class principles in HOL.*

10.5 LTL_3 and ptLTL (Alternative Semantics)

As a preliminary work, the semantics of LTL_3 and the equivalence between semantics of ptLTL and LTL_3 (at last index), Theorem 7.2.2, has been formalized by the author of this thesis. (The formal proofs have been committed to HOL4 official, as part of the `temporal_deep` example.)

Below is the formalization of LTL_3 semantics (Definition 4.1.3):

```
[LTL3_SEM_DEF]
⊢ LTL3_SEM u f =
   if ∀v. LTL_SEM (u + v) f then LTL3_T
   else if ∀v. ¬LTL_SEM (u + v) f then LTL3_F
   else LTL3_U
```

The formalization of ptLTL (alternative) semantics (Definition 7.1.2):

```
[PTLTL_SEM_ALT_def]
⊢ (∀u p. PTLTL_SEM_ALT u (LTL_PROP p) ⟺ P_SEM (LAST u) p) ∧
  (∀u f. PTLTL_SEM_ALT u (LTL_NOT f) ⟺ ¬PTLTL_SEM_ALT u f) ∧
  (∀u f2 f1.
     PTLTL_SEM_ALT u (LTL_AND (f1,f2)) ⟺ PTLTL_SEM_ALT u f1 ∧ PTLTL_SEM_ALT u f2) ∧
  (∀u f. PTLTL_SEM_ALT u (LTL_PREV f) ⟺ 1 < LENGTH u ∧ PTLTL_SEM_ALT (BUTLASTN 1 u) f) ∧
  ∀u f2 f1.
    PTLTL_SEM_ALT u (LTL_SINCE (f1,f2)) ⟺
    ∃k. k < LENGTH u ∧ PTLTL_SEM_ALT (BUTLASTN k u) f2 ∧
        ∀j. j < k ⇒ PTLTL_SEM_ALT (BUTLASTN j u) f1
```

And the formalization of Theorem 7.2.2:

```
[PTLTL_SEM_ALT_LTL3]
⊢ IS_PAST_LTL f ∧ 0 < LENGTH u ⇒
  (PTLTL_SEM_ALT u f ⟺ THE (LTL3_SEM_TIME (LENGTH u − 1) u f))
```

Furthermore, we can show that, by taking the time parameter to the last position of the input trace prefix, if the monitoring properties contain only past temporal operators, then the monitoring verdict from an LTL$_3$ monitor (as a reduced version of ABRV monitor without assumptions) is exactly the same as alternative ptLTL semantics of the same ptLTL properties:

```
[PTLTL_monitor_thm]
⊢ IS_PAST_LTL f ∧ 0 < LENGTH u ⇒
  (PTLTL_SEM_ALT u f ⟺ THE (GEN_LTL3_monitor f u (LENGTH u − 1)))
```

Chapter 11

Conclusions

In this thesis, we proposed an extended RV framework called ABRV, where assumptions, partial observability and resets are considered in monitor synthesis. We proposed a new four-valued LTL semantics called ABRV-LTL and have shown its necessity in RV monitors under assumptions. As the solution, we gave a simple symbolic LTL monitoring algorithm and demonstrated that, under certain assumptions the resulting monitors are predictive, while some non-monitorable properties becomes monitorable.

The ABRV framework has been also extended to assumptions defined as infinite-state system, where infinite-state belief states are represented as quantifier-free first-order formulas and the emptiness checkings are reduced to SMT-based model checking. We start from a trivial reduction from RV to MC, and eventually obtained an highly optimized RV algorithm, based on Incremental BMC. The final version is hundreds of times faster than the initial one. But as observed in [146], a "major question regarding the use of SMT solvers in performing runtime monitoring is whether they are fast enough." We argue that, for some partially-observable systems, like planets explorers, where the frequency of observations is low, there is a trade-off between the required speed of the monitor and the complexity of the assumptions needed to reason on the non-observable parts.

We also present NuRV, a nuXmv extension for Runtime Verification. It supports assumption-based RV for propositional LTL with both future and past operators, with the supports of partial observability and resets. It has functionalities for offline and online monitoring, and code generation of the monitors in various programming languages. The software engineer aspects of NuRV include its ability to generate standalone monitor code in various programming language, and the ability to do network-based (remote) monitoring based on CORBA.

The experimental evaluations on Dwyer's LTL patterns and other examples show that NuRV is quite efficient in both generation-time and runtime.

11.1 Future Directions

Someone said that a section about future directions is the most important part of a thesis. After all, a scientific research direction without possible future extension is considered as a dead direction, either because every possible directions were already touched, or because it is too hard to make even one more footstep. Runtime Verification is not a dead direction, of course, at least not in the assumption-based branch. There are still some important work to do.

11.1.1 Multi-property monitoring

In ABRV, each monitor is essentially synthesized from the combination of two inputs: the monitoring property and the assumptions. One can imagine that the computation efforts regarding each monitor at runtime, when processing each input states, is a combination of the computation efforts caused by the assumptions and those by the monitoring property itself, and their relative complexities roughly lead the same proportions of the total computation efforts, either at monitor-synthesis time or at runtime.

In practice, it is usually the case that multiple monitors are synthesized from different properties but all under the same assumptions (e.g. the system model). If each of these monitors were synthesized on its own, all the computation efforts regarding the assumptions would have to be repeated. This issue is especially critical when the assumptions are very complex while each monitoring properties are relatively simple: either the formulas are short, or the formulas only involve a very small portions of all variables defined in the model as assumptions.

Of source, having each monitor working in a standalone manner brings flexibilities: the end user has freedom to choose which subset of these monitors to be used. But if multiple monitors of different properties could be synthesized into a single monitor, which takes the same inputs as before, but can outputs multiple verdicts at once: each representing the monitoring outputs of one monitor, just like it were synthesized alone.

More formally speaking, in the problem of multi-property monitoring in ABRV, there is a single FTS K as the RV assumptions, and a *vector* of n LTL formulae $\overline{\varphi} \doteq \langle \varphi_1, \varphi_2, \ldots, \varphi_n \rangle$ as the monitoring properties. The goal is to generate a single *combined* monitor $\mathcal{M}^K_{\overline{\varphi}}$ such that, for all input trace prefix μ (with reset signals),

$$\mathcal{M}^K_{\overline{\varphi}}(\mu) = \overline{v} \doteq \langle v_1, v_2, \ldots, v_n \rangle, \tag{11.1}$$

where

$$v_i = \mathcal{M}^K_{\varphi_i}(\mu) \doteq [\![\textsc{obs}(\mu), \textsc{mrr}(\mu) \models \varphi_i]\!]^K_4 \quad (i = 1, \ldots, n) \ . \tag{11.2}$$

Let call each $\mathcal{M}^K_{\varphi_i}$ a *sub-monitor* of the combined monitor $\mathcal{M}^K_{\overline{\varphi}}$. A major drawback here is that, if such a monitor must be reset, essentially all sub-monitors are being reset, and there is no way to selectively do resets on only some of them. But in practice this limitation should

not be a big problem: usually all sub-monitors should emit inconclusive verdicts, until one of them reports a conclusive verdict (and then all sub-monitors would have to be reset, for the next rounds.)

Note that, even without any assumptions, the idea of multi-property monitoring is still useful. For example, if multiple LTL formulas as monitoring properties share a large piece of sub-formulas (in the extreme case, one property is just the sub-formula of another property), then we can imagine that, the computation efforts regarding shared formulas could have been once just once for multiple monitors involving them.[1]

It is not hard to imagine that, in the worse case, the required computation efforts of the combined monitor is just a literal summation (and perhaps also with extra costs) of the computations involved in each sub-monitors. For example, there is no assumptions, and each LTL properties involve different disjoint sets of variables. Better situations, however, occur in the following cases:

1. The assumptions are relatively complex but each properties are relatively simple (short),

2. Multiple properties as LTL formulas share large piece of sub-formulas.

Below we give an idea for multi-property monitoring algorithm. It works (i.e. can save some computation efforts) because of the following two reasons, corresponding to the above two cases:

1. The components (initial condition, transition relation and fairness) of assumptions, either as raw formulas or being encoded into BDDs, occurs only one time in the combined monitor, thus the computation efforts regarding assumptions should be the same as in the case of single-property monitors.

2. The set of elementary variables due to LTL translations of multiple LTL properties, is a union of elementary variables of each LTL properties, while the same sub-formula occurring in multiple LTL properties only leads to the same elementary variable.

The above first reason should be easy to understand. Now we explain the second one. Note that, in LTL to ω-automata translation (Section 3.7), for each LTL property φ the initial condition Θ_φ of the translated ω-automata represents the LTL formula itself, which is now converted into a propositional formula where elementary variables are used to represent sub-formulas leading by temporal operators. The transition relation ρ_φ, instead, is irrelevant to φ itself but is a conjunction of equations, each passing values of an elementary variable from the past to present,

[1] In rewriting-based monitoring approach using ptLTL, the monitoring verdicts are always conclusive, i.e. Boolean, and for any input trace prefix, the semantics of any ptLTL formula can be inductively defined by the verdicts of its sub-formulas. For ABRV-LTL and LTL_3, we already know that such inductive definition is impossible. Besides, here we are attacking a more general problem where multiple formulas may only share some sub-formulas but not necessary in sub-formula relations.

and from the present to the future:

$$\rho_\varphi \doteq \bigwedge_{\mathrm{X}_\psi \in \mathrm{el}(\varphi)} \left(\mathrm{X}_\psi \leftrightarrow \chi'(\psi)\right) \ \wedge \bigwedge_{\mathrm{Y}_\psi \in \mathrm{el}(\varphi)} \left(\chi(\psi) \leftrightarrow \mathrm{Y}'_\psi\right).$$

It is easy to see that the same elementary variable, which represents the same model variable or temporal sub-formula, always leads to the same equation in the overall transition relation, even in another LTL translation process. Furthermore, if the above transition relation were further conjuncted with another transition relation from another LTL property φ', i.e.

$$\begin{aligned}\rho_\varphi \wedge \rho_{\varphi'} = &\left[\bigwedge_{\mathrm{X}_\psi \in \mathrm{el}(\varphi)} \left(\mathrm{X}_\psi \leftrightarrow \chi'(\psi)\right) \ \wedge \bigwedge_{\mathrm{Y}_\psi \in \mathrm{el}(\varphi)} \left(\chi(\psi) \leftrightarrow \mathrm{Y}'_\psi\right)\right] \\ \wedge &\left[\bigwedge_{\mathrm{X}_\psi \in \mathrm{el}(\varphi')} \left(\mathrm{X}_\psi \leftrightarrow \chi'(\psi)\right) \ \wedge \bigwedge_{\mathrm{Y}_\psi \in \mathrm{el}(\varphi')} \left(\chi(\psi) \leftrightarrow \mathrm{Y}'_\psi\right)\right]\end{aligned}$$

Then we can imagine that, e.g., those equations from elementary variables $\mathrm{x}_\psi \in \mathrm{el}(\varphi') \setminus \mathrm{el}(\varphi)$ will not cause anything wrong, if we have had used $\rho_\varphi \wedge \rho_{\varphi'}$ instead of ρ_φ for the monitor synthesis of φ. Besides, the size of $\rho_\varphi \wedge \rho_{\varphi'}$ may not be that long: perhaps $\mathrm{el}(\varphi')$ and $\mathrm{el}(\varphi)$ are very similar due to large piece of sharing in the two LTL formulas. This idea of combining transition relation without hurting monitor correctness is a key to our multi-property monitor synthesis algorithm.

The other key idea is the disjoint combination of belief states. Recall in the symbolic monitoring algorithm, Algorithm 1, the two belief states r_φ and $r_{\neg\varphi}$ are initially given by the initial condition of LTL translations, Θ_φ and $\Theta_{\neg\varphi}$ (plus the initial condition of assumptions), and is repeatedly transformed by forwarding images (w.r.t. inputs and fairness), while the monitoring outputs are based on their emptiness. Now support we have two such belief states (the positive parts, for example) r_{φ_1} and r_{φ_2} from the monitor construction of two LTL properties φ_1 and φ_2. By introducing a Boolean flag t, we construct the following new belief states formula:

$$r_{\overline{\varphi}} \doteq \text{if } t \text{ then } r_{\varphi_1} \text{ else } r_{\varphi_2} \tag{11.3}$$

For more than two properties, in general, such as $\varphi_1, \varphi_2, \ldots \varphi_n$, we can introduce a finite-domain integer t (of values from 1 to n) as the tag and construct the new belief states formulas in the following way:

$$r_{\overline{\varphi}} \doteq \bigwedge_{i=1\ldots n} \left[(t = i) \rightarrow r_{\varphi_i}\right] \tag{11.4}$$

By having $t = t'$ (t' is the *next* version of t in transition relations) as part of the combined transition relation $\rho_{\overline{\varphi}}$, we can imagine that the forward images of $r_{\overline{\varphi}}$ must also have the same form in names of forwarding images of each r_{φ_i}. To retrieve each r_{φ_i} from the combined $r_{\overline{\varphi}}$, it suffices to just do one more conjunction: $r_{\overline{\varphi}} \wedge (t = i) \equiv r_{\varphi_i}$.

Finally, to support resets of monitors (by resetting all sub-monitors at once), it suffices to do a union of the combined belief states $r_{\overline{\varphi}}$ and $r_{\overline{\neg\varphi}}$ (here $\overline{\neg\varphi}$ is an abbreviation of the vector of the negations of all involved LTL properties). Under the protection of tags the previous smart idea of computing belief states representing only the history of SUS, still works.

We omit the actual symbolic monitoring algorithm which reflects the above ideas.

11.1.2 SAT-based finite-state monitoring

When the RV assumptions are complex with too many variables, and maybe the LTL monitoring properties are also complex. The performance of BDD-based monitoring algorithms represented in Chapter 5 may be too slow. This is not surprised, as the same phenomenon has also been observed in BDD-based symbolic model checking scenarios, and the attempts to resolve this problem has lead to SAT-based symbolic model checking such as Bounded Model Checking [22] in 1999 and then the "Incremental Construction of Inductive Clauses for Indubitable Correctness" (IC3) algorithm [28] in 2011.

When NuRV were used in real projects, we have found that sometimes the BDD-based monitors cannot give satisfied performance on large finite-state models used as RV assumptions. In this case, the infinite-state monitoring algorithms can be directly used as an alternative solution, because any SMT solver is also a SAT solver and there is nothing wrong of applying the SMT-based monitoring algorithms given in Chapter 6, to finite-state systems with finite-domain and Boolean variables. In case the performance is still not good enough, the next idea would be using pure SAT solvers and pure Boolean (original) IC3 model checkers instead of the SMT-based ones.[2] This works because, in general, the performance of dedicated SAT solvers (such as MiniSAT) are better than SMT solvers designed for various theory domains.

Instead of raw formulas, when applying SAT solvers and SAT-based model checkers, Reduced Boolean Circuits (RBC) [96] can be used instead. Note that RBCs do not have canonical forms like BDDs, thus there is no way to generate standalone monitors with this SAT-based approach. Note also that, Boolean quantifier elimination procedures are still needed. In the actual implementation, if such QE procedures are not available, those provided by SMT solvers (like MathSAT5) can still be used as intermediate steps before calling other SAT-based procedures.

[2] A small story here. The author of this thesis was ever faced on a task of inventing a new runtime monitoring algorithm for finite-state systems, where *only* SAT solvers were needed, i.e. without using any model checker. This was after the RV 2019 paper publications but right before RV 2021 where the infinite-state ABRV paper got published. That invented algorithm, called "Bounded Trace Contraction" by the author, was correct (at least it looks so), but it involves SAT formulas whose length is near the *diameter* of combined automata of the assumptions and the translated monitoring property. In fact, it is equivalent to the current last version of the infinite-state monitoring algorithm based on Incremental Bounded Model Checking, just the max_k of BMC procedure must be set to the diameter of the problem so that neither quantifier elimination nor IC3 model checker will not be actually called. That "Bounded Trace Contraction" was invented before the author realized that ABRV can be reduced to Model Checking (and vice versa), which indicates that during ABRV monitoring, the number of SAT solving calls and the size of SAT input formulas cannot be polynomial to the combined size of LTL property and RV assumptions, because ABRV/MC is PSPACE-complete while SAT is only NP-complete.

11.1.3 Parametric Trace Slicing

In general, both events and states can be monitored, even when they go beyond Boolean variables having infinite-state states. For example, an SUS may have two events $open(file)$ and $close(file)$, where *file* can be any string representing a file. Of course, from the view of infinite-state systems we can think them as two infinite-state variables *open* and *close*, and the two events can be then represented by value assignments (or equations) $open = file$ and $close = file$. For any literal *file*, one can easily construct monitoring properties using these equations involving infinite-state variables and their values in LTL formula. But if the goal is to monitor something like "*for all the files*, whenever it is open there must be a close operation *on the same file* within certain time." Note that the outmost quantification is not always universal. In another case, it may be that "*there exists* a file, which is eventually closed twice." Thus, roughly speaking safety properties are implicitly universal, while liveness properties are implicitly existential, while all other cases of quantifications including nesting of quantifications are assumed to not exist.

Currently the existing ABRV framework lacks the support of such *parametric* events with implicit quantifications. But it is not hard to add such supports on top of the existing ABRV framework. The *Parametric Trace Slicing* [41] technique implemented by the MOP framework of runtime monitoring [123] represents one such solution. Roughly speaking, in MOP, an input trace such as

$$s = open(f_1), open(f_2), close(f_1), close(f_2), \ldots$$

is interpreted internally as two traces:

$$s_1 = open(f_1), \text{noop}, close(f_1), \text{noop}, \ldots,$$
$$s_2 = \text{noop}, open(f_2), \text{noop}, close(f_2), \ldots$$

The noop states in the above traces are placeholders only for keeping all other states in the two *sliced* traces aligned with the original input trace. The trace slicing operations happen when a new file is encountered by the monitor. The monitor reports violation (or verification, depending on the setting) once there is one *sub-monitor* (of sliced traces) reports so, and each sub-monitors only need to support standard LTL monitoring. The MOP framework supports many different plugins which covers a wide range of different monitoring specification languages, since the MOP framework itself only take care of the trace slicing while leave the actual monitoring to the plugins.

Thus, in theory, NuRV can be modified to servce as a MOP plugin, to support Parametric Trace Slicing. The support of implicit quantifications in parametric events, although leading to efficient implementations, is often too limited in practice. A perfect approach is to support first-order quantifications (over data) in general.

11.1.4 LTL with first-order quantification over data

Many authors talk about First-Order LTL without actually supporting first-order quantifiers in their LTL definitions. Perhaps they were just having in mind the classical result that LTL (with the *until* operator, possibly without past operators) has the same expressiveness with first-order logics (FOL). If not wrong, Kamp's this impressive LTL expressiveness result [97], with proof later refined by Shelah et al. [76], was the very reason that the *until* operator was added into the initial version of LTL invented by Amir Pnueli [128]. Note that LTL semantics (Definition 3.3.2 is given in FOL, i.e. by using universal and existential quantifiers over time, while it is a bit surprising that arbitrary FOL formula can be back translatd to LTL formula using temporal operators including *until*, although the formula size may blow up. (To add this, translating LTL past operators to future operators may also cause a blow up in formula size. [112])

Thus "first-order quantification over time" is not meaning as a possible new feature in runtime monitoring, because this is just what the standard LTL can do. Parametric Trace Slicing can be seen as a special case: implicit first-orders quantification over data. Many RV researchers have attacked "LTL with first-order quantification over data" in general, but the existing work and progress is not quite satisfied. Among these works, the author highlights the following two of them:

1. Monitoring over Quantifier Temporal Logic (QTL) with bounded quantifiers [83].

2. Monitoring over Metric First-Order Temporal Logic (MFOTL) [16].

Both QTL and MFOTL originally came from Model Checking community. These logics are strong enough, however, existing RV work is never able to support monitoring arbitrary formulas given in these logics. (Whenever a new RV paper on these logics were published, the audience should carefully find sentences around words "limitations" or "fragments" in the paper, to see what kind of properties are *actually* supported, and such limitations usually are not mentioned in the paper title or abstract.) For instance, in the above QTL work, two specialized quantifiers $\tilde{\forall}$ and $\tilde{\exists}$ are added, which quantify over only seen values. In the work of MFOTL, on the other hand, only properties of the form $\mathbf{G}\,\varphi$, where φ is bounded, is supported. *It is basically imposible to extend the existing work, to support arbitrary nesting of first-order quantifiers over data in the monitoring property, no matter how slow the resulting monitor will be.*

Below is the author's proposal to a complete solution of monitoring LTL with first-order quantification over data. The key idea can be summarized into the following *thesis* (cf. *Church–Turing thesis*, just for the meaning of "thesis" here):

Thesis 11.1.1. *In linear-time logics, first-order quantification over data can be reduced to second-order quantification over time.*

To illustrate this point, let us think again the sample property given in Section 11.1.3: "*for all the files*, whenever it is open there must be a close operation *on the same file* within certain time." For each possible file, the previous mentioned Parametric Trace Slicing technique essentially does a filtering on the input trace, to construct a sub-trace which does not contain operations for other files. In words, the actual monitoring is done only on such sub-traces whose indexes as a set of (non-negative) integers (used as discrete time) is a subset of all integers.

Obviously, for any file f occurred in the trace, there exists a *maximal* set of integers N containing all $open(f)$ and $close(f)$. By maximal we mean that any other set of integers N' also containing the same operators, is a subset of N. Then, the actual monitoring property can be written either in standard LTL over N or directly in first-order logic where all quantifications are implicitly over time points in N. In more formally language, the monitoring property is something like this:

$$\begin{aligned} &\forall N.\,(\forall t, t' \in N.\,(operand(t) = operand(t')))\wedge \\ &\quad (\forall N'.\,(\forall t, t' \in N.\,(operand(t) = operand(t'))) \Rightarrow N' \subseteq N) \Rightarrow \\ &\quad \mathbf{G}_N((operation = open) \Rightarrow \mathbf{F}_N\,(operation = close)) \end{aligned} \tag{11.5}$$

or

$$\begin{aligned} &\forall N.\,(\forall t, t' \in N.\,(operand(t) = operand(t')))\wedge \\ &\quad (\forall N'.\,(\forall t, t' \in N.\,(operand(t) = operand(t'))) \Rightarrow N' \subseteq N) \Rightarrow \\ &\quad \forall t \in N((operation(t) = open) \Rightarrow \exists t' \in N.\,t \leqslant t' \wedge (operation(t) = close)) \end{aligned} \tag{11.6}$$

Note that, for any operation like $open(f)$ we have used two functions *operation* and *operand* to retrieve *open* and f, respectively. They are essentially labelling functions over time. On the other hand, temporal operators like $\mathbf{G}_N$ must be associated with a set of time (by default, without such association, it should be understood as all time, i.e. the set of all integers $\mathbb{N}$). In our LTL syntax (Definition 3.3.1), $\mathbf{G}$ is not primitive. However, if we rewrite all temporal operators to the primitives ones, then their semantics (over infinite traces) should be understood w.r.t. N, for example, the following semantics definition of *until* operator (cf. Definition 3.3.2) should be understood as all involved time $i, j, k \in N$ if $\mathbf{U}_N$ were used in place of $\mathbf{U}$:

$$\begin{aligned} &w, i \models \varphi\,\mathbf{U}_N\,\psi \text{ (assuming } i \in N) \quad \Leftrightarrow \\ &\exists k.\,k \in N \wedge i \leqslant k \wedge [w, k \models \psi] \;\wedge\; (\forall j.\,j \in N \wedge i \leqslant j < k \Rightarrow [w, j \models \varphi]) \end{aligned}$$

Some cautions must be taken for the *next* operator $\mathbf{X}_N$: because it may not hold that $\forall i \in N.\,i + 1 \in N$. In practice, one may choose to add more variants of the *next* operator, to allow access either the *absolute* time or *relative* time w.r.t. N. [3] Due to these potential difficulties of interpreting LTL operators, it seems better to abandon LTL, is only used as a syntactic sugar for

[3] So far the author do not have a full picture on this modifiction to LTL syntax and semantics used a syntax sugar of FO fragments in MSO.

easily writing monitoring properties. For the desired monitor synthesis task, instead of working on (11.5) one would rather work directly on (11.6).

In (11.6) there are two kind of quantifiers, one is the quantifier over N, a set of time, the other is the quantifier over variables like t, the time itself. In general, both universal and existential quantifications may occur for N and t. This kind of logic, is called *Monadic Second-Order Logic*, abbreviated as MSOL (or MSO). It is said *monadic* to distinguish from (general) *Second-Order Logic*, where second-order quantifiers (like N) may occur as a *relation* (at least binary) of first-order quantifiers (like t). It is well known that, in certain scenarios like logic of graphs, MSO is decidable while second-order logic itself is not in general [58].

Roughly speaking, finite traces and RV assumptions are both graphs. In particular, a single finite trace is nothing but a chain of time vertices, where the actual state observation occurs as labelling functions of each time vertex. A monitoring property expressed in MSO essentially gives a set of finite graphs, while Courcelle has proved that *every set of finite graphs, that is definable in monadic second-order logic is recognizable.* [57]. Note that, by *recognizable* essentially it means that one can build a finite-state machine which has the same language as the MSO formula. There are similar further results for infinite graphs, when infinite-state systems are under consideration.

The semantics of MSO, once the syntax is fixed, should be clear and self-explaining. The MSO syntax suitable for monitor synthesis should be minimized by defining derivative operators in primitive operators. Following some ideas from Courcelle [57], the first step of the MSO syntax minimization is to eliminate first-order quantifiers by introducing a unary operator SING for detecting if a set is singleton. Thus, instead of $\forall t.\, t \in N \Rightarrow \cdots$, we can use $\forall P.\, \mathrm{SING}(P) \wedge P \subseteq N \Rightarrow \cdots$. Note how set member tests ($\in$) also get all eliminated if no set elements (individual time points) directly occur in the MSO formula. Furthermore, it is well known that universal quantifiers can be rewritten by negation and existential quantifiers, in both first-order and second-order cases. Thus, eventually a possible minimized MSO syntax (with only primitive operators) should include at least the following: basic Boolean connectives, second-order existential quantification ($\exists$ over sets), set singleton test (SING), subset test ($\subseteq$), and labelling functions (but working on sets instead of individual set elements) for actually storing the information.

We omit further details here, only to mention that MSO-based approach was ever used in Model Checking. One such fruit is the MONA model checker [105] for WS1S (Weak Second-Order Logic of One Successor)[4] MONA was not very successful in practice, because it generates huge explicit-state automata translated from WS1S. To the best of our knowledge, nobody ever tried to generate symbolic automata instead, which may resolve the potential state-explosion

[4]By *weak* it means that the second-order quantification is not only over sets but also finite sets. Because runtime monitors, unlike the case of model checking, only see finite traces as inputs, thus, in theory, any second-order (set) variable used in monitoring properties should implicitly be assumed as finite. Thus perhaps only Weak MSO are needed in RV. So far the author do not know the possible benefits of this observation.

issues found in explicit-state MSO-based model checkers. So far there is no known work on monitoring MSO in general, except that many temporal logics found in RV field can be reduced to MSO. Besides QTL and MFOTL mentioned above, the author believe that hyperproperties [54] can also be expressed in MSO. Existing work on monitoring hyperproperties only focus on some easy fragments [70].

11.1.5 Monitoring with out-of-order inputs

The infinite-state monitoring algorithm given in Section 6.6, based on Incremental Bounded Model Checking, can be easily extended to support *out-of-order inputs*. This is mostly because, in the BMP loops, new step constraints (as observations) added incrementally can be injected into any position in the SMT formula being constructed.

To illustrate out-of-order input traces, we need to slightly modify the definition of finite and infinite traces given in Section 3.1. Instead, a finite- or infinite-trace can be seen as a sequence of pair $\langle t, s\rangle$ where t is the time and s is the actual word from the alphabet Σ. The part t can be seen as a time tag explicitly supplied to the monitor. For normal (ordered) traces, the time tag is always the same as the corresponding trace index. More generally, we can think a trace as a function f mapping natural numbers $\mathbb{N}$ (or just a subset of it) to the set of pairs in form of $\langle t, s\rangle$. Let us use fst and snd to access the inside elements in the pairs. Thus, for normal traces we have $\forall i \in \mathbb{N}.\, \mathrm{fst}(f(i)) \equiv i$, while for out-of-order traces this property does not hold. For example, for a trace like $s = s_0 s_1 s_2 \cdots$, the monitor may first receive s_1, then s_2, and then s_0. The goal is to make sure that the monitoring output after receiving these three input states is always the same, regardless of the order of these states.

Furthermore, the behavior of this monitor which is capable to process out-of-order inputs must at least support partial observability (even without supporting assumptions and resets). Using again the above sample trace, the desired monitoring output after receiving s_1 and s_2 but without s_0 should be interpreted as, at time 0 there is no observation except for knowing the discrete time has advanced by one. Then, the monitor received s_0, and the internal belief states must be updated according to s_0 such that some paths previously not compatible with s_0 must now be excluded, and this sometimes may change the monitoring output (at time 2) from inconclusive to conclusive verdicts.

In practice, such out-of-ordering of trace states may only happen locally, i.e. the correct position for a new input state should be *close* to the end of the current trace prefix. For example, a monitor after taking trace inputs from s_2 to s_{100} but now suddenly faces s_1. Such a monitor can be very inefficient in practice, even its output is always correct.

In the existing ABRV framework using monitoring algorithms based on Incremental BMC, the monitor can naturally accept out-of-order inputs between the present time and last time when the monitor is reset, or internal QE is called to accumulate a new pair of belief states. In

other words, as long as only Incremental BMC gets involved, it is always possible to inject new observations into the SMT formula for BMC checking, no matter if the observation happens at the present or earlier time.

Out-of-order traces may occur in distributed environments where multiple components of the SUS may not be totally synchronized, thus producing events (belonging to the same discrete time) at slightly different real time. To the best of our knowledge, currently there is no RV research on supporting out-of-order traces.[5]

11.1.6 Monitoring hybrid systems

In the so-called hybrid systems, both discrete- and continuous-time transitions come into play. Hybrid systems can be described by timed automata [5]. The language of timed automata is called *timed language* defined in the following way:

Definition 11.1.2 (Timed language (of timed automata))**.** *Let $\tau = \tau_1\tau_2\cdots$, where $\tau_i in \mathbb{R}$ are real values, be called a* time sequence*, then a* timed word *over an alphabet Σ is a pair (σ, τ) where $\sigma = \sigma_1\sigma_2\cdots$ in an infinite word of over Σ as usual. A timed language over Σ is a set of timed words over Σ.*

When monitoring LTL (possibly with additional operators designed for dense time) under assumptions given by timed automata, the basic idea is the same: the monitoring property is translated into FTS, which is then composed with the timed automaton representing RV assumptions. The resulting time automaton will be used as a runtime monitor, which also takes timed words as inputs. Another way is to rely on the ABRV-MC reduction (Section 4.6.3) and use timed model checkers (e.g. nuXmv 2.0.0 supports a timed checking mode) or model checker specially designed for hybrid systems.

Note, however, that Bounded Model Checking cannot be used any more, because it is no more possible to construct any SMT formula by unrolling the transition relations in a timed automata. Instead, the monitor must rely on repeated computations of forwarding images, for each new inputs, to get the belief states for deciding the monitor outputs. Note also that, this forwarding image computations may not terminate at all, unless the timed automata is guaranteed to be *zeno-free*: any possible time sequence from the timed language must not have limiting points except at the infinite. No further details can be said here.

11.1.7 ABRV and probabilistic models

There are still many other possible directions. For one last example, we know assumptions may come from system models, but what if the system is stochastic and is described by a Markov

[5]However, the author has ever seen a paper submission on supporting out-of-order traces in monitoring of hybrid systems with dense time setting. But unfortunately that submission was rejected finally. Let us hope it will re-appear.

chain? It is not easy for the existing ABRV framework to support probabilistic systems, mostly because probabilistic systems are rarely described in symbolic manner. (It is hard to assign transition probabilities to symbolic automata with transition relation for each pair of states s, s' satisfying $T(V, V')$.) However, conceptually we can imagine that, LTL, after being translated into ω-automata, can still be composed with the probabilistic models of the SUS. Previously we said that those impossible transitions make non-monitorable properties monitorable, now we should say that *transitions with low probabilities* make non-monitorable properties monitorable, if we cut off those paths with low probabilities. Another way is to assign probability values to the monitor outputs, e.g. to give the probabilities for each possible outputs.

The author hopes that the above discussions on future directions may inspire some new research work on the side of the audience of this thesis.

Bibliography

[1] Parker Abercrombie and Murat Karaorman. jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java. *Electr. Notes Theor. Comput. Sci.*, 70(4):55–79, December 2002.

[2] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karolina Lehtinen. An Operational Guide to Monitorability. In Peter Csaba Ölveczky and Gwen Salaün, editors, *LNCS 11724 - Software Engineering and Formal Methods (SEFM 2019)*, pages 433–453. Springer International Publishing, Cham, 2019. doi: 10.1007/978-3-030-30446-1_23.

[3] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019.

[4] W. Ackermann. *Solvable cases of the Decision Problem*. North-Holland Publishing Company, 1954. ISBN 1024589568. doi: 10.2307/2964059.

[5] Rajeev Alur and David L Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126 (2):183–235, 1994.

[6] Oliver Arafat, Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification revisited. Technical Report Technical Report TUM-I0518, Technische Universität München, München, 2005.

[7] Duncan Paul Attard and Adrian Francalanza. A Monitoring Tool for a Branching-Time Logic. In *LNCS 10012 - Runtime Verification (RV 2016)*, pages 473–481. Springer, Cham, September 2016.

[8] Shaun Azzopardi, Christian Colombo, and Gordon J Pace. A Model-Based Approach to Combining Static and Dynamic Verification Techniques. In Tiziana Margaria and Steffen Bernhard, editors, *LNCS 9952 - Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016, Part I)*, pages 416–430. Springer, October 2016. doi: 10.1007/978-3-319-47166-2_29.

[9] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, January 2009. doi: 10.3233/978-1-58603-929-5-825.

[10] Howard Barringer and Klaus Havelund. TraceContract - A Scala DSL for Trace Analysis. In *LNCS 6664 - FM 2011: Formal Methods*, pages 57–72. Springer Berlin Heidelberg, April 2011. doi: 10.1007/978-3-642-21437-0_7.

[11] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In Bernhard Steffen and Giorgio Levi, editors, *LNCS 2937 - Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, pages 44–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi: 10.1007/978-3-540-24622-0_5.

[12] Howard Barringer, David E Rydeheard, and Klaus Havelund. Rule Systems for Run-Time Monitoring: From Eagle to RuleR. In *LNCS 4389 - Runtime Verification (RV 2007)*, pages 111–125. Springer Berlin Heidelberg, December 2007. doi: 10.1007/978-3-540-77395-5_10.

[13] Detlef Bartetzko, Clemens Fischer, Michael Möller 0002, and Heike Wehrheim. Jass - Java with Assertions. *Electron. Notes Theor. Comput. Sci.*, 2001.

[14] Ezio Bartocci, Yliès Falcone, and Giles Reger. International Competition on Runtime Verification (CRV). In *LNCS 11429 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019, Part III)*, pages 1–9. Springer, New York, NY, March 2019.

[15] Bartocci, Ezio and Falcone, Yliès. *LNCS 10457 - Lectures on Runtime Verification*, volume 10457 of *Introductory and Advanced Topics*. Springer, Cham, August 2017. doi: 10.1007/978-3-319-75632-5.

[16] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring Metric First-Order Temporal Properties. *J. ACM*, 62(2):15:1–15:45, 2015.

[17] Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1-2):46–93, April 2016. doi: 10.1007/s10703-016-0253-8.

[18] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, February 2010. doi: 10.1093/logcom/exn075.

[19] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14–64, September 2011. doi: 10.1145/2000799.2000800.

[20] Anna Bernasconi, Claudio Menghi, Paola Spoletini, Lenore D. Zuck, and Carlo Ghezzi. From Model Checking to a Temporal Proof for Partial Models. In Antonio Cerone and Marco Roveri, editors, *LNCS 10469 - Software Engineering and Formal Methods (SEFM 2017)*, pages 54–69. Springer, Cham, 2018. doi: 10.1007/978-3-319-66197-1_4.

[21] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Coq'Art: The Calculus of Inductive Constructions. Springer Science & Business Media, Berlin, Heidelberg, March 2013. doi: 10.1007/978-3-662-07964-5.

[22] Armin Biere, Alessandro Cimatti, Edmund M Clarke Jr, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *LNCS 1579 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*, pages 193–207. Springer, Berlin, Heidelberg, 1999. doi: 10.1007/3-540-49059-0_14.

[23] Armin Biere, Alessandro Cimatti, Edmund M Clarke Jr, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. In *Advances in Computers: Highly Dependable Software*, pages 117–148. Academic Press, 2003. doi: 10.1016/s0065-2458(03)58003-2.

[24] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, Marco Gario, Stefano Tonetta, and Viktoria Vozarova. Diagnosability of Fair Transition Systems. In Press, 2022.

[25] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic. In *LNCS 8413 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, pages 326–340. Springer, Berlin, Heidelberg, February 2014. doi: 10.1007/978-3-642-54862-8_22.

[26] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Design of Asynchronous Fault Detection and Identification Components using Temporal Epistemic Logic. *Logical Methods in Computer Science*, 11(4):1–33, 2015. doi: 10.2168/LMCS-11(4:4)2015.

[27] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. *nuXmv 2.0.0 User Manual*, 2019. URL `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`.

[28] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In Ranjit Jhala and David Schmidt, editors, *LNCS 6538 - Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, pages 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi: 10.1007/978-3-642-18275-4_7.

[29] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *LNCS 3472 - Model-Based Testing of Reactive Systems*. Springer, Berlin, Heidelberg, 2005. doi: 10.1007/b137241.

[30] Glenn Bruns and Patrice Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In Nicolas Halbwachs and Doron A Peled, editors, *LNCS 1633 - Computer Aided Verification (CAV 1999)*, pages 274–287. Springer Berlin Heidelberg, Berlin, Heidelberg, July 1999. doi: 10.1007/3-540-48683-6_25.

[31] Randal E Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 1986. doi: 10.1109/TC.1986.1676819.

[32] Randal E Bryant. Binary Decision Diagrams. In Edmund M Clarke Jr, Thomas A Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 191–217. Springer International Publishing, Cham, May 2018. doi: 10.1007/978-3-319-10575-8_7.

[33] J Richard Büchi. On a decision method in restricted second order arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.

[34] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, June 1992. doi: 10.1016/0890-5401(92)90017-A.

[35] Ian Cassar, Adrian Francalanza, Duncan Paul Attard, Luca Aceto, and Anna Ingólfsdóttir. A Generic Instrumentation Tool for Erlang. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, 2017.

[36] Ian Cassar, Adrian Francalanza, Duncan Paul Attard, Luca Aceto, and Anna Ingólfsdóttir. A Suite of Monitoring Tools for Erlang. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, 2017.

[37] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *LNCS 8559 - Computer Aided Verification (CAV 2014)*, pages 334–342. Springer, Cham, June 2014. doi: 10.1007/978-3-319-08867-9_22.

[38] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.6 User Manual*, October 2015. URL `https://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`.

[39] Feng Chen and Grigore Roşu. Towards Monitoring-Oriented Programming. *Electr. Notes Theor. Comput. Sci.*, 89(2):108–127, November 2003.

[40] Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *22nd annual ACM SIGPLAN conference*, pages 569–588, New York, USA, October 2007. ACM Press.

[41] Feng Chen and Grigore Roşu. Parametric Trace Slicing and Monitoring. In *LNCS 5505 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 246–261. Springer, Berlin, Heidelberg, March 2009. doi: 10.1007/978-3-642-00768-2_23.

[42] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP. *Electr. Notes Theor. Comput. Sci.*, 144(4):3–20, May 2006.

[43] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940. doi: 10.2307/2266170.

[44] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In *LNCS 7358 - Computer Aided Verification (CAV 2012)*, pages 277–293. Springer, Berlin, Heidelberg, July 2012. doi: 10.1007/978-3-642-31424-7_23.

[45] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In Agostino Cortesi, editor, *LNCS 2294 - Verification, Model Checking, and Abstract Interpretation (VMCAI 2002)*, pages 196–207. Springer Berlin Heidelberg, Berlin, Heidelberg, April 2002. doi: 10.1007/3-540-47813-2_14.

[46] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014. doi: 10.1007/978-3-642-54862-8_4.

[47] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-Based Runtime Verification with Partial Observability and Resets. In *LNCS 11757 - Runtime Verification (RV 2019)*, pages 165–184. Springer, 2019. doi: 10.1007/978-3-030-32079-9_10.

[48] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. NuRV: A nuXmv Extension for Runtime Verification. In *LNCS 11757 - Runtime Verification (RV 2019)*, pages 382–392. Springer, 2019. doi: 10.1007/978-3-030-32079-9_23.

[49] Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta. SMT-based satisfiability of first-order LTL with event freezing functions and metric operators. *Inf. Comput.*, 272:104502, 2020. doi: 10.1016/j.ic.2019.104502.

[50] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-Based Runtime Verification of Infinite-State Systems. In *LNCS 12974 - Runtime Verification (RV 2021)*, pages 207–227. Springer International Publishing, October 2021. doi: 10.1007/978-3-030-88494-9_11.

[51] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. *NuRV 1.9.0 User Manual*, September 2022. URL `https://es-static.fbk.eu/tools/nurv/nurv-manual_190.pdf`.

[52] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997. doi: 10.1023/A:1008615614281.

[53] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, May 2018. doi: 10.1007/978-3-319-10575-8.

[54] Michael R Clarkson and Fred B Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65. IEEE, 2008.

[55] Aaron R. Coble. Anonymity, information, and machine-assisted proof. Technical Report UCAM-CL-TR-785, University of Cambridge, Computer Laboratory, July 2010. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-785.pdf`.

[56] Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1):109–158, May 2016. doi: 10.1007/s10703-016-0251-x.

[57] Bruno Courcelle. The Monadic Second-Order Logic of Graphs I. Recognizable Sets of Finite Graphs. *Information and Computation*, 85(1):12–75, 1990. doi: 10.1016/0890-5401(90)90043-H.

[58] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Encyclopedia of mathematics and its applications, 2012.

[59] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, 18(2):205–225, 2015. doi: 10.1007/s10009-015-0380-3.

[60] Xiaoning Du, Yang Liu, and ALwen Tiu. Trace-Length Independent Runtime Monitoring of Quantitative Policies in LTL. In Nikolaj Bjørner and Frank de Boer, editors, *LNCS 9109 - FM 2015: Formal Methods*, pages 231–247. Springer, Cham, May 2015. doi: 10.1007/978-3-319-19249-9_15.

[61] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, New York, USA, 1999. ACM Press. doi: 10.1145/302405.302672.

[62] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *LNCS 2725 - Computer Aided Verification (CAV 2003)*, pages 27–39. Springer, Berlin, Heidelberg, 2003. doi: 10.1007/978-3-540-45069-6_3.

[63] E Allen Emerson and Chin-Laung Lei. Temporal Reasoning Under Generalized Fairness Constraints. In *LNCS 210 - Theoretical Aspects of Computer Science (STACS 1986)*, pages 21–36. Springer, Berlin, Heidelberg, 1986. doi: 10.1007/3-540-16078-7_62.

[64] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.*, 14(3):349–382, 2012. doi: 10.1007/s10009-011-0196-8.

[65] Ylies Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, 2013. doi: 10.3233/978-1-61499-207-3-141.

[66] Yliès Falcone, Srdjan Krstic, Giles Reger, and Dmitriy Traytel. A Taxonomy for Classifying Runtime Verification Tools. In Christian Colombo and Martin Leucker, editors, *LNCS 11237 - Runtime Verification (RV 2018)*, pages 241–262. Springer, Cham, 2018. doi: 10.1007/978-3-030-03769-7_14.

[67] Davide Fauri, Daniel Ricardo dos Santos, Elisa Costante, Jerry den Hartog, Sandro Etalle, and Stefano Tonetta. From System Specification to Anomaly Detection (and back). In *CPS-SPC*, pages 13–24, 2017. doi: 10.1145/3140241.3140250. URL `https://doi.org/10.1145/3140241.3140250`.

[68] Jeanne Ferrante and Charles Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM Journal on Computing*, 4(1):69–76, March 1975. doi: 10.1137/0204006.

[69] Jean-Christophe Filliatre and Clément Pascutto. Ortac: Runtime Assertion Checking for OCaml (Tool Paper). In *LNCS 12974 - Runtime Verification (RV 2021)*, pages 1–10. Springer International Publishing, October 2021.

[70] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods in System Design*, 18(6):1–28, June 2019.

[71] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *LNCS 9333 - Runtime Verification (RV 2015)*, pages 71–86. Springer, Cham, September 2015.

[72] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the Hennessy–Milner Logic with recursion. *Formal Methods in System Design*, 51(1):87–116, March 2017.

[73] Lars-Åke Fredlund. Guaranteeing Correctness Properties of a Java Card Applet. *Electr. Notes Theor. Comput. Sci.*, 113:217–233, January 2005.

[74] Lars-Åke Fredlund, Julio Mariño, Sergio Pérez, and Salvador Tamarit. Runtime Verification in Erlang by Using Contracts. In *WFLP*, 2018.

[75] Ariel Damián Fuxman. *Formal Analysis of Early Requirements Specifications*. PhD thesis, University of Toronto, 2001. URL `https://tspace.library.utoronto.ca/handle/1807/15905`.

[76] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the Temporal Analysis of Fairness. In *The 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, January 1980.

[77] Dov M Gabbay, Renate A Schmidt, and Andrzej Szałas. *Second-Order Quantifier Elimination*. Foundations, Computational Aspects and Applications. College Publications, 2008.

[78] D Garbervetsky, C Nakhli, S Yovine, and H Zorgati. Program Instrumentation and Run-Time Analysis of Scoped Memory in Java. *Electr. Notes Theor. Comput. Sci.*, 113: 105–121, January 2005.

[79] Sahika Genc and Seéphane Lafortune. Predictability of event occurrences in partially-observed discrete-event systems. *Automatica*, 45(2):301–311, February 2009. doi: 10.1016/j.automatica.2008.06.022.

[80] Sahika Genc and Stéphane Lafortune. Predictability in Discrete-Event Systems Under Partial Observation. *IFAC Proceedings Volumes*, 39(13):1461–1466, 2006. doi: 10.3182/20060829-4-CN-2909.00243.

[81] Michael J C Gordon and Thomas F Melham. *Introduction to HOL.* A Theorem Proving Environment for Higher Order Logic. Cambrige University Press, New York, NY, USA, 1993.

[82] Michael J. C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelberg, 1979. doi: 10.1007/3-540-09724-4.

[83] Klaus Havelund and Doron A Peled. Efficient Runtime Verification of First-Order Temporal Properties. In *LNCS 10869 - Model Checking Software (SPIN 2018)*, pages 26–47. Springer, Cham, June 2018. doi: 10.1007/978-3-319-94111-0_2.

[84] Klaus Havelund and Doron A Peled. Runtime Verification: From Propositional to First-Order Temporal Logic. In *LNCS 11237 - Runtime Verification (RV 2018)*, pages 90–112. Springer, Cham, October 2018. doi: 10.1007/978-3-030-03769-7_7.

[85] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. *Electr. Notes Theor. Comput. Sci.*, 55(2):200–217, 2001.

[86] Klaus Havelund and Grigore Roşu. Synthesizing Monitors for Safety Properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *LNCS 2280 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, pages 342–356. Springer, Berlin, Heidelberg, 2002. doi: 10.1007/3-540-46002-0_24.

[87] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, March 2004.

[88] Klaus Havelund, Grigore Roşu, and Daniel Clancy. Java PathExplorer: A Runtime Verification Tool. In *Proceeding of International Symposium on Artificial Intelligence, Robotics and Automation in Space*, pages 1–8, Montreal, Canada, January 2001.

[89] Klaus Havelund, Doron A Peled, and Dogan Ulus. First order temporal logic monitoring with BDDs. In *Formal Methods in Computer-Aided Design (FMCAD 2017)*, pages 116–123. IEEE, September 2017. doi: 10.23919/FMCAD.2017.8102249.

[90] Klaus Havelund, Doron A Peled, and Dogan Ulus. First-order temporal logic monitoring with BDDs. *Formal Methods in System Design*, 2(3):117–23, January 2019. doi: 10.1007/s10703-018-00327-4.

[91] Thomas A Henzinger and N Ege Saraç. Monitorability Under Assumptions. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *LNCS 12399 - Runtime Verification (RV 2020)*, pages 3–18. Springer International Publishing, Cham, 2020. doi: 10.1007/978-3-030-60508-7_1.

[92] Jörg Hoffmann and Ronen I. Brafman. Contingent planning via heuristic forward search with implicit belief states. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, pages 71–80. AAAI, 2005. URL `http://www.aaai.org/Library/ICAPS/2005/icaps05-008.php`.

[93] HOL4 contributors. *The HOL System DESCRIPTION (Kananaskis-13 release)*, August 2019. URL `http://sourceforge.net/projects/hol/files/hol/kananaskis-13/kananaskis-13-description.pdf`.

[94] HOL4 contributors. *The HOL System LOGIC (Kananaskis-13 release)*, August 2019. URL `http://sourceforge.net/projects/hol/files/hol/kananaskis-13/kananaskis-13-logic.pdf`.

[95] Joe Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, May 2003. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-566.pdf`.

[96] Paul B Jackson and Daniel Sheridan. Clause Form Conversions for Boolean Circuits. In Holger H Hoos and David G Mitchell, editors, *LNCS 3542 - Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 183–198. Springer, Berlin, Heidelberg, 2005. doi: 10.1007/11527695_15.

[97] Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, UCLA, 1979.

[98] Murat Karaorman and Jay Freeman. jMonitor: Java Runtime Event Specification and Monitoring Library. *Electr. Notes Theor. Comput. Sci.*, 113:181–200, January 2005.

[99] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Springer, Boston, MA, July 2000.

[100] Katarína Kejstová, Petr Rockai, and Jiri Barnat. From Model Checking to Runtime Verification and Back. In *LNCS 10548 - Runtime Verification (RV 2017)*, pages 225–240. Springer, Cham, August 2017. doi: 10.1007/978-3-319-67531-2_14.

[101] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic Verification of Linear Temporal Logic Specifications. In *LNCS 1443 - Automata, Languges and Programming (ICALP 1998)*, pages 1–16. Springer, Berlin, Heidelberg, May 1998. doi: 10.1007/BFb0055036.

[102] Leonid Khachiyan. Fourier–Motzkin Elimination Method. In *Encyclopedia of Optimization*, pages 1074–1077. Springer US, Boston, MA, 2009. doi: 10.1007/978-0-387-74759-0_187.

[103] Moonjoo Kim, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java MaC: a Run-time Assurance Tool for Java Programs. *Electr. Notes Theor. Comput. Sci.*, 55(2):1–18, October 2001. doi: 10.1016/S1571-0661(04)00254-3.

[104] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational Analysis of Run-time Monitoring - Fundamentals of Java-MaC. *Electr. Notes Theor. Comput. Sci.*, 70(4):80–94, December 2002.

[105] Nils Klarlund, Anders Møller, and Michael I Schwartzbach. MONA Implementation Secrets. *BRICS Report Series*, 7(40), June 2000.

[106] Stephen Cole Kleene. *Introduction to Metamathematics*. Wolthers-Noordhoff, New York, 1971.

[107] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[108] Martin Leucker. Teaching Runtime Verification. In Sarfraz Khurshid and Koushik Sen, editors, *LNCS 7186 - Runtime Verification (RV 2011)*, pages 34–48. Springer, Berlin, Heidelberg, 2012.

[109] Martin Leucker. Sliding between Model Checking and Runtime Verification. In *LNCS 7687 - Runtime Verification (RV 2012)*, pages 82–87. Springer, Berlin, Heidelberg, January 2013. doi: 10.1007/978-3-642-35632-2_10.

[110] Martin Leucker. Runtime Verification for Linear-Time Temporal Logic. In Jonathan P Bowen, Zhiming Liu, and Zili Zhang, editors, *LNCS 10215 - Engineering Trustworthy Software Systems (SETSS 2016)*, pages 151–194. Springer International Publishing, Cham, April 2017. doi: 10.1007/978-3-319-56841-6.

[111] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. doi: 10.1016/j.jlap.2008.08.004.

[112] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The Glory of the Past. In *Logics of Programs*, pages 196–218, 1985.

[113] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The computer journal*, 36(5):450–462, 1993. URL `https://dblp.org/rec/journals/cj/LoosW93`.

[114] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian-Florin Serbanuta, and Grigore Roşu. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *LNCS 8734 - Runtime Verification (RV 2014)*, pages 285–300. Springer, Cham, August 2014. doi: 10.1007/978-3-319-11164-3_24.

[115] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag New York, 1992. doi: 10.1007/978-1-4612-0931-7.

[116] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, 1995. doi: 10.1007/978-1-4612-4222-2.

[117] Annalisa Marcja and Carlo Toffalori. Quantifier Elimination. In *A Guide to Classical and Modern Model Theory*, pages 43–83. Springer Netherlands, Dordrecht, 2003. doi: 10.1007/978-94-007-0812-9_2.

[118] C Mascle, D Neider, M Schwenger, and P Tabuada. From LTL to rLTL monitoring: improved monitorability through robust semantics. In *23rd International Conference on Hybrid Systems Computation and Control*, pages 1–12, New York, NY, USA, April 2020. ACM. doi: 10.1145/3365365.3382197.

[119] Kenneth L McMillan. *Symbolic Model Checking*. Springer Science+Business Media, 1993. doi: 10.1007/978-1-4615-3190-6.

[120] Thomas F Melham. Automating Recursive Type Definitions in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, page 64. Springer Science & Business Media, 1989. doi: 10.1007/978-1-4612-3658-0_9.

[121] Thomas F Melham. A Package For Inductive Relation Definitions In HOL. In *HOL Theorem Proving System and Its Applications (HOL 1991)*, pages 350–357. IEEE, Davis, CA, USA, August 1991. doi: 10.1109/HOL.1991.596299.

[122] Claudio Menghi, Paola Spoletini, and Carlo Ghezzi. Dealing with Incompleteness in Automata-Based Model Checking. In *LNCS 9995 - FM 2016: Formal Methods*. Springer, October 2016. doi: 10.1007/978-3-319-48989-6.

[123] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on*

Software Tools for Technology Transfer, 14(3):249–289, April 2011. doi: 10.1007/s10009-011-0198-6.

[124] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford Univ., Dept. of Computer Science, 1972. URL `http://www.dtic.mil/dtic/tr/fulltext/u2/785072.pdf`.

[125] Jeremy W Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants - Integrating Daikon and ESC/Java. *Electron. Notes Theor. Comput. Sci.*, 55(2):255–276, October 2001.

[126] Doron A Peled and Klaus Havelund. Refining the Safety-Liveness Classification of Temporal Properties According to Monitorability. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, pages 218–234. Springer, June 2019. doi: 10.1007/978-3-030-22348-9_14.

[127] Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. Predictive runtime verification of timed properties. *Journal of Systems and Software*, 132:353–365, October 2017. doi: 10.1016/j.jss.2017.06.060.

[128] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977.

[129] Amir Pnueli and A Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *FM 2006: Formal Methods*, pages 573–586. Springer, Berlin, Heidelberg, August 2006. doi: 10.1007/11813040_38.

[130] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005. doi: 10.1007/s10515-005-6205-y.

[131] Kristin Yvonne Rozier and Johann Schumann. R2U2: Tool Overview. *Kalpa Publications in Computing*, 3:138–156, 2017. doi: 10.29007/5pch.

[132] Usa Sammapun, Raman Sharykin, Margaret DeLap, Myong Kim, and Steve Zdancewic. Formalizing Java-MaC. *Electr. Notes Theor. Comput. Sci.*, 89(2):171–190, November 2003.

[133] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, September 1995. doi: 10.1109/9.412626.

[134] Joshua Schneider, David A Basin, Srdjan Krstic, and Dmitriy Traytel. A Formally Verified Monitor for Metric First-Order Temporal Logic. In Bernd Finkbeiner and Leonardo

Mariani, editors, *LNCS 11757 - Runtime Verification (RV 2019)*, pages 1–19. Springer International Publishing, Porto, Portugal, October 2019.

[135] Klaus Schneider. Translating linear temporal logic to deterministic ω-automata. In *GI/ITG/GMM WORKSHOP METHODEN DES ENTWURFS UND DER VERIFIKATION DIGITALER SYSTEME*, pages 149–158, 1997.

[136] Klaus Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 39–54. Springer, 2001. doi: 10.1007/3-540-45653-8_3.

[137] Klaus Schneider. Temporal Logics. In *Verification of Reactive Systems - Formal Methods and Algorithms*, pages 279–404. Springer-Verlag, Berlin, Heidelberg, 2004. doi: 10.1007/978-3-662-10778-2_5.

[138] Klaus Schneider and Dirk W Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata. In *LNCS 1690 - Theorem Proving in Higher Order Logics (TPHOLs 1999)*, pages 255–272. Springer, Berlin, Heidelberg, September 1999.

[139] Viktor Schuppan, Marcel Baur, and Armin Biere. JVM Independent Replay in Java. *Electr. Notes Theor. Comput. Sci.*, 113:85–104, January 2005.

[140] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Runtime Monitoring with Recovery of the SENT Communication Protocol. In Rupak Majumdar and Viktor Kunčak, editors, *LNCS 10426 - Computer Aided Verification (CAV 2017, Part I)*. Springer, July 2017. doi: 10.1007/978-3-319-63387-9_17.

[141] A. Prasad Sistla, Min Zhou, and Lenore D. Zuck. Monitoring Off-the-Shelf Components. In E Allen Emerson and Kedar S Namjoshi, editors, *LNCS 3855 - Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, pages 222–236. Springer, Berlin, Heidelberg, 2006. doi: 10.1007/11609773_15.

[142] A Prasad Sistla, Milos Zefran, and Yao Feng. Monitorability of Stochastic Dynamical Systems. In *LNCS 6806 - Computer Aided Verification (CAV 2011)*, pages 720–736. Springer, Berlin, Heidelberg, September 2018. doi: 10.1007/978-3-642-22110-1_58.

[143] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *LNCS 5170 - Theorem Proving in Higher Order Logics (TPHOLs 2008)*. Springer, Berlin, Heidelberg, 2008. doi: 10.1007/978-3-540-71067-7_6.

[144] Volker Stolz and Frank Huch. Runtime Verification of Concurrent Haskell Programs. *Electr. Notes Theor. Comput. Sci.*, 113:201–216, January 2005.

[145] Li Tan, Jesung Kim, O Sokolsky, and Insup Lee. Model-based testing and monitoring for hybrid embedded systems. In *IEEE International Conference on Information Reuse and Integration*, pages 487–492. IEEE, November 2004. doi: 10.1109/IRI.2004.1431508.

[146] Vidhya Tekken Valapil, Sorrachai Yingchareonthawornchai, Sandeep Kulkarni, Eric Torng, and Murat Demirbas. Monitoring Partially Synchronous Distributed Systems Using SMT Solvers. In Shuvendu Lahiri and Giles Reger, editors, *LNCS 10548 - Runtime Verification (RV 2017)*, pages 277–293. Springer, Cham, 2017. doi: 10.1007/978-3-319-67531-2_17.

[147] Chun Tian and Davide Sangiorgi. Unique solutions of contractions, CCS, and their HOL formalisation. *Information and Computation*, pages 104606–37, July 2020. doi: 10.1016/j.ic.2020.104606.

[148] Thomas Tuerk and Klaus Schneider. Relationship Between Alternating Omega-Automata and Symbolically Represented Nondeterministic Omega-Automata. Technical report, University of Kaiserslautern, November 2005.

[149] Thomas Tuerk, Klaus Schneider, and Mike Gordon. Model Checking PSL Using HOL and SMV. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *LNCS 4383 - Hardware and Software: Verification and Testing (HVC 2006)*, pages 1–15. Springer, Berlin, Heidelberg, 2007. doi: 10.1007/978-3-540-70889-6_1.

[150] Xian Zhang, Martin Leucker, and Wei Dong. Runtime Verification with Predictive Semantics. In *LNCS 7226 - NASA Formal Methods (NFM 2012)*, pages 418–432. Springer, Berlin, Heidelberg, March 2012. doi: 10.1007/978-3-642-28891-3_37.

[151] Y Zhao, S Oberthür, M Kardos, and F J Rammig. Model-based Runtime Verification Framework for Self-optimizing Systems. *Electr. Notes Theor. Comput. Sci.*, 144(4): 125–145, May 2006. doi: 10.1016/j.entcs.2006.02.008.

[152] Yuhong Zhao and Franz Rammig. Model-based Runtime Verification Framework. *Electr. Notes Theor. Comput. Sci.*, 253(1):179–193, October 2009. doi: 10.1016/j.entcs.2009.09.035.

Appendix A

Data and Tables

A.1 SMV Models

The MODEL 2 (M2) used in this thesis (Section 9.1.2), generated from Dwyer's Pattern 10, is represented by the following SMV program. We first use NuSMV's `ltl2smv` program to translate the LTL, then fill in other system variables. All `x_` variables are elementary temporal variables. It also shows how the symbolic LTL to Büchi translation described in Sec. 3.7 works.

```
MODULE main
VAR
   p : boolean;   q : boolean;   r : boolean;
   s : boolean;   t : boolean;   z : boolean;
   x_11 : boolean;   x_9 : boolean;   x_7 : boolean;
   x_5  : boolean;   x_3 : boolean;   x_1 : boolean;
DEFINE
   x_2 := ((!x_4 | x_6) | (!s & x_3));
   x_6 := ((!x_0 | x_8) | (s & x_7));
   x_8 := ((!x_4 | x_10) | (!s & x_9));
   x_10 := (!x_0 | (s & x_11));
   x_4 := (!s | x_5);
   x_0 := (s | x_1);
INIT
   (!x_0 | x_2)
JUSTICE
   (!x_2 | (!x_4 | x_6))
JUSTICE
   (!x_6 | (!x_0 | x_8))
JUSTICE
   (!x_8 | (!x_4 | x_10))
JUSTICE
   (!x_10 | !x_0)
TRANS   next(x_2) = x_3
TRANS   next(x_6) = x_7
```

```
TRANS   next(x_8) = x_9
TRANS   next(x_10) = x_11
TRANS   next(x_4) = x_5
TRANS   next(x_0) = x_1
```

A.2 Dwyer's LTL Patterns

Dwyer's LTL patterns [61] in NuSMV's LTL syntax are given in Table A.1 and A.2.

Table A.1: Dwyer's LTL patterns

ID	Pattern	LTL
0	p is false (Globally)	`G (!p)`
1	p is false (Before r)	`F r -> (!p U r)`
2	p is false (After q)	`G (q -> G (!p))`
3	p is false (Between q and r)	`G ((q & !r & F r) -> (!p U r))`
4	p is false (After q until r)	`G (q & !r -> ((G !p) | (!p U r)))`
5	p becomes true (Globally)	`F p`
6	p becomes true (Before r)	`(G !r) | (!r U (p & !r))`
7	p becomes true (After q)	`G (!q) | F (q & F p)`
8	p becomes true (Between q and r)	`G (q & !r -> ((G !r) | (!r U (p & !r))))`
9	p becomes true (After q until r)	`G (q & !r -> (!r U (p & !r)))`
10	trans to p occ. $\leqslant 2$ (Globally)	`G (!p) | (!p U (G p | (p U (G (!p) | (!p U (G p | (p U (G (!p)))))))))`
11	trans to p occ. $\leqslant 2$ (Before r)	`F r -> ((!p & !r) U (r | ((p & !r) U (r | ((!p & !r) U (r | ((p & !r) U (r | (!p U r)))))))))`
12	trans to p occ. $\leqslant 2$ (After q)	`F q -> (!q U (q & (G (!p) | (!p U (G p | (p U (G (!p) | (!p U (G p | (p U (G !p))))))))))))`
13	trans to p occ. $\leqslant 2$ (Betw. q and r)	`G ((q & F r) -> ((!p & !r) U (r | ((p & !r) U (r | ((!p & !r) U (r | ((p & !r) U (r | (!p U r)))))))))))`
14	trans to p occ. $\leqslant 2$ (After q until r)	`G (q -> ((!p & !r) U (r | ((p & !r) U (r | ((!p & !r) U (r | ((p & !r) U (r | (G (!p) | (!p U r)) | G p)))))))))`
15	p is true (Globally)	`G p`
16	p is true (Before r)	`F r -> (p U r)`
17	p is true (After q)	`G (q -> G p)`
18	p is true (Between q and r)	`G ((q & !r & F r) -> (p U r))`
19	p is true (After q until r)	`G (q & !r -> (G p | (p U r)))`
20	s precedes p (Globally)	`G (!p) | (!p U s)`
21	s precedes p (Before r)	`F r -> (!p U (s | r))`
22	s precedes p (After q)	`G (!q) | F (q & (G (!p) | (!p U s)))`
23	s precedes p (Between q and r)	`G ((q & !r & F r) -> (!p U (s | r)))`
24	s precedes p (After q until r)	`G (q & !r -> (G (!p) | (!p U (s | r))))`

Table A.2: Dwyer's LTL patterns (continued)

ID	Pattern	LTL
25	s responds to p (Globally)	`G (p -> F s)`
26	s responds to p (Before r)	`F r -> (p -> (!r U (s & !r))) U r`
27	s responds to p (After q)	`G (q -> G (p -> F s))`
28	s responds to p (Between q and r)	`G ((q & !r & F r) -> (p -> (!r U (s & !r))) U r)`
29	s responds to p (After q until r)	`G (q & !r -> (G (p -> (!r U (s & !r))) |` `((p -> (!r U (s & !r))) U r)))`
30	s,t precedes p (Globally)	`F p -> (!p U (s & !p & X (!p U t)))`
31	s,t precedes p (Before r)	`F r -> (!p U (r | (s & !p & X (!p U t))))`
32	s,t precedes p (After q)	`(G !q) | (!q U (q & F p ->` `(!p U (s & !p & X (!p U t)))))`
33	s,t precedes p (Between q and r)	`G ((q & F r) -> (!p U (r | (s & !p & X(!p U t)))))`
34	s,t precedes p (After q until r)	`G (q -> (F p -> (!p U (r | (s & !p & X(!p U t))))))`
35	p precedes s,t (Globally)	`(F (s & X (F t))) -> ((!s) U p)`
36	p precedes s,t (Before r)	`F r -> ((!(s & (!r) & X (!r U (t & !r)))) U (r | p))`
37	p precedes s,t (After q)	`(G !q) |` `((!q) U (q & ((F (s & X (F t))) -> ((!s) U p))))`
38	p precedes s,t (Between q and r)	`G ((q & F r) ->` `((!(s & (!r) & X (!r U (t & !r)))) U (r | p)))`
39	p precedes s,t (After q until r)	`G (q -> (!(s & (!r) & X(!r U (t & !r)))` `U (r | p) | G (!(s & X (F t)))))`
40	p responds to s,t (Globally)	`G (s & X (F t) -> X (F (t & F p)))`
41	p responds to s,t (Before r)	`F r -> (s & X (!r U t) -> X (!r U (t & F p))) U r`
42	p responds to s,t (After q)	`G (q -> G (s & X (F t) -> X (!t U (t & F p))))`
43	p responds to s,t (Between q and r)	`G ((q & F r) ->` `(s & X (!r U t) -> X (!r U (t & F p))) U r)`
44	p responds to s,t (After q until r)	`G (q -> (s & X (!r U t) -> X (!r U (t & F p)))` `U (r | G (s & X (!r U t) -> X (!r U (t & F p)))))`
45	s,t responds to p (Globally)	`G (p -> F(s & X (F t)))`
46	s,t responds to p (Before r)	`F r -> (p -> (!r U (s & !r & X (!r U t)))) U r`
47	s,t responds to p (After q)	`G (q -> G (p -> (s & X (F t))))`
48	s,t responds to p (Between q and r)	`G ((q & F r) ->` `(p -> (!r U (s & !r & X (!r U t)))) U r)`
49	s,t responds to p (After q until r)	`G (q -> (p -> (!r U (s & !r & X (!r U t))))` `U (r | G (p -> (s & X (F t)))))`
50	s,t w/o z resp. to p (Globally)	`G (p -> F (s & !z & X (!z U t)))`
51	s,t w/o z resp. to p (Before r)	`F r ->` `(p -> (!r U (s & !r & !z & X ((!r & !z) U t)))) U r`
52	s,t w/o z resp. to p (After q)	`G (q -> G (p -> (s & !z & X (!z U t))))`
53	s,t w/o z resp. to p (Betw. q and r)	`G ((q & F r) -> (p -> (!r U` `(s & !r & !z & X ((!r & !z) U t)))) U r)`
54	s,t w/o z resp. to p (After q until r)	`G (q -> (p -> (!r U (s & !r & !z & X ((!r & !z)` `U t)))) U (r | G (p -> (s & !z & X (!z U t)))))`

Table A.3: Size of monitors built from LTL patterns

ID	$\lvert M_1\rvert$	$\lVert M_1\rVert$	$\lvert M_2\rvert$	$\lVert M_2\rVert$	$\lvert M_1^2\rvert$	$\lVert M_1^2\rVert$	$\lvert M_2^2\rvert$	$\lVert M_2^2\rVert$	$\lvert M_3^2\rvert$	$\lVert M_3^2\rVert$	$\lvert M_1^1\rvert$	$\lVert M_1^1\rVert$	$\lvert M_2^1\rvert$	$\lVert M_2^1\rVert$
0	5 (2)	12	7	28	11 (5)	22	16	58	20	72	4 (1)	9	6	18
1	15 (8)	56	23	184	36 (20)	116	56	404	60	432	8 (2)	24	14	56
2	13 (4)	72	17	136	31 (10)	152	41	296	45	324	8 (1)	28	11	44
3	33 (8)	400	41	656	81 (20)	880	101	1456	105	1512	13 (1)	60	17	85
4	21 (4)	272	33	528	51 (10)	592	81	1168	85	1224	9 (1)	40	13	65
5	5 (2)	12	7	28	11 (5)	22	16	58	20	72	4 (1)	9	6	18
6	9 (6)	24	19	152	21 (15)	44	46	332	50	360	5 (2)	12	11	44
7	13 (4)	72	17	136	31 (10)	152	41	296	45	324	8 (1)	28	11	44
8	27 (8)	304	35	560	66 (20)	664	86	1240	90	1296	9 (1)	40	13	65
9	27 (8)	304	35	560	66 (20)	664	86	1240	90	1296	9 (1)	40	13	65
10	13 (2)	44	15	60	31 (5)	94	36	130	40	144	10 (1)	27	12	36
11	23 (8)	120	31	248	56 (20)	260	76	548	80	576	14 (2)	48	20	80
12	29 (4)	200	33	264	71 (10)	440	81	584	85	612	16 (1)	60	19	76
13	49 (8)	656	57	912	121 (20)	1456	141	2032	145	2088	21 (1)	100	25	125
14	37 (4)	528	49	784	91 (10)	1168	121	1744	125	1800	17 (1)	80	21	105
15	5 (2)	12	7	28	11 (5)	22	16	58	20	72	4 (2)	6	5	15
16	15 (8)	56	23	184	36 (20)	116	56	404	60	432	7 (2)	20	13	52
17	13 (4)	72	17	136	31 (10)	152	41	296	45	324	5 (1)	16	8	32
18	33 (8)	400	41	656	81 (20)	880	101	1456	105	1512	10 (1)	45	14	70
19	21 (4)	272	33	528	51 (10)	592	81	1168	85	1224	6 (1)	25	10	50
20	5 (3)	8	10	40	5 (3)	8	20	72	26	92	4 (2)	6	8	24
21	16 (10)	48	22	176	28 (16)	88	52	376	58	416	8 (3)	20	13	52
22	16 (4)	96	20	160	38 (8)	212	46	324	50	352	10 (1)	36	13	52
23	33 (8)	400	41	656	81 (20)	880	101	1456	105	1512	12 (1)	55	16	80
24	19 (2)	272	33	528	47 (6)	592	81	1168	85	1224	8 (1)	35	12	60
25	6 (0)	24	6	24	13 (1)	46	14	50	18	64	5 (0)	15	5	15
26	14 (8)	48	22	176	34 (20)	100	54	388	58	416	7 (2)	20	13	52
27	15 (0)	120	15	120	35 (2)	248	37	264	41	292	10 (0)	40	10	40
28	31 (8)	368	39	624	77 (20)	816	97	1392	101	1448	12 (1)	55	16	80
29	31 (8)	368	39	624	71 (18)	784	93	1360	97	1416	12 (1)	55	16	80
30	14 (8)	48	22	176	23 (15)	60	44	316	53	376	7 (2)	20	13	52
31	34 (20)	224	46	736	76 (48)	408	108	1560	117	1680	12 (3)	45	19	95
32	31 (20)	176	43	688	51 (36)	224	95	1376	109	1560	12 (5)	35	17	85
33	66 (16)	1600	82	2624	163 (40)	3536	203	5840	207	5952	17 (1)	96	22	132
34	42 (8)	1088	66	2112	103 (20)	2384	163	4688	167	4800	12 (1)	66	17	102
35	16 (8)	64	24	192	24 (12)	88	48	344	55	396	8 (2)	24	14	56
36	36 (20)	256	48	768	68 (40)	400	100	1424	107	1528	13 (3)	50	20	100
37	99 (0)	1584	99	1584	165 (8)	2416	165	2480	191	2888	40 (0)	200	40	200
38	75 (16)	1888	91	2912	165 (48)	3552	197	5600	201	5712	19 (1)	108	24	144
39	51 (8)	1376	75	2400	117 (32)	2656	165	4704	169	4816	14 (1)	78	19	114
40	16 (0)	128	16	128	33 (2)	228	35	244	39	272	11 (0)	44	11	44
41	36 (12)	384	40	640	74 (27)	680	87	1256	91	1312	17 (2)	75	20	100
42	39 (0)	624	39	624	85 (8)	1152	85	1216	89	1272	19 (0)	95	19	95
43	77 (0)	2464	77	2464	167 (16)	4512	167	4768	171	4880	30 (0)	180	30	180
44	67 (0)	2144	67	2144	147 (16)	3968	147	4224	151	4336	24 (0)	144	24	144
45	14 (0)	112	14	112	31 (2)	220	33	236	37	264	9 (0)	36	9	36
46	30 (16)	224	46	736	73 (40)	472	113	1624	117	1680	11 (2)	45	19	95
47	29 (4)	400	41	656	69 (12)	832	97	1408	101	1464	10 (1)	45	14	70
48	67 (16)	1632	83	2656	165 (40)	3584	205	5888	209	6000	18 (1)	102	23	138
49	111 (20)	2912	123	3936	243 (46)	5792	277	8096	281	8208	18 (1)	102	23	138
50	26 (0)	416	26	416	63 (8)	840	63	904	67	960	11 (0)	55	11	55
51	58 (32)	832	90	2880	143 (80)	1808	223	6416	227	6528	13 (2)	66	23	138
52	57 (16)	1312	73	2336	137 (40)	2816	177	5120	181	5232	12 (1)	66	17	102
53	131 (32)	6336	163	10432	325 (80)	14080	405	23296	409	23520	21 (1)	140	27	189
54	203 (48)	9920	219	14016	459 (118)	20032	501	29248	505	29472	21 (1)	140	27	189

A.3 Formal proofs

The following HOL proof scripts require HOL4's `temporal_deep` example. In particular, the `runtime_verificationTheory` [1] which contains the formal proof of formal theorems in Section 10.5, was contributed by the same author of this thesis.

```
open HolKernel Parse boolLib bossLib;
open pred_setTheory hurdUtils Omega_AutomataTheory Temporal_LogicTheory;
open full_ltlTheory ltl_to_automaton_formulaTheory runtime_verificationTheory;

val _ = new_theory "paper";
val _ = hide "K";

val _ = set_fixity "extends" (Infix(NONASSOC, 450));

Definition extends_def : (* no partial observability *)
    (i extends u) = ?c. i = concat u c
End

Definition compatible_def :
    compatible u K = ?c. concat u c IN K
End

Definition LTL3_output_def :
   (LTL3_output T T = LTL3_U) /\
   (LTL3_output T F = LTL3_T) /\
   (LTL3_output F T = LTL3_F)
End

(* "belief run" after taking a finite trace u *)
Definition LTL3_belief_run_def :
    LTL3_belief_run l u = {i | i extends u /\ LTL_SEM i l}
End

(* LTL3 monitor (abstract version) *)
Definition LTL3_monitor_def :
    LTL3_monitor l u = LTL3_output (LTL3_belief_run l           u <> EMPTY)
                                   (LTL3_belief_run (LTL_NOT l) u <> EMPTY)
End

(* correctness of LTL3 monitor, an abstract version *)
Theorem LTL3_monitor_thm :
    !l u. LTL3_monitor l u = LTL3_SEM u l
Proof
    RW_TAC std_ss [LTL3_monitor_def, LTL3_SEM_def]
```

[1] https://github.com/HOL-Theorem-Prover/HOL/blob/develop/examples/temporal_deep/src/examples/runtime_verificationScript.sml

```
 >> Cases_on `LTL3_belief_run          l  u = {}`
 >> Cases_on `LTL3_belief_run (LTL_NOT l) u = {}`
 >> fs [LTL3_output_def] (* 4 subgoals *)
 >| [ (* goal 1 (of 4) *)
      fs [LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_def, LTL_SEM_TIME_def] \\
      Q.ABBREV_TAC `i = u ++ (\n. {})` \\
     `i extends u` by PROVE_TAC [extends_def] \\
      METIS_TAC [],
      (* goal 2 (of 4) *)
      RW_TAC std_ss [Once EQ_SYM_EQ, LTL3_SEM_def, LTL3_SEM_TIME_F] \\
      Q.PAT_X_ASSUM `LTL3_belief_run (LTL_NOT l) u <> {}` K_TAC \\
      fs [LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_def, LTL_SEM_TIME_def] \\
      POP_ASSUM (MP_TAC o (Q.SPEC `u ++ v`)) \\
     `u ++ v extends u` by PROVE_TAC [extends_def] \\
      METIS_TAC [],
      (* goal 3 (of 4) *)
      RW_TAC std_ss [Once EQ_SYM_EQ, LTL3_SEM_def, LTL3_SEM_TIME_T] \\
      Q.PAT_X_ASSUM `LTL3_belief_run l u <> {}` K_TAC \\
      fs [LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_def, LTL_SEM_TIME_def] \\
      POP_ASSUM (MP_TAC o (Q.SPEC `u ++ v`)) \\
     `u ++ v extends u` by PROVE_TAC [extends_def] \\
      METIS_TAC [],
      (* goal 4 (of 4) *)
      RW_TAC std_ss [Once EQ_SYM_EQ, LTL3_SEM_def, LTL3_SEM_TIME_def] >|
      [ (* goal 4.1 (of 2) *)
        Q.PAT_X_ASSUM `LTL3_belief_run l u <> {}` K_TAC \\
        fs [LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
            LTL_SEM_def, LTL_SEM_TIME_def, extends_def] \\
        POP_ASSUM (STRIP_ASSUME_TAC o (Q.SPEC `c`)) \\
        METIS_TAC [],
        (* goal 4.2 (of 2) *)
        Q.PAT_X_ASSUM `~!w. P` K_TAC \\
        Q.PAT_X_ASSUM `LTL3_belief_run (LTL_NOT l) u <> {}` K_TAC \\
        fs [LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
            LTL_SEM_def, LTL_SEM_TIME_def, extends_def] \\
        POP_ASSUM (STRIP_ASSUME_TAC o (Q.SPEC `c`)) \\
        METIS_TAC [] ] ]
QED

Definition GEN_LTL3_belief_run_def :
    GEN_LTL3_belief_run l u t = {i | i extends u /\ LTL_SEM_TIME t i l}
End

(* alternative definition *)
```

```
Theorem LTL3_belief_run_alt :
    !l u. LTL3_belief_run l u = GEN_LTL3_belief_run l u 0
Proof
    RW_TAC std_ss [GEN_LTL3_belief_run_def, LTL3_belief_run_def, LTL_SEM_def]
QED

(* LTL3 monitor (abstract version) *)
Definition GEN_LTL3_monitor_def :
    GEN_LTL3_monitor l u t =
        LTL3_output (GEN_LTL3_belief_run l           u t <> EMPTY)
                    (GEN_LTL3_belief_run (LTL_NOT l) u t <> EMPTY)
End

(* alternative definition *)
Theorem LTL3_monitor_alt :
    !l u. LTL3_monitor l u = GEN_LTL3_monitor l u 0
Proof
    RW_TAC std_ss [GEN_LTL3_monitor_def, GSYM LTL3_belief_run_alt,
                   LTL3_monitor_def]
QED

(* correctness of LTL3 monitor, an abstract version *)
Theorem GEN_LTL3_monitor_thm :
    !l u t. GEN_LTL3_monitor l u t = LTL3_SEM_TIME t u l
Proof
    RW_TAC std_ss [GEN_LTL3_monitor_def]
 >> Cases_on `GEN_LTL3_belief_run          l  u t = {}`
 >> Cases_on `GEN_LTL3_belief_run (LTL_NOT l) u t = {}`
 >> fs [LTL3_output_def] (* 4 subgoals *)
 >| [ (* goal 1 (of 4) *)
      fs [GEN_LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_TIME_def] \\
      Q.ABBREV_TAC `i = u ++ (\n. {})` \\
     `i extends u` by PROVE_TAC [extends_def] \\
      METIS_TAC [],
      (* goal 2 (of 4) *)
      RW_TAC std_ss [Once EQ_SYM_EQ, LTL3_SEM_TIME_F] \\
      Q.PAT_X_ASSUM `GEN_LTL3_belief_run (LTL_NOT l) u t <> {}` K_TAC \\
      fs [GEN_LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_TIME_def] \\
      POP_ASSUM (MP_TAC o (Q.SPEC `u ++ v`)) \\
     `u ++ v extends u` by PROVE_TAC [extends_def] \\
      METIS_TAC [],
      (* goal 3 (of 4) *)
      RW_TAC std_ss [Once EQ_SYM_EQ, LTL3_SEM_TIME_T] \\
      Q.PAT_X_ASSUM `GEN_LTL3_belief_run l u t <> {}` K_TAC \\
      fs [GEN_LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
```

```
          LTL_SEM_TIME_def] \\
      POP_ASSUM (MP_TAC o (Q.SPEC `u ++ v`)) \\
     `u ++ v extends u` by PROVE_TAC [extends_def] \\
      METIS_TAC [],
      (* goal 4 (of 4) *)
      RW_TAC std_ss [Once EQ_SYM_EQ, LTL3_SEM_TIME_def] >|
      [ (* goal 4.1 (of 2) *)
        Q.PAT_X_ASSUM `GEN_LTL3_belief_run l u t <> {}` K_TAC \\
        fs [GEN_LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
            LTL_SEM_TIME_def, extends_def] \\
        POP_ASSUM (STRIP_ASSUME_TAC o (Q.SPEC `c`)) \\
        METIS_TAC [],
        (* goal 4.2 (of 2) *)
        Q.PAT_X_ASSUM `~!w. P` K_TAC \\
        Q.PAT_X_ASSUM `GEN_LTL3_belief_run (LTL_NOT l) u t <> {}` K_TAC \\
        fs [GEN_LTL3_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
            LTL_SEM_TIME_def, extends_def] \\
        POP_ASSUM (STRIP_ASSUME_TAC o (Q.SPEC `c`)) \\
        METIS_TAC [] ] ]
QED

(* PTLTL monitor *)
Theorem PTLTL_monitor_thm :
    !f u. IS_PAST_LTL f /\ 0 < LENGTH u ==>
          (PTLTL_SEM_ALT u f = THE (GEN_LTL3_monitor f u (LENGTH u - 1)))
Proof
    RW_TAC std_ss [PTLTL_SEM_ALT_LTL3, GEN_LTL3_monitor_thm]
QED

(* ------------------------------------------------------------------------- *)
(*   ABRV monitor (abstract version)                                         *)
(* ------------------------------------------------------------------------- *)

(* ABRV-LTL *)
Type ABRV_LTL[pp] = ``:LTL3 option``

Overload true    = ``SOME LTL3_T``
Overload false   = ``SOME LTL3_F``
Overload unknown = ``SOME LTL3_U``
Overload error   = ``NONE :ABRV_LTL``

Definition LTL4_output_def :
   (LTL4_output T T = unknown) /\
   (LTL4_output T F = true) /\
   (LTL4_output F T = false) /\
   (LTL4_output F F = error)
End
```

```
Definition LTL4_SEM_TIME_def :
    LTL4_SEM_TIME K phi (u :'a set list) t =
    if ~compatible u K then error
    else if compatible u K /\
           (!w. (concat u w) IN K ==> LTL_SEM_TIME t (concat u w) phi) then true
    else if compatible u K /\
           (!w. (concat u w) IN K ==> ~LTL_SEM_TIME t (concat u w) phi) then false
    else unknown
End

Definition GEN_LTL4_belief_run_def :
    GEN_LTL4_belief_run K phi u t =
      {i | i extends u /\ LTL_SEM_TIME t i phi /\ i IN K}
End

Definition ABRV_monitor_def :
    ABRV_monitor K phi u t =
    LTL4_output (GEN_LTL4_belief_run K phi           u t <> EMPTY)
                (GEN_LTL4_belief_run K (LTL_NOT phi) u t <> EMPTY)
End

Theorem ABRV_monitor_thm :
    !K phi u t. ABRV_monitor K phi u t = LTL4_SEM_TIME K phi u t
Proof
    RW_TAC std_ss [ABRV_monitor_def]
 >> Cases_on `GEN_LTL4_belief_run K phi           u t = {}`
 >> Cases_on `GEN_LTL4_belief_run K (LTL_NOT phi) u t = {}`
 >> RW_TAC std_ss [LTL4_output_def] (* 4 subgoals *)
 >| [ (* goal 1 (of 4) *)
      fs [GEN_LTL4_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_TIME_def] \\
      rw [LTL4_SEM_TIME_def, compatible_def] >| (* 3 subgoals *)
      [ (* goal 1.1 (of 3) *)
        CCONTR_TAC \\
        Q.PAT_X_ASSUM `!w. u ++ w IN K ==> _` (MP_TAC o Q.SPEC `c`) >> rw [] \\
       `u ++ c extends u` by METIS_TAC [extends_def] \\
        METIS_TAC [],
        (* goal 1.2 (of 3) *)
        CCONTR_TAC >> fs [] \\
       `u ++ w extends u` by METIS_TAC [extends_def] \\
        METIS_TAC [],
        (* goal 1.3 (of 3) *)
        CCONTR_TAC >> fs [] \\
       `u ++ w extends u` by METIS_TAC [extends_def] \\
        METIS_TAC [] ],
      (* goal 2 (of 4) *)
```

```
      fs [GEN_LTL4_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_TIME_def] \\
      rw [LTL4_SEM_TIME_def, compatible_def] >| (* 3 subgoals *)
      [ (* goal 2.1 (of 3) *)
       `?c. x = u ++ c` by METIS_TAC [extends_def] \\
        Q.PAT_X_ASSUM `!w. u ++ w IN K ==> _` (MP_TAC o (Q.SPEC `c`)) >> rw [],
        (* goal 2.2 (of 3) *)
        fs [] >> Q.EXISTS_TAC `w` >> rw [],
        (* goal 2.3 (of 3) *)
        fs [] \\
       `u ++ w' extends u` by METIS_TAC [extends_def] \\
        METIS_TAC [] ],
      (* goal 3 (of 4) *)
      fs [GEN_LTL4_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_TIME_def] \\
      rw [LTL4_SEM_TIME_def, compatible_def] >| (* 3 subgoals *)
      [ (* goal 3.1 (of 3) *)
       `?c. x = u ++ c` by METIS_TAC [extends_def] \\
        Q.EXISTS_TAC `c` >> rw [],
        (* goal 3.2 (of 3) *)
        fs [] \\
        Q.PAT_X_ASSUM `!w. u ++ w IN K ==> _` (MP_TAC o (Q.SPEC `w`)) >> rw [] \\
       `u ++ w extends u` by METIS_TAC [extends_def] \\
        METIS_TAC [],
        (* goal 2.3 (of 3) *)
        fs [] \\
       `u ++ w extends u` by METIS_TAC [extends_def] \\
        METIS_TAC [] ],
      (* goal 4 (of 4) *)
      fs [GEN_LTL4_belief_run_def, EXTENSION, NOT_IN_EMPTY, GSPECIFICATION,
          LTL_SEM_TIME_def] \\
      rw [LTL4_SEM_TIME_def, compatible_def] >| (* 3 subgoals *)
      [ (* goal 4.1 (of 3) *)
       `?c. x' = u ++ c` by METIS_TAC [extends_def] \\
        Q.PAT_X_ASSUM `!w. u ++ w IN K ==> _` (MP_TAC o (Q.SPEC `c`)) >> rw [],
        (* goal 4.2 (of 3) *)
        fs [] \\
       `?c. x = u ++ c` by METIS_TAC [extends_def] \\
        Q.PAT_X_ASSUM `!w. u ++ w IN K ==> _` (MP_TAC o (Q.SPEC `c`)) >> rw [],
        (* goal 4.3 (of 3) *)
        fs [] \\
        Q.EXISTS_TAC `w` >> rw [] ] ]
QED

val _ = export_theory ();
```

Appendix B

CORBA-Based Client-Server Monitoring

NuRV supports online remote monitoring (i.e. *monitor server*), based on CORBA (The *Common Object Request Broker Architecture*). After executing a command, NuRV stops the interactive shell and starts to listen on network so that user code can remotely execute the `heartbeat` command (not directly but in an equivalent way) for online monitoring.

In some senses this is the "real" online monitoring, although it is in theory possible to run NuRV as an interactive session inside another program. Furthermore, if there is only one client and one server, NuRV can also be used as a dynamic library. This is like using SQLlite instead of full RDBMS such as Oracle or MySQL. In any case, both finite- and infinite-state monitoring are supported.

B.1 About CORBA

Object Management Group, Inc. describe their CORBA architecture as follows:

> The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate. The ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine

or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems. In fielding typical client/server applications, developers use their own design or a recognized standard to define the protocol to be used between the devices. Protocol definition depends on the implementation language, network transport and a dozen other factors. ORBs simplify this process. With an ORB, the protocol is defined through the application interfaces via a single implementation language-independent specification, the IDL. And ORBs provide flexibility. They let programmers choose the most appropriate operating system, execution environment and even programming language to use for each component of a system under construction. More importantly, they allow the integration of existing components. In an ORB-based solution, developers simply model the legacy component using the same IDL they use for creating new objects, then write "wrapper" code that translates between the standardized bus and the legacy interfaces. CORBA is a signal step on the road to object-oriented standardization and interoperability. With CORBA, users gain access to information transparently, without them having to know what software or hardware platform it resides on or where it is located on an enterprise's network. The communications heart of object-oriented systems, CORBA brings true interoperability to today's computing environment.

B.2 Client-Server Monitoring

With help of CORBA, NuRV supports multiple clients connecting to multiple servers. Here each "monitor server" is a NuRV running process in which multiple LTL properties are added with their corresponding runtime monitors built by `build_monitor` command. Note that a single NuRV process can indeed provide multiple monitors corresponding to different LTL properties, however all these monitors must share the same ground model as the RV assumptions. Thus the ability of letting a single monitor client to connect to multiple NuRV processes (without extra programming overheads) is necessary in certain applications.[1]

The following further descriptions may settle the potential concern on the scalability:

- Multiple NuRV processes (monitor server) can run together, on same or different machines,

[1] By running multiple NuRV processes one can also benefit from multiple CPUs in the host machine, as NuRV does not support multi-threading in its core monitoring algorithms.

without any concern on the potential conflict of listening ports. In another words, multiple NuRV processes can start up in any order (even after the clients, as long as they are not queried yet). However, each NuRV processes must have a unique "instance name". (This should be a common need, as otherwise there's no way to know who has what properties being monitored.)

- All monitor servers register their names in a small central server daemon (the name service) provided by third-party software (with multiple choices), just like Internet DNS (domain name system).
- One or multiple monitor clients can connect to these monitor servers by only their instance names. Client programming has the same complexity for one server or multiple servers, because clients only need to know the location (address and port, etc.) of the central name server. (We provide sample client code in C, C++, Java, Lisp and Python.)

Technically speaking, the monitor server currently supported by NuRV is based on a *synchronous, push model*: the monitor clients are responsible to activately send observations to the monitor server, and such sending operations must wait until the server finished the calculations, i.e. the execution of underlying monitoring algorithms.[2]

Remark B.2.1. *Other client-server protocols such as JSON-RPC 2.0 for runtime monitoring are in consideration, due to the fact that in many recent programming languages there is no CORBA support available. However, the support of these "other" protocols, when they come in future versions of NuRV, will be designed as a CORBA bridge (or gateway): NuRV will behave as a CORBA client (and also server) for translating other protocols into CORBA calls to the same or another NuRV-based monitor server. This design will eliminate the need of CORBA programming at the side of end users but CORBA itself will be always there.*

B.3 Additional Software Dependencies

The monitor server functionality is provided by another execution `NuRV_orbit`. `NuRV_orbit` has all functionalities of `NuRV`, plus the CORBA-based monitor server support. The reason we provide two executions is that:

1. The additional software (by dynamically linking) required by `NuRV_orbit` may not be available in some versions of operating systems of end users;
2. In the future, we may provide more execution variants linking different libraries.

[2] In the future, NuRV may additionally support the *asynchronous, push model*. This is particularly important when users want to send a single observation to get the verdicts of many or all monitors, since checking the monitoring results for many properties may take a long time while the client cannot wait.

On Linux (e.g. Debian and Ubuntu), `NuRV_orbit` requires a package called `orbit2`. On Mac OS X, it requires the same package from MacPorts [3]. (In future versions of NuRV, we plan to switch to omniORB, which is a more active CORBA implementation than ORBit.)

To use the monitor server, third-party naming service is required. This service is provided by many packages. The following 3 software are confirmed working:

- Java SE (JDK) 1.8 (execution name: `tnameserv`);
- Linux package `orbit2-nameserver` (execution name: `orbit-name-server-2`);
- Linux/Mac[4] package `omniorb` (execution name: `omniNames`).

B.4 A Tutorial of CORBA-based Monitor

Let's continue the online monitoring example in Section 8.4 (with the SMV file `disjoint.smv` found in Section 8.5) and turn NuRV into a monitor server. [5]

1. Open a Terminal window and start the third-party name service from JDK 1.8: (JDK prior to 1.8 also provides it.)

   ```
   $ tnameserv
   ```

   The above command will print a long string starting with "IOR:" (which stands for an *Interoperable Object Reference*) and listen on a TCP port.

2. Open another Terminal window and start NuRV (the variant with monitor server support) in the same directory with `disjoint.smv` [6] (otherwise a new LTL property can be added by the command `add_property`):

   ```
   $ NuRV_orbit -int disjoint.smv
   ```

   Keep in mind that an LTL property $p \mathbf{U} q$ has been defined already and stored at index 0 of property manager. Following Section 8.4, the following NuRV commands builds the internal monitor for it:

   ```
   NuSMV> go
   NuSMV> build_monitor -n 0
   ```

[3] `https://www.macports.org`

[4] On Mac OS X, the package is provided by MacPorts.

[5] This tutorial is tested on Mac OS X 10.15. The same steps should work on all supported versions of NuRV on Mac OS X and Linux. Currently NuRV does not support monitor server on MS Windows.

[6] This SMV file is also in the directory `client/c` of the shipped common files.

3. Start the monitor server by executing the command `monitor_server` with the IOR string returned by `tnameserv` in the first step: (you must substitute "`IOR:...`" with the actual (long) string returned by `tnameserv`)

   ```
   NuSMV> monitor_server -N IOR:...
   ```

   The above command should return something like "Binding service reference at name service against id: NuRV/Monitor/Service". Note that this monitor server is uniquely identified by "NuRV/Monitor/Service". [7]

   (Note: use Ctrl+C to terminate the monitor server and return to the shell prompt.)

4. Make sure Homebrew [8] or MacPorts packages `orbit` and `pkg-config` are installed in your Mac system. Go to directory `client/c` and execute `make` to build the C-based monitor client:

   ```
   $ make
   ```

   If everything goes correctly, at the end there will be an execution `monitor_client` being built. It hardcoded a monitor client which sends the trace $pppqqq$ (3 times p and 3 times q, just like the sample in Section 8.4) to the monitor server. Again, this client program must know the location of the name server (by knowing the IOR string): (once again, you must substitute "`IOR:...`" with the actual (long) string returned by `tnameserv` in the first step)

   ```
   $ ./monitor_client -ORBInitRef NameService=IOR:...
   ```

   If everything goes fine, you should see the following outputs by the above monitor client. The first line is a reminder of its usage, and next line says that it has found the monitor server identified by the name "NuRV/Monitor/Service". And the rest is the monitoring outputs (3 times "unknown" and 3 times "true"):

   ```
   *** Usage: ./monitor-client -ORBInitRef NameService=IOR:...

   Resolving service reference from name-service with id "NuRV/Monitor/Service"
   unknown
   unknown
   unknown
   true
   true
   true
   ```

[7] The first two parts of the ID are always "NuRV/Monitor", while the third part "Service" can be changed by using command option `-i`. When starting multiple monitor servers (i.e. multiple NuRV processes, each of them should have different IDs.

[8] `https://brew.sh`

The business logic of the monitor client can be found near the end of the C code file `monitor-client.c` (see Section B.6.2 for more details of the sample code and other files in the same directory.)

B.5 The Simple Monitor Interface

It can be undersood that, the monitor server running inside NuRV is an instance (or object) of a class, which has the following interface given in an Interface Definition Language (IDL):

```
#pragma prefix "eu.fbk"

module Monitor {
  #pragma version Monitor 1.0

  enum Verdict {RV_True, RV_False, RV_Unknown, RV_Error};
  #pragma version Verdict 1.0

  interface MonitorService {
    #pragma version MonitorService 1.0

    // this "any" index can only be string or long
    Verdict heartbeat (in any index, in string state);
    #pragma version heartbeat 1.0

    oneway void reset (in any index, in boolean hard_p);
    #pragma version reset 1.0
  };
};
```

This class/interface is called `MonitorService`, which currently has two methods: `heartbeat` and `reset`. (It is safe to completely ignore those "progma" lines in the above IDL definition, as they are totally internal matters.) The method `heartbeat` is for sending an observation to the monitor. It takes two parameters: an integer- or string-valued `index` of LTL properties (each LTL property corresponds one internal monitor, to be created by the command `build_monitor`), and a string-valued `state` as a logical expression following NuSMV syntax representing the current observation (e.g. "`p & !q`" means $p \wedge \neg q$). The return value is in an enumeration type `Verdict` which has four possible values: `RV_True`, `RV_False`, `RV_Unknown`, and `RV_Error`. The method `reset` is for resetting the monitor. It is a "oneway" method, which immediately returns without any return value. Beside the same `index` parameter for identifying the internal monitor (or property), the Boolean parameter `hard_p` is used for choosing between hard and soft resets: *if this parameter is true, then it is a hard reset, otherwise it is a soft reset.*

The job of a monitor client is to mapping this monitor service instance from remote (i.e. the process space of monitor server) to local. The monitor server(s), after startup, will register

their instance to the central third-party naming service, while the monitor client(s) also find the needed monitor service from the same naming service. Obviously each such monitor service instance should have a unique name. This name is by default "NuRV/Monitor/Service" and can be customized by the `-i` parameter of the command `monitor_server`. (See NuRV User Manual [51] for more details.)

Once the monitor client succeed in mapping the monitor service instance to its local process space, it will behave just like a normal object in its own (object-oriented) programming language (for non-OO languages like C, the method calls are simulated by normal C functions on instance pointers).

The present interface is "simple" in the sense that any method call of `heartbeat` must wait until the related monitoring computation finished on the server side. In another words, this is a *synchronous* interface.[9]

B.6 Monitor Client Programming

In this section we describe the monitor client programming in five supported programming languages: C, C++, Java, Common Lisp and Python. All involved sample client code can be found in NuRV common files shipped with the main executions.

NOTE: starting from version 1.7.0, NuRV ships with two execution files on platforms with monitor server supports: `NuRV[.exe]` and `NuRV_orbit`, only the latter supports monitor server. Using the command `monitor_server` on the normal `NuRV[.exe]` will cause the program immediately quit. (Currently `NuRV_orbit.exe` is not available on MS Windows, but this is not due to any essential technical difficulities.)

B.6.1 Preliminaries

By default, the monitor server only listen on Unix domain sockets and can only be connected from monitor clients written in C (see Section B.6.2 for more details). To enable the monitor server listening on TCP/IP (or even IPv6) ports, a file named "`.orbitrc`" must be created and put into the home directory with the following contents:

```
ORBIIOPUSock=1
ORBIIOPIPv4=1
ORBIIOPIPv6=0
```

For instance, the above recommended config file enables Unix domain sockets, TCP/IPv4 but keeps TCP/IPv6 disabled.

[9] In the future, NuRV may support a *synchronous* interface: the method calls immediately returns, while the monitor server will later contact the monitor client (which must have an internal event loop to listen for such contacts) with the monitoring results returned.

B.6.2 CORBA client in C

C is the native language in which NuRV and its monitor server support is written. When using the C-based monitor client on the same machine with the monitor server, client and server does not need TCP/IP at all: instead they can communicate directly by Unix domain sockets.

The C-based monitor client requires linking a library called `orbit2`, which can be easily installed on many Linux and FreeBSD systems. On Mac OS X, users can install it from Homebrew, Fink or MacPorts. The sample client code finds the needed library by `pkg-config`, which therefore must be installed together by the same packaging system. Any client code must include three C headers:

```
#include <orbit/orbit.h>
#include "monitor.h"
#include "toolkit.h" /* ie. etk_abort_if_exception() */
```

The header file `orbit/orbit.h` is provided by the `orbit2` package. The header file `monitor.h` is generated from the IDL interface, together with the C source `monitor-common.c` and `monitor-stubs.c`. (End users do not need to re-generate them and can just copy the already generated interface code for their own uses.) The header file `toolkit.h`, together with `toolkit.c`, are small toolkit files for the ease use of `orbit2`. Users can freely use them too.

The only manually written code file is thus only `monitor-client.c`. The following global variables are needed for holding the connection information.

```
static CORBA_ORB global_orb = CORBA_OBJECT_NIL; /* global orb */
static Monitor_MonitorService service = CORBA_OBJECT_NIL;
static CORBA_Environment ev[1];
```

Note that the variable service of the type `Monitor_MonitorService`. Each variable of this type holds one monitor server. If a single monitor client needs to connect to multiple monitor servers (by running multiple NuRV processes), multiple variables (or an array) of this type must be used.

The following main stages are needed for a typical monitor client programming:

1. *Initialization.* The string "`orbit-local-orb`" can be arbitrary.

   ```
   CORBA_exception_init(ev);
   global_orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
   ```

2. *Binding the name service.* The sample code connects to the default monitor service name ("NuRV/Monitor/Service" by default. Change the code if the monitor server is started with different instance names.)

   ```
   CosNaming_NamingContext name_service = CORBA_OBJECT_NIL;
   gchar *id[] = {"NuRV", "Monitor", "Service", NULL};
   ```

```
name_service = etk_get_name_service (global_orb, ev);
service = (Monitor_MonitorService) etk_name_service_resolve (name_service, id, ev);
```

3. *Sending observations to the monitor.* The following code prepare the monitor index at 0, and the variables holding two observations "p" and "q":

```
CORBA_long id = 0;
CORBA_any index;
index._type = TC_CORBA_long;
index._value = &id;

CORBA_char *state_p = "p";
CORBA_char *state_q = "q";
```

   Then the following code can be used for sending an observation to the monitor:

```
Monitor_Verdict res;
res = Monitor_MonitorService_heartbeat (service, &index, state_p, ev);
```

4. *Processing the monitor verdicts.* The following same code pieces can translated the values of the enum type `Monitor_Verdict` into different string-based outputs (and print out them):

```
switch (res) {
case Monitor_RV_True:
  g_print("true\n");
  break;
case Monitor_RV_False:
  g_print("false\n");
  break;
case Monitor_RV_Unknown:
  g_print("unknown\n");
  break;
default:
  g_print("error\n");
}
```

5. *Resetting the monitor.* The following code does a hard reset to the monitor at the previous index:

```
CORBA_boolean hard_p = CORBA_TRUE;
Monitor_MonitorService_reset (service, &index, hard_p, ev);
```

6. *Uninitialization and cleanup.*

```
CORBA_Object_release(service, ev);

if (orb != CORBA_OBJECT_NIL) {
```

```
    CORBA_ORB_destroy(orb, ev);
}
```

More details can be found in "ORBit Beginners Documentation V1.6" available on Internet.

B.6.3 C++

The sample C++ client code provided at `client/cpp` requires a library called `omniorb`, which is provided by most Linux distributions. On Mac, bot MacPorts and Homebrew provide them.

Monitor client code in C++ is more natural than the above code in C, in the sense that the monitor service is represented by a real C++ object. Below we quickly give the relevant code pieces corresponding to each stages: (Check the actual sample code for C++ exception handling. Also note that the monitor server must enable TCP/IPv4.)

1. Initialization. (The C++ header file `monitor.hh` is generated from the IDL file.)

```
#include "monitor.hh"

CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB4");
```

2. Binding the name service.

```
CORBA::Object_var obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var rootContext = CosNaming::NamingContext::_narrow(obj);

CosNaming::Name name;
  name.length(3);
  name[0].id = (const char*) "NuRV"; // string copied
  name[0].kind = (const char*) "";  // string copied
  name[1].id = (const char*) "Monitor";
  name[1].kind = (const char*) "";
  name[2].id = (const char*) "Service";
  name[2].kind = (const char*) "";

CORBA::Object_var obj2 = rootContext->resolve(name);
Monitor::MonitorService_var service = Monitor::MonitorService::_narrow(obj2);
```

3. Sending the observations.

```
CORBA::String_var state = (const char*) "p";
CORBA::Any index;
CORBA::Long l = 0;
index <<= l;

Monitor::Verdict res = service->heartbeat (index, state);
```

4. Processing the monitor verdicts.

```
string result;
switch (res) {
case Monitor::RV_True:
  result = "true";
  break;
case Monitor::RV_False:
  result = "false";
  break;
case Monitor::RV_Unknown:
  result = "unknown";
  break;
default:
  result = "error";
}
```

5. Resetting the monitor.

```
CORBA::Boolean hard_p = false;
service->reset (index, hard_p);
```

6. Ending the client.

```
orb->destroy();
```

More details can be found in omniORB 4 documents at `http://omniorb.sourceforge.net/docs.html`.

B.6.4 Java

The Java client code (at `client/java`, as an Eclipse project) only supports JDK before or equals to 1.8, in which JDK directly provides the needed libraries (Thus no third-party JAR is needed). The monitor interface IDL is translated to some Java classes under the prefix `eu.fbk.monitor`.

Java program musst have a entry/main class. The connection information is held in its public static member variable:

```
public static org.omg.CORBA.ORB orb = null;
```

1. Initialization.

```
Properties props = new Properties();
String ior = "IOR:...";
props.put("org.omg.CORBA.ORBInitRef", "NameService=" + ior);
orb = org.omg.CORBA.ORB.init(args, props);
```

2. Binding name service.

```
org.omg.CORBA.Object ncRef = orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext nc =
    org.omg.CosNaming.NamingContextHelper.narrow(ncRef);

org.omg.CosNaming.NameComponent[] monitorName =
    new org.omg.CosNaming.NameComponent[3];
monitorName[0] = new org.omg.CosNaming.NameComponent("NuRV", "");
monitorName[1] = new org.omg.CosNaming.NameComponent("Monitor", "");
monitorName[2] = new org.omg.CosNaming.NameComponent("Service", "");

org.omg.CORBA.Object monitorRef = nc.resolve(monitorName);
MonitorService service = MonitorServiceHelper.narrow(monitorRef);
```

3. Sending the observation.

```
org.omg.CORBA.Any index = orb.create_any();
index.insert_long(0); // monitor 0
Verdict res = service.heartbeat(index, "TRUE");
```

4. Processing the outputs.

```
String result = new String();
if (res == Verdict.RV_True) {
    result = "true";
} else if (res == Verdict.RV_False) {
    result = "false";
} else if (res == Verdict.RV_Unknown) {
    result = "unknown";
} else { // res == Monitor.Verdict.RV_Error
    result = "error";
}
```

5. Resetting the monitor.

```
service.reset(index, false);
```

6. Shutdown.

```
orb.shutdown(true);
orb.destroy();
```

More details can be found in JDK 1.8 documentation.

B.6.5 Common Lisp

The Common Lisp client code is based on LispWorks Enterprise Edition. The interface IDL file is part of the running program without any pre-translation.

1. Initialization. (Suppose the IOR string is stored in a variable *ior*.)

```
(defvar *client-orb*   nil) ; ORB
(defvar *name-service* nil) ; NS
(defvar *service*      nil) ; Monitor:Service instance

(setq *client-orb* (op:orb_init nil "LispWorks ORB"))
```

2. Binding the name service.

```
(corba:set-pluggable-module-details "NameService" :ior-string *ior*)

(defun get-name-service (orb)
  (let ((ref (op:resolve_initial_references orb "NameService")))
    (when ref
      (op:narrow 'CosNaming:NamingContext ref))))

(setq *name-service* (get-name-service *client-orb*))

(defun name-components (names)
  (mapcar #'(lambda (name) (CosNaming:NameComponent :id name :kind "")) names))

(setq *monitor-name* (name-components '("NuRV" "Monitor" "Service")))

(defun resolve-object (name)
  (unless *name-service*
    (warn "No name service found")
    (return-from resolve-object nil))
  (handler-case
      (op:resolve *name-service* name)
    (CosNaming:NamingContext/NotFound nil)))

(setq *service*
      (op:narrow 'Monitor:MonitorService (resolve-object *monitor-name*)))
```

3. Sending the observation.

```
(defgeneric monitor (index))
(defmethod monitor ((index integer))
  (corba:any :any-typecode corba:_tc_long :any-value index))

(defmethod monitor ((index string))
  (corba:any :any-typecode corba:_tc_string :any-value index))

(op:heartbeat *service* (monitor 0) "p & q") ; 0 is the id of an LTL property
(op:heartbeat *service* (monitor "p0") "!p") ; "p0" is the name of an LTL property
```

4. Resetting the monitor.

```
(op:reset *service* (monitor 0) nil) ; nil means soft reset (hard_p = false)
```

B.6.6 Python

The Python client code requires a Python package `py-omniORBpy` which can be found in MacPorts. Similar packages (but with different names) are available on Linux, searching keywords "py" + "omniorb". The monitor interface IDL is directly read by Python code, some minor stub Python code are also generated from the IDL file. (They are accessible by Python code `import Monitor`.)

1. Initialization.

```
import sys
from omniORB import CORBA
from omniORB import any
import Monitor
import CosNaming

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
```

2. Binding name service.

```
obj = orb.resolve_initial_references("NameService");
rootContext = obj._narrow(CosNaming.NamingContext)

name = [CosNaming.NameComponent("NuRV", ""),
        CosNaming.NameComponent("Monitor", ""),
        CosNaming.NameComponent("Service", "")]
obj = rootContext.resolve(name)
service = obj._narrow(Monitor.MonitorService)
```

3. Sending the observation (and reset):

```
service.heartbeat(to_any(0), "p␣&␣q")
service.reset(to_any("p0"))
```

More details can be found in omniORB 4 Python documents at `http://omniorb.sourceforge.net/docs.html`.

Index

ω-automata, 19

ABRV problems, 32
alarm condition, 40
ANSI C, 96
Automata Theory, 15

Büchi Automata, 21
Belief states, 14
Binary decision diagrams (BDD), 18
Bounded Model Checking, 6
Bounded Model Checking (BMC), 55, 66
branching-time logics, 23

C++, 101
Calculus of Inductive and Co-inductive Constructions, 119
canonical form, 18
Cantor's Ternary Set, 75
Common Lisp, 102, 178
compassion requirements, 18
CORBA, 167
CUDD, 18

DejaVu, 77, 113
Deterministic Finite Automata, 22
diagnoser, 40
diagnosers, 41
diagnostic monitor, 35
diameter, 67
Discrete-Event Systems, 12
DIVINE model checker, 14
Dwyer's LTL patterns, 7, 158

elementary variables, 19
Emerson-Lei, 19
equisatisfiable, 69
Execution Analysis, 2
Expansion Laws, 20
explicit-state automata, 21

Fair Kripke Structure (FKS), 18
fair states, 19
Fair Transition System (FTS), 18
Fault Detection, Identification (FDI), 3, 40
Ferrante-and-Rackoff, 23
Finite-State Machine (FSM), 22
Finite-State Transducer (FST), 22
First-order formulas, 16
First-Order Quantifier Elimination, 23, 55
first-order quantifiers, 135
forward image, 19
Fourier-Motzkin, 23, 114
future temporal operators, 19

Higher Order Logic, 119
Hilbert's choice operator, 119
HOL4, 7
hybrid systems, 139
hyperproperties, 138

IC3, 55
IC3-IA, 55
Incremental BMC, 67
Incremental Bounded Model Checking, 138
initial condition, 20

Instrumentation, 1
Interface Definition Language (IDL), 167

Java PathExplorer, 77
JavaMOP, 77
justice set, 21

Linear Temporal Logic (LTL), 16
LLVM, 103
Logic of Computable Functions, 119
Loos-and-Weispfenning, 23, 114
LTL Modulo Theory, 16
LTL Syntax, 16

Markov chain, 140
Metric First-Order Temporal Logic, 135
Model-based Runtime Verification, 13
MONA model checker, 137
Monadic Second-Order Logic, 137
monitor recovery, 13
Monitor Synthesis, 2
Monitorability, 13
monitorability under assumptions, 38
Monitoring Modulo Theories, 13
MOP framework, 134
motivating example, 33, 56
multi-property monitoring, 130

non-monitorable, 2
Nondeterministic Finite-State Automata, 22
NuRV, 7, 85

ordered binary decision diagrams (OBDD), 18
out-of-order inputs, 138

Parametric Trace Slicing, 134
Partial observability, 12
past temporal operators, 19
Past-time LTL, 77
predictive monitor, 35
probabilistic systems, 140
Prolog, 94
ptLTL, 77, 111
ptLTL semantics, 77

Quantifier Temporal Logic, 111, 135
quantifier-free formulae, 19

R2U2, 77
resettable monitors, 3
RuleR, 77
RV-Monitor, 77

SAT-based symbolic model checking, 133
Satisfiability Modulo Theory (SMT), 16, 55
Second-Order Logic, 137
Second-Order Quantifier Elimination, 23
Simple Theory of Types, 119
SMT-based symbolic model checking, 13
state-explosion, 1, 137
step constraint, 67
system under scrutiny (SUS), 1

tableau, 19
temporal operator, 16
Testing, 1
Theorem Proving, 1
timed language, 139
trace non-storing, 52
trace-length independent, 52, 75
TraceContract, 77
Traditional RV, 4
transition relation, 18, 20

undecidability, 37
univariate quadratic polynomial, 23
ustice requirements, 18

well-formed formulae, 17
WS1S, 137