Assumption-Based Runtime Verification of Infinite-State Systems

Alessandro Cimatti, Chun Tian, and Stefano Tonetta

Fondazione Bruno Kessler, Trento, Italy
{cimatti,ctian,tonettas}@fbk.eu

Abstract. Runtime Verification (RV) basically means monitoring an execution trace of a system under scrutiny and checking if the trace satisfies or violates a specification. In Assumption-Based Runtime Verification (ABRV), runtime monitors may be synthesized from not only the specification but also a system model (either full or partial), which represents the assumptions on which the input traces are expected to follow. With assumptions the monitor can additionally check if the input traces actually follow the assumptions. Some previous research has shown that monitors under assumptions can be more precise or even predictive, while non-monitorable specifications may become monitorable under assumptions.

The question of synthesizing runtime monitors for finite-state systems and propositional or first-order temporal logics, with or without assumptions, has mostly been answered by prior work. For monitoring infinite-state systems, however, most existing approaches focus on supporting parametric or first-order specifications while they cannot be easily extended to support assumptions.

This paper presents a general solution for ABRV of infinite-state systems by a reduction of RV problems to LTL Model Checking (MC), which is further based on Satisfiability Modulo Theories and other techniques. When First-Order Quantifier Elimination (QE) is also available, the corresponding algorithm can be greatly optimized. This solution is general because in theory any LTL MC (and QE) algorithms can be used, and the supported types of infinite-state variables also depend on these underlying algorithms. In particular, the relatively expensive model checking can be minimized by a modified version of Bounded Model Checking algorithm which performs model checking incrementally on each input of the monitor.

1 Introduction

Runtime Verification (RV) [15, 20] is a lightweight verification technique aiming at monitoring the execution trace of a system under scrutiny (SUS) and checking if the trace satisfies or violates a specification. The central task in RV is *monitor synthesis*, i.e. generating from the specification a *runtime monitor*, which takes a run (execution trace) from the SUS and outputs *verdicts* for each states of the

run. Although a specification[1] exists within the context of a system model, i.e. the abstraction of the system being specified, the current taxonomy for classifying RV tools [21] does not consider synthesizing runtime monitors from a system model, in addition to the specification.

Assumption-Based Runtime Verification (ABRV) [13] extends the traditional Runtime Verification (RV) [20, 29] by additionally assuming an underlying system model that the input traces are expected to follow. The resulting runtime monitor checks if, under the assumptions given by the model, the SUS execution satisfies or violates the property (and additionally if the execution is compliant with the assumption). Prior research [13, 24, 28] has shown that, for certain combinations of models and properties, assumption-based monitors are more precise (i.e. arriving at a conclusion based on the assumption while traditional monitors would be inconclusive), or even *predictive* (i.e. 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. Another advantage of the assumption-based approach is the possibility of monitoring properties over partially-observable systems, capturing as the assumption the relationship between observable and internal states of the SUS.

The question of synthesizing runtime monitors for finite-state systems and propositional temporal logics, with or without assumptions, has mostly been answered by prior work [1, 4, 13]. In particular, for ABRV, there exist effective automata-based approaches using Binary Decision Diagrams (BDD) [7] to represent belief states, i.e., the set of automata states where the system can be according to a sequence of (partial) observations.

In the case of infinite-state systems, of which the state variables may have infinite domains (such as integer, rational and real variables), the corresponding RV problem (i.e. monitoring LTL) can be in theory resolved by evaluating the property (Boolean) propositions over the non-Boolean variables. Going from propositional to first-order temporal logic (or even further), existing work mostly put a focus on supporting things like parametric specifications [34] or specifications with first-order quantifiers [23].

This paper presents a general new approach for ABRV of infinite-state systems (the related algorithms can also be applied to finite-state systems). Instead of relying on BDDs, which is used by NuRV [14] (the previous tool implementation of ABRV), the idea is based on Satisfiability Modulo Theories (SMT) [2]. We show how to reduce RV problems directly to SMT-based LTL Model Checking (MC) problems, then solvable by model checkers like nuXmv [8]. This solution is general because in theory any LTL MC (and QE) algorithms can be used, and the supported types of infinite-state variables also depend on these underlying algorithms. For the LTL semantics over finite traces, which is also the semantics of monitoring outputs, our choice is still based on LTL_3 [1] with respect to extra

[1] According to [21], a *specification* is a concrete description of a *property* (a partition of traces) using a well-defined formalism (like LTL). However, this difference is not very important here, and thus we use the words “property” and “specification” interchangeably for the rest of this paper.

verdicts (out-of-model) due to RV assumptions. In comparison with other possible LTL semantics for RV purposes [3], it turns out that, with our choice, there exists a simple and elegant reduction from RV to MC problems. If, additionally, First-Order Quantifier Elimination [33] is also available (for the chosen first-order theory), the corresponding algorithm can be greatly optimized, without seeing the monotonically growing of SMT formulas during the monitoring. In this case, the algorithm keeps track of a belief state, representing all states in which the system can be according to the assumption after a sequence of observation. The RV problem is then reduced to checking, after each observation, the emptiness checking of symbolic automata with the belief states as initial conditions.

However, there are 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. To this purpose, we extend the basic RV-MC reductions with optimizations that perform (relatively cheap) incomplete checks instead of the more expensive model checking calls. One such optimization is to always check first the literal emptiness of the belief state by SMT solvers, the other is to use the incomplete plain Bounded Model Checking (BMC) [5], with improved encodings for the full class of LTL properties [12], only for detecting counterexamples (the plain use). With these significantly faster checks, the full IC3-based model checker [10] is now rarely called (at most twice in each run).

To obtain some empirical results, we have implemented our algorithms in a new version of NuRV, which is based on nuXmv and MathSAT SMT solver [11] (MathSAT provides some quantifier elimination procedures). We present an experimental evaluation of the performance of the basic monitoring algorithms and various optimizations. Results on the best optimized algorithm seem to be promising for practical applications.

Outline of the paper. In Section 2 we recall some related concepts and definitions. In Section 3, we describe an example of ABRV with infinite-state assumptions. In Section 4 we give two basic RV algorithms and prove their correctness proofs. Furthermore, Section 4.3 discusses various optimizations of the basic algorithms. Some experimental evaluations and results are given in Section 5. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Preliminaries

2.1 Satisfiability Modulo Theory

We work in the setting of Satisfiability Modulo Theory (SMT) [2] and LTL Modulo Theory (see, e.g., [9]). 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.

2.2 First-Order Quantifier Elimination

First-order quantifier elimination [33] 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. 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. Quantified elimination is possible only for some first-order theories. In practice, for $\mathcal{LRA}$, most SMT solvers use methods like Fourier-Motzkin [27], Ferrante-and-Rackoff [22] or Loos-and-Weispfenning [30]. Note that QE procedures do not guarentee any kind of boundedness of the resulting formulas.

2.3 Fair Transition System

Infinite-state systems (used as RV assumptions) in this paper are described as *Fair Transition Systems* (FTS) [32], denoted by $\langle V, \Theta, \rho, \mathcal{J} \rangle$, where $V = \{x_1, \ldots, x_n\}$ is a finite set of variables, Θ the *initial condition*, ρ the *transition relation*, and $\mathcal{J}$ a (finite) set of *justice conditions*. (Θ, ρ and each element of $\mathcal{J}$ are quantifier-free $\mathcal{T}$-formulas.) Given an FTS $K = \langle V, \Theta, \rho, \mathcal{J} \rangle$, a *state* s of K is just a value assignment of variables in V. Any formula using variables in V can be interpreted as the set of states satisfying the formula. Θ and each $J \in \mathcal{J}$ are such formulas, while ρ is a formula about V and its *primed version* $V' = \{x'_1, \ldots, x'_n\}$ indicating the relationship between the *current* and *next* states. If s is an assignment to V, s' is the corresponding assignment to V' such that $s'(v') = s(v)$ for all $v \in V$.

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

$$\mathrm{fwd}(\psi(V), \rho(V, V'))(V) \doteq (\exists V.\ \rho(V, V') \wedge \psi(V))[V/V'] \tag{1}$$

where $[V/V']$ denotes the substitution of (free) variables in V' with the corresponding one in V. The existential quantifiers in forward images can be eliminated by QE procedures.

A *fair path* $\sigma = s_0 s_1 \cdots$ of K is an infinite sequence of states such that: (1) $s_0 \models_{\mathcal{T}} \Theta(V)$; (2) for each i, $s_i \cup s'_{i+1} \models_{\mathcal{T}} \rho(V, V')$; (3) for each $J \in \mathcal{J}$ there are infinitely many i such that $s_i \models_{\mathcal{T}} J(V)$. Let $\mathcal{L}(K)$ be the set of all fair paths of K. Sometimes we write σ_i for the zero-indexed i-th element of σ, i.e. $\sigma = \sigma_0 \sigma_1 \ldots$. A *trace* is a finite or infinite sequence of value assignments of V.

2.4 Linear Temporal Logic

In this paper, we consider properties specified in first-order quantifier-free Linear Temporal Logic (LTL) [31] with both future and past operators. The set of LTL formulas can be inductively defined as follows:

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

where the (quantifier-free) formula α is built by a set of variables V and a first-order signature Σ, and is interpreted according to a Σ-theory $\mathcal{T}$. 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 like false, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are used as syntactic abbreviations with their standard meanings in propositional logic. We also use the metric operators $\mathbf{X}^n$ and $\mathbf{F}^{\leq k}$ here defined as an abbreviations: $\mathbf{X}^0\varphi := \varphi$, $\mathbf{X}^{k+1}\varphi := \mathbf{X}\mathbf{X}^k\varphi$ for $k \geq 0$, and $\mathbf{F}^{\leq k}\varphi := \bigvee_{0\leq i\leq k} \mathbf{X}^i\varphi$.

The semantics of LTL formulas over infinite traces are standard for propositional and temporal operators (see, e.g. [13] for the full definitions). For atoms the semantics is reduced to the theory-specific semantics:

$$\sigma, i \models \alpha(V, V') \quad \text{iff} \quad \sigma_i \cup \sigma'_{i+1} \models_{\mathcal{T}} \alpha(V, V')$$

Any LTL formula can be translated into an equivalent FTS such that the set of fair paths of the FTS, when projected to V, coincides with the set of infinite traces satisfying the same LTL formula. Here we use essentially the same LTL translation algorithm in [13] (Section 2) except that the atomic formulas are translated syntactically as Boolean variables. Note that, for any LTL formula φ and its negation $\neg\varphi$, their LTL translations (as FTS) only differ at the initial conditions (this property is indeed leveraged in all our RV algorithms to be presented in this paper).

2.5 Assumption-Based Runtime Verification

The definition of the ABRV problem and the related ABRV-LTL semantics adopted in this paper are essentially the same as in the authors' previous paper for the finite-state case [13]. There are some minor changes for the support of infinite-state systems and non-Boolean variables.

Let K be an FTS as the RV assumption on the behavior of the SUS. When the SUS is *partially observable*, the monitor has only partial information on the actual state of the SUS. For simplicity purposes we assume that the monitor receives a sequence of value assignments for a subset O of all state variables V of the FST K (see [13] for a more general setting). A trace over O, also called a trace of observations, is a finite or infinite sequence of value assignments of O.

Given a finite trace u over O, the set of fair paths *compatible* with u is defined below: (roughly speaking, each u_i is a subset of variable assignments of σ_i)

$$\mathcal{L}^K(u) \doteq \left\{\sigma \in \mathcal{L}(K) \;\middle|\; \forall i < |u|.\, \sigma_i \models_{\mathcal{T}} u_i\right\} \ . \tag{2}$$

The LTL semantics of φ over the finite trace u at index i, having four possible values: *conditionally true* ($\top^a$), *conditionally false* ($\bot^a$), *inconclusive* (?) and *out-of-model* ($\times$), is defined below:

$$[\![u, i \models \varphi]\!]_4^K \doteq \begin{cases} \times, & \text{if } \mathcal{L}^K(u) = \emptyset \\ \top^a, & \text{if } \mathcal{L}^K(u) \neq \emptyset \text{ and } \forall w \in \mathcal{L}^K(u).\ w, i \models \varphi \\ \bot^a, & \text{if } \mathcal{L}^K(u) \neq \emptyset \text{ and } \forall w \in \mathcal{L}^K(u).\ w, i \models \neg\varphi \\ ? & \text{otherwise} \end{cases} \quad (3)$$

In the finite-state ABRV, we also consider a Boolean reset signal that is used to reset the index used as reference to evaluate the property. In this paper, to simplify the presentation, we omit this additional feature (although it is implemented and supported by the tool implementation) and define the infinite-state ABRV problem as the problem of constructing a function (as *runtime monitor*) taking a finite trace u over O and returning an ABRV-LTL verdict:

$$\mathcal{M}_\varphi^K(u) \doteq [\![u, 0 \models \varphi]\!]_4^K \quad . \quad (4)$$

3 Motivating Example

In this section, we describe a use case of ABRV with an infinite-state assumption using 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 $\mathbf{G}(s \to \mathbf{F}^{\leq 7}(t = 100))$, 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.

4 ABRV Algorithms for Infinite-State Systems

4.1 ABRV reduced to Model Checking

We first revisit the relationship between runtime verification and model checking, as clarified in [28], to conceive a trivial solution ABRV based on calling a model checker at every observation.

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),

$$\mathcal{M}_\varphi^K(u) = [\![u, i \models \varphi]\!]_4^K = \begin{cases} \times, & \text{if } K \times S_u \models \varphi \text{ and } K \times S_u \models \neg\varphi \\ \top^{\mathrm{a}}, & \text{if } K \times S_u \models \varphi \text{ and } K \times S_u \not\models \neg\varphi \\ \bot^{\mathrm{a}}, & \text{if } K \times S_u \not\models \varphi \text{ and } K \times S_u \models \neg\varphi \\ ?, & \text{otherwise} . \end{cases} \quad (5)$$

From this equation, we can derive a simple monitor called `monitor1`, which calls the model checker twice for each input state. It is also depicted in Fig. 1, where the output is defined as in (5).

4.2 ABRV reduced to MC and quantifier elimination

In `monitor1` the entire input trace (the prefix received so far) must be encoded into a model (i.e. an FTS) S_μ, and obviously the model checker is called on

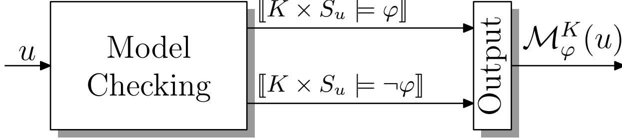

Fig. 1. ABRV reduced to MC

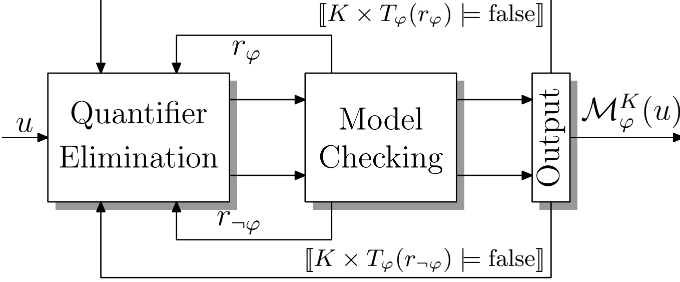

Fig. 2. ABRV reduced to MC and QE

increasingly bigger problems linear to the length of the trace prefix. In practice `monitor1` will be too slow after receiving even a small number of input states. The key for getting a better RV algorithm is to find a way to maintain some internal status which is updated by each input state in the trace. For automata-based RV monitors, the status is the location of monitor automata. For rewriting-based RV approaches, the status is the current form of the monitoring property after rewriting.

Recall in the finite-state ABRV algorithm [13], 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 [13] is trace-length independent [17], i.e. having bounded memory consumption (with also a time complexity linear to the length of the trace prefix).

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

Table 1. Output Table of Fig. 2 and Algorithm 1

Algorithm 1: 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
```

The monitor monitor2 detailed in Algorithm 1 is very similar to the symbolic algorithm given [13]. 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 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. 2, 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 2.2) such that

$$\texttt{quantifier_elimination}(V,\ \alpha(V \cup V')) \doteq (\exists V.\alpha(V \cup V'))[V/V'] = \beta(V) \quad (6)$$

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 [13]) 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 [19]) 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 1.

The correctness of Algorithm 1 is given by the following theorem: (the proof is omitted due to page limits.)

Theorem 1. *The function* `monitor2` *given in Algorithm 1 correctly implements the ABRV monitor* $\mathcal{M}_\varphi^K(\cdot)$.

4.3 Optimizations

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 counterexamples, e.g. the plain Bounded Model Checking (BMC). (Also note that, for infinite-state systems, the property may be violated but no lasso-shaped counterexample exists; in this case, neither BMC or the full IC3-IA algorithm can find it.) The following 4 *basic* optimizations, namely *o1–o4*, are identified:

- o1 If the monitor has already reached conclusive verdicts ($\top^a$ or $\perp^a$), then for the runtime verification of the next input state *at most one* MC call is need. 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 $\perp$, then there is no need to further call model checkers to detect its emptiness.
- 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. 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 counterexamples. This optimization is to call the incomplete plain BMC (or any

other MC procedure which detects counterexamples) 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 counterexample 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.

Algorithm 2: The optimized version of Algorithm 1

```
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 ⊥;            // counterexample 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 2. *Assuming* `BMC` *always find the counterexample whenever it exists,* `IC3_IA` *is called at most twice in the "online" version of Algorithm 2 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 $\bot$ (counterexamples 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 $\top^{\mathrm{a}}$, 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 counterexample.)
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 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. □

4.4 ABRV reduced to Model checking and Incremental BMC

Further optimizations can be done by leveraging the *incrementality* of Bounded Model Checking occurred in Algorithm 2, where the function `BMC` are called as incomplete preliminary steps before the full `IC3_IA` calls. In the following discussions we assume the audience is familiar with the internal work of BMC algorithms (otherwise see [5] and [12]).

We first define a BMC encoding of the belief states after a sequence of observations $u_0 u_1 \cdots u_n$, denoted by $\mathrm{bs}(u_0 u_1 \cdots u_n)$. These are inductively given by

$$\mathrm{bs}(u_0)(V) = I(V) \wedge u_0(V), \tag{7}$$

$$\mathrm{bs}(u_0 u_1 \cdots u_{i+1})(V) = \mathrm{fwd}\big(\mathrm{bs}(u_0 u_1 \cdots u_i)(V), T(V, V')\big)(V) \wedge u_{i+1}(V) \ . \tag{8}$$

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

Theorem 3 (Equisatisfiability). *When $k > 1$, the following two formulas*

$$I(V_0) \wedge u_0(V_0) \wedge \bigwedge_{j=0}^{k-1} \big[T(V_j, V_{j+1}) \wedge u_{j+1}(V_{j+1})\big], \tag{9}$$

and

$$\mathrm{bs}(u_0 u_1 \cdots u_k)(V) \tag{10}$$

are equi-satisfiable.

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.[2]

In Algorithm 3 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 4 and 5):

- `init_nonemptiness` for creating a persistent SMT solver instance,
- `update_nonemptiness` for checking the nonemptiness 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 3 (relative to the correctness of undefined methods) can be seen by a comparison with Algorithm 1. 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`.

In Algorithm 4 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 $\left(\bigwedge_{\psi \in \mathcal{J}_K \cup \mathcal{J}_\varphi} \mathbf{GF}\, \psi\right) \rightarrow \mathrm{false}$ as in Algorithm 2. 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*.

The core of incremental BMC algorithm for RV, `update_nonemptiness`, is finally given in Algorithm 5.

[2] In the ideal case (when BMC stopped by having found a counterexample, and the overall monitoring verdicts is conclusive), the monitor only needs to call SMT solver *once* to decide the next monitoring output.

Algorithm 3: The optimized RV monitor based on incremental BMC

```
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;
```

5 Experimental Evaluation

The RV algorithms presented in this paper have been implemented in NuRV [14][3]. The usefulness of RV assumptions has been explored in previous papers (see, e.g., Section 5 of [13]), thus the focus of experimental evaluations here is mainly at the correctness and performance of ABRV algorithms for infinite-state systems. All performance results are obtained on a MacBook Pro laptop with an 8-core Intel Core i9 (2.3GHz).

The correctness of these RV algorithms, 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.

[3] The official site of NuRV is now at `https://es-static.fbk.eu/tools/nurv/`.

Algorithm 4: 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.max_k := max_k;                          // a local copy of max_k
```

5.1 Tests on the motivating example (Section 3)

The actual monitoring results on the motivating example in Section 3 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.

5.2 Tests on Dwyer's LTL patterns

We use again Dwyer's LTL patterns [18] (55 in total[4]) 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.

Fig. 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

[4] See also `https://matthewbdwyer.github.io/psp/patterns/ltl.html`.

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

as an infinite-state model. The length of input traces increases 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.

Fig. 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

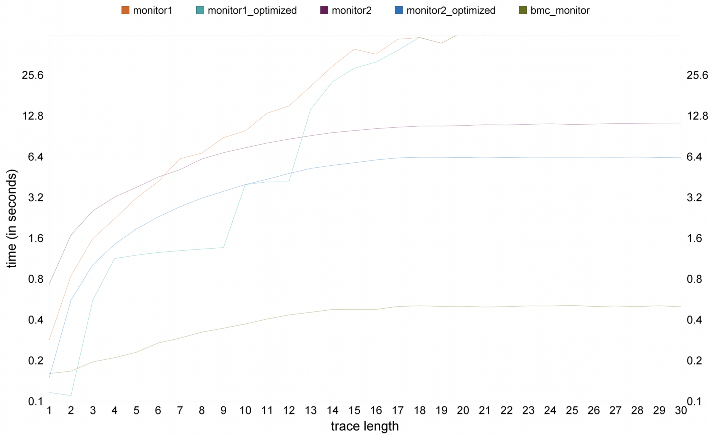

Fig. 3. Performance of five RV algorithms on Pattern 49

on the two monitors. For most patterns (and also on average), `bmc_monitor` is about 10x faster than `monitor2_optimized`.

6 Related Work

Despite the vast literature on SAT- and SMT-based symbolic model checking [6], 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) [16] 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. In [35], 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 [28]. Adapting existing model checkers for RV purposes is a natural idea for reducing the costs of tool development from scratch. Similar with NuRV (which is adapted from nuXmv), the DIVINE model checker was adapted to perform runtime verification [26]. 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. [36].

Belief states have been used in planning under partial observability. See, for example, the work of in [25], from which we borrow the idea of representing them

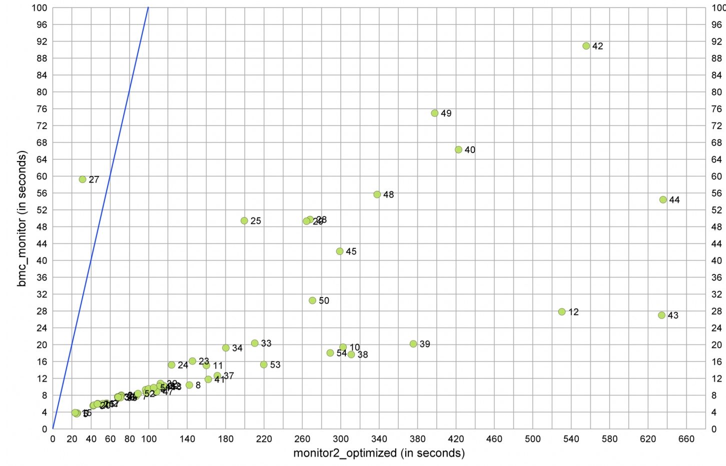

Fig. 4. Performance of `bmc_monitor` and `monitor2_optimized` on all patterns

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.

7 Conclusion

ABRV is a recently proposed framework for RV based on the definition of some assumption on the SUS behavior, which is exploited by the runtime monitor to achieve early detection, prediction and partial observability. The framework has been extended in this paper 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.

As observed in [35], 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. In the future, we plan to investigate such trade-off in realistic scenarios. We will also consider real-time temporal properties with timed assumptions and address the problem of generating monitor's code taking into account infinite-state assumptions.

References

1. Arafat, O., Bauer, A., Leucker, M., Schallhart, C.: Runtime verification revisited. Tech. Rep. Technical Report TUM-I0518, Technische Universität München, München (2005)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (Jan 2009). https://doi.org/10.3233/978-1-58603-929-5-825
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Journal of Logic and Computation **20**(3), 651–674 (Feb 2010). https://doi.org/10.1093/logcom/exn075
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20**(4), 14–64 (Sep 2011). https://doi.org/10.1145/2000799.2000800
5. Biere, A., Cimatti, A., Clarke Jr, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. In: Advances in Computers: Highly Dependable Software, pp. 117–148. Academic Press (2003)
6. Biere, A., Cimatti, A., Clarke Jr, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: LNCS 1579 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999), pp. 193–207. Springer, Berlin, Heidelberg (Jun 2013). https://doi.org/10.1007/3-540-49059-0_14
7. Bryant, R.E.: Binary decision diagrams. In: Handbook of Model Checking, pp. 191–217. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_7
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: LNCS 8559 - Computer Aided Verification (CAV 2014), pp. 334–342. Springer, Cham (Jun 2014). https://doi.org/10.1007/978-3-319-08867-9_22
9. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: SMT-based satisfiability of first-order LTL with event freezing functions and metric operators. Inf. Comput. **272**, 104502 (2020). https://doi.org/10.1016/j.ic.2019.104502
10. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 Modulo Theories via Implicit Predicate Abstraction. In: LNCS 8413 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014), pp. 46–61. Springer, Berlin, Heidelberg (Feb 2014). https://doi.org/10.1007/978-3-642-54862-8_4
11. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: LNCS 7795 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013), pp. 93–107. Springer, Berlin, Heidelberg (Feb 2013). https://doi.org/10.1007/978-3-642-36742-7_7
12. Cimatti, A., Pistore, M., Roveri, M., Sebastiani, R.: Improving the Encoding of LTL Model Checking into SAT. In: Cortesi, A. (ed.) LNCS 2294 - Verification, Model Checking, and Abstract Interpretation (VMCAI 2002), pp. 196–207. Springer Berlin Heidelberg, Berlin, Heidelberg (Apr 2002). https://doi.org/10.1007/3-540-47813-2_14
13. Cimatti, A., Tian, C., Tonetta, S.: Assumption-Based Runtime Verification with Partial Observability and Resets. In: LNCS 11757 - Runtime Verification (RV 2019), pp. 165–184. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_10
14. Cimatti, A., Tian, C., Tonetta, S.: NuRV: A nuXmv Extension for Runtime Verification. In: LNCS 11757 - Runtime Verification (RV 2019), pp. 382–392. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_23

15. Colin, S., Mariani, L.: Run-Time Verification. In: LNCS 3472 - Model-Based Testing of Reactive Systems, pp. 525–555. Springer, Berlin, Heidelberg (Jan 2008). https://doi.org/10.1007/11498490_24
16. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. International Journal on Software Tools for Technology Transfer **18**(2), 205–225 (2015). https://doi.org/10.1007/s10009-015-0380-3
17. Du, X., Liu, Y., Tiu, A.: Trace-Length Independent Runtime Monitoring of Quantitative Policies in LTL. In: Bjørner, N., de Boer, F. (eds.) LNCS 9109 - FM 2015: Formal Methods, pp. 231–247. Springer, Cham (May 2015). https://doi.org/10.1007/978-3-319-19249-9_15
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering. pp. 411–420. ACM Press, New York, USA (1999). https://doi.org/10.1145/302405.302672
19. Emerson, E.A., Lei, C.L.: Temporal Reasoning Under Generalized Fairness Constraints. In: LNCS 210 - Theoretical Aspects of Computer Science (STACS 1986), pp. 21–36. Springer, Berlin, Heidelberg (1986). https://doi.org/10.1007/3-540-16078-7_62
20. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Engineering Dependable Software Systems **34**, 141–175 (2013). https://doi.org/10.3233/978-1-61499-207-3-141
21. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A Taxonomy for Classifying Runtime Verification Tools. In: Colombo, C., Leucker, M. (eds.) LNCS 11237 - Runtime Verification (RV 2018), pp. 241–262. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_14
22. Ferrante, J., Rackoff, C.: A Decision Procedure for the First Order Theory of Real Addition with Order. SIAM Journal on Computing **4**(1), 69–76 (Mar 1975). https://doi.org/10.1137/0204006
23. Havelund, K., Peled, D.A.: Runtime Verification: From Propositional to First-Order Temporal Logic. In: LNCS 11237 - Runtime Verification (RV 2018), pp. 90–112. Springer, Cham (Oct 2018). https://doi.org/10.1007/978-3-030-03769-7_7
24. Henzinger, T.A., Saraç, N.E.: Monitorability Under Assumptions. In: Deshmukh, J., Nickovic, D. (eds.) LNCS 12399 - Runtime Verification (RV 2020), pp. 3–18. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_1
25. Hoffmann, J., Brafman, R.I.: Contingent planning via heuristic forward search witn implicit belief states. In: Biundo, S., Myers, K.L., Rajan, K. (eds.) Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA. pp. 71–80. AAAI (2005), `http://www.aaai.org/Library/ICAPS/2005/icaps05-008.php`
26. Kejstová, K., Rockai, P., Barnat, J.: From Model Checking to Runtime Verification and Back. In: LNCS 10548 - Runtime Verification (RV 2017), pp. 225–240. Springer, Cham (Aug 2017). https://doi.org/10.1007/978-3-319-67531-2_14
27. Khachiyan, L.: Fourier–Motzkin Elimination Method. In: Encyclopedia of Optimization, pp. 1074–1077. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-74759-0_187
28. Leucker, M.: Sliding between Model Checking and Runtime Verification. In: LNCS 7687 - Runtime Verification (RV 2012), pp. 82–87. Springer, Berlin, Heidelberg (Jan 2013). https://doi.org/10.1007/978-3-642-35632-2_10

29. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
30. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The computer journal **36**(5), 450–462 (1993), `https://dblp.org/rec/journals/cj/LoosW93`
31. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag New York (1992). https://doi.org/10.1007/978-1-4612-0931-7
32. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer-Verlag New York (1995). https://doi.org/10.1007/978-1-4612-4222-2
33. Marcja, A., Toffalori, C.: Quantifier Elimination. In: A Guide to Classical and Modern Model Theory, pp. 43–83. Springer Netherlands, Dordrecht (2003). https://doi.org/10.1007/978-94-007-0812-9_2
34. Reger, G., Rydeheard, D.E.: From First-order Temporal Logic to Parametric Trace Slicing. In: LNCS 9333 - Runtime Verification (RV 2015). Springer (Sep 2015). https://doi.org/10.1007/978-3-319-23820-3
35. Tekken Valapil, V., Yingchareonthawornchai, S., Kulkarni, S., Torng, E., Demirbas, M.: Monitoring Partially Synchronous Distributed Systems Using SMT Solvers. In: Lahiri, S., Reger, G. (eds.) LNCS 10548 - Runtime Verification (RV 2017), pp. 277–293. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_17
36. Zhang, X., Leucker, M., Dong, W.: Runtime Verification with Predictive Semantics. In: LNCS 7226 - NASA Formal Methods (NFM 2012), pp. 418–432. Springer, Berlin, Heidelberg (Mar 2012). https://doi.org/10.1007/978-3-642-28891-3_37