NuRV: A nuXmv Extension for Runtime Verification

Alessandro Cimatti, Chun Tian(✉), and Stefano Tonetta

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

Abstract. We present NuRV, an extension of the nuXmv model checker for assumption-based LTL runtime verification with partial observability and resets. The tool provides some new commands for online/offline monitoring and code generations into standalone monitor code. Using the online/offline monitor, LTL properties can be verified incrementally on finite traces from the system under scrutiny. The code generation currently supports C, C++, Common Lisp and Java, and is extensible. Furthermore, from the same internal monitor automaton, the monitor can be generated into SMV modules, whose characteristics can be verified by Model Checking using nuXmv. We show the architecture, functionalities and some use scenarios of NuRV, and we compare the performance of generated monitor code (in Java) with those generated by a similar tool, RV-Monitor. We show that, using a benchmark from Dwyer's LTL patterns, besides the capacity of generating monitors for long LTL formulae, our Java-based monitors are about 200x faster than RV-Monitor at generation-time and 2–5x faster at runtime.

1 Introduction

Symbolic Model Checking [16] is a powerful formal verification technique for proving temporal properties of transition systems (a.k.a. models) represented by logical formulae. In the case of Linear Temporal Logic (LTL) [15], the properties can be translated into symbolically represented ω-automata, which is then conjoined with the model and proved by search-based techniques that exhaustively analyze the infinite traces of the system [7]. Runtime Verification (RV) [10,13] on the other hand, is a lightweight verification technique for checking if a given property is satisfied (or violated) on a finite trace of the system under scrutiny (SUS). In general, LTL-based RV problems can be resolved by automata-based [1], rewriting-based [17], or rule-based [11] approaches.

In this paper, we present a new tool called NuRV, an extension of the nuXmv [4] model checker for LTL-based RV. To the best of our knowledge, this is the first time that a model checker is directly modified (or extended) into a runtime monitor (or monitor generator). It is natural to do so, as nuXmv has

This work has received funding from European Union's *Horizon 2020* research and innovation programme under the Grant Agreement No. 700665 (Project *CITADEL*).

already provided the needed infrastructure, such as a symbolic translation from LTL to ω-automata, an algorithm for computing the "fair states" (those leading to infinite paths), together with an interface to BDD library [3] based on CUDD 2.4.1 [18].

For the monitoring algorithm implemented in NuRV (c.f. [6] for more details), our start point is the automata-based approach [1] based on LTL$_3$, implemented *symbolically*. Suppose the monitoring property is φ, we first run the LTL translations twice, on φ and $\neg\varphi$, to get two symbolic automata T_φ and $T_{\neg\varphi}$, resp. Then an input trace u is synchronously *simulated* on T_φ and $T_{\neg\varphi}$, by repeatedly computing forward images w.r.t. all fair states[1]. For each input state of u, we get two sets of *belief states*, r_φ and $r_{\neg\varphi}$. Based on their emptinesses, the monitor returns one of the following verdicts:

- *conclusive true* ($\top$), if $r_\varphi \neq \emptyset$ and $r_{\neg\varphi} = \emptyset$. φ is *verified* for all future inputs;
- *conclusive false* ($\bot$), if $r_\varphi = \emptyset$ and $r_{\neg\varphi} \neq \emptyset$. φ is *violated* for all future inputs;
- *inconclusive* (?), if $r_\varphi \neq \emptyset$ and $r_{\neg\varphi} \neq \emptyset$. In this case, the knowledge of the monitor is limited by the finiteness of u.

Besides the property φ, the monitoring algorithm takes in input a model K of the SUS. This is used to declare the variables in which the properties are expressed, but more importantly to define some constraints on their temporal evolution, which represent assumptions on the behavior of the SUS. By considering only (infinite) traces of K, the above algorithm may give more *precise* outputs (turning ? into $\top/\bot$). This is obtained by using $K \otimes T_\varphi$ (the *synchronous product* of K and T_φ) and $K \otimes T_{\neg\varphi}$ instead of T_φ and $T_{\neg\varphi}$, respectively. This coincides with [12], where the resulting monitor is called to be *predictive*.

The model is used by NuRV in different novel ways. First of all, there is the possibility that $u \notin L(K)$, because the model may be wrong, or it only captures a partial knowledge of the SUS, or due to unexpected faults. In this case we have $r_\varphi = r_{\neg\varphi} = \emptyset$ in above algorithm, and we naturally let the monitor returns a fourth verdict called *out-of-model* ($\times$). This is why we call K an *assumption*, and the two verdicts $\top/\bot$ are only conclusive under assumptions, thus renamed to $\top^{\mathrm{a}}/\bot^{\mathrm{a}}$. This extended RV approach may be called *assumption-based*. In particular, if one only cares whether the SUS always follows its model, we can use a dummy LTL property true in above procedure, so that $K \otimes T_{\neg\varphi}$ is always empty, and the monitor will output either $\top^{\mathrm{a}}$ or $\times$, indicating whether $u \in L(K)$. This application coincides with *model-based RV* [19].

Second, the above monitoring algorithm directly supports *partially observable* traces, i.e. variables appeared in the monitoring property are not (always) known in each state of the input trace. This is because the symbolic forward-image computations do not require full observability—less restrictive inputs result to

[1] Emerson-Lei algorithm [9] is used here. This corresponds to the NBA-to-NFA conversions based on SCC (strongly connected components) detections in [1], while the forward-image computations determinize NFAs into DFAs *on the fly*. Thus, NuRV provides a full implementation of [1].

coarser belief states. Partial observability becomes more useful under assumptions, as an assumption may express a relation between observable and unobservable variables of the SUS.

Third, NuRV supports *resettable* monitors, i.e. it can evaluate an LTL property at arbitrary positions of the input trace. This idea was inspired by the observation that, in r_φ and $r_{\neg\varphi}$, all variables (some are generated by the LTL translations) related to the present and the past have the same values, while all variables related to the future have opposite values. There is no easy way to distinguish these two groups of variables. However, by taking $r_\varphi \cup r_{\neg\varphi}$ we smartly get a new belief state which represents the history of the system after a run given by the input trace seen so far. If we restart the monitor algorithm at state i using this history as the new initial condition of K (also with a reduced version of initial conditions of T_φ and $T_{\neg\varphi}$), the new monitor is essentially evaluating $[\![u, i \models \varphi]\!]$ for $|u| > i$, with the underlying assumptions taken into account. This is again an orthogonal feature, but having an assumption makes resetting of the monitor more interesting as the assumption evolves to take into consideration the history of the system.

Furthermore, NuRV can synthesize the symbolic monitors into explicit-state monitor automata and then generate them into standalone monitor code in various programming languages (currently we support C, C++, Java, and Common Lisp). Besides, it is possible to dump the monitor automata into SMV modules, which can be further analyzed in nuXmv for their correctness and other properties.

The rest of this paper is organized as follows: In Sect. 2 we describe its architecture and functionalities. Some use case scenarios (as running examples) are given in Sect. 3. Section 4 shows some experimental evaluation results. Finally, we conclude the paper in Sect. 5 with some directions for future work.

2 Architecture and Functionalities

NuRV implements the Assumption-based Runtime Verification (ABRV) with partial observability and resets described in [6]. Monitoring properties are expressed in Propositional Linear Temporal Logic (LTL) [15] with both future and past temporal operators. For each input state, the monitor outputs one of four verdicts in $\mathbb{B}_4 \doteq \{\top^{\mathrm{a}}, \bot^{\mathrm{a}}, ?, \times\}$. 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 reader may refer to [6] for the formal definition of the LTL semantics and the related RV problems.

2.1 Architecture of NuRV

The internal structure of NuRV is shown in Fig. 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

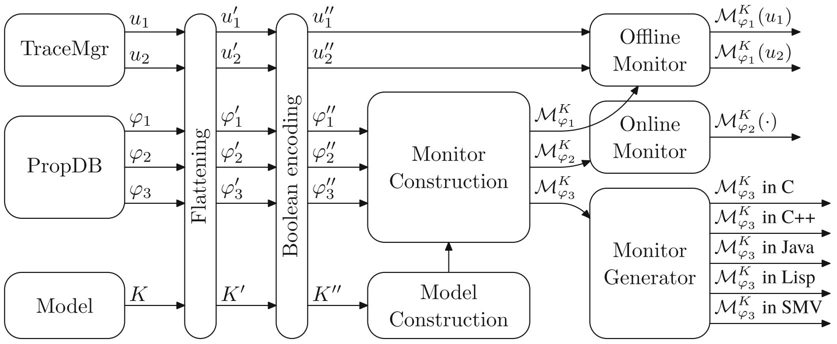
TraceMgr
u_1
u_2
PropDB
φ_1
φ_2
φ_3
Model
K
Flattening
u'_1
u'_2
φ'_1
φ'_2
φ'_3
K'
Boolean encoding
u''_1
u''_2
φ''_1
φ''_2
φ''_3
K''
Monitor Construction
Model Construction
$\mathcal{M}^K_{\varphi_1}$
$\mathcal{M}^K_{\varphi_2}$
$\mathcal{M}^K_{\varphi_3}$
Offline Monitor
Online Monitor
Monitor Generator
$\mathcal{M}^K_{\varphi_1}(u_1)$
$\mathcal{M}^K_{\varphi_1}(u_2)$
$\mathcal{M}^K_{\varphi_2}(\cdot)$
$\mathcal{M}^K_{\varphi_3}$ in C
$\mathcal{M}^K_{\varphi_3}$ in C++
$\mathcal{M}^K_{\varphi_3}$ in Java
$\mathcal{M}^K_{\varphi_3}$ in Lisp
$\mathcal{M}^K_{\varphi_3}$ in SMV

Fig. 1. The internal structure of NuRV

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. 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.

The commands `build_monitor` and `verify_property` together implemented the offline monitoring algorithm described in [6]. 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 [2] to be useful.

2.2 Structure of Explicit-State Monitors

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 three levels of explicit-state monitors:

L1 The monitor synthesis stops at all conclusive states;
L2 The monitor synthesis explores all states;
L3 The monitor synthesis explores all states and reset states.

A sample explicit-state monitor for LTL property $p\,\mathbf{U}\,q$ generated by NuRV is shown in Fig. 2. The monitor is generated under the assumption that either p or q is true in the input. The monitor 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 $\top^{\mathrm{a}}$ (`Y`). The **L1** 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 **L2** monitor contains all locations and transitions, thus it may return $\times$ even after the monitor reached conclusive verdicts. The **L3** monitor additionally contains information for the resets: in case the monitor is reset, the current location will first *jump* to the location indicated in the bracket `[]`, of current location, 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.

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.

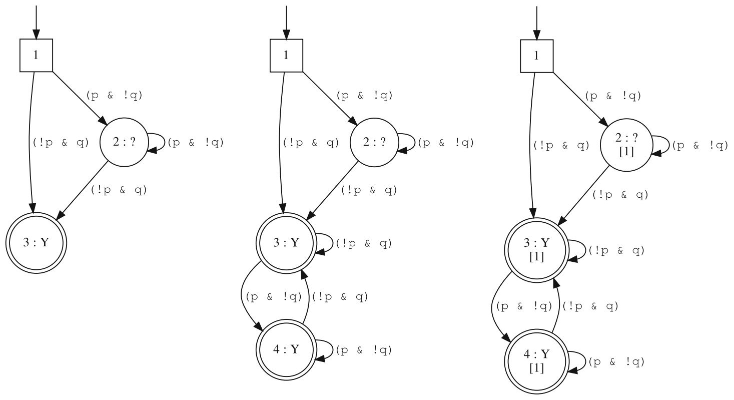

Fig. 2. Explicit-state monitors of $p\,\mathbf{U}\,q$ (assuming $p \neq q$) (L1–L3)

2.3 API of Generated Code

NuRV currently supports monitor code generation into five languages: C, C++, Java, Common Lisp and SMV. The structure of monitor code is simple yet efficient: it simply mimics the simulations of deterministic FSMs.

The monitor code generated (in C, for example) has the following 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] */);
```

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.

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. 3 does the work (also c.f. Fig. 4 for the contents of two helper files).

```
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
```

Fig. 3. The batch commands

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)

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

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

```
p
q
```

Fig. 4. `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).

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

```
p
q
```

Fig. 5. Offline monitoring in NuRV

For offline monitoring, there is no need to call `generate_monitor` in above batch command. Suppose a trace $u = pppqqq$ has been loaded (by `read_trace`), the command `verify_property` verifies the trace against the symbolic monitor of φ_0, shown in Fig. 5 (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 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"`.

4 Experimental Evaluation

We have done some comparison tests[2] between NuRV and the latest release of RV-Monitor [14]. To show the feasibility and effectiveness of RV tools, we tried to generate LTL monitors from a wide coverage of practical specifications, i.e. Dwyer's LTL patterns[3] [8]. The purpose 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.

Table 1. Eight long formulae from Dwyer's patterns

ID	Pattern	LTL
13	Trans to p occur at most twice (between 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 occur 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 (between 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)	$\mathbf{G}\,(q \rightarrow (p \rightarrow (\neg r\,\mathbf{U}\,(s \wedge \neg r \wedge \neg z \wedge \mathbf{X}\,((\neg r \wedge \neg z)\,\mathbf{U}\,t))))\mathbf{U}\,(r \vee \mathbf{G}\,(p \rightarrow (s \wedge \neg z \wedge \mathbf{X}\,(\neg z\,\mathbf{U}\,t)))))$

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 1. Also it

[2] All test data and materials for reproducing these experiments are available at https://es.fbk.eu/people/ctian/papers/rv2019/rv2019-data.tar.gz.

[3] The latest version (55 in total) is available at http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml. We call them Pattern $0, 1, \ldots, 54$ in the same order.

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 0.467 s (MacBook Pro with Intel Core i7 2.6 GHz, 4 cores) using a single core, while the 37 rvm monitors were generated in 78.619 s on the same machine using multiple cores.

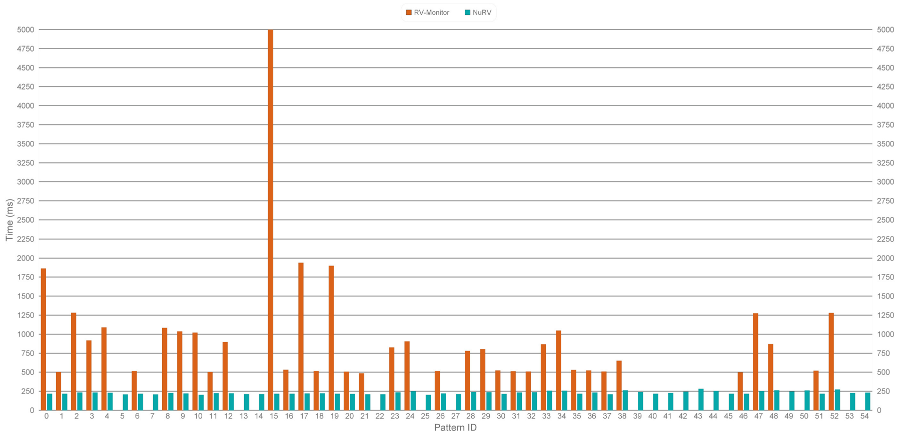

Fig. 6. Performance of generated Java monitors on 10^7 states.

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 [5], 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. 6. Our monitors (in Java) have shown a constant-like time complexity (approx. 250 ms), 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 500 ms to more than 6 s, depending on the number of resets.

5 Conclusions and Future Work

We presented 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

[4] The error message is "violation is not a supported state in this logic, ltl.".

for offline and online monitoring, and code generation of the monitors in various programming languages. The experimental evaluation on standard LTL patterns shows that NuRV is quite efficient in both generation and running time. In the future, we plan to participate in the RV competition to broaden the tool comparison and to extend the monitor specification language beyond the propositional case.

Acknowledgment. We thank the anonymous reviewers for their helpful comments.

References

1. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14–64 (2011). https://doi.org/10.1145/2000799.2000800
2. Bozzano, M., et al.: nuXmv 1.1.1 User Manual (2016). https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf
3. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 191–217. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_7
4. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
5. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
6. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, 165–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_10
7. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods Syst. Des. **10**(1), 47–71 (1997). https://doi.org/10.1023/A:1008615614281
8. 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 (1999). https://doi.org/10.1145/302405.302672
9. Allen Emerson, E., Lei, C.-L.: Temporal reasoning under generalized fairness constraints. In: Monien, B., Vidal-Naquet, G. (eds.) STACS 1986. LNCS, vol. 210, pp. 21–36. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-16078-7_62
10. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Eng. Dependable Softw. Syst. **34**, 141–175 (2013). https://doi.org/10.3233/978-1-61499-207-3-141
11. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transfer **17**(2), 143–170 (2014). https://doi.org/10.1007/s10009-014-0309-2
12. Leucker, M.: Sliding between model checking and runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 82–87. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_10
13. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004

14. Luo, Q., et al.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_24
15. 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
16. McMillan, K.L.: Symbolic Model Checking. Springer, Heidelberg (1993). https://doi.org/10.1007/978-1-4615-3190-6
17. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. **12**(2), 151–197 (2005). https://doi.org/10.1007/s10515-005-6205-y
18. Somenzi, F.: CUDD: CU Decision Diagram Package, Release 2.4.1. University of Colorado at Boulder (2005)
19. Zhao, Y., Rammig, F.: Model-based runtime verification framework. Electron. Notes Theoret. Comput. Sci. **253**(1), 179–193 (2009). https://doi.org/10.1016/j.entcs.2009.09.035