NuRV: a nuXmv Extension for Runtime Verification

(RV 2019 tool paper)

Alessandro Cimatti [1] Chun Tian [1,2] Stefano Tonetta [1]

{cimatti,ctian,tonettas}@fbk.eu

[1]Fondazione Bruno Kessler, Italy

[2]University of Trento, Italy

October 2019

Outline

1. The idea
2. The interface
3. The (generated) code
4. The tests

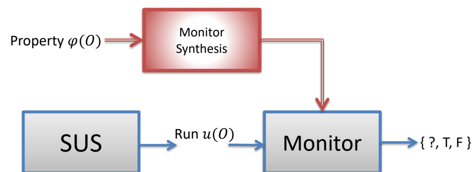
From Symbolic Model Checking to Runtime Verification

1. Symbolic Model Checking: checking a temporal property on *all infinite* traces of a given model.
2. Runtime Verification (RV): verifying if a temporal property is satisfied (or violated) on *one finite* trace of a system under scrutiny (SUS).
3. Different RV approaches: automata-based, rewriting-based, rule-based, etc.
4. nuXmv has provided all necessary infrastructures for building an *automata-based* LTL RV tool:
   - LTL (Linear-Temporal Logic) parser;
   - LTL to ω-automata translation;
   - BDD (Binary Decision Diagram) library;
   - FSM (Finite-State Machine) library, etc.

The Birth of NuRV (1)

We started from a *symbolic* implementation of LTL_3 monitor[1]:

1. An LTL property φ;
2. Two symbolic automata T_φ and $T_{\neg\varphi}$;
3. A finite input trace $u \in \Sigma^*$;
4. Two belief states $T_\varphi(u)$ and $T_{\neg\varphi}(u)$;



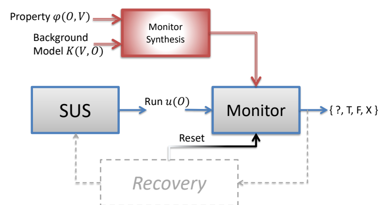
The monitor output is given by:

$T_\varphi(u)$	$T_{\neg\varphi}(u)$	$\mathcal{M}_\varphi(u)$
not $\emptyset$	not $\emptyset$	? (inconclusive)
not $\emptyset$	$\emptyset$	$\top$ (conclusively true)
$\emptyset$	not $\emptyset$	$\bot$ (conclusively false)
$\emptyset$	$\emptyset$	(impossible)

[1] A. Bauer, M. Leucker, and C. Schallhart. *Runtime Verification for LTL and TLTL.* (ACM TOSEM 2011)

The Birth of NuRV (2)

Then we found it natural to have the following extension[2]



1. An LTL property φ;
2. A model K used as the assumption;
3. Two symbolic automata $K \otimes T_\varphi$ and $K \otimes T_{\neg\varphi}$;
4. A finite input trace $u \in \Sigma^*$;
5. Two belief states $(K \otimes T_\varphi)(u)$ and $(K \otimes T_{\neg\varphi})(u)$;

The monitor output is given by:

$(K \otimes T_\varphi)(u)$	$(K \otimes T_{\neg\varphi})(u)$	$\mathcal{M}_\varphi^K(u)$
not $\emptyset$	not $\emptyset$	? (inconclusive)
not $\emptyset$	$\emptyset$	$\top^{\mathrm{a}}$ (true under assumption)
$\emptyset$	not $\emptyset$	$\bot^{\mathrm{a}}$ (false under assumption)
$\emptyset$	$\emptyset$	$\times$ (out of model)

[2] A. Cimatti, C. Tian, and S. Tonetta. *Assumption-based Runtime Verification with Partial Observability and Resets.* (RV 2019)

Other Benefits of the Symbolic Approach

- Symbolic LTL translation has an $O(n)$ complexity.[3]
- Partial Observability is supported in computing $T_\varphi(u)$ or $(K \otimes T_\varphi)(u)$:

$$\forall p \in AP.\ p \text{ is non-observable } \Rightarrow p \text{ can be any value.}$$

- The monitor can be *reset* by taking $(K \otimes T_\varphi)(u) \cup (K \otimes T_{\neg\varphi})(u)$ as the new initial belief states. Roughly speaking,

$$\mathcal{M}^K_\varphi(u) = [\![u, 0 \models \varphi]\!] \qquad \text{(old monitor)}$$
$$\mathcal{M}'^K_\varphi(u \cdot v) = [\![u \cdot v, |u| \models \varphi]\!] \qquad \text{(new monitor)}$$

[3] K. Schneider. *Improving Automata Generation for Linear Temporal Logic by Considering the Automaton Hierarchy.* (LPAR 2001)

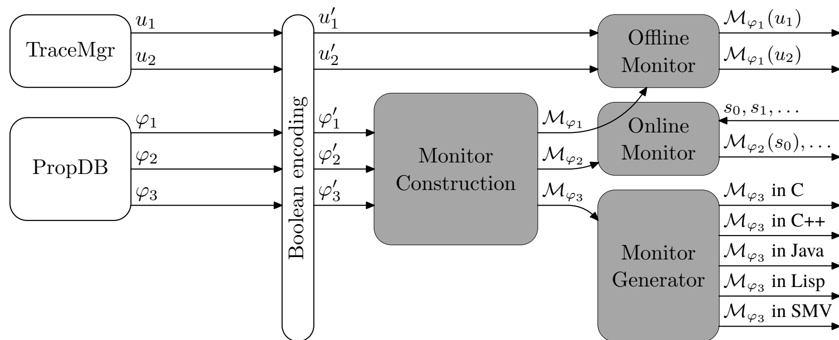
NuRV: The Features

- “Traditional” RV plus ABRV
- Offline vs. Online Monitors;
- BDD-based (deductive) vs. Code Generation (reactive)
- Code generation in various programming languages;
- Code generation in SMV, allowing formal verifications of the correctness or other properties of the monitor.

Availability

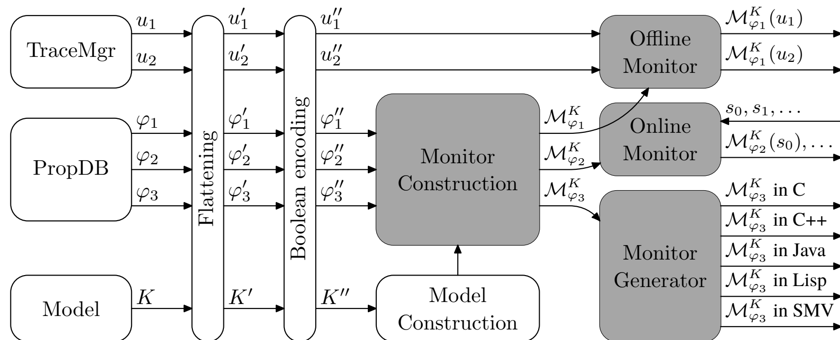
- Currently available as part of the paper artifacts.
- Free for academic use. (We are preparing for the initial release.)

NuRV: The Architecture (1)



- nuXmv components: TraceMgr, PropDB, Boolean encoding;
- NuRV components: Monitor Construction, Offline Monitor, Online Monitor, Monitor Generator.

NuRV: The Architecture (2)



- NUXMV components: TraceMgr, PropDB, Boolean encoding, Model, Flattening, Model Construction;
- NuRV components: Monitor Construction, Offline Monitor, Online Monitor, Monitor Generator.

Outline

1 The idea

2 The interface

3 The (generated) code

4 The tests

NuRV = nuXmv Extended with RV commands

New interactive commands added by NuRV:

1. `build_monitor`: building the symbolic monitor for an LTL property;
2. `verify_property`: verify a loaded trace in the symbolic monitor;
3. `heartbeat`: verify one input state (online monitoring);
4. `generate_monitor`: generate standalone monitor code.

Typical Use Scenarios

- Online Monitoring: `build_monitor` + `heartbeat`;
- Offine Monitoring: `build_monitor` + `verify_property`;
- Monitor Generation: `build_monitor` + `generate_monitor`.

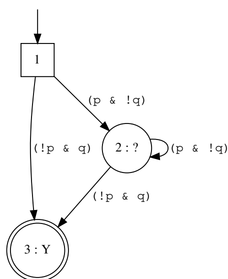
Structure of Explicit-State Monitors (1) - $p \, \mathrm{U} \, q$

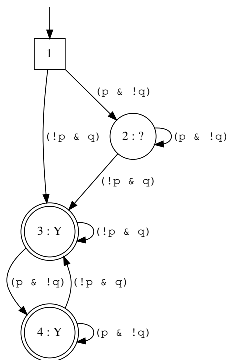
Monitor Levels for Optimization Purposes:

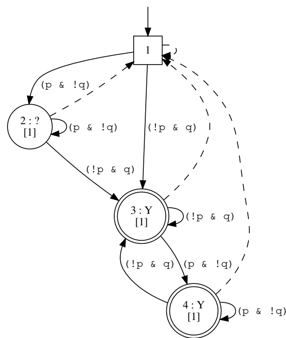
L1 The monitor synthesis stops at all conclusive states;

L2 The monitor synthesis explores all states;

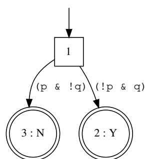
L3 The monitor synthesis explores all states and reset states.







Structure of Explicit-State Monitors (2) - The “Iceberg” of Y $p \vee q$



API of Generated Code

C header of generated monitor functions

```
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] */);
```

Two encodings of input state

1. *Static* partial observability: `state` denotes a full assignment of the observables, encoded in binary bits: 0 for *false* ($\bot$), 1 for *true* ($\top$);
   e.g. $p \wedge q = (11)_2$, $p \wedge \neg q = (01)_2$.
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* ($\bot$).
   e.g. $p \wedge q = (11)_3$, $p \wedge \neg q = (21)_2$, $p = (01)_3$, $\top = (00)_3$.

Use Case Scenario (Offline Monitoring)

disjoint.smv:

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

default.ord:

```
p
q
```

monitor.cmd:

```
set input_file "disjoint.smv"
set input_order_file "default.ord"
go
add_property -l -p "p U q"
build_monitor -n 0
read_trace "trace.xml"
verify_property -n 0 1
quit
```

trace.xml ($u = pppqqq$):

```
<?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>
            ...
    <loops> </loops>
</counter-example>
```

Command line:

```
> NuRV -source monitor.cmd
1, unknown
2, unknown
3, unknown
4, true
5, true
6, true
```

Use Case Scenario (Code Generation)

disjoint.smv:

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

default.ord:

```
p
q
```

synthesis.cmd:

```
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
```

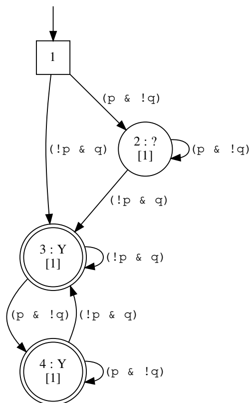
Command line

```
NuRV -source synthesis.cmd
```

Outline

1. The idea
2. The interface
3. The (generated) code
4. The tests

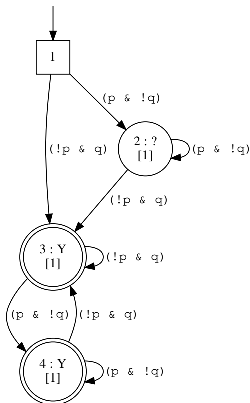
Structure of Generated Code in C (1)




```
int /* [out] (0 = unknown, 1 = true, 2 = false, 3 = out-of-model) */
  M0 (long state,        /* [in] */
      int  reset,        /* [in] (0 = none, 1 = hard, 2 = soft) */
      int *current_loc) /* [in/out] */
{
  /* local variables */
  long input = 0L, base = 1L;
  int  output;

  /* handle hard resets */
  if (1 == reset) *current_loc = 1; /* initial location */
  else if (2 == reset) {
    switch (*current_loc) {
    case 4: *current_loc = 1; break;
    case 3: *current_loc = 1; break;
    case 2: *current_loc = 1; break;
    case 1: *current_loc = 1; break;
    default: ;
    }
  }
  input = state;
M0_start: ...
M0_loc_4: ...
M0_loc_3: ...
M0_loc_2: ...
M0_loc_1: ...
  /* end of function */
M0_exit:
  return output;
}
```

Structure of Generated Code in C (2)




```
M0_start:
  /* go to current location */
  switch (*current_loc) {
  case 4:
    output = 1; goto M0_loc_4;  /* true */
  case 3:
    output = 1; goto M0_loc_3;  /* true */
  case 2:
    output = 0; goto M0_loc_2;  /* unknown */
  case 1:
    output = 0; goto M0_loc_1;  /* unknown */
  default:
    output = -1; goto M0_exit;  /* invalid location */
  }
M0_loc_4:
  switch (input) {
  case 2L: /* (!p & q) */
    output = 1; /* true */
    *current_loc = 3; break;
  case 1L: /* (p & !q) */
    output = 1; /* true */
    *current_loc = 4; break;
  default:
    output = 3; /* out-of-model */
  }
  goto M0_exit;
M0_loc_3: ...
M0_loc_2: ...
M0_loc_1: ...
M0_exit: return output;
```

Structure of Generated Code in C++

```
namespace rvsynth {
  class monitor {
  private:
    int current_loc;
  public:
    monitor () { current_loc = 1; }
    int run (long state, int reset);
  };
}

#include "monitor.hpp"

namespace rvsynth {

int /* out (0 = unknown, 1 = true, 2 = false, 3 = error) */
  monitor::run
    (long state,
     int  reset) /* in (0 = none, 1 = initial, 2 = soft) */
{
  /* local variables */
  long input = 0L, base = 1L;
  int  output;

  /* handle hard resets */
  if (1 == reset) current_loc = 1; /* initial location */
  else if (2 == reset) {
    switch (current_loc) {
    case 4: current_loc = 1; break;
    case 3: current_loc = 1; break;
    case 2: current_loc = 1; break;
```

Structure of Generated Code in Java

```
package rvsynth;

public class monitor {
  int current_loc = 1;

  public int /* out (0 = unknown, 1 = true, 2 = false, 3 = error) */
    run (long state,
         int reset) /* in (0 = none, 1 = hard, 2 = soft) */
  {
    /* local variables */
    long input = 0L, base = 1L;

    /* handle hard resets */
    if (1 == reset) current_loc = 1;
    else if (2 == reset) {
      switch (current_loc) {
      case 4: current_loc = 1; break;
      case 3: current_loc = 1; break;
      case 2: current_loc = 1; break;
      case 1: current_loc = 1; break;
      default: ;
      }
    }

    input = state;

    return monitor_start(input);
  }

  private int monitor_start(long input)
```

Structure of Generated Code in Common Lisp

```
(in-package :rvsynth)

(defclass monitor ()
  ((current-loc :type fixnum :accessor current-loc :initform 1)))

(defmethod run ((m monitor) state reset &aux (input 0) (base 1))
  (declare (ignorable base)
           (type symbol reset) (type fixnum state input))
  (if (eq :hard reset)
      (setf (current-loc m) 1)
    (when (eq :soft reset)
      (case (current-loc m)
        (4 (setf (current-loc m) 1))
        (3 (setf (current-loc m) 1))
        (2 (setf (current-loc m) 1))
        (1 (setf (current-loc m) 1))
        (t nil))))

  (setq input state)

  (prog (output)
    (case (current-loc m)
     (4
      (setq output :true)
      (go monitor-loc-4))
     (3
      (setq output :true)
      (go monitor-loc-3))
     (2
      (setq output :unknown)
```

Structure of Monitor Code in SMV

```
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;
```

Outline

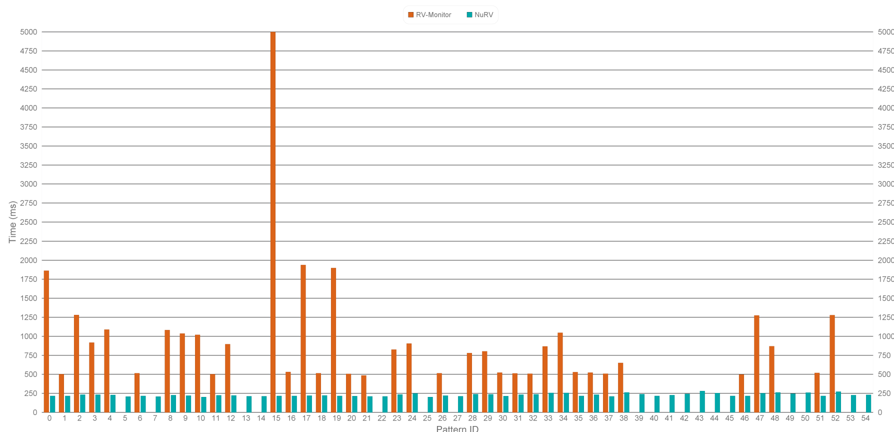
1. The idea
2. The interface
3. The (generated) code
4. The tests

Tests on Dwyer's LTL patterns

ID	Pattern	LTL
13	Trans to p occ. at most twice (betw. q and r)	$\mathsf{G}\,((q \wedge \mathsf{F}\,r) \rightarrow ((\neg p \wedge \neg r)\,\mathsf{U}\,(r \vee ((p \wedge \neg r)\,\mathsf{U}\,(r \vee ((\neg p \wedge \neg r)\,\mathsf{U}\,(r \vee ((p \wedge \neg r)\,\mathsf{U}\,(r \vee (\neg p\,\mathsf{U}\,r)))))))))$
14	Trans to p occ. at most twice (after q until r)	$\mathsf{G}\,(q \rightarrow ((\neg p \wedge \neg r)\,\mathsf{U}\,(r \vee ((p \wedge \neg r)\,\mathsf{U}\,(r \vee ((\neg p \wedge \neg r)\,\mathsf{U}\,(r \vee ((p \wedge \neg r)\,\mathsf{U}\,(r \vee (\neg p\,\mathsf{W}\,r) \vee \mathsf{G}\,p))))))))$
39	p precedes s, t (after q until r)	$\mathsf{G}\,(q \rightarrow (\neg(s \wedge (\neg r) \wedge \mathsf{X}\,(\neg r\,\mathsf{U}\,(t \wedge \neg r)))\,\mathsf{U}\,(r \vee p) \vee \mathsf{G}\,(\neg(s \wedge \mathsf{X}\,\mathsf{F}\,t))))$
43	p responds to s, t (between q and r)	$\mathsf{G}\,((q \wedge \mathsf{F}\,r) \rightarrow (s \wedge \mathsf{X}\,(\neg r\,\mathsf{U}\,t) \rightarrow \mathsf{X}\,(\neg r\,\mathsf{U}\,(t \wedge \mathsf{F}\,p)))\,\mathsf{U}\,r)$
44	p responds to s, t (after q until r)	$\mathsf{G}\,(q \rightarrow (s \wedge \mathsf{X}\,(\neg r\,\mathsf{U}\,t) \rightarrow \mathsf{X}\,(\neg r\,\mathsf{U}\,(t \wedge \mathsf{F}\,p)))\,\mathsf{U}\,(r \vee \mathsf{G}\,(s \wedge \mathsf{X}\,(\neg r\,\mathsf{U}\,t) \rightarrow \mathsf{X}\,(\neg r\,\mathsf{U}\,(t \wedge \mathsf{F}\,p)))))$
49	s, t responds to p (after q until r)	$\mathsf{G}\,(q \rightarrow (p \rightarrow (\neg r\,\mathsf{U}\,(s \wedge \neg r \wedge \mathsf{X}\,(\neg r\,\mathsf{U}\,t))))\,\mathsf{U}\,(r \vee \mathsf{G}\,(p \rightarrow (s \wedge \mathsf{X}\,\mathsf{F}\,t))))$
53	s, t without z responds to p (betw. q and r)	$\mathsf{G}\,((q \wedge \mathsf{F}\,r) \rightarrow (p \rightarrow (\neg r\,\mathsf{U}\,(s \wedge \neg r \wedge \neg z \wedge \mathsf{X}\,((\neg r \wedge \neg z)\,\mathsf{U}\,t))))\,\mathsf{U}\,r)$
54	s, t without z responds to p (after q until r)	$\mathsf{G}\,(q \rightarrow (p \rightarrow (\neg r\,\mathsf{U}\,(s \wedge \neg r \wedge \neg z \wedge \mathsf{X}\,((\neg r \wedge \neg z)\,\mathsf{U}\,t))))\,\mathsf{U}\,(r \vee \mathsf{G}\,(p \rightarrow (s \wedge \neg z \wedge \mathsf{X}\,(\neg z\,\mathsf{U}\,t)))))$

```
MODULE main
VAR     p : boolean; q : boolean; r : boolean;
        s : boolean; t : boolean; z : boolean;
INVAR  (p & !q & !r & !s & !t & !z) |
       (!p &  q & !r & !s & !t & !z) |
       (!p & !q &  r & !s & !t & !z) |
       (!p & !q & !r &  s & !t & !z) |
       (!p & !q & !r & !s &  t & !z) |
       (!p & !q & !r & !s & !t &  z)
```

Comparison with RV-Monitor



	NuRV	**RV-Monitor**
Code generation (Java)	0.467s (55 monitors)	78.619s (37 monitors)
Offline monitor (10^7 states)	250ms	from 500ms to 6s

Conclusions

1. NuRV is a nuXmv extension for Runtime Verification.
2. It supports assumption-based RV for propositional LTL with both future and past operators, with the supports of partial observability and resets.
3. It has functionalities for offline and online monitoring, and code generation of the monitors in various programming languages. (Extending to new languages is *easy* due to the automata-based approach.)
4. NuRV is quite efficient in both monitor generation and monitoring.