NuRV 2.0.0 User Manual

Alessandro Cimatti,
Chun Tian,
Stefano Tonetta

Fondazione Bruno Kessler
Via Sommarive 18, 38123 Povo (TN), Italy

Email: nurv@fbk.eu

Contents

Chapter 1

Introduction

NuRV [CTT19b] is an implementation of Assumption-Based Runtime Verification (ABRV) [CTT19a].

The input of ABRV is a model K as assumptions on the behavior of the system under scrutiny (SUS), plus one or more monitoring properties φ. The output of ABRV is a monitor $\mathcal{M}_\varphi^K$, which further takes a finite trace u from the SUS and outputs verdicts for each input state of u. The workflow of ABRV is shown in Fig. 1.1. NuRV currently supports monitoring properties expressed in extensions of Linear Temporal Logic (LTL).

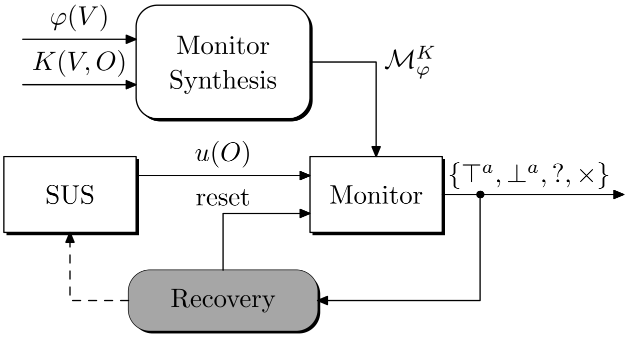


Figure 1.1: Assumption-Based Runtime Verification

The detailed functionalities of NuRV can be described by the classification of RV tools according to the taxonomy proposed in [FKRT18], as shown in Table 1.1 and 1.2.

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

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

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

The commands `build_monitor` and `verify_property` together implemented the offline monitoring algorithm described in [CTT19a]. 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 [BCC+19] to be useful.

- `read_model`: reads a SMV file into NUXMV;
- `flatten_hierarchy`: flattens the hierarchy of modules;
- `encode_variables`: builds the BDD variables necessary to compile the model into a BDD;

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

Table 1.1: Classification of NuRV according to the taxonomy [FKRT18]

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

Table 1.2: Distinguished features of three NuRV modes

- `build_flat_model`: compiles the flattened hierarchy into a Scalar FSM;
- `build_model`: compiles the flattened hierarchy into a BDD;
- `add_property`: adds an (LTL) property to the list of properties;
- `read_trace`: loads a previously saved trace (in XML format).

In model checking scenario, the commands `read_model`, `flatten_hierarchy`, `encode_variables`, `build_flat_model` and `build_model` essentially initialize the system model for further verification. If all these commands take default parameters (while the input model is given in other ways, e.g. by environment variable or command line argument), the user could instead use a single command go, which is equivalent to the command sequence 4–8. The command `add_property` can be used to add new LTL properties, with each of them a monitor can be built and associated by calling the command `build_monitor`. (An alternative way of adding properties is to put them into SMV files as LTL specifications, i.e. `LTLSPEC` [BCC^{+}19].)

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

1.1 New features in NuRV 2.0

1. Scalar API for code generation in C with support of SMV array variables (limited to one dimension);
2. Code generation in FMU (FMI 2.0 and 3.0 standards).

1.2 New features in NuRV 1.9

1. Scalar API for code generation in Python (beside in C).
2. Symbolic finite- or infinite-state monitors calling NuRV as a shared/dynamic library.

1.3 Input language

NuRV supports the same input language (SMV language and LTL specifications) inherited from NUXMV (and then NuSMV). Such "languages" include:

- The SMV language for defining RV assumptions, either finite-state or infinite-state;
- The LTL specifications used for LTL-based Runtime Verification;
- The XML-based trace format (used by NuSMV for expressing counterexamples) for offline monitoring;
- The input order file for monitor code generation.

However, not all language features from nuXmv/NuSMV are confirmed working or being tested in NuRV.

Linear Temporal Logic (LTL) can be used to specify complex ordering relationship between events. In Appendix A, Dwyer's LTL patterns are given in NuSMV's LTL syntax.

1.4 Running NuRV

NuRV inherites the running modes of NuSMV and nuXmv (see [BCC+19] for more details.)

The main interaction mode of NuRV is through an interactive shell. In this mode NuRV enters a read-eval-print loop. The user can activate the various NUSMV computation steps as system commands with different options. These steps can therefore be invoked separately, possibly undone or repeated under different modalities.

The interactive shell of NuRV is activated from the system prompt as follows ('`NuRV>`' is the default NuRV shell prompt): [1]

```
$ NuRV -int <RET>
NuRV>
```

When the `-int` option is not specified, NuRVruns as a batch program performing (some of) the steps in a fixed sequence.

```
$ NuRV [command line options] smv-file <RET>
```

The model described in *smv-file* is processed as RV assumptions. Even without assumption, this file serves as the alphabet in which the monitoring properties are expressed. Then, if *smv-file* contains LTL formulas (given in `LTLSPEC` sections), these properties can be directly referenced by RV commands.

[1] To disable the banner, use additionally `-quiet` when calling NuRV.

Chapter 2

Offline Monitoring

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

For RV on finite-state systems, it is also possible to generate monitor code into various programming languages (see Chapter 4 for more details). Then it is up to the user to use such generated monitors in a online or offline manner. However, essentially the generated monitor are online monitors taking input states one by one.

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

Figure 2.1: The SMV file `disjoint.smv` for $p\ \mathbf{U}\ q$ (assuming $p \neq q$)

The SMV model file including the LTL property is given in Fig. 2.1. Suppose we wanted to monitor a trace $u = \{p, \neg q\}\{p, \neg q\}\{p, \neg q\}\{\neg p, q\}\{\neg p, q\}\{\neg p, q\}$, that is, for the first 3 states p is true (and q is false), then q becomes true (and p becomes false). The following XML file (saved as `trace.xml`, for example) should be prepared (or generated by the user using other programs) as the trace:

```
<?xml version="1.0" encoding="UTF-8"?>
<counter-example type="0" id="1" desc="LTL Counterexample">
    <node>
        <state id="1">
            <value variable="p">TRUE</value>
            <value variable="q">FALSE</value>
        </state>
    </node>
    <node>
        <state id="2">
            <value variable="p">TRUE</value>
            <value variable="q">FALSE</value>
        </state>
    </node>
    <node>
        <state id="3">
            <value variable="p">TRUE</value>
            <value variable="q">FALSE</value>
        </state>
    </node>
    <node>
        <state id="4">
            <value variable="p">FALSE</value>
            <value variable="q">TRUE</value>
        </state>
    </node>
    <node>
        <state id="5">
            <value variable="p">FALSE</value>
            <value variable="q">TRUE</value>
        </state>
    </node>
    <node>
        <state id="6">
            <value variable="p">FALSE</value>
            <value variable="q">TRUE</value>
        </state>
    </node>
</counter-example>
```

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

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

Now NuRV can be called in this way:

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

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

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

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

Chapter 3

Online Monitoring

In online monitoring, the synthesized runtime monitor takes a single input state and immediately output a verdict corresponding to the input state. The related NuRV command is `heartbeat`. (For the "real" online monitor which can be called from remote, see Section 5.) The monitor maintains it internal states (aka belief states) for handling future inputs. The monitor can be softly or hardly reset when taking an input state.

For RV on finite-state systems, it is also possible to generate monitor code into various programming languages (see Chapter 4 for more details). Then it is up to the user to use such generated monitors in a online or offline manner. However, essentially the generated monitor are online monitors taking input states one by one.

As an example of online monitoring (using the same setting given in Fig. 2.1), the following batch command (saved as `online.cmd`) can be used for demo purposes:

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

If one calls NuRV in the following way:

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

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

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

Currently the online monitoring support of NuRV can only be used for debugging or manual testing purposes (of the monitors), because interactively calling `heartbeat` commands is not very useful when NuRV is actually used as online monitors. In future versions NuRV may provide a network-based monitor server so that external callers may call `heartbeat` commands, among other commands, remotely.

Chapter 4

Code Generation

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

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

The code generation facility gets major updates in NuRV 1.8.0 with the following changes and new features:

1. Extended API with support of arbitrary number of Boolean variables/bits.
2. Two new target languages: Prolog and LLVM IR.
3. Structure-based API supporting scalar variables (for C target language only).

The code generation facility gets major updates in NuRV 1.9.0 with the following changes and new features:

1. Structure-based API supporting scalar variables for Python target language.
2. Generating "symbolic" monitors calling NuRV itself as a shared/dynamic library (currently on non-Windows platforms only). (See the `-S` and `-x` option of `generate_monitor`).
3. The options `-C` and `-c` have been moved from `generate_monitor` to `build_monitor` to better support observable expressions.

4.1 Code generation in DOT language; monitor levels

The DOT language of open-source software Graphiviz can be used to describe explicit-state monitors synthezied by NuRV.

The code generation of monitors in DOT language (by using command-line option `-L "c"` when calling `generate_monitor`) can be used for paper representations of runtime monitors, also for understanding the structure of generated monitor code in other languages.

For the efficiency of generated monitors in different scenarios, NuRV supports different levels of explicit-state monitors, controlled by the command-line option `-L` of `generate_monitor`:

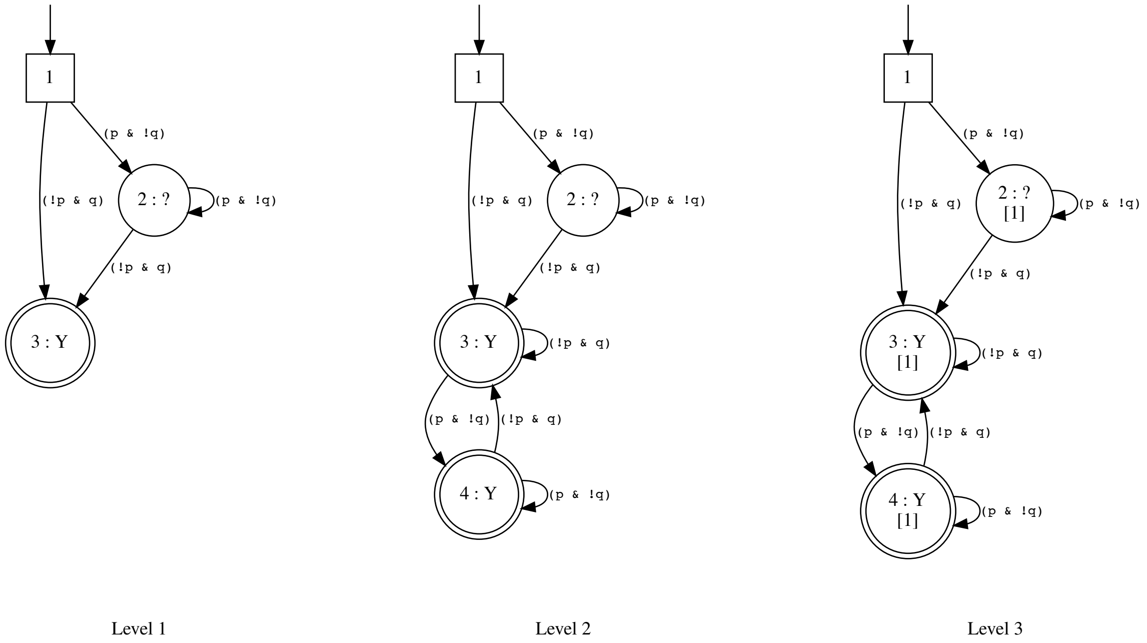
- Level 1: the monitor synthesis stops at all conclusive states;
- Level 2: the monitor synthesis explores all states;
- Level 3: the monitor synthesis explores all states and reset states;
- Level 4: the monitor always resets before taking next inputs.

Level 1 monitors are usually very small, but still fully functional when the monitor is never reset and the underlying assumption is never violated. (In this case the monitor is monotonic and never outputs ×.) Also, comparison tests to other RV tools are usually done by level 1 monitors, as they give exactly the same outputs in comparion with LTL$_3$ monitors. Level 2 monitors are full automata whose language is exactly the same as the language of the monitor specification (filtered or restricted by the assumption). Level 3 monitors are full-featured monitors supporting arbitrary resets, supporting all features of the ABRV framework. In addition, level 4 monitors consider the special use scenario in which the monitor is repeatedly reset before each new input state. This scenario is particularly useful for monitoring ptLTL.

In the example below, the first three levels of monitors for LTL property $p\,\mathbf{U}\,q$ (assuming $p \neq q$) are generated. We use the same SMV file as in Fig. 2.1. To generate three DOT files `pUq_L1.dot`, `pUq_L2.dot` and `pUq_L3.dot` corresponding to the monitors of $p\,\mathbf{U}\,q$ (assuming $p \neq q$) at different levels, the following batch command can be used:

```
go
build_monitor -n 0
generate_monitor -n 0 -L "dot" -l 1 -o "pUq_L1"
generate_monitor -n 0 -L "dot" -l 2 -o "pUq_L2"
generate_monitor -n 0 -L "dot" -l 3 -o "pUq_L3"
quit
```

The generated monitors, when converting to PDF by the `dot` command (provided by Graphviz) are shown in Fig. 4.1. The monitors generated by NuRV can be understood as finite-state machines with locations numbered from 1 (the initial location). By taking input states (e.g. `p & !q`, encoded as integers), the current location changes. Each location holds at most three data: (i) the ID of current location, (ii) the monitor verdict, and (iii) the ID of location for (soft) resets. (Only level 3 monitors have reset data.)



Level 1

Level 2

Level 3

Figure 4.1: LTL monitors of $p\,\mathbf{U}\,q$ (level 1–3), assuming $p \neq q$

4.2 Code generation in C

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

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

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

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

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

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

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

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

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

1. state (or states): an encoded long integer (or an array of long integers) representing the current input state of the trace,

2. width: in the new signature, this is the length of states supplied by the user.

3. reset, an integer representing the possible reset signal, and

4. current_loc: a pointer of integer holding the internal state of the monitor.

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

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

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

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

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

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

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

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

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

2. How many long integers are necessary?

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

4.2.1 Structural interface of generated monitors in C

This is a major new feature in NuRV 1.8.0. Besides Boolean variables, the SMV language also supports finite-domain integers and fixed-length machine words, which is called *scalar variables*. In the BDD-based symbolic monitoring, scalar variables can be freely used in the model and monitoring properties, because internally they are encoded as Boolean bits. (In the SMT-based monitoring, scalar variables are directly handled by the underlying SMT solvers.)

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

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

```
/* input data */
typedef struct {
  short p; /* 0: false, other: true */
  short q; /* 0: false, other: true */
  short r; /* 0: false, other: true */
  short s; /* 0: false, other: true */
} monitor_input_t;

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

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

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

Remark 4.1. (The other structure monitor_mask_t is intended to support partial observability of values in the structure monitor_input_t, i.e. a input value is considered observable whenever the corresponding slot in monitor_mask_t is non-zero. Currently this is not implemented yet, and user can just supply NULL as the value of masks.)

In another case, the model contains the following variables:

```
VAR
```

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

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

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

/* input data */
typedef struct {
  int i;
  int k;
  int l;
} m1_input_t;

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

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

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

Support of SMV array variables Starting from NuRV 2.0.0, SMV array variables (limited to one dimension) are also supported. For instance, with the following variables definitions in SMV model: (Note that SMV supports integer-ranged array indexes whose elements can be negative)

```
MODULE main
VAR
    p : array 0..10 of boolean;
    q : array 10..11 of {OK, y, z};
    r : array -10..0 of unsigned word[16];
```

NuRV now generates the following monitor input C structure:

```
typedef struct {
  int p[11];
  int q[12];
  uint16_t r_padding_[10];
  uint16_t r[1];
} monitor_input_t;
```

where the additional "padding" slot makes C expressions like input->q[-10] (assuming input is a pointer allocated from the above structure definition) possible and valid, without causing any memory violations, and then the lower level C code generated by NuRV will correctly transfer the input values for driving the underlying monitor code.

4.2.2 Observable expressions in generated monitor

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

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

To generate monitor with *observable expressions*, a new parameter `-C` is added into the command `generate_monitor`. `-C` takes a file name, in which each line is an observable expression. See Section 7 for more details.

4.3 Code generation in C++

Since NuRV 1.8.0, the monitor code generated in C++ (by using command-line option `-L "cpp"` when calling `generate_monitor`) has class headers like the following:

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

private:
  ...
}
```

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

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

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

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

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

4.4 Code generation in Java

Since NuRV 1.8.0, the monitor code generated in Java (by using command-line option `-L "java"` when calling `generate_monitor`) has the following structure:

```
package eu.fbk.rvsynth;

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

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

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

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

```
package eu.fbk.rvsynth;

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

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

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

4.5 Code generation in Common Lisp

Since NuRV 1.8.0. the monitor code generated in Common Lisp (by using command-line option -L "lisp" when calling generate_monitor) has the following structure:

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

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

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

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

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

4.6 Code generation in Prolog

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

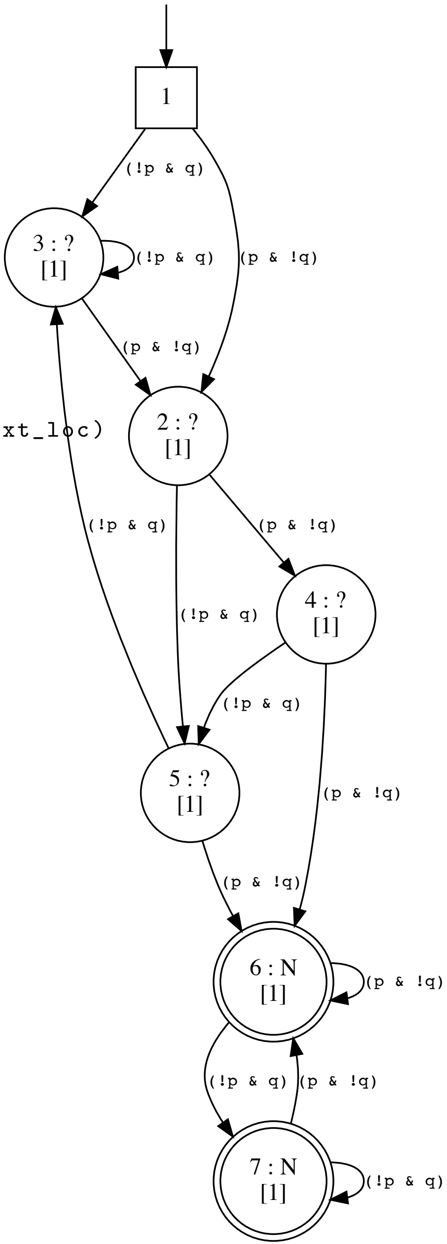
```
:- module(rvsynth, [monitor/5, monitor_old/5]).
```

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.7 Code generation in LLVM IR

Starting from NuRV 1.8.0. the monitor code can be generated in LLVM IR (by using command-line option `-L "llvm"` when calling `generate_monitor`). This is done by outputting LLVM IR code in plain text without linking any library from LLVM. The resulting LLVM IR code does not call any external function.

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

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

```
    i32        %width,       ; [in]
    i32        %reset,       ; [in]
    i32*       %current_loc) ; [in/out]
{
    ...
}
```

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

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

4.8 Code generation in SMV

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

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

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

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

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

  _rloc := _reset ? case
    (_loc = 4) : 1; (_loc = 3) : 1; (_loc = 2) : 1; (_loc = 1) : 1; TRUE : 0;
  esac : _loc;

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

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

4.9 Code generation in FMU (FMI 2.0 and 3.0)

Starting from NuRV 2.0, monitors (currently finite-state only) can be generated as Functional Mock-up Units (FMUs). The interface follows The Functional Mock-up Interface standards[1], version 2.0 or 3.0.

To generate monitor FMUs, user should use either `FMU` (FMI 2.0) or `FMU3` (FMI 3.0) as the target language (the `-L` option) when calling `generate_monitor`. The output (given by `-o`) is a directory holding all necessary source code (XML and C code).

To obtain binary FMUs, user needs to go to the generated FMU directory and call `make` with suitable parameters for building the platform-specific FMUs (see the generated `Makefile` for more details.)

Remark 4.2. This functionality is not tested on Windows. The generated `Makefile` only considers Linux and macOS, but the generated C source code can support more platforms.

[1] `https://fmi-standard.org`

Chapter 5

Monitor Server

NuRV supports network-based online monitoring (i.e. *monitor server*) since version 1.7.0.

5.1 Introduction

Starting from version 1.7.0, NuRV supports a network-based "monitor server" mode. After executing a command, NuRV stops the interactive shell and starts to listen on network so that user code can remotely execute the `heartbeat` command (not directly but in an equivalent way) for online monitoring. (In some senses this is the "real" online monitoring, although it is in theory possible to run NuRV as an interactive session inside another program.)

NuRV supports multiple clients connecting to multiple servers. Here each "monitor server" is a NuRV running process in which multiple LTL properties are added with their corresponding runtime monitors built by `build_monitor` command. Note that a single NuRV process can indeed provide multiple monitors corresponding to different LTL properties, however all these monitors must share the same ground model as the RV assumptions. Thus the ability of letting a single monitor client to connect to multiple NuRV processes (without extra programming overheads) is necessary in certain applications.[1]

The following further descriptions may settle the potential concern on the scalability:

- Multiple NuRV processes (monitor server) can run together, on same or different machines, without any concern on the potential conflict of listening ports. In another words, multiple NuRV processes can start up in any order (even after the clients, as long as they are not queried yet). However, each NuRV processes must have a unique "instance name". (This should be a common need, as otherwise there's no way to know who has what properties being monitored.)
- All monitor servers register their names in a small central server daemon (the name service) provided by third-party software (with multiple choices), just like Internet DNS (domain name system).
- One or multiple monitor clients can connect to these monitor servers by only their instance names. Client programming has the same complexity for one server or multiple servers, because clients only need to know the location (address and port, etc.) of the central name server. (We provide sample client code in C, C++, Java, Lisp and Python.)

Technically speaking, the monitor server currently supported by NuRV is based on a *synchronous, push model*: the monitor clients are responsible to activately send observations to the monitor server, and such sending operations must wait unti the server finished the calculations, i.e. the execution of underlying monitoring algorithms.[2]

[1] By running multiple NuRV processes one can also benefit from multiple CPUs in the host machine, as NuRV does not support multithreading in its core monitoring algorithms.

[2] In the future, NuRV may additionally support the *asynchronous, push model*. This is particularily important when users want to send a single observation to get the verdicts of many or all monitors, since checking the monitoring results for many properties may take a long time while the client cannot wait.

5.2 Additional software dependencies

The monitor server functionality is provided by another execution `NuRV_orbit`, which currently is only available on Mac OS X (only for `x86_64`), Linux and Solaris. `NuRV_orbit` has all functionalities of `NuRV`, plus the monitor server support. The reason we provide two executions is that:

1. The additional software (by dynamically linking) required by `NuRV_orbit` may not be available in some versions of operating systems of end users;
2. In the future, we may provide more execution variants linking different libraries.

On Linux (e.g. Debian and Ubuntu), `NuRV_orbit` requires a package called `orbit2`. On Mac OS X, it requires the same package from MacPorts [3].

To use the monitor server, third-party naming service is required. This service is provided by many packages. The following 3 software are confirmed working:

- Java SE (JDK) 1.8 (execution name: `tnameserv`);
- Linux package `orbit2-nameserver` (execution name: `orbit-name-server-2`);
- Linux/Mac[4] package `omniorb` (execution name: `omniNames`).

5.3 A tutorial of monitor server

Let's continue the online monitoring example in Chapter 3 (with the SMV file `disjoint.smv` found in Chapter 2) and turn NuRV into a monitor server. [5]

1. Open a Terminal window and start the third-party name service from JDK 1.8: (JDK prior to 1.8 also provides it.)

   ```
   $ tnameserv
   ```

   The above command will print a long string starting with "IOR:" (which stands for an *Interoperable Object Reference*) and listen on a TCP port.

2. Open another Terminal window and start NuRV (the variant with monitor server support) in the same directory with `disjoint.smv` [6] (otherwise a new LTL property can be added by the command `add_property`):

   ```
   $ NuRV_orbit -int disjoint.smv
   ```

   Keep in mind that an LTL property $p\,\mathbf{U}\,q$ has been defined already and stored at index 0 of property manager. Following Chapter 3, the following NuRV commands builds the internal monitor for it:

   ```
   NuRV> go
   NuRV> build_monitor -n 0
   ```

3. Start the monitor server by executing the command `monitor_server` with the IOR string returned by `tnameserv` in the first step: (you must substitute "`IOR:...`" with the actual (long) string returned by `tnameserv`)

   ```
   NuRV> monitor_server -N IOR:...
   ```

[3] `https://www.macports.org`

[4] On Mac OS X, the package is provided by MacPorts.

[5] This tutorial is tested on Mac OS X 10.15. The same steps should work on all supported versions of NuRV on Mac OS X and Linux. Currently NuRV does not support monitor server on MS Windows.

[6] This SMV file is also in the directory `client/c` of the shipped common files.

The above command should return something like “Binding service reference at name service against id: NuRV/Monitor/Service”. Note that this monitor server is uniquely identified by “NuRV/Monitor/Service”.[7]

(Note: use Ctrl+C to terminate the monitor server and return to the shell prompt.)

4. Make sure Homebrew[8] or MacPorts packages `orbit` and `pkg-config` are installed in your Mac system. Go to directory `client/c` and execute `make` to build the C-based monitor client:

```
$ make
```

If everything goes correctly, at the end there will be an execution `monitor_client` being built. It hardcoded a monitor client which sends the trace *pppqqq* (3 times p and 3 times q, just like the sample in Chapter 3) to the monitor server. Again, this client program must know the location of the name server (by knowing the IOR string): (once again, you must substitute “`IOR:...`” with the actual (long) string returned by `tnameserv` in the first step)

```
$ ./monitor_client -ORBInitRef NameService=IOR:...
```

If everything goes fine, you should see the following outputs by the above monitor client. The first line is a reminder of its usage, and next line says that it has found the monitor server identified by the name “NuRV/Monitor/Service”. And the rest is the monitoring outputs (3 times “unknown” and 3 times “true”):

```
*** Usage: ./monitor-client -ORBInitRef NameService=IOR:...

Resolving service reference from name-service with id "NuRV/Monitor/Service"
unknown
unknown
unknown
true
true
true
```

The business logic of the monitor client can be found near the end of the C code file `monitor-client.c` (see Section 5.5.2 for more details of the sample code and other files in the same directory.)

5.4 The “simple” monitor interface (IDL definition)

It can be undersood that, the monitor server running inside NuRV is an instance (or object) of a class, which has the following interface given in an Interface Definition Language (IDL):

```
#pragma prefix "eu.fbk"

module Monitor {
  #pragma version Monitor 1.0

  enum Verdict {RV_True, RV_False, RV_Unknown, RV_Error};
  #pragma version Verdict 1.0

  interface MonitorService {
    #pragma version MonitorService 1.0

    // this "any" index can only be string or long
    Verdict heartbeat (in any index, in string state);
    #pragma version heartbeat 1.0
```

[7] The first two parts of the ID are always “NuRV/Monitor”, while the third part “Service” can be changed by using command option `-i`. When starting multiple monitor servers (i.e. multiple NuRV processes, each of them should have different IDs.

[8] `https://brew.sh`

```
    oneway void reset (in any index, in boolean hard_p);
    #pragma version reset 1.0
  };
};
```

This class/interface is called `MonitorService`, which currently has two methods: `heartbeat` and `reset`. (It is safe to completely ignore those "progma" lines in the above IDL definition, as they are totally internal matters.) The method `heartbeat` is for sending an observation to the monitor. It takes two parameters: an integer- or string-valued `index` of LTL properties (each LTL property corresponds one internal monitor, to be created by the command `build_monitor`), and a string-valued `state` as a logical expression following NuSMV syntax representing the current observation (e.g. "`p & !q`" means $p \wedge \neg q$). The return value is in an enumeration type `Verdict` which has four possible values: `RV_True`, `RV_False`, `RV_Unknown`, and `RV_Error`. The method `reset` is for resetting the monitor. It is a "oneway" method, which immediately returns without any return value. Beside the same `index` parameter for identifying the internal monitor (or property), the Boolean parameter `hard_p` is used for choosing between hard and soft resets: *if this parameter is true, then it is a hard reset, otherwise it is a soft reset.*

The job of a monitor client is to mapping this monitor service instance from remote (i.e. the process space of monitor server) to local. The monitor server(s), after startup, will register their instance to the central third-party naming service, while the monitor client(s) also find the needed monitor service from the same naming service. Obviously each such monitor service instance should have a unique name. This name is by default "NuRV/Monitor/Service" and can be customized by the `-i` parameter of the command `monitor_server` (see Chapter 7 for more details.)

Once the monitor client succeed in mapping the monitor service instance to its local process space, it will behave just like a normal object in its own (object-oriented) programming language (for non-OO languages like C, the method calls are simulated by normal C functions on instance pointers).

The present interface is "simple" in the sense that any method call of `heartbeat` must wait until the related monitoring computation finished on the server side. In another words, this is a *synchronous* interface.[9]

5.5 Client programming in various programming languages

In this section we describe the monitor client programming in five supported programming languages: C, C++, Java, Common Lisp and Python. All involved sample client code can be found in NuRV common files shipped with the main executions.

NOTE: starting from version 1.7.0, NuRV ships with two execution files on platforms with monitor server supports: `NuRV[.exe]` and `NuRV_orbit`, only the latter supports monitor server. Using the command `monitor_server` on the normal `NuRV[.exe]` will cause the program immediately quit. (Currently `NuRV_orbit.exe` is not available on MS Windows, but this is not due to any essential technical difficulities.)

5.5.1 Preliminaries

By default, the monitor server only listen on Unix domain sockets and can only be connected from monitor clients written in C (see Section 5.5.2 for more details). To enable the monitor server listening on TCP/IP (or even IPv6) ports, a file named "`.orbitrc`" must be created and put into the home directory with the following contents:

```
ORBIIOPUSock=1
ORBIIOPIPv4=1
ORBIIOPIPv6=0
```

For instance, the above recommended config file enables Unix domain sockets, TCP/IPv4 but keeps TCP/IPv6 disabled.

[9] In the future, NuRV may support a *synchronous* interface: the method calls immediately returns, while the monitor server will later contact the monitor client (which must have an internal event loop to listen for such contacts) with the monitoring results returned.

5.5.2 C

C is the native language in which NuRV and its monitor server support is written. When using the C-based monitor client on the same machine with the monitor server, client and server does not need TCP/IP at all: instead they can communicate directly by Unix domain sockets.

The C-based monitor client requires linking a library called `orbit2`, which can be easily installed on many Linux and FreeBSD systems. On Mac OS X, users can install it from Homebrew, Fink or MacPorts. The sample client code finds the needed library by `pkg-config`, which therefore must be installed together by the same packaging system. Any client code must include three C headers:

```c
#include <orbit/orbit.h>
#include "monitor.h"
#include "toolkit.h" /* ie. etk_abort_if_exception() */
```

The header file `orbit/orbit.h` is provided by the `orbit2` package. The header file `monitor.h` is generated from the IDL interface, together with `monitor-common.c` and `monitor-stubs.c`. (End users do not need to re-generate them and can just copy the already generated interface code for their own uses.) The header file `toolkit.h`, together with `toolkit.c`, are small toolkit files for the ease use of `orbit2`. Users can freely use them too.

The only manually written code file is thus only `monitor-client.c`. The following global variables are needed for holding the connection information.

```c
static CORBA_ORB global_orb = CORBA_OBJECT_NIL; /* global orb */
static Monitor_MonitorService service = CORBA_OBJECT_NIL;
static CORBA_Environment ev[1];
```

Note that the variable service of the type `Monitor_MonitorService`. Each variable of this type holds one monitor server. If a single monitor client needs to connect to multiple monitor servers (by running multiple NuRV processes), multiple variables (or an array) of this type must be used.

The following main stages are needed for a typical monitor client programming:

1. *Initialization.* The string “`orbit-local-orb`” can be arbitrary.

   ```c
   CORBA_exception_init(ev);
   global_orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
   ```

2. *Binding the name service.* The sample code connects to the default monitor service name (“NuRV/Monitor/Service” by default. Change the code if the monitor server is started with different instance names.)

   ```c
   CosNaming_NamingContext name_service = CORBA_OBJECT_NIL;
   gchar *id[] = {"NuRV", "Monitor", "Service", NULL};

   name_service = etk_get_name_service (global_orb, ev);
   service = (Monitor_MonitorService) etk_name_service_resolve (name_service, id, ev);
   ```

3. *Sending observations to the monitor.* The following code prepare the monitor index at 0, and the variables holding two observations “p” and “q”:

   ```c
     CORBA_long id = 0;
     CORBA_any index;
     index._type = TC_CORBA_long;
     index._value = &id;

     CORBA_char *state_p = "p";
     CORBA_char *state_q = "q";
   ```

   Then the following code can be used for sending an observation to the monitor:

   ```c
     Monitor_Verdict res;
     res = Monitor_MonitorService_heartbeat (service, &index, state_p, ev);
   ```

4. *Processing the monitor verdicts.* The following same code pieces can translated the values of the enum type `Monitor_Verdict` into different string-based outputs (and print out them):

```
switch (res) {
case Monitor_RV_True:
  g_print("true\n");
  break;
case Monitor_RV_False:
  g_print("false\n");
  break;
case Monitor_RV_Unknown:
  g_print("unknown\n");
  break;
default:
  g_print("error\n");
}
```

5. *Resetting the monitor.* The following code does a hard reset to the monitor at the previous index:

   ```
   CORBA_boolean hard_p = CORBA_TRUE;
   Monitor_MonitorService_reset (service, &index, hard_p, ev);
   ```

6. *Uninitialization and cleanup.*

   ```
   CORBA_Object_release(service, ev);

   if (orb != CORBA_OBJECT_NIL) {
     CORBA_ORB_destroy(orb, ev);
   }
   ```

More details can be found in “ORBit Beginners Documentation V1.6” available on Internet.

5.5.3 C++

The sample C++ client code provided at client/cpp requires a library called omniorb, which is provided by most Linux distributions. On Mac, bot MacPorts and Homebrew provide them.

Monitor client code in C++ is more natural than the above code in C, in the sense that the monitor service is represented by a real C++ object. Below we quickly give the relevant code pieces corresponding to each stages: (Check the actual sample code for C++ exception handling. Also note that the monitor server must enable TCP/IPv4.)

1. Initialization. (The C++ header file monitor.hh is generated from the IDL file.)

   ```
   #include "monitor.hh"

   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB4");
   ```

2. Binding the name service.

   ```
   CORBA::Object_var obj = orb->resolve_initial_references("NameService");
   CosNaming::NamingContext_var rootContext = CosNaming::NamingContext::_narrow(obj);

   CosNaming::Name name;
     name.length(3);
     name[0].id = (const char*) "NuRV"; // string copied
     name[0].kind = (const char*) "";  // string copied
     name[1].id = (const char*) "Monitor";
     name[1].kind = (const char*) "";
     name[2].id = (const char*) "Service";
     name[2].kind = (const char*) "";

   CORBA::Object_var obj2 = rootContext->resolve(name);
   Monitor::MonitorService_var service = Monitor::MonitorService::_narrow(obj2);
   ```

3. Sending the observations.

```
CORBA::String_var state = (const char*) "p";
CORBA::Any index;
CORBA::Long l = 0;
index <<= l;

Monitor::Verdict res = service->heartbeat (index, state);
```

4. Processing the monitor verdicts.

```
    string result;
    switch (res) {
    case Monitor::RV_True:
      result = "true";
      break;
    case Monitor::RV_False:
      result = "false";
      break;
    case Monitor::RV_Unknown:
      result = "unknown";
      break;
    default:
      result = "error";
    }
```

5. Resetting the monitor.

```
CORBA::Boolean hard_p = false;
service->reset (index, hard_p);
```

6. Ending the client.

```
orb->destroy();
```

More details can be found in omniORB 4 documents at http://omniorb.sourceforge.net/docs.html.

5.5.4 Java

The Java client code (at client/java, as an Eclipse project) only supports JDK before or equals to 1.8, in which JDK directly provides the needed libraries (Thus no third-party JAR is needed). The monitor interface IDL is translated to some Java classes under the prefix eu.fbk.monitor.

Java program musst have a entry/main class. The connection information is held in its public static member variable:

```
public static org.omg.CORBA.ORB orb = null;
```

1. Initialization.

```
Properties props = new Properties();
String ior = "IOR:...";
props.put("org.omg.CORBA.ORBInitRef", "NameService=" + ior);
orb = org.omg.CORBA.ORB.init(args, props);
```

2. Binding name service.

```
org.omg.CORBA.Object ncRef = orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext nc =
    org.omg.CosNaming.NamingContextHelper.narrow(ncRef);

org.omg.CosNaming.NameComponent[] monitorName =
    new org.omg.CosNaming.NameComponent[3];
monitorName[0] = new org.omg.CosNaming.NameComponent("NuRV", "");
monitorName[1] = new org.omg.CosNaming.NameComponent("Monitor", "");
monitorName[2] = new org.omg.CosNaming.NameComponent("Service", "");

org.omg.CORBA.Object monitorRef = nc.resolve(monitorName);
MonitorService service = MonitorServiceHelper.narrow(monitorRef);
```

3. Sending the observation.

```
org.omg.CORBA.Any index = orb.create_any();
index.insert_long(0); // monitor 0
Verdict res = service.heartbeat(index, "TRUE");
```

4. Processing the outputs.

```
String result = new String();
if (res == Verdict.RV_True) {
    result = "true";
} else if (res == Verdict.RV_False) {
    result = "false";
} else if (res == Verdict.RV_Unknown) {
    result = "unknown";
} else { // res == Monitor.Verdict.RV_Error
    result = "error";
}
```

5. Resetting the monitor.

```
service.reset(index, false);
```

6. Shutdown.

```
orb.shutdown(true);
orb.destroy();
```

More details can be found in JDK 1.8 documentation.

5.5.5 Common Lisp

The Common Lisp client code is based on LispWorks Enterprise Edition. The interface IDL file is part of the running program without any pre-translation.

1. Initialization. (Suppose the IOR string is stored in a variable `*ior*`.)

```
(defvar *client-orb*   nil) ; ORB
(defvar *name-service* nil) ; NS
(defvar *service*      nil) ; Monitor:Service instance

(setq *client-orb* (op:orb_init nil "LispWorks ORB"))
```

2. Binding the name service.

```
(corba:set-pluggable-module-details "NameService" :ior-string *ior*)

(defun get-name-service (orb)
  (let ((ref (op:resolve_initial_references orb "NameService")))
    (when ref
      (op:narrow ’CosNaming:NamingContext ref))))

(setq *name-service* (get-name-service *client-orb*))
```

```lisp
(defun name-components (names)
  (mapcar #'(lambda (name) (CosNaming:NameComponent :id name :kind "")) names))

(setq *monitor-name* (name-components '("NuRV" "Monitor" "Service")))

(defun resolve-object (name)
  (unless *name-service*
    (warn "No name service found")
    (return-from resolve-object nil))
  (handler-case
      (op:resolve *name-service* name)
    (CosNaming:NamingContext/NotFound nil)))

(setq *service*
      (op:narrow 'Monitor:MonitorService (resolve-object *monitor-name*))))
```

3. Sending the observation.

```lisp
(defgeneric monitor (index))
(defmethod monitor ((index integer))
  (corba:any :any-typecode corba:_tc_long :any-value index))

(defmethod monitor ((index string))
  (corba:any :any-typecode corba:_tc_string :any-value index))

(op:heartbeat *service* (monitor 0) "p & q") ; 0 is the id of an LTL property
(op:heartbeat *service* (monitor "p0") "!p") ; "p0" is the name of an LTL property
```

4. Resetting the monitor.

```lisp
(op:reset *service* (monitor 0) nil) ; nil means soft reset (hard_p = false)
```

5.5.6 Python

The Python client code requires a Python package py-omniORBpy which can be found in MacPorts. Similar packages (but with different names) are available on Linux, searching keywords “py” + “omniorb”. The monitor interface IDL is directly read by Python code, some minor stub Python code are also generated from the IDL file. (They are accessible by Python code import Monitor.)

1. Initialization.

```python
import sys
from omniORB import CORBA
from omniORB import any
import Monitor
import CosNaming

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
```

2. Binding name service.

```python
obj = orb.resolve_initial_references("NameService");
rootContext = obj._narrow(CosNaming.NamingContext)

name = [CosNaming.NameComponent("NuRV", ""),
        CosNaming.NameComponent("Monitor", ""),
        CosNaming.NameComponent("Service", "")]
obj = rootContext.resolve(name)
service = obj._narrow(Monitor.MonitorService)
```

3. Sending the observation (and reset):

   ```
   service.heartbeat(to_any(0), "p & q")
   service.reset(to_any("p0"))
   ```

More details can be found in omniORB 4 Python documents at `http://omniorb.sourceforge.net/docs.html`.

Chapter 6

RV on Infinite-State Systems

NuRV supports infinite-state monitoring [CTT21] since version 1.5.0.

For LTL properties using infinite-domain variables such as integers or real numbers, the RV problem without assumption can be resolved as in the Boolean case. Although non-Boolean variables are involved, the monitoring property is essentially still propositional. For example, to monitor $\mathbf{G}\,(i \leqslant 5)$, where $i \in \mathbb{Z}$ is unbounded, we can instead synthesize a monitor from $\mathbf{G}\,p$ where p is a Boolean variable. Then, at runtime, it only remains to convert the original input traces about i into equivalent traces about p by the formula $p = (i \leqslant 5)$. The situation is the same for real variables.

The ABRV algorithm for infinite-state systems is implemented by leveraging NUXMV's functionality of verifying LTL properties on infinite-state models using IC3IA engine [CGMT14], which depends on the MathSAT5 SMT solver [CGSS13] for infinite-state models. The QE procedure is also provided by MathSAT. Most NuRV features, such as partial observability and resets, are also supported for infinite-state systems, while the explicit-state code generation is not supported.

Starting from version 1.6.0, NuRV adopts a new technique called *Incremental Bounded Model Checking*. and other optimizations. This results to further performance improvements (at least *10x* faster).

To support RV on infinite-state systems, the following changes must be made when using NuRV:

- Use `go_msat` instead of `go` to prepare the models (as RV assumptions);
- Use `-x` when calling `build_monitor` [1], `verify_property` and `heartbeat` commands.

6.1 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)

[1] This step is optional for BDD-based monitors as the monitors will be built automatically when necessary. For RV on infinite-state systems, calling `build_monitor` is now a must before further RV commands, because an important parameter, the maximal bound k for BMC solvers, must be specified here.

- $h \rightarrow (h' \leftrightarrow \neg f)$ (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 can be switched on with the command s)
- $f \rightarrow f'$ (the fault is permanent)

Suppose that we can only observe the temperature and the switching command, and that we want to monitor the following property: $\varphi_1 = \mathbf{G}(s \rightarrow \mathbf{F}(t = 100))$, i.e., whenever the heating system is switched on, the temperature will eventually reach the temperature of 100. The assumption can be exploited by the ABRV monitor to deduce for example that 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 that, without assumption, 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 traces of K compatible with u violate φ. Thus, $[\![u, 0 \models \varphi_1]\!]_4^K = \bot^a$.

Suppose instead the we monitor the stronger property: $\mathbf{G}(s \rightarrow \mathbf{F}^{\leq 7}(t = 100))$, i.e, whenever the heating system is switched on, the temperature will reach the temperature of 100 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 the level 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.

The following SMV file (saved as `example_temp.smv`) implements the above RV assumption:

```
MODULE main
VAR heating: boolean;
VAR temp: real;
VAR switch_on: boolean;
VAR fault_heat: boolean;

INIT
temp=0

INVAR
temp>=0 & temp<=100

TRANS
heating -> ((temp=100 & next(temp)=100) | (next(temp)>=temp + 10 & next(temp)<=temp + 20))

TRANS
(!heating) -> ((temp=0 & next(temp)=0) | (next(temp)<=temp - 10 & next(temp)>=temp - 20))

TRANS
heating -> (next(heating)=!fault_heat)

TRANS
(!heating) -> (next(heating)=(switch_on & !fault_heat))

TRANS
fault_heat -> next(fault_heat)
```

The above mentioned sample trace can be implemented below (saved as `trace.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<counter-example type="0" id="1" desc="LTL Counterexample">
  <node><state id="1">
    <value variable="temp">0</value>
    <value variable="switch_on">TRUE</value>
  </state></node>
  <node><state id="2">
    <value variable="temp">20</value>
    <value variable="switch_on">FALSE</value>
  </state></node>
  <node><state id="3">
    <value variable="temp">10</value>
    <value variable="switch_on">TRUE</value>
  </state></node>
</counter-example>
```

The next batch command, saved in test.cmd, first defines two new LTL properties and then verify them againt the above trace:

```
go_msat

add_property -l -n "p1" -p "G (switch_on -> F (temp=100))"
add_property -l -n "p2" -p "G (switch_on -> F [0,7] (temp=100))"

read_trace trace.xml
show_traces 1

show_property -P "p1"
echo Monitoring p1 ...
verify_property -x -P "p1" 1

show_property -P "p2"
echo Monitoring p2 ...
verify_property -x -P "p2" 1

quit
```

Below are outputs of the above batch commands:

```
Trace is stored at 1 index
    <!-- ################### Trace number: 1 ################### -->
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    temp = 0
    switch_on = TRUE
  -> State: 1.2 <-
    temp = 20
    switch_on = FALSE
  -> State: 1.3 <-
    temp = 10
    switch_on = TRUE
**** PROPERTY LIST [ Type, Status, Counter-example Number, Name ] ****
--------------------------  PROPERTY LIST  -------------------------
005 : G (switch_on ->  F temp = 100)
  [LTL            Unchecked      N/A    p1]
Monitoring p1 ...
2, unknown
3, unknown
Maximum bound reached
4, false
**** PROPERTY LIST [ Type, Status, Counter-example Number, Name ] ****
--------------------------  PROPERTY LIST  -------------------------
006 : G (switch_on ->  F [0,7] temp = 100)
  [LTL            Unchecked      N/A    p2]
Monitoring p2 ...
2, unknown
3, unknown
```

```
4, false
```

Chapter 7

Commands

7.1 Commands available in all NuRV variants

The command `verify_property` can be used as an offline monitor: it takes input from a trace, and output runtime verification results on each state of the trace, outputting 4-valued results: *true*, *false*, *unknown* and *error* (*out-of-model*). The output can be written into a file, in CSV-based text format.

The command `build_monitor` and `generate_monitor` together synthesize the symbolic runtime monitor into explicit-state monitor program in C, Lisp or SMV. The C and Lisp versions can be compiled into monitor program modules as part of a large runtime monitoring solution. The SMV version is intented for model checking on the correctness of runtime monitors.

verify_property - *Verify a property against a trace using runtime verification techniques* Command

```
verify_property [-h] [-n number | -P "name"] | [-p "formula [IN context]"] [-r]
[-x] [-e number] [-o filename] trace_number[.from_state[:[to_state]]]
```

Verifies the specified property taken from the property list, or adds the new specified property and verifies it.

Command Options:

`-h`	Prints the command usage.
`-n number`	Verifies the property with index `number` in the property database.
`-P "name"`	Verifies the property named "name" in the property database.
`-p "formula [IN context]"`	Verifies the `formula` specified on the command-line. `context` is the module instance name where the variables in `formula` must be evaluated.
`-r`	Always reset (ptLTL special mode).
`-o filename`	Save verification results to the file given by `filename`.
`-x`	RV on infinite-state systems.
`-e number`	Engine of RV for infinite-state systems (0: simple, 1: optimized, 2: incremental (default))
`trace_number`	The (ordinal) identifier number of the trace to be verified.

`from_state`	Denotes left end of the trace slice to be verified.
`to_state`	Denotes right end of the trace slice to be verified.

build_monitor - *Builds a runtime monitor for a `ltl-expr` property* Command

```
build_monitor [-h] [-x] [-k number] [-c "constr"] [-C file] [[-n number | -P
"name"] | [-p "formula [IN context]"]]
```

Command Options:

`-h`	Prints the command usage.
`-n number`	Build monitor for the property with index `number` in the property database.
`-P name`	Build monitor for the property named `name` in the property database.
`-p "formula [IN context]"`	Build monitor for the `formula` specified on the command-line. `context` is the module instance name where the variables in `formula` must be evaluated.
`-x`	RV on infinite-state systems.
`-k number`	The `max_k` (maximal search bound) of Bounded Model Checking.
`-c "constr"`	A Boolean expression of observables as a cube of Boolean variables.
`-C filename`	A file listing each observables (scalar variable or Boolean expression)

Remark 7.1. Each line of the file supplied by the parameter `-C` can be either a proposition (i.e. expression of type Boolean) or a single scalar variable. In the latter case, all underlying bits of the scalar variable are considered observable. If `-c` is also specified, it is combined with observable propositions given by `-C`.

generate_monitor - *Generate standalone explicit-state monitor code.* Command

```
generate_monitor [-h] [-x] [-S] [-n number | -P "name"] [-l level] [-L "lang"] [-v
"vars"] [-p] [-f "function"] [-m "module"] -o filename
```

Command Options:

`-h`	Prints the command usage.
`-n number`	Outputs the monitor for property with index `number` in the property database.
`-P name`	Outputs the monitor for property named `name` in the property database.
`-l number`	Monitor construction level (1-4). Default: 3.
`-L lang`	Target language for code generation.
`-v "vars"`	Cube of all variables in the model.
`-p`	Partial observablity support (powerset construction)
`-x`	RV on infinite-state systems. Use with -S to generate symbolic infinite-state monitors calling NuRV as a shared/dynamic library.

`-S` Generating symbolic monitors calling NuRV as a shared/dynamic library.

`-f "function"` Name of entry function/class in the target language.

`-m "module"` Name of module/package in the target language.

`-o filename` The filename of generated monitor code (usually also the name of entry function)

heartbeat - *Send one input state to the monitor (online monitoring).* Command

```
heartbeat [-h] [-x] [-n number] [-P "name"] [-c "constr" | -s trace.state] [-r]
[-R]
```

Command Options:

`-h` Prints the command usage.

`-n number` outputs the monitor for property with index `number` in the property database.

`-P name` outputs the monitor for property named `name` in the property database.

`-s trace.state` Pick an input state from the trace manager.

`-r` Soft reset the monitor before processing the current input state.

`-R` Hard reset the monitor before processing the current input state.

`-x` RV on infinite-state systems.

7.2 Commands available in certain NuRV variants

The following command is only available in NuRV with network support:

monitor_server - *Start an online, CORBA-based monitor server* Command

```
monitor_server [-h] [-N IOR] [-i instance]
```

Command Options:

`-h` Prints the command usage.

`-N IOR` IOR string of CORBA name service (default: from the `nameservice` variable)

`-i instance` monitor instance name (default: `Service`)

Bibliography

[BCC$^+$19] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. *nuXmv 2.0.0 User Manual*, 2019.

[CGMT14] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *LNCS 7795 - Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, pages 93–107. Springer, Berlin, Heidelberg, February 2013.

[CTT19a] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-Based Runtime Verification with Partial Observability and Resets. In Bernd Finkbeiner and Leonardo Mariani, editors, *LNCS 11757 - Runtime Verification (RV 2019)*, pages 165–184. Springer International Publishing, Porto, Portugal, October 2019.

[CTT19b] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. NuRV: A nuXmv Extension for Runtime Verification. In Bernd Finkbeiner and Leonardo Mariani, editors, *LNCS 11757 - Runtime Verification (RV 2019)*, pages 382–392. Springer International Publishing, Porto, Portugal, October 2019.

[CTT21] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-Based Runtime Verification of Infinite-State Systems. To be published by Runtime Verification 2021 (RV '21), 2021.

[DAC99] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, New York, USA, 1999. ACM Press.

[FKRT18] Yliès Falcone, Srdjan Krstic, Giles Reger, and Dmitriy Traytel. A Taxonomy for Classifying Runtime Verification Tools. In Christian Colombo and Martin Leucker, editors, *LNCS 11237 - Runtime Verification (RV 2018)*, pages 241–262. Springer, Cham, 2018.

Appendix A

LTL patterns

Dwyer's LTL patterns [DAC99] in NuSMV's LTL syntax are given in Table A.1 and A.2.

Table A.1: Dwyer's LTL patterns

ID	Pattern	LTL
0	p is false (Globally)	`G (!p)`
1	p is false (Before r)	`F r -> (!p U r)`
2	p is false (After q)	`G (q -> G (!p))`
3	p is false (Between q and r)	`G ((q & !r & F r) -> (!p U r))`
4	p is false (After q until r)	`G (q & !r -> ((G !p) | (!p U r)))`
5	p becomes true (Globally)	`F p`
6	p becomes true (Before r)	`(G !r) | (!r U (p & !r))`
7	p becomes true (After q)	`G (!q) | F (q & F p)`
8	p becomes true (Between q and r)	`G (q & !r -> ((G !r) | (!r U (p & !r))))`
9	p becomes true (After q until r)	`G (q & !r -> (!r U (p & !r)))`

10	trans to p occ. $\leqslant 2$ (Globally)	`G (!p) | (!p U (G p | (p U (G (!p) |`
		`(!p U (G p | (p U (G (!p)))))))))`

11	trans to p occ. $\leqslant 2$ (Before r)	`F r -> ((!p & !r) U (r | ((p & !r) U (r |`
		`((!p & !r) U (r | ((p & !r) U (r | (!p U r)))))))))`

12	trans to p occ. $\leqslant 2$ (After q)	`F q -> (!q U (q & (G (!p) | (!p U (G p`
		`| (p U (G (!p) | (!p U (G p | (p U (G !p)))))))))))`

13	trans to p occ. $\leqslant 2$ (Betw. q and r)	`G ((q & F r) -> ((!p & !r) U (r | ((p & !r) U (r |`
		`((!p & !r) U (r | ((p & !r) U (r | (!p U r))))))))))`

14	trans to p occ. $\leqslant 2$ (After q until r)	`G (q -> ((!p & !r) U (r | ((p & !r) U (r | ((!p & !r) U`
		`(r | ((p & !r) U (r | (G (!p) | (!p U r)) | G p)))))))))`

15	p is true (Globally)	`G p`
16	p is true (Before r)	`F r -> (p U r)`
17	p is true (After q)	`G (q -> G p)`
18	p is true (Between q and r)	`G ((q & !r & F r) -> (p U r))`
19	p is true (After q until r)	`G (q & !r -> (G p | (p U r)))`
20	s precedes p (Globally)	`G (!p) | (!p U s)`
21	s precedes p (Before r)	`F r -> (!p U (s | r))`
22	s precedes p (After q)	`G (!q) | F (q & (G (!p) | (!p U s)))`
23	s precedes p (Between q and r)	`G ((q & !r & F r) -> (!p U (s | r)))`
24	s precedes p (After q until r)	`G (q & !r -> (G (!p) | (!p U (s | r))))`

Table A.2: Dwyer's LTL patterns (continued)

ID	Pattern	LTL
25	s responds to p (Globally)	G (p -> F s)
26	s responds to p (Before r)	F r -> (p -> (!r U (s & !r))) U r
27	s responds to p (After q)	G (q -> G (p -> F s))
28	s responds to p (Between q and r)	G ((q & !r & F r) -> (p -> (!r U (s & !r))) U r)
29	s responds to p (After q until r)	G (q & !r -> (G (p -> (!r U (s & !r))) | ((p -> (!r U (s & !r))) U r)))
30	s, t precedes p (Globally)	F p -> (!p U (s & !p & X (!p U t)))
31	s, t precedes p (Before r)	F r -> (!p U (r | (s & !p & X (!p U t))))
32	s, t precedes p (After q)	(G !q) | (!q U (q & F p -> (!p U (s & !p & X (!p U t)))))
33	s, t precedes p (Between q and r)	G ((q & F r) -> (!p U (r | (s & !p & X(!p U t)))))
34	s, t precedes p (After q until r)	G (q -> (F p -> (!p U (r | (s & !p & X(!p U t))))))
35	p precedes s, t (Globally)	(F (s & X (F t))) -> ((!s) U p)
36	p precedes s, t (Before r)	F r -> ((!(s & (!r) & X (!r U (t & !r)))) U (r | p))
37	p precedes s, t (After q)	(G !q) | ((!q) U (q & ((F (s & X (F t))) -> ((!s) U p))))
38	p precedes s, t (Between q and r)	G ((q & F r) -> ((!(s & (!r) & X (!r U (t & !r)))) U (r | p)))
39	p precedes s, t (After q until r)	G (q -> (!(s & (!r) & X(!r U (t & !r))) U (r | p) | G (!(s & X (F t)))))
40	p responds to s, t (Globally)	G (s & X (F t) -> X (F (t & F p)))
41	p responds to s, t (Before r)	F r -> (s & X (!r U t) -> X (!r U (t & F p))) U r
42	p responds to s, t (After q)	G (q -> G (s & X (F t) -> X (!t U (t & F p))))
43	p responds to s, t (Between q and r)	G ((q & F r) -> (s & X (!r U t) -> X (!r U (t & F p))) U r)
44	p responds to s, t (After q until r)	G (q -> (s & X (!r U t) -> X (!r U (t & F p))) U (r | G (s & X (!r U t) -> X (!r U (t & F p)))))
45	s, t responds to p (Globally)	G (p -> F(s & X (F t)))
46	s, t responds to p (Before r)	F r -> (p -> (!r U (s & !r & X (!r U t)))) U r
47	s, t responds to p (After q)	G (q -> G (p -> (s & X (F t))))
48	s, t responds to p (Between q and r)	G ((q & F r) -> (p -> (!r U (s & !r & X (!r U t)))) U r)
49	s, t responds to p (After q until r)	G (q -> (p -> (!r U (s & !r & X (!r U t)))) U (r | G (p -> (s & X (F t)))))
50	s, t w/o z resp. to p (Globally)	G (p -> F (s & !z & X (!z U t)))
51	s, t w/o z resp. to p (Before r)	F r -> (p -> (!r U (s & !r & !z & X ((!r & !z) U t)))) U r
52	s, t w/o z resp. to p (After q)	G (q -> G (p -> (s & !z & X (!z U t))))
53	s, t w/o z resp. to p (Betw. q and r)	G ((q & F r) -> (p -> (!r U (s & !r & !z & X ((!r & !z) U t)))) U r)
54	s, t w/o z resp. to p (After q until r)	G (q -> (p -> (!r U (s & !r & !z & X ((!r & !z) U t)))) U (r | G (p -> (s & !z & X (!z U t)))))

Command Index