VERIFY - a verifier for real-time systems

Jonathan S. Ostroff*

June 1, 1992

Abstract

VERIFY is a tool for checking finite state real-time systems in the temporal logic RTTL. Each process in the system to be verified is declared, in Prolog, as a transition system with lower and upper time bounds on each transition (0 and ∞ if the bounds are unknown). The processes can be composed in parallel to form more complex systems of concurrent processes. To check an RTTL property (such as freedom from deadlock or a real time response property), VERIFY generates a state reachability graph, and checks the RTTL property over the reachability graph. The processes may have data variables so that various data types and operations on data can be accommodated. If the property does not hold, the state (or sequence of states) for which the property fails to hold is provided. This information is useful in debugging the design.

VERIFY is part of a toolset that is being developed at York University (Canada) for the design of real-time safety critical systems. The TTM/RTTL framework forms the underlying theoretical basis for the tools. TTMs are timed transition systems, and RTTL stands for real-time temporal logic.

Another tool called BUILD is currently under development. BUILD is a visual specification tool for constructing complex systems as hierarchical TTMs. BUILD also allows for the execution of the TTMs, as well as producing the input Prolog descriptions of the systems as Prolog code suitable for input to VERIFY.

*Jonathan Ostroff is with the Department of Computer Science, York University 4700 Keele Street, North York, Ontario, Canada, M3J 1P3. Email: jonathan@cs.yorku.ca. This work is supported by the Natural Sciences and Engineering Research Council of Canada

Contents

1 Getting Started

This document describes a Prolog program called VERIFY to verify the correctness of real-time reactive systems. The references in [Ost90a,Ost92b] describe the TTM/RTTL framework that is used to model and check real-time systems.

VERIFY was written in Quintus Prolog[1] Version 3.0, and has been tested on both SUN and DEC workstations. VERIFY should work on any platform that supports Quintus Prolog Version 3.0 (hereafter referred to as QP)[2].

The program was written as a prototype for use on small systems with a couple of hundred states, although it has also been used on systems with about 10,000 states. It should not be relied on as the sole check of correctness of safety critical software.

1.1 VERIFY

VERIFY implements the decision procedures described in [Ost90a] for automatically checking a small but important class of properties for any finite state TTM (timed transition model). The procedures are linear in the size of the system state reachability graph. Real-time temporal logic (RTTL) is used for specifying the properties to be verified.

The following steps are involved in verifying a given property.

1. (Modelling step) Given a system composed of a set of interacting real-time discrete event processes, construct the corresponding TTM M. TTMs can be composed in parallel, e.g. $M = M_1 \| M_2 \| M_3$, where M_1 may itself be the parallel composition of many TTMs.

2. Compute the reachability graph G_M of M using the RG2 procedure described in [Ost90a]. The RG2 procedure ensures that only those states that are actually reachable are computed — lower and upper time bounds on events may constrain the TTM to fewer states than events with no bounds on them.

[1] Quintus Prolog is a copyright of Quintus Computer Systems, Inc.

[2] There is also a QOF file avialable for Quintus Prolog 2.5.1

3. Specify a formula w in RTTL that describes the behaviour that M must satisfy.

4. Apply the appropriate decision procedure for w. All decision procedures perform an analysis of the RG2 reachability graph G_M.

1.2 Quintus Prolog

There are three ways to run VERIFY under QP:

On terminals: VERIFY can be run in a fashion similar to a line editor, or under GNU Emacs (or Unipress Emacs). Running under Emacs allows the user to examine input files in one window, while executing commands in another window. Sessions can be saved on file, and previous commands can be repeated by appealing to command histories.

X11 — QUI: The Quintus User Interface (QUI) is based on the X11 windowing system and the use of the mouse. VERIFY can also be run under QUI.

QP allows a program to be saved in a compiled form called a QOF file (Quintus Object Format). QOF files contain a machine independent representation of both compiled and dynamic Prolog procedures. This means that QOF files are completely portable between different platforms running Quintus Prolog.

VERIFY is made available in QOF format at no charge to researchers. The author retains the copyright to the VERIFY software. No changes may be made to the software without the written permission of the author.

1.3 Installing VERIFY

The file `verify31.tar.Z` must be obtained by anonymous ftp [3] from the `pub/AUTOMATE` directory on the server `nexus.yorku.ca`. Then, uncompress and restore all the required files by executing:

[3] If you are using Quintus Prolog 2.5, then ftp the file `verify251.tar.Z`. When using this version of prolog, the file `dynam.ttm` must be available if the `compose` procedure is used. Also the lines beginning with the `multifile` and `dynamic` declarations in the .ttm files must be commented out.

```
uncompress verify31.tar.Z
tar -xvf verify31.tar
```

This will create a directory `verify` which will contain the QOF file `verify31`, as well as various examples.

1.4 Invoking VERIFY

VERIFY can be started up under the assumption that (a) Quintus Prolog 3.0 is installed, and (b) the `verify` directory has been properly installed as described in Section 1.3.

There are three ways to get started. Either invoke the QOF file `verify` from the Unix command line:

```
unix prompt>  ./verify
% loading file ...

VERIFY Version 1.0 May 7, 1992
 Copyright Jonathan Ostroff, York University.
 email : jonathan@cs.yorku.ca


                Type "routines" for a list of available routines.



Quintus Prolog Release 3.1.1 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800

| ?-
```

Alternatively, Quintus Prolog can be invoked under Emacs. First execute the command `% prolog +` to run Emacs. At the Prolog prompt in the Emacs window, VERIFY can be invoked by executing the Prolog command `restore(verify).` (Help for either prolog or emacs can be obtained by executing the prolog command `manual.`).

A third possibility is to call `qui` from the Unix command line to invoke the X11 interface QUI (QUI can usually be invoked by executing `doing the Prolog procedure restore(verify).  as before.`

The Prolog command ‘‘routines.’’ will provide a list of all the VERIFY procedures that can be invoked.

```
| ?- routines.


        Graph Routines          rg2(TTM).
        ---------------         compose([TTM1...TTMn], NewTTM).

        Display Routines :      statistics(TTM).
        ----------------        graph(TTM).
                                uniqueStates(TTM).
                                allStates(TTM).
                                initialStates(TTM).

        Verification Routines:  henceForth(TTM, Property1, Property2,
        ----------------------                 Answer, Location).

                                unless(TTM, Property1, Property2, Property3,
                                              Answer, Location).
                                eventually(TTM, StateFormula1, StateFormula2,
                                              Answer, Location).
                                rtrfRG2(TTM, InitialStateFormulaNumber,
                                              FinalStateFormulaNumber,
                                              MinLo, MaxHi).
                                          MaxHi).


        Predicates :            prop(TTM, Property, [StateContents,
        ----------                                 NextTransition]).
                                sf(TTM, Property, StateContents).


        If RoutineName is not unified, the descriptions of all the routines
will be displayed.

yes
| ?-
```

In addition, some routines are available for examining the files in the current directory, or changing to a new directory in the Unix file system.

`cd(directory).` — changes the current directory to be `directory`. For example, cd(..) moves up one level, and cd(sensor) changes the current directory to the subdirectory `sensor`.

`pwd.` — returns the path of the current directory.

Other commands — Other commands include: `cat(filename)`, `date`, `ls(directory)`, `ls-l(directory)`, `ls-l(directory)`, `ls-a(directory)` and `ls-al(directory)`.

Although VERIFY is up and running, nothing can be verified until two files are created:

`sample.ttm` — this is the file containg the Prolog code describing the real-time system as a TTM called `sample`.

`sample.sf` — this file contains state-formulas that are used to specify the property to be verified.

2 An example

In this section, we will use an example TTM called SAMPLE to illustrate the workings of the verifier. SAMPLE is a single simple TTM, i.e. it is not the parallel composition of many TTMs.

An example involving the parallel composition of processes, and using shared variables, may be found in [Ost92b]. VERIFY has a `compose` command which allows for TTMs to be composed in parallel. It allows for the possibility of both (a) shared variables, and (b) shared transitions. Thus, both asynchronous and synchronous communication can be modelled. Concurrent and nondeterministic behaviour can also be represented. TTMs can model most features of real-time programs, Petri Nets, Statecharts and Communicating Sequential Processes. In particular, TTMs have data variables that can be used to represent program variables, arrays, records and other data structures.

2.1 The TTM SAMPLE

When using VERIFY we distinguish between:

- The formal mathematical description of a TTM.

  (A TTM M is a 3-tuple $M = \{\mathcal{V}, \Theta, \mathcal{T}\}$ where $\mathcal{V}$ is a variables set, Θ is an initial condition, and $\mathcal{T}$ is the set of all transitions. A precise operational semantics describes all *legal trajectories* that the TTM can execute.)

- The visual presentation of TTMs via TTM-CHARTS (a modified form of statecharts) — other visual methods can just as easily be used, e.g Petri Nets.

- the Prolog representation of the TTM M in a file called `m.ttm`.

To use VERIFY, all that is needed is the Prolog representation of the TTM which is quite similar to the mathematical description. A visual tool called BUILD[4] has been developed, which ties together all three aspects of the TTM. BUILD allows the designer to enter the TTM visually as a TTM-CHART, uses the semantics to execute computations, and generates the required Prolog representations.

Consider the TTM-CHART of the timed transition model SAMPLE in Figure 1.

Formally the timed transition model SAMPLE is a 3-tuple

$$SAMPLE = (\mathcal{V}_{SAMPLE}, \Theta_{SAMPLE}, \mathcal{T}_{SAMPLE})$$

consisting of a a set of variables $\mathcal{V}_{SAMPLE}$, an initial condition Θ_{SAMPLE}, and a set of transitions $\mathcal{T}_{SAMPLE}$.

The set of TTM variables is $\mathcal{V}_{SAMPLE} = \{\mathbf{n}, t, x, c, y, z\}$ where each variable has a type given by:

$$\begin{aligned} type(t) &= \{0, 1, 2, \cdots\} = type(y) \\ type(\mathbf{n}) &= \mathcal{T}_{SAMPLE} \\ type(x) &= \{\texttt{off}, \texttt{idle}, \texttt{slow}, \texttt{fast}\} \\ type(z) &= \{\cdots, -2, -1, 0, 1, 2, \cdots\} \end{aligned}$$

[4] BUILD has not yet been made publicly avialable as it is still being tested.

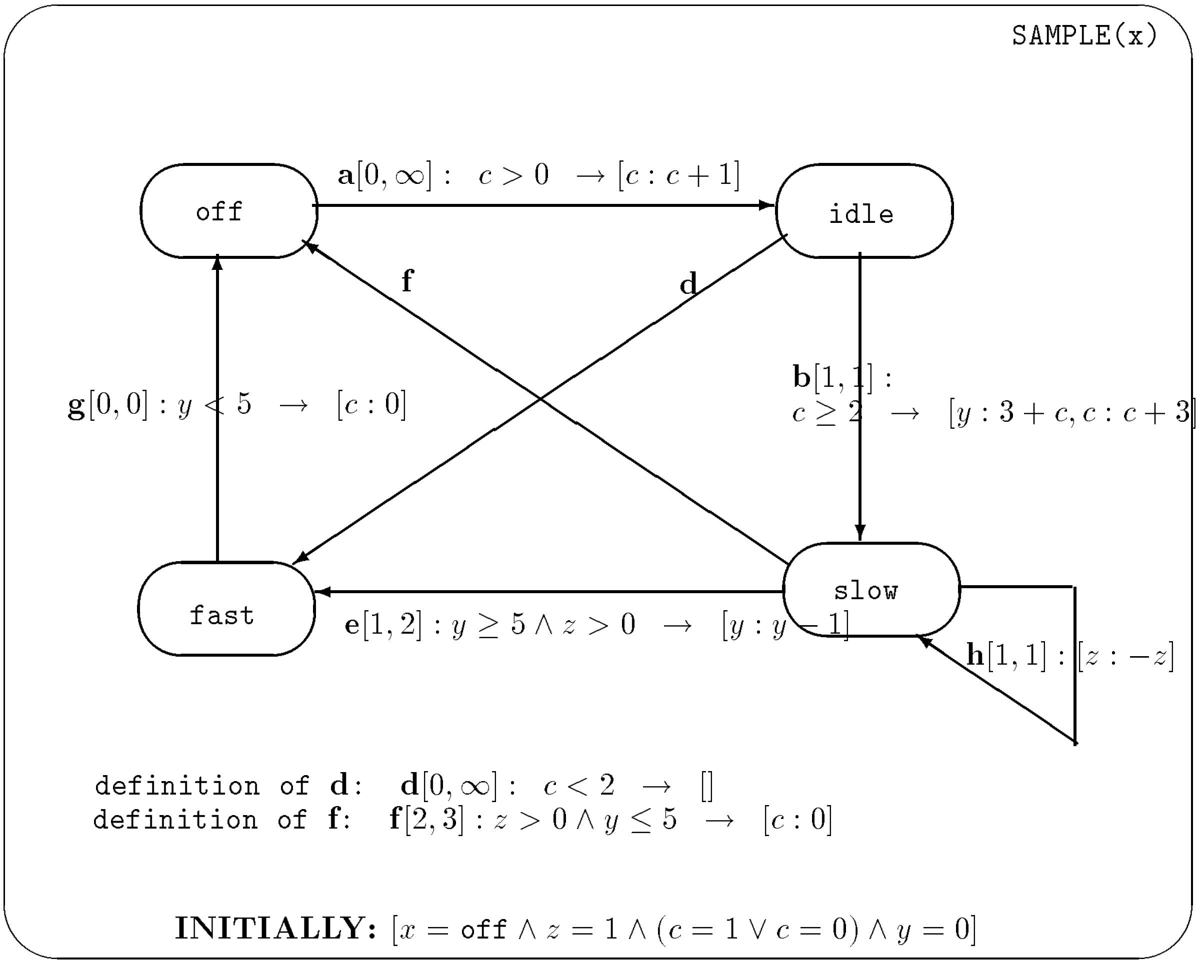

Figure 1: A TTM-CHART of the timed transition model sample

and the initial condition is:

$$\Theta_{SAMPLE} = [x = \texttt{off} \wedge z = 1 \wedge (c = 1 \vee c = 0) \wedge y = 0]$$

and the set of transitions is:

$$\mathcal{T}_{SAMPLE} = \{tick, a, b, d, e, f, g\}$$

Each transition $\tau \in \mathcal{T}$ is a 4-tuple $\tau = (e_\tau, h_\tau, l_\tau, u_\tau)$, where e_τ is the enabling condition, h_τ is the transformation function, l_τ is the lower time bound and u_τ is the upper time bound.

Tran	Guard:	Function:	Low	Upper:
a	(X=off,(C>0))	[X:idle,C:C+1]	0	infinity
b	(X=idle,(C>=2))	[X:slow,Y:3+C,C:C+3]	1	1
d	(X=idle,(C<2))	[X:fast]	0	infinity
e	(X=slow,(Y>=5,Z>0))	[X:fast,Y:Y-1]	1	2
f	(X=slow,(Z>0,Y=<5))	[X:off,C:0]	2	3
g	(X=fast,(Y<5))	[X:off,C:0]	0	0
h	(X=slow)	[X:slow,Z:-Z]	1	1

The above table of transitions is written in Prolog style in which the TTM variables are written in upper case (e.g. `Y` instead of y). Conjunction is denoted by a comma (e.g. `X=off, Z>0`), disjunction is denoted by a semi-colon (e.g. `X=off; Z>0`), and negation is denoted by `not` (e.g. `not ( X=off; Z>0) `).

Enabling conditions and initial conditions are examples of *state-formulas*, i.e. boolean-valued expressions in the TTM variables that can be evaluated to true or false in a single state. RTTL formulas are built up from state formulas, e.g.

S1: $\Box\texttt{not}(\texttt{X} = \texttt{fast}, \texttt{Y} = 3)$ — henceforth (i.e. in the current state and all future states), the assertion `not( X=fast, Y=3 )` is always true.

The symbol $\Box$ therefore stands for *henceforth*.

S2: $(\texttt{C} = 0) \rightarrow \Box\texttt{isTick(N)}$ — in any state in which `(C=0)` becomes true, from then onwards the only transition that can occur is a tick. (The state-formula `isTick(N)` asserts that $(\mathbf{n} = tick)$, i.e. the next transition taken is the clock tick.)

S3: $\Box(\mathtt{Z} = \mathtt{1}; \mathtt{N} = \mathtt{h}; \mathtt{isTick}(\mathtt{N}))$ — henceforth, in any legal state, either `Z=1`, or the transition `h` occurs, or the tick transition ocurs.

S4: $\Box(\mathtt{Z} = \mathtt{1}; \mathtt{isTick}(\mathtt{N}))$ — henceforth, either `Z=1`, or the tick transition occurs.

S5: $(\mathtt{N} = \mathtt{b}) \rightarrow (\mathtt{C} >= \mathtt{5}; \mathtt{Y} = \mathtt{0})\ \mathbf{U}\ (\mathtt{N} = \mathtt{e})$ — if the transition `b` occurs, then, from that state and onwards `(C>=5) ; Y=0` holds true up to (but not necessarily including) the state in which the transition `e` occurs. If `e` does not occur, then `(C>=5) ; Y=0` holds true forever (after the occurrence of `b`).

The symbol **U** stands for *unless*.

S6: $(\mathtt{X} = \mathtt{idle}) \rightarrow \Diamond_{[1,2]}\ (\mathtt{X} = \mathtt{fast}; \mathtt{Z} = -\mathtt{1})$ — if `X=idle` becomes true, then in no less than 1 clock tick, but no more than 2 clock ticks, a state is eventually reached in which either `(X = fast)`, or `(Z = -1)`.

S7: $(\mathtt{X} = \mathtt{off}) \rightarrow \Diamond(\mathtt{X} = \mathtt{idle})$ — if the TTM is `off`, then eventually it must become `idle`.

S8 — Deadlock Free The requirement below specifies that in every (reachable) state, some transition other than *tick* must be enabled.

$$\Box \bigvee_{\tau \in \mathcal{T}_{\mathtt{sample}} \wedge \mathbf{n} \neq tick} e_\tau$$

(The notation e_τ stands for the enabling condition of the transition τ and $\mathcal{T}_{\mathtt{sample}}$ stands for the set of all transitions of the TTM `sample`).

VERIFY can be used to check the correctness of all the above specifications. Note that since the set of legal trajectories is suffix closed, in RTTL the validity of $\Box w$ follows from the validity of w. Thus, in RTTL

$$[p_1 \rightarrow \Diamond p_2] \equiv [\Box(p_1 \rightarrow \Diamond p_2)]$$

In order to check SAMPLE for the above properties, two files are needed:

`sample.ttm` — the Prolog description of the TTM SAMPLE (see Appendix A).

`sample.sf` — the file containing the atomic state-formulas from which specifications in RTTL are constructed (see Appendix B).

Once these files are present, the RG2 reachability graph must be constructed, after which the various specifications can be checked.

2.2 Construction of the RG2 reachability graph

The RG2 reachability graph is constructed using the `sample.ttm` file as follows:

```
%%%  Make sure the current directory contains the sample.ttm file.
| ?- ls.
sample.sf  sample.ttm
yes
%%% Use "cd(directory)" to change to the correct directory if
%%% necessary.
%%%
%%% Construct the rg2 reachability graph.
| ?- rg2(sample).
...
yes
%%% Obtain the number of states and edges in the graph.
| ?- statistics(sample).
...
Statistics for TTM :  sample

Unique States = 7
Total States  = 10
Total Edges   = 12
yes
%%% Obtain the state contents as well as state names.
| ?- uniqueStates(sample).
 State State
Number Contents [X,C,Y,Z]
====== ========
0 <=> [off,1,0,1]
1 <=> [off,0,0,1]
2 <=> [idle,2,0,1]
```

```
3 <=> [slow,5,5,1]
4 <=> [fast,5,4,1]
5 <=> [slow,5,5,-1]
6 <=> [off,0,4,1]
There are 7 unique states for TTM sample.
yes
| ?- allStates(sample).
...
      State      History
      Number      Number
      ======      ======

                0              0
                1              1
                2              2
                2              3
                3              4
                3              5
                4              6
                5              7
                6              1
                5              8
There are 10 states for TTM sample.
yes
%%% Thus a state numbered 2-8 means state number 2 with transition
%%% history 8. State 2-5 has the same values of [X,X,Y,Z] as state
%%% 2-8, the only difference being the transition histories.

%%% Now examine the reachability graph.
| ?- graph(sample).
Source          Destination
State   Event   State
======  =====   ===========
     [tick,1]
0-0 --------> 0-0

       a
0-0 --------> 2-2
```

```
    [tick,1]
1-1 --------> 1-1

     [tick,1]
2-2 --------> 2-3

       b
2-3 --------> 3-4

      [tick,1]
3-4 --------> 3-5

       e
3-5 --------> 4-6

       h
3-5 --------> 5-7

       g
4-6 --------> 6-1

    [tick,1]
5-7 --------> 5-8

    [tick,1]
6-1 --------> 6-1

       h
5-8 --------> 3-4

There are 12 edges for TTM sample.
yes
| ?-
```

2.3 Verifying RTTL specifications

Once the reachability graph is generated, the VERIFY procedures listed below may be used to check RTTL specifications. The notation $p, p_1, p_2, p_3, \cdots$ stands for state-formulas (atomic properties of RTTL declared in the `sample.sf` file).

Safety properties: Two safety properties can be checked, viz. invariance ($p_1 \rightarrow \Box p_2$) and *unless* properties ($p_1 \rightarrow p_2 \mathbf{U} p_3$).

For safety properties, use the `prop(sample,p,[[X,C,Y,Z],N])` style in `sample.sf` for specifying an atomic property p. The variable `N` is used to specify event occurrences whereas the variables `X,C,Y` and `Z` are used to specify activity and data values.

`henceForth(sample,p1,p2,Answer,Location)` — is used to check the invariance property $p_1 \rightarrow \Box p_2$.

To check $\Box p$, invoke:
`henceForth(sample,initial,p,Answer,Location)`.

`unless(sample,p1,p2,p3,Answer,Location)` — is used to check the property $p_1 \rightarrow p_2 \mathrm{U} p_3$.

To check $p_1 \mathrm{U} p_2$, invoke the VERIFY procedure:
`unless(sample,initial,p1,p2,Answer,Location)`.

Bounded response and liveness properties: The bounded response time property ($p_1 \rightarrow \Diamond_{[l,u]} p_2$), and the liveness property ($p_1 \rightarrow \Diamond p_2$) can be checked using the VERIFY query `rtrfRG2(sample,p1,p2,Lo,Hi)`. The `rtrfRG2` name stands for Real-Time Response First (occurrence of p_2 after p_1) using a check of the RG2 graph.

Use the `sf(sample, p, [X,C,Y,Z])` style in `sample.sf` for specifyingatomic properties such as p (or p_1, p_2). The next transition variable, e.g. ($\mathbf{n} = \mathrm{g}$), cannot be used for response or liveness properties in the current version of VERIFY.

`rtrfRG2(sample,p1,p2,Lo,Hi)` — is used to check the property given by the RTTL formula $p_1 \rightarrow \Diamond_{[l,u]} p_2$.

The `rtrfRG2` query checks the minimal and maximal times of the *first* occurrence of p_2 after p_1. Thus, the property really being checked is:

$$p_1 \rightarrow (\neg p_2 \mathcal{U}_{[l,u]} p2)$$

where $\mathcal{U}$ is the *until* operator, i.e. if p_1 becomes true, then eventually within l to u ticks p_2 must occur, and $\neg p_2$ is true until then. Note that

$$\begin{aligned} p_1 \rightarrow (\neg p_2 \mathcal{U}_{[l,u]} p2) & \equiv & p_1 \rightarrow \Diamond p_2 \quad \text{for some finite } l \text{ and } u \\ & \Rightarrow & p_1 \rightarrow \Diamond_{[l,u]} p_2 \end{aligned}$$

The procedure `rtrfRG2` either (a) returns finite values for `Lo` and `Hi`, or (b) returns an uninstantiated value for `Lo` and `Hi`.

In case (b), both $p_1 \rightarrow \Diamond_{[l,u]} p_2$ and $p_1 \rightarrow \Diamond p_2$ are invalid.

In case (a), the liveness property $p_1 \rightarrow \Diamond p_2$ is valid. The real-time response property $p_1 \rightarrow \Diamond_{[l,u]} p_2$ is valid if `Lo` $\leq l \leq u \leq$ `Hi`. The values returned for `Lo` and `Hi` apply to the first occurrence of p_2 after p_1.

2.3.1 Checking safety properties S1 to S5, and S8

Recall that safety properties (henceforth and unless) are checked with reference to the `prop` declarations in `sample.sf`. For example, to check property S5 below, the following atomic `prop` decalarations are made in `sample.sf`:

```
prop(sample, 5, [[X,C,Y,Z], N]) :-
    N=b.
prop(sample, 6, [[X,C,Y,Z], N]) :-
    (C >= 5) ; (Y = 0).
prop(sample, 7, [[X,C,Y,Z], N]) :-
    N=e.

%%% Checking S1
| ?- henceForth(sample, initial, 1, Answer, Location).
...
The present application of henceForth produced a true result.
The result holds for all the states tested.
Answer = true,
Location = all <Press CR>

%%% Checking S2
```

```
| ?- henceForth(sample, 3, 4, Answer, Location).
...
The present application of henceForth produced a true result.
The result holds for all the states tested.
Answer = true,
Location = all

%%% Checking S3
| ?- henceForth(sample, initial, 2, Answer, Location).
...
The present application of henceForth produced a true result.
The result holds for all the states tested.
Answer = true,
Location = all

%%% Checking S4
| ?- henceForth(sample, initial, -3, Answer, Location).
...
The present application of henceForth produced a false result.
The result failed for State Number "5" with History Number "8"
via Transition "h" to State Number "3" with History Number "4".
The State Contents are:
                    [slow,5,5,-1]
Answer = false,
Location = [[5,8],[slow,5,5,-1],h,[3,4]]
%%% In the state 5-8, Z = -1 and N = h. Hence S5 is falsified
%%% in this state.
%%% Knowing the failing state can help the designer debug the TTM.
%%% Press ";" for additional failing states.

%%% Checking S5
| ?- unless(sample, 5, 6 ,7, A ,L).
...
Send results to sample.output file (type "y." or "n.") ? y.
sending results to sample.output file...
A = true,
L = all
```

```
%%% Checking deadlock freedom S8
| ?- henceForth(sample, initial, enabled, A, L)
   The present application of henceForth produced a false result.
   The result failed for State Number "1" with History Number "1".
   The State Contents are:

                  [off,0,0,1]



A = false,
L = [[1,1],[off,0,0,1]] ;


   The present application of henceForth produced a false result.
   The result failed for State Number "6" with History Number "1".
   The State Contents are:

                  [off,0,4,1]



A = false,
L = [[6,1],[off,0,4,1]] ;


   The present application of henceForth produced a true result.
   The result holds for all the states tested.


A = true,
L = all

%%% The TTM sample deadlocks at the above two states.
```

All the above properties are valid except for S4 and S8. VERIFY produces a failing state for S4 and S8 (pressing ";" returns all the other failing states). These failing states can help the designer debug the TTM design.

2.3.2 Checking real-time response properties S6 and S7

Recall that real-time response and liveness properties must be checked by declaring the atomic properties using `sf` declarations in sample.sf.

```
%%%% Checking property S6
| ?- rtrfRG2(sample, 3, 2, Lo, Hi).
...
Send results to sample.output file (type "y." or "n.") ? y.
sending results to sample.output file...
Lo = 1,
Hi = 2
| ?-
%%% Checking property S7
| ?- rtrfRG2(sample, 4, 3, Lo, Hi).
...
Send results to sample.output file (type "y." or "n.") ? y.
sending results to sample.output file...
Lo = _12771,
Hi = _12788
```

The property S7 is thus invalid. The file `sample.output` contains the failing legal trajectories.

3 Composing TTMs

See [Ost92b] for a process control example where:

$$SUD = SENSOR \| FILTER \| ALARM$$

This example is stored in the directory `examples/sensor`.

The three TTMs (sensor, filter and alarm) can be created individually and the `compose` command of VERIFY can be used to run them together:

```
| ?- compose([sensor,filter,alarm], sud).
...
There are variables in the TTMs to be  composed which are not
unique to a particular TTM.  Should all these variables be
considered to be shared variables?  Answering Y. will treat
all like variables as shared variables.  Answering N. will
show each non-unique variable and ask whether or not it
should be considered a shared variable. ( Y. / N. ) : Y.

The composed TTM will be found in the file - sud.ttm
yes
| ?-
```

The TTMs found in the directory `examples/textbook-examples` are the TTMs used in Chapter 7 of [Ost89].

4 Acknowledgements

John Dangov and Olaf von Breman wrote the original software as part of an undergraduate project in the Department of Computer Science at York University. Don Laws improved the eficiency of the software and removed bugs that had crept into the original code.

A The Prolog file `sample.ttm`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                          %
%                                                                          %
%                 Quintus Prolog Code for TTM: "sample"                    %
%                                                                          %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
%  Multifile and Dynamic declarations:
%
%
```

```
:- multifile map/3, type/3, initialcondition/2, en/3, h/4, lohi/4.
:- dynamic map/3, type/3, initialcondition/2, en/3, h/4, lohi/4.
%
%
%  Mapping and Types:
%
%
map(sample, [x,c,y,z], [X,C,Y,Z]) :-
        type(x, X),
        type(c, C),
        type(y, Y),
        type(z, Z).
type(sample, x, X) :-
        (X=fast; X=slow; X=idle; X=off).
type(sample, c, C) :-
        C >= 0.
type(sample, y, Y) :-
        Y >= 0.
type(sample, z, Z) :-
        Z >= 0; Z < 0.
%
%
%  Initial Condition:
%
%
initialcondition(sample, [off, 1, 0, 1]).
initialcondition(sample, [off, 0, 0, 1]).
%
%
%  Enabling Conditions:
%
%
en(sample, tick, [X,C,Y,Z]).
en(sample, a, [X,C,Y,Z]) :-
        (X=off,
         C>0).
en(sample, b, [X,C,Y,Z]) :-
        (X=idle,
         C>=2).
```

```
en(sample, d, [X,C,Y,Z]) :-
        (X=idle,
         C<2).
en(sample, e, [X,C,Y,Z]) :-
        (X=slow,
         (Y>=5,
          Z>0)).
en(sample, f, [X,C,Y,Z]) :-
        (X=slow,
         (Z>0,
          Y=<5)).
en(sample, g, [X,C,Y,Z]) :-
        (X=fast,
         Y<5).
en(sample, h, [X,C,Y,Z]) :-
        X=slow.
%
%
%  Transformation Functions:
%
%
h(sample, tick, [X,C,Y,Z], [X,C,Y,Z]).
h(sample, a, [X,C,Y,Z], [idle,Cnew,Y,Z]) :-
     Cnew is C+1.
h(sample, b, [X,C,Y,Z], [slow,Cnew,Ynew,Z]) :-
     Cnew is C+3,
     Ynew is 3+C.
h(sample, d, [X,C,Y,Z], [fast,C,Y,Z]).
h(sample, e, [X,C,Y,Z], [fast,C,Ynew,Z]) :-
     Ynew is Y-1.
h(sample, f, [X,C,Y,Z], [off,Cnew,Y,Z]) :-
     Cnew is 0.
h(sample, g, [X,C,Y,Z], [off,Cnew,Y,Z]) :-
     Cnew is 0.
h(sample, h, [X,C,Y,Z], [slow,C,Y,Znew]) :-
     Znew is -Z.
%
%
%  Lower and Upper Bounds:
```

```
%
%
lohi(sample, tick, 0, infinity).
lohi(sample, a, 0, infinity).
lohi(sample, b, 1, 1).
lohi(sample, d, 0, infinity).
lohi(sample, e, 1, 2).
lohi(sample, f, 2, 3).
lohi(sample, g, 0, 0).
lohi(sample, h, 1, 1).
```

B The Prolog file sample.sf

```
:- multifile prop/3, sf/3.
:- dynamic prop/3, sf/3.

prop(sample, 1, [[X,C,Y,Z], N]) :-
     not ( X=fast , Y=3 ).
prop(sample, 2, [[X,C,Y,Z], N]) :-
    ( Z=1 ; N=h; isTick(N) ).
prop(sample, 3, [[X,C,Y,Z], N]) :-
    C=0.
prop(sample, 4, [[X,C,Y,Z], N]) :-
    isTick(N).
prop(sample, 5, [[X,C,Y,Z], N]) :-
    N=b.
prop(sample, 6, [[X,C,Y,Z], N]) :-
    (C >= 5) ; (Y = 0).
prop(sample, 7, [[X,C,Y,Z], N]) :-
    N=e.
prop(sample, 8, [[X,C,Y,Z], N]) :-
    Y>=4.
prop(sample, 9, [[X,C,Y,Z], N]) :-
    Y=4.
prop(sample, -1, [[X,C,Y,Z], N]) :-
     (Y=3);N=g.
prop(sample, -2, [[X,C,Y,Z], N]) :-
```

```
     (Y=3); isTick(N).
prop(sample, -3, [[X,C,Y,Z], N]) :-
     Z=1; isTick(N).
prop(sample, -4, [[X,C,Y,Z], N]) :-
    not ( Z=1; isTick(N) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% state-formula "enabled" to check for deadlock
?-consult('sample.ttm').
prop(sample, enabled, [[X,C,Y,Z], N]) :-
       en(sample,Transition, [X,C,Y,Z]), not (Transition = tick).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sf(sample, 1, [X,C,Y,Z]) :-
    X=slow.
sf(sample, 2, [X,C,Y,Z]) :-
    X=fast ; Z= -1.
sf(sample, 3, [X,C,Y,Z]) :-
    X=idle.
sf(sample, 4, [X,C,Y,Z]) :-
    X=off.
```

C Additional Notes

C.1 Files in different directories

In VERIFY routines such as `rg2(sample)` the TTM `sample` refers to a file that is in the current directory.

If the file is in a different directory, then the file name must be single quoted, e.g:

```
rg2('./example/sample')
restore('../verify')
```

C.2 Compiling, saving and restoring VERIFY

To start Quintus Prolog (in an Emacs window)

```
unix prompt> % prolog +
Quintus Prolog Release 3.1.1 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800
| ?
```

In order to compile `verify.pl` to obtain a compiled version of the code called `verify`, the following procedure is invoked:

```
| ?- compile(newverify).
...
% verify.pl compiled in module user, 23.483 sec 193,596 bytes
yes
| ?-
```

The state can now be saved as A QOF file.

```
| ?- save_program(verify, start).
% Saving QOF file ...
yes
| ?-
```

The second argument of `save_program`, i.e. `start`, determines which Prolog prdicate is called on entry into the saved state.

D Archiving the files

Assuming all files are in a directory VERIFY, do

```
tar -cvf verify.tar VERIFY
compress verify.tar
```

to create the archive file `verify.tar.Z`. This file can then be placed in `ftp/AUTOMATE` on `nexus`.

References

[Ost89] J.S. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press Limited (distributed by John Wiley and Sons), England, 1989.

[Ost90a] J.S. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[Ost90b] J.S. Ostroff. A logic for real-time discrete event processes. *IEEE Control Systems Magzine*, June 1990.

[Ost91a] J.S. Ostroff. Constraint logic programming for reasoning about discrete event processes. *The Journal of Logic Programming*, Volume II, numbers 3 and 4, October/November 1991.

[Ost91b] J.S. Ostroff. Systematic development of real-time discrete event systems. In *Proceedings of the ECC91 European Control Conference*, pages 522–533, Paris, France, July 1991. Hermes Press.

[Ost92a] J.S. Ostroff. Verification of safety critical systems using TTM/RTTL. In *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600, Springer-Verlag, 1992. (In Press).

[Ost92b] J.S. Ostroff. A verifier for real-time properties. *The Journal of Real-Time Systems*, 4, pages 5–35, 1992.

[OW90] J.S. Ostroff and W.M. Wonham. A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, April 1990.