Time Rover Product and Research Overview

Language and Tool Properties

Time Rover product are concerned with efficient executable specifications and associated applications. We have developed the technology to execute specifications that enjoys the following properties (papers are available upon request):

1. *Powerful specification/rules language.* We believe a specification language needs to be capable of capturing real-life requirements and constraints, such as real-time constrains, while lending itself to natural language and the human cognitive process. Our tools support forms of linear time temporal logic that include real-time constraints, time series constraints, multi-instancing, and past-time (LTL/MTL/MTLS). Consider the following examples:

   a. *Real-time constraints*: `Always(isRED Implies Eventually` $_{C1<120}$ `isGREEN)` which specifies that a traffic light must change lights from red to green within less than 120 `C1` clock ticks. This specification uses an extension to LTL known as MTL, suggested by Zohar Manna of Stanford.

   b. *Time-series constraints*:
   ```
   Always  (cruiseSet Implies
        (speed*0.95 < speed' && speed' < speed*1.05)
        Until $speed$
   (cruiseChange || cruiseOff))
   ```
   which specifies a constraint on a cruise control system where, as of the `cruiseSet` event, speed must be 5% stable until one of the `cruiseChange` or `cruiseOff` events fire. This specification uses a time-series extension to LTL and MTL which I have recently suggested (Papers #P10).

   c. *Multi-instancing:* consider a system of *n* traffic light controllers, where the traffic light requirement listed in item (a) above needs to be applied to each controller individually, i.e., without the light changing in one controller affecting the evaluation of the requirement in another controller. Our form specification languages and tools support multi-instancing.

   d. *Counting operators*, such as *eventA* occurs between 10 and 20 times until eventB occurs (and when using real-time constraints, the occurrence of *eventB* can be constraint to occur, say, within 2 to 5 seconds).

2. *Graphical user interface* for: rules, simulation (see below), build, and administration (actions such as e-mail notifications or custom actions).

3. *On-line monitoring*. On-line monitoring implies that no postmortem processing is used and history traces of the inputs are not preserved. As a counter example consider a method where the history trace of all input events is stored all in a database and used by SQL based program to query those tables at a later time. The SQL program is a SQL representation of the first-order logic equivalent of the LTL formula. On-line

monitoring preserves no history traces and assumes no termination time for the monitored application. On-line monitoring expands the application range of temporal monitoring enabling non-verification applications described below.

4. *Well founded semantics for finite sequences.* Temporal logic is typically defined and used in the context of infinite length input sequences. This is one more reason it has not been used for conventional testing, which deals with finite length executions. Our finite-sequence temporal logic semantics is based on a 4-level logic using 4 possible pairs of Booleans in the form of <*Logic*, *Final*>. The *Logic* bit is the logical outcome of the temporal logic formula as evaluated *so far*. The *Final* bit provides extra information about possible extensions of the input sequence, i.e., *information about the future*. Consider for example the assertion: `Always x>0`. When evaluated every cycle (using *on-line* methods (2)), as long as `x>0` is true then the *Logic* bit is 1 while the *Final* bit is 0, representing the fact that the *Logic* outcome might change in the future. When `x==0` then `x>0` is false, the *Logic* bit turns 0, and the *Final* bit turns 1, representing the fact that the *Logic* result will never change in the future.

   Note that our support for real-time constraints exacerbates the need to the *Final* result bit. For example the traffic-light controller requirement `Always(isRED Implies Eventually` $_{C1<120}$ `isGREEN)` has a bounded eventuality requirement. When observing a sequence of *Logic* results bits, such as 00011000, the results do not provide information about the real-time constraint being violated or not; namely, the *Logic* bit result sequence is similar to the evaluation of the same requirement with no constraint. The *Final* bit however will be one if the eventuality is not satisfied within the specified time constraint.

5. *Code generation*. The ability to generate source code from a temporal formula has proven to be useful. For example, generated code can be placed *in process* on an embedded target for monitoring purposes even if the target cannot communicate with a remote temporal logic monitor. Similarly, generated code is useful for applications which use formal specifications inside exception handlers (see attached paper #P8). Using generated code, our technology supports both *remote and in-process monitoring.*

6. *Simulation*. As pointed out by Garth Watney of JPL in the attached feedback report (paper #P9), "simulation is essential". The inclusion of real-time and time-series constraints in temporal specifications exacerbates the need to visual simulation.

7. *Actions language.* Time Rover tools support an *action language* that enables the test engineer to specify actions that fire on success or fail (*Logic* bit) and *Final*-bit values of the temporal rule evaluation (see attached paper #P7). The DBRover supports a UI based action specification interface and e-mail notification actions, as well as custom actions.

8. *Serialization*. DBRover and Temporal Rover have the option to perform serialization of the monitored trace. Consider a very long execution being monitored on a remote monitor such as DBRover or PaX. Typically, for temporal logic monitors that do not preserve history traces (on-line monitors cannot preserve traces), if the remote monitor shuts down for any reason, it will loose its ability to recover its monitoring state, i.e., when restarted, temporal rule

evaluation must begin as if this is time 0. Serialization of the monitors composite state enables graceful recovery without loosing the ability to continue monitoring the rules from their last computed state.

Application domains

There are two main application domains for our tools: verification and run-time rule evaluation. Some *Verification applications* are the following:

1. *Testing*: using a temporal monitor to automatically detect errors in testing phase of the design, as used by JPL (paper #P9). The tools supports similar testing capabilities on a high-level algorithmic prototype (using Matlab interface) and in a rapid prototyping environment (paper #P3).
2. *Field monitoring*. This is an extension of (1) above, where temporal requirements are either evaluated in the field or communicated from the field to a remote monitor. Our tools support both options, and being on-line they have a higher likelihood of successfully monitoring an speedy embedded application for a long period of time. Note that field monitoring is important for the validation of real-time requirements.

Run time applications use temporal monitoring either to provide information about the observed system, or use the specification language as a high-level rules specification and development language. Some run-time applications are the following:

1. *Field monitoring for non-verification purposes.* This application is (2) above but applied in the actual deployed code setting, such as having the Mars Rover monitored while on Mars. The requirement for this application is true on-line and serializable monitoring capabilities.
2. True run-time applications such as *business rule checking, transportation and security rule checking, intrusion detection.* See papers #P4, P5, P6. When used for these applications, temporal logic is used as a *programming* language enabling fast and accurate rule development. The Temporal Rover and DBRover generate code that is then used like any C, C++, or Java code inside the application. The DBRover extends this capability by allowing dynamic rule modification without rebuilding the entire application.

Products

1. *Temporal Rover*: a code generator for the temporal logics described earlier (LTL/MTL/MTLS). Generates source code in C, C++, Java, VHDL, Verilog, with connectivity to Matlab and Ada. Generated code enjoys the properties described earlier. In addition, customizable actions are enabled in the program domain. See attached paper #P8.
2. *DBRover*: A graphical remote monitoring version of the Temporal Rover GUI, editor, and simulator. Temporal rules are developed graphically, using an IDE. Once developed, a graphical simulator enables thorough debugging of the requirement rules using graphical simulation. The DBRover then automatically

and transparently generates code (using the Temporal Rover), compiles and links the rules into the remote execution engine component of the DBRover. At this point the DBRover listens to messages communicated from the target application (via sockets, http, or serial port communication) and evaluates the rules accordingly. For real-time constraint evaluation it supports two forms of real-time measurement: server-side and client-side. For example, when connecting to a simulated application such as a Matlab simulation, real-time information is simulated real-time and must therefore be provided by the Matlab client application.

Being remote, the DBRover can monitor applications on a wide range of platforms and languages, such as: database, Windows, Linux, Unix, Java, VxWorks, Matlab, and Ada. The DBRover can be used for across three common design phases common at NASA: (i) Matlab modeling, (ii) Unix/PC simulation, (iii) embedded target code implementation (e.g., on VxWorks).

The DBRover can be used for verification purposes as well as for business and security rule checking, as described in the technical papers attached (papers #P1, P3, P4, P5, P6, P9).

The DBRover also integrates Havelunds PaX. The DBRover extends Pax's deadlock detection capabilities in the following ways: (i) *visualization*, where the DBRover uses UML MSC to visualize potential deadlocks (ii*) connection to embedded targets* such as VxWorks, for the detection of potential deadlock on an embedded target.

3. ATG-Rover and MC-Rover. The ATG-Rover automatically generates test sequences from formal specifications. This product is not used currently by customers. Rather, its underlying technology is being used by the MC-Rover, a model checker currently under development. The MC-Rover is *a real-time model-checker* (see attached paper #P2) capable of validating real-time properties of software systems. It is distinguished from other model-checkers (such as NASA ARC's -- Willem Wiser's JPF) in that:
    a. It is not limited to Java and does not require any customized VM or OS kernel.
    b. Its focus is on validation of MTL requirements.
    c. Optimization reduces the search space considerably. Unlike JPF, it is stateless (does not record state visitation for backtracking purposes); this enables it to consume far less memory and time than JPF.