The Temporal Rover and the ATG Rover

Doron Drusinsky
Time-Rover, Inc.
11425 Charsan Ln., Cupertino, CA 95014, USA
doron@time-rover.com, www.time-rover.com

Abstract. The Temporal Rover is a specification based verification tool for applications written in C, C++, Java, Verilog and VHDL. The tool combines formal specification, using Linear-Time Temporal Logic (LTL) and Metric Temporal Logic (MTL), with conventional simulation/execution based testing. The Temporal Rover is tailored for the verification of complex protocols and reactive systems where behavior is time dependent. The Temporal Rover generates executable code from LTL and MTL assertions written as comments in the source code. This executable source code is compiled and linked as part of the application under test. During application execution the generated code validates the executing program against the formal temporal specification requirements. Using MTL, real time and relative time constraints can be validated. A special code generator supports validation of such constraints in the field, on an embedded target.

1. Temporal Logic Overview

Temporal Logic [5] is a special branch of modal logic that investigates the notion of time and order. In [6], Pnueli suggested using Linear-Time Propositional Temporal Logic (LTL) for reasoning about concurrent programs. Since then, several researchers have used LTL to state and prove correctness of concurrent programs, protocols, and hardware (e.g., [2], [3], [4]).

Metric Temporal Logic (MTL) extends Temporal Logic with real time constraints and was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [1].

Throughout this article we will refer to TL assertions being LTL assertions with or without MTL extensions.

2. Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the propositional logic operators (and (&&), or (||), xor (^), not (!), etc.) there are four future-time operators and four dual past time operators. The following syntax is used by the Temporal Rover for these four operators:

- Always in the future, and Always in the past. You can use the English keywords *Always* and *AlwaysInThePast*, or a box ([], [-]) to represent the always operators.

- Sometime in the future, and Sometime in the past. You can use the English keywords *Sometime* and *SometimeInThePast*, or a diamond (<>, <->) to represent a sometime operators.

- Until (for the future), and Since (for the past). You can use the English keywords *Until* and *Since*, or U and S to represent the until and since operators.

- Next cycle (for the future), and Previous cycle (for the past). You can use the English keywords *Next* and *Previous*, or a circle ((), (-)) to represent these operators.

3. Metric Temporal Logic

Metric Temporal Logic (MTL) extends LTL by supporting the specification of relative time and real time constraints. All four LTL future time operators (*Sometime*, *Always*, *Until*, *Next*) can be characterized by relative time and real time constraints specifying the duration of the temporal operator.

The following are some examples of MTL constraints (text inside curly brackets implies a prepositional-logic formula):

1. $\text{Always}_{<10}$({readySignal == 1} Implies (){ackSignal == 0})
which reads: a*lways, within the next ten cycles, readySignal == 1 implies that one cycle later ackSignal == 0.*
2. $\text{Always}_{\text{timer1}[5,10]}$ ({readySignal == 1} Implies
$\text{Eventually}_{\text{timer2}>=\ 20}${ackSignal == 0})
which reads: *always, between 5 and 10 timer1 real time units in the future, readySignal == 1 implies that eventually, at least 20 timer2 real time units further in the future, ackSignal == 0.*

MTL constraints can specify lower bounds, upper bounds, and ranges for relative time and real time constraints. In the first example a relative time upper bound is specified for the always operator. It is relative in that the bound is counted in clock cycles (see *The Temporal Rover's Notion of Time* below). In the second example two real time constraints are specified, using virtual clocks named *timer1* and *timer2*. A separate *TRClock* statement maps these virtual clocks to system calls, system clocks or any other counting device.

4. Proprietary Operators

The following two operators are Temporal Rover specific and are not part of LTL or MTL as described in the literature. Both operators can be used only in the MTL form namely, with a constraint.

- ρ `RepeatedUntil`$_{\text{constraint}}$ φ. This operator counts the number of occurrences of ρ until the occurrence of φ. It is used with MTL syntax, such as ρ `RepeatedUntil`$_{<=5}$ φ which succeeds if and only if ρ occurred five times or less until the occurrence of φ. Note that the constraint (written as *<=5*) does not represent real-time or relative clock ticks but rather is the number of times ρ succeeds in a *final* way (finality issues are discussed in the sequel).

- `Repeated`$_{\text{constraint}}$ ρ. This operator counts the number of occurrences of ρ in the future. It is used with MTL syntax, such as `Repeated`$_{[4,7]}$ ρ which succeeds if and only if ρ occurred between four and seven times in the future.

5. Using the Temporal Rover

The Temporal Rover is a code generator whose input is a Java, C, C++, Verilog, or VHDL source code program, where TL assertions are written as source code comments. The Temporal Rover parser converts this program file into a new file, which is identical to the original file except for the TL assertions that are now implemented in source code.

5.1 Writing Temporal Logic Assertions within a Program

The following example illustrates the way assertions are written inside source code. A Traffic-Light Controller (TLC) function is repeatedly invoked by a scheduler. Being a typical state machine, in each invocation cycle it computes its next state based on it's current state and the values of variables at the time. The TL assertions describes temporal requirements for this TLC.

```
void myScheduler() {
      while (1) {
            … /* schedule various system modules */
            trafficLightController()
      }
…
void trafficLightController() {
… /* Traffic Light Controller functionality */

/* TRBegin
// Asserting that always, if light is Green, camera
// is not on  (not shooting), namely CameraOn == 0.
TRAssert{ Always({Color_Main == GREEN} Implies
                 {CameraOn == 0})
         }
=>
// These actions are customizable and provided by
// the user
  {
      printf("Assertion 1: so far: SUCCESS\n");
      printf("Assertion 1: so far: FAIL\n");
      TRProcessLastResult("Assertion1\n");
      printf("Assertion 1: DONE! your last result is FINAL!\n");
  }
TREnd */

} // end of trafficLightController function
```

The assertion has the following general syntax:

`TRAssert` {<TL-formula>} `=>` {<*Four optional actions*>}

The list of four optional actions are, in order:

1. Action to perform every cycle in which the TL formula succeeds.
2. Action to perform every cycle in which the TL formula fails.
3. Action to perform every cycle (see *The Temporal Rover's Notion of Time* below).
4. Action to perform every cycle in which the TL formula succeeds or fails in a final way (see *Finite Executions vs. Infinite TL Sequences* below).

5.2 The Temporal Rover's Notion of Time

In the Traffic Light Controller example above, the assertion is written inside the *trafficLightController* function. This location of the assertion within the program defines the clock tick for the TL assertion. A new cycle for this assertion is considered to occur when the assertions location in the code is reached in run-time. For example, an assertion

Always({Color_Main == GREEN} Implies Next {CameraOn == 0})

Within the *trafficLightController* function will consider the *Next* cycle to be the next time *trafficLightController* is reached.

5.3 Finite Executions vs. Infinite TL Sequences

TL is semantically defined for infinite sequences or at least for sequences whose end point is known. For example, the TL formula *Eventually {x>0}* semantically evaluates to true if there is a point within the input sequence where *x>0*. However, when executing an on-going reactive program it is not known a priori whether the program has ended yet or not. Hence, if x is 0 until cycle 1000 there are two possibilities:

- The program will not continue executing.
- The program will continue executing and possibly *x>0* sometime thereafter

.

In the first case *Eventually {x>0}* should semantically fail, whereas in the second it might succeed. The Temporal Rover cannot know in run time whether the program will continue executing or not. Therefore, the best it can conclude is that *so far, the assertion failed*. In contrast, when *x>0* this assertion succeeds and the Temporal Rover knows that no future value of x will be able to change this fact. This is the reason for the fourth optional Temporal Rover action. This action executes when the success/fail result is *final* and cannot change in the future.

Another example is *Always{x>0}*. It will succeed in a non final way as long as *x>0*. It will fail in a final way once x is not greater than 0.

5.4 The Embedded Systems Code Generator

Embedded systems applications face the following special constraints:

- Limited memory. The target application in many automotive, consumer, and communication applications is limited in RAM.
- The target application has real-time constraints.
- Due to the memory size limitations many embedded applications have no on-board operating system. Such systems have no file system for logging verification and test reports. In addition, they cannot perform dynamic memory allocation, which is essential for all TL implementations.

The Temporal-Rover has a special code generator targeted at embedded applications. The generated verification code has almost no memory overhead on the target, and performs overhead computation on a host PC. The generated verification code requires no OS on the target side

Using this code generator, a host computer performs almost all verification code. The target is required to perform only basic computations of propositional sub-formulae and to communicate the results to the host.

The host and target code segment communicate via serial port, RPC, or any other communication protocol, using a customizable protocol. Since all verification code is executing on host side, there is full access to a file system for logging and comparing test results.

5.5 Concurrency

The Temporal Rover supports assertions for concurrent systems using the following approach. Let A and B be two concurrent entities, such as processes, tasks, threads, modules, etc. Entity A may mark a block of code as a section, such as CriticalSectionA. Having done that, entity B can refer to this block of code in it's assertions. For example:

```
Eventually(
      {y==0} And
      ({x!=0) Until {TRWithin("CriticalSectionA")})
)
```

Where TRWithin is a special TemporalRover construct that examines, during execution whether entity A is indeed inside the marked block of code or not.

6. Complexity and Scalability

The Tableau method is commonly used by model-checkers for formal verification of TL properties. It was conceived as a method for solving the validity problem for LTL ([7]). As such, it is considered as a possible method for run-time evaluation of LTL. The Tableau method creates a non-deterministic state machine (called the Tableau) whose size is exponential in the size of the TL formula. For example, the size of the formula/assertion *Always (x==1 Implies Next y==1)* is 5, counting 3 operators and 2 propositions. The corresponding Tableau will therefore have 32 states. Being non-deterministic, a Tableau requires an implementation where most states are in memory every clock cycle. Obviously, such an exponential method is not scaleable, and cannot be used for the verification of large applications and protocols. Consider a temporal logic specification of a real-life protocol such as the PCI-bus protocol, with over 100 assertions, of length 15 each. Consider also an efficient Tableau implementation of 50 bytes per state. The run-time memory requirement will be 160MBytes, leaving almost no memory for the actual PCI-bus simulation.

Some PCI-bus assertions are listed on our Web site at http://www.time-rover.com, where it is evident that assertions are often of length 20 or more, requiring 1,000,000 states or more per assertion, implying memory consumption in the order of Gigabytes. Note that the size of the Tableau is directly related to speed in that it represents the number of states that need to be evaluated and processed every clock cycle.

The Temporal Rover does not use the Tableau method. The complexity of it's algorithm is n^2 thereby yielding much better scalability than the Tableau. This complexity is not in contrast with known exponential lower bounds for the TL validity problem, as the Temporal Rover does not attempt to solve the validity problem.

7. The ATG Rover

The ATG-Rover is a program that automatically generates test sequences that cover high-level specifications written in TL, using the same specification syntax used by the Temporal Rover.

The sequences generated by the ATG-Rover are well suited for testing a requirement driven application in that:

- Only sequences relevant to the specification requirements are generated thereby reducing the size of the test bench considerably.

- The generated sequences have the same flexibility as the high-level requirements. Hence for example, a requirement that *Every two consecutive Requests must be separated by an Acknowledgement* has the flexibility of allowing the *Acknowledgement* to appear any time between two consecutive *Requests*. Therefore, ATG should generate test sequences that pass for any combination of <*Request, Acknowledgement, Request*> triplet, no matter how far apart the events are from one another. Formally speaking, the ATG Rover needs to accommodate TL's inherent non-determinism.

- Redundant test sequences are not generated.

Rather than generating the test bench itself, the output of the ATG-Rover is a program that generates tests, given in source code. This eliminates the need to store a very large amount of tests, and enables the test engineer to modify the generated program so that only a particular subset of tests will be generated. The generated program becomes the driver of the system under test, for the purpose of testing particular units within the system. The ATG-Rover accommodates TL's inherent non-determinism by incorporating a TemporalRover verification cycle within each ATG-Rover test cycle.

The generated program generates all possible input sequences for a given maximal sequence length. For every time stamp within the sequence it generates all possible input combinations. For each such combination, the generated program does the following:

1. Feeds the unit under test with it's current inputs.
2. Fires the unit.
3. Fires the Temporal Rover to validate the unit's response against the specified response.

References

1. E. Chang, A. Pnueli, Z. Manna - Compositional Verification of Real-Time Systems, Proc. 9'th *IEEE Symp. On Logic In Computer Science*, 1994, pp. 458-465.

2. B. T. Hailpern, S. Owicki - *Modular Verification of Communication Protocols*. IEEE Trans of comm. **COM-31**(1), No. 1, 1983, pp. 56-68.

3. Z. Manna, A. Pnueli - *Verification of Concurrent Programs: Temporal Proof Principles*, Proc. of the Workshop on Logics of Programs, Springer Verlag, LNCS, 1981 pp. 200-252.

4. S. Owicki, L. Lamport - *Proving Liveness Properties of Concurrent Programs*, TOPLAS 4(3) (1982), 455-495.

5. A. Prior - *Past, Present and Future*, Oxford University Press, 1967.

6. A. Pnueli - *The Temporal Logic of Programs*, Proc. 181977 *IEEE Symp.. on Foundations of Computer Science*, pp. 46-57.

7. A. P. Sistla, E. M. Clarke - *The Complexity of Linear Propositional Temporal Logic*, Journal of the ACM **32** (1985), pp. 733-749.

Appendix: Assertion Examples

- ev1 and ev2 happen or do not happen simultaneously:

  Always ({ ev1 } Iff { ev2 })

- if ev1 then ev2 two cycles later:

  Always ({ ev1 } Implies (2){ ev2 })

- ev2 not before ev1:

  Always (Not{ev2} Until {ev1})

- ev2 within n cycles after ev1:

  Always ({ev1} Implies Eventually $_{<=n}${ev2})

- ev2 within n1 and n2 cycles after ev1:

  Always ({ev1} Implies Eventually $_{[n1,n2]}${ev2})

- ev2 any number of cycles after ev1:

  Always ({ev1} Implies Eventually {ev2})

- ev2 after n cycles of no ev1:

  Always $_{<=n}$Not{ev1} Implies Eventually $_{>n}${ev2}

- ev2 any number of cycles after ev1, with no ev3 in between:

  Always ({ev1} Implies ((Not{ev3} Until {ev2}) And

  Eventually {ev2}))

- ev1 after the last ev2 and before ev3:

  Always (Last{ev2} Implies Eventually ({ev1} And AlwaysInThePast Not{ev3}))

- if ev1, then ev2 must not occur for n cycles:

  Always ({ev1} Implies Always $_{<=n}$Not{ev2})

- if ev2, then ev1 must have occurred 3 cycles earlier:

```
Always ( {ev2} Implies Previous Previous Previous {ev1} )
```

- if ev2, then ev1 must have occurred sometime earlier (not including present time):

```
Always ( {ev2} Implies Previous SometimeInThePast {ev1} )
```

- evMid occured at least once between evStart and evStop:

```
Always ( { evStart } Implies {evMid}Before{evStop} )
```

- val==VAL between evStart and evStop:

```
Always ( { evStart } Implies ( {val==VAL} Until {evStop} ) )
```