Monitoring Temporal Rules Combined with Time Series

Doron Drusinsky[1,2]

[1] Naval Postgraduate School, Monterey, CA, USA
ddrusins@nps.navy.mil

[2] Time-Rover, Inc., 11425 Charsan Ln., Cupertino, CA 95014, USA
doron@time-rover.com, www.time-rover.com

Abstract. Run-time monitoring of temporal properties and assertions is used for testing and as a component of execution-based model checking techniques. Traditional run-time monitoring however, is limited to observing sequences of pure Boolean propositions. This paper describes tools, which observe temporal properties over time series, namely, sequences of propositions with constraints on data value changes over time. Using such temporal logic with time series (LTLD) it is possible to monitor important properties such as stability, monotonicity, temporal average and sum values, and temporal min/max values. The paper describes the Temporal Rover and the DBRover, which are in-process and remote run-time monitoring tools, respectively, that support linear time temporal logic (LTL) with real-time (MTL) and time series (LTLD) constraints.

1. Temporal Logic and Run-Time Monitoring Overview

Temporal Logic 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.

Linear-Time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the propositional logic operators there are four future-time operators and four dual past time operators: always in the future (always in the past), eventually, or sometime in the future (sometime in the past), until (Since), and next cycle (previous cycle). Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [1]. MTL extends LTL by supporting the specification of relative time and real time constraints. All four LTL future time operators can be constrained by relative time and real time constraints specifying the duration of the temporal operator. This paper described additional extension to LTL and MTL suitable for the specification of time-series constraints.

Run time Execution Monitoring (REM) is a class of methods of tracking temporal requirements for an underlying application. First applications of REM were verification oriented where REM methods were used to track whether an executing system conforms to formal specification requirements. Recent adaptations of REM methods enable run time monitoring for non-verification purposes such as temporal business rule checking and temporal security rule checking [5]. Unlike previously published

W.A. Hunt, Jr. and F. Somenzi (Eds.): CAV 2003, LNCS 2725, pp. 114–117, 2003.

methods [7], the newer methods are *on-line*, namely, temporal rules are evaluated without storing an ever growing and potentially unbounded history trace. The TemporalRover and DBRover tools described in this paper perform on-line REM using executable alternating finite automata. The technique enables on-line monitoring complex Kansas State Specification Pattern assertions at a rate of 6000 to 60,000 cycles per second on a 1GHz CPU [4], and is capable of monitoring past-time and future-time temporal logic augmented with real-time constraints, time-series constraints, and special counting operators described in [2]. High-speed on-line REM enables demanding applications such as formal specification based exception handling [3].

2. Run Time Monitoring Tools: The Temporal Rover and DBRover

The Temporal Rover [2] is a code generator whose input is a Java, C, C++, or HDL source code program, where LTL/MTL assertions are embedded 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 assertions that are now implemented in source code. The following example contains an embedded MTL assertion for a Traffic Light Controller (TLC) written using the Temporal Rover syntax asserting that *for 100 milliseconds, whenever light is red, camera s.b. on*:

```
void tlc(int Color_Main, boolean CameraOn) {
  … /* Traffic Light Controller functionality */
  /* TRBegin
   TRClock{C1=getTimeInMillis()} // get time from OS
   TRAssert{ Always({Color_Main == RED} Implies
               Eventually_C1<1000_{CameraOn == 1})
           } =>
      //         Customizable        user        actions
    {printf("SUCCESS");printf("FAIL");printf("DONE!");}
  TREnd */
} /* end of tlc */
```

The TemporalRover generates code that replaces the embedded LTL/MTL assertion with real C, C++, Java, or HDL code, which executes in-process, i.e., as part of the underlying application. The DBRover is a remote monitor version of the TemporalRover whereby assertions are monitored on a remote machine, using HTTP, sockets, or serial communication with the underlying target application.

3. LTL and MTL with Time Series Constraints (LTLD)

While LTL and MTL assert about sequences of pure Boolean propositions, it is often required to assert about sequences of propositions over time series, i.e., series of data values with constraints on the change of those values over time. For example, consider a requirement *R,* stating that *for one minute as of eventA, the value of variable x should be 10% stable.* Such a requirement combines MTL with propositions

based on temporal instances of a variable x. The need for such time series assertions typically involves the validation of statistical and algebraic artifacts such as stability, monotonicity, averaging and expectancy, sum and product values, time-series properties, min/max values, etc. Specific examples of such time series assertions are listed in the sequel.

Like LTL, LTL augmented with time series (LTLD) assertions are non-deterministic and might have multiple overlapping instances active simultaneously. For example, in requirement *R* above, the values of a same variable name *x* are referred to and compared with one another in multiple points in time, for a plurality of *eventA*'s, i.e., for a plurality of initial *x* values. One of many possible scenarios is where *eventA* occurs first when *x* =100, and then occurs again 30 seconds later when *x* =110; hence, in the overlapping 30 second time-segment, *x* values must range between 99 and 110. Clearly, the number and timing of *eventA* occurrences is unknown in advance, and the simple 1-minute end condition could, in general, be non-deterministic, rendering the task of monitoring all possible scenarios non-trivial.

LTLD enables the specification of requirements in which propositions include temporal instances of variables. Consider the following *automotive cruise control* code with an embedded stability assertion requiring speed to be 5% stable while cruise is set and not changed (uses TemporalRover syntax):

```
Void cruise(boolean cruiseSet, boolean cruiseChange,
boolean cruiseOff, boolean cruiseIncr, int speed){
     … /* Cruise Controller functionality */
    /* TRBegin
      TRAssert{Always ({cruiseSet}Implies
                 {speed*0.95<speed' && speed'<speed*1.05}
                 Until $speed$
                 {cruiseChange || cruiseOff}
                ) }=> {…} // user actions

    TREnd */
```

In the this example speed is a temporal data variable, which is associated with the Until temporal operator. This association implies that every time the Until operator begins its evaluation, possibly in multiple instances (due to non-determinism), the speed value is sampled and preserved in speed variable of this instance of the *Until* operator; this value is referred to as the pivot value for this *Until* operator instance. Future speed values used by this particular evaluation of the *Until* statement are referred to using the prime notation, i.e., as *speed'*, and are referred to as primed values. Hence, if speed is 100Kmh when *cruiseSet* is true, then the pivot value for speed is 100, while every subsequent speed value is referred to as speed' and must be within 5% of the (pivot) speed.

Note how *speed* is *declared* using the *$speed$* notation to be a temporal data variable associated with the *Until* operator. This declaration indicates to the Temporal Rover that it should be sampling a pivot value from the environment in the first cycle of the *Until* operators lifecycle, and to refer to all subsequent samples of speed as *speed'*.

Similarly, the following example consists of a *monotonicity* requirement for the cruise control system, where speed is monotonically increasing while Cruise Increase (*cruiseIncr*) command is active:

```
TRAssert {Always({cruiseIncr}Implies
                 {(speed<=speed') && (speed=speed')>=0}
                   Until       $speed$       {!cruiseIncr}
                )}=>  { => {…} // user actions
```

In this example the temporal data variable *speed* is sampled upon the *cruiseIncr* event, and is compared to the current value (*speed'*) every cycle. The latest speed value is then saved in the pivot for next cycle's comparison.

The following example consists of a temporal averaging and min/max requirement for the cruise control system, requiring that while cruise is set and unchanged the difference between average speed and minimum speed is always less than 1% of speed.

```
TRAssert{Always ({cruiseSet}Implies
           {(n++ >=0)&& ((sum+=speed') >= 0) &&
           ((average=sum/n) >=0) &&
           ((min=(speed'<min?speed':min) >=0) &&
           (average-min < speed'/100)
           }
     Until $speed,min=1000,n=0,average=0,sum=0$
           {cruiseChange || cruiseOff}
)}=> {…} // user actions
```

In this example the only data value that is sampled from the environment (the *cruise* method/function) is *speed*. All other pivots (i.e., for *min, n, average,* and *sum*) are initialized upon the construction of the *Until* object. Likewise, the only prime value that is sampled from the environment is *speed'*, whereas all other primed variables are assigned as specified in the assignment statements (e.g. average'=sum'/n'). The TemporalRover makes this distinction when is recognizes an assignment in the declaration statement, such as *sum*=0 above.

4. 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. D. Drusinsky - *The Temporal Rover and ATG Rover*. Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science, 1885, pp. 323-329.
3. D. Drusinsky - *Formal Specs Can Handle Exceptions*, CMP Embedded Developers Journal, Nov. 2001, pp., 10-14.
4. D. Drusinsky, *On-line Efficient Monitoring of Metric Temporal Logic Specifications using Alternating Automata*, submitted for publication.
5. D. Drusinsky and J. Fobes - *Real-time,On-line,Low Impact, Temporal Pattern Matching*, 7th World Multiconference on Systemics, Cybernetics and Informatics, Orlando FL, 2003; accepted for publication.
6. A. Pnueli - *The Temporal Logic of Programs*, Proc.18th IEEE Symp. on Foundations of Computer Science, pp. 46-57, 1977.
7. A. P. Sistla and O. Wolfson - *Temporal Conditions and Integrity Constraints in Active Database Systems*, Proceedings of the ACM-SIGMOD 1995, International Conference on Management of Data, San Jose, CA, May 1995.