

# Real-time, On-line, Low Impact, Temporal Pattern Matching

Doron Drusinsky

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

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

**Abstract.** A temporal pattern matching technique based on formal specifications using Linear Time Temporal Logic (LTL) is described. The method is based on Remote Execution and Monitoring (REM) of LTL assertions. Unlike verification applications where the formal specification is used to locate errors in a corresponding program, REM is not concerned with correctness but rather with temporal pattern detection. Unlike comparable techniques, such as SQL in temporal databases, the method is completely on-line, and does not require storage of the input sequence. This makes REM especially suitable for low-impact, real-time, temporal pattern matching of potentially never ending applications such as security monitoring and financial temporal business rule checking.

## Introduction

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. In [7], 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., [5], [6]).

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 (*Eventually*, *Always*, *Until*, *Next*) can be characterized by relative time and real time constraints specifying the duration of the temporal operator.

Run-time Execution and Monitoring (REM) is a method of tracking the behavior of an underlying application, such as an embedded system, financial database, or airline reservation systems. REM methods range from simple print-statement logging methods to run-time tracking of complete formal requirements (e.g. written in LTL/MTL) for verification purposes. Indeed, first applications of REM were for verification applications, where REM was and is used to track how formal specification requirements are conformed to by the actual executing system.

Recent adaptation of REM methods enable run-time monitoring for non-verification purposes, such as temporal business rule checking and temporal security rule checking. This adaptation uses REM methods as on-line, real-time, and low-impact temporal pattern matchers.

This paper describes the DBRover, an on-line, real-time, low impact REM tool based on LTL and MTL, and its application to temporal pattern detection for financial and security applications.

### **Online Temporal Pattern Detection**

Consider the following two airline security related temporal pattern rules, both concerned with detecting a foreign national male passenger with a student visa flying to the Harrisburg International Airport near the Three Mile Island nuclear power plant:

- R1. Detect such a passenger if he has traveled to the Middle East at least once within a year of obtaining his student visa.
- R2. Detect such a passenger if he has traveled to the Middle East at least once within a year of obtaining his student visa and he received two or more direct deposits from non-US banks within the last year.

Both rules describe temporal patterns that contain potentially discernable elements from an airline security system operating automatically and in real-time. The two primary methods for performing such temporal pattern detection are *off-line* and *on-line*, as described in the sequel.

The Federal Aviation Administration (FAA) has an automated profiling system originally termed Computer Assisted Passenger Screening (CAPS) [5] that relies upon the data in each Passenger Name Record. This profiling system is being upgraded to access a more extensive range of data. CAPPS-II will profile airline passengers based on secret criteria in order to identify potential terrorists. Personal information about passengers may additionally include that from INS, law enforcement, and customs. Having such history information stored in the system, or in constituent subsystems, enables SQL based implementation of a temporal pattern rule such as R1. We call such an implementation *off-line* because it relies on storing and querying historical information. An off-line solution induces the following three impact consequences.

1. Temporal, historical, information is stored within the system (e.g. within CAPPS-II and/or its constituent subsystems).
2. Temporal and non-temporal pattern detection is initiated by the security related query, querying the constituent resources at will. We regard this as *high-impact* solution because a temporal query is initiated from outside the original scope of the queried system, thereby impacting the performance of the queried system. For example, the INS subsystem of CAPPS-II being impacted by repeated external queries from CAPPS-II proper, which sooner or later will degrade the INS

system performance. Performance degradation will occur because of the actual query processing forced on the INS subsystem and because of the fact that with time, temporal queries can query monotonically increasing data-sets of historical data.

In addition to performance issues, CAPPs and its constituent subsystems need to agree on a shared data representation for merging query results from multiple subsystems (e.g. merging INS and law enforcement query results).

This paper is concerned with *low-impact, on-line* temporal pattern detection. It uses run-time temporal logic monitoring (REM) to detect temporal patterns without using historical data (i.e., it is *on-line*), and without querying the underlying application (i.e., it is *low-impact*). The only communicated information it requires from the underlying application (e.g. CAPPs-II constituent subsystems) are Boolean messages for basic propositions such as *deposit of more than \$1000 was made to account of SSN=222 11 2222*.

While pattern rule R1 described earlier is programmable within the suggested CAPPs-II framework, R2 requires extension which includes banking information. Such an extension however, will not lend itself to off-line temporal pattern detection methods, for the following reasons:

1. The banking data systems only store historical/temporal information for a limited duration (e.g. 3 months). The industry is unlikely to make any significant change to this policy.
2. Banking data systems are not likely to permit high-impact, CAPPs-II initiated, queries, because of the performance consequences discussed earlier as well as their own security need to be in full control over the content of any such query.

In contrast, a REM temporal pattern detection method, being on-line and low-impact, can be used in tandem with CAPPs-II, while supporting extensions that support rules such as R2.

## The Temporal Rover and DBRover

The Temporal Rover [3] 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 should be on*:

```
void tlc(int Color_Main, boolean CameraOn) {
... /* Traffic Light Controller functionality */
/* TRBegin
TRClock{C1=getTimeInMillis()} // get time from the OS
TRAssert{ Always({Color_Main == RED} Implies
```

```

        Eventually_C1<1000_{CameraOn == 1})
    } =>
// Customizable user actions
{printf("SUCCESS\n");printf("FAIL\n");printf("DONE!\n");}
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 Temporal Rover is also used for formal specification based exception handling [4].

The DBRover [5] is a REM version of the TemporalRover whereby assertions are monitored on a remote machine, using HTTP, sockets, or serial communication with the underlying target application. The DBRover includes a graphical temporal rule editor, a temporal rule simulator, and a temporal rule execution engine based on the TemporalRover code generator.

The DBRover is concerned with the following main objectives:

1. *Monitoring on-line*, namely no *postmortem* processing is used. A counter example would be to store all events in a database and use a SQL based method to query those tables at a later time. The motivation for this requirement is that no expected termination time for the underlying application (e.g. security application) should be assumed. With no expected termination time the size of the stored information will be monotonically increasing, which is unacceptable in most cases. The algorithm chosen for the DBRover is fully on-line and does not require monotonically increasing storage space.
2. *Low impact*. The DBRover does not interrogate the underlying application (e.g. the banking system in the R2 example). Rather, it listens to simple events pertaining to basic propositions, such as *deposit-occurred*, or *balance<0* which are sent to the DBRover by the underlying application via sockets or http.
3. *Rule flexibility*. Pattern specification rules, being domain specific and evolving, change frequently and are often written by domain experts, not programmers. The DBRover uses a GUI for LTL pattern rule entry; rules can be changed in the UI and automatically mounted to be monitored, all with almost no change to the application being monitored.
4. *Powerful rules language*. For the same reasons discussed earlier, pattern specification rules need to be able to capture real-life patterns and concerns, such as real-time constraints, all while being close to natural language. LTL satisfy this requirement; a large body of research points to its expressiveness and usefulness as a specification language. MTL adds real-time constraints to LTL specifications.

The DBRover listens to sockets, http, or serial communication messages sent to it from the main application (APP), such as a banking application. Messages are organized in streams that represent sequences of events or conditions in APP. Every cycle, such as whenever the bank account balance changes, the APP send 1 bit per basic proposition (e.g. deposit of more than \$500 made to account) and possibly an ID of the underlying entity being tracked (e.g. SSN or bank account number) indicating whether that proposition is true or false in that particular cycle. The DBRover will repeatedly re-evaluate the temporal pattern rule, either in a single instance, or using multi-instancing (e.g., one instance per SSN or bank account number). An administrative part of the DBRover can be programmed to send out e-mail's or to invoke custom actions (e.g. external, user written program or script) based on the success or failure of chosen rules.

## 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. Fobes, J. L. Computer Assisted Passenger Screening (CAPS), DOT/FAA/AR-96/38, 1996.
3. D. Drusinsky, *The Temporal Rover and ATG Rover*. Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science, 1885, pp. 323-329.
4. D. Drusinsky, *Formal Specs Can Handle Exceptions*, CMP Embedded Developers Journal, Nov. 2001, pp., 10-14.
5. B. T. Hailpern, S. Owicki - *Modular Verification of Communication Protocols*. IEEE Trans of comm. **COM-31**(1), No. 1, 1983, pp. 56-68.
6. Z. Manna, A. Pnueli - *Verification of Concurrent Programs: Temporal Proof Principles*, Proc. of the Workshop on Logics of Programs, Springer LNCS, 1981 pp. 200-252.
7. A. Pnueli - The Temporal Logic of Programs, Proc. 181977 IEEE Symp.. on Foundations of Computer Science, pp. 46-57.