Journal of Universal Computer Science, vol. 12, no. 5 (2006), 482-498
submitted: 21/10/05, accepted: 27/4/06, appeared: 28/5/06 © J.UCS

On-line Monitoring of Metric Temporal Logic with Time-Series Constraints Using Alternating Finite Automata

Doron Drusinsky
(Naval Postgraduate School, Monterey, CA, USA
ddrusins@nps.edu
and
Time Rover, Inc., Cupertino, CA, USA
www.time-rover.com)

Abstract: In this paper we describe a technique for monitoring and checking temporal logic assertions augmented with real-time and time-series constraints, or Metric Temporal Logic Series (MTLS). The method is based on Remote Execution and Monitoring (REM) of temporal logic assertions. We describe the syntax and semantics of MTLS and a monitoring technique based on alternating finite automata that is efficient for a large set of frequently used formulae and is also an on-line technique. We investigate the run-time data-structure size for several interesting assertions taken from the Kansas State specification patterns.

Keywords: formal specification, temporal logic, run-time monitoring, run-time verification, alternating automata, time series, assertions, reactive systems
Categories: F.4.1, F.1.1, D.2.4

1 Introduction

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. It was Pnueli [17] who first suggested using Linear-Time Temporal Logic (PLTL) for reasoning about concurrent programs. Since then, several researchers have used PLTL to state and measure the correctness of concurrent programs, protocols, and hardware (e.g., [11, 16]).

PLTL is an extension of propositional logic in which, in addition to the well-known propositional logic operators, there are four future-time operators (◊-Eventually, □-Always, U-Until, O-Next) and four dual-past time operators. A well known library known as the Kansas State University (KSU) specification patterns library, contains patterns of PLTL specifications that encode the knowledge of experts in finite state verification [2].

Chang, Pnueli, and Manna suggested Metric Temporal Logic (MTL) as a vehicle for verifying real time systems [3]. MTL extends PLTL by supporting the specification of relative-time and real-time constraints. Using MTL, all four PLTL future-time operators can be characterized by relative-time and real-time constraints specifying the duration of the temporal operator.

In [1], Alur and Henzinger classify a variety of real-time logics according to their complexity and expressiveness. In particular, they investigate the expressive power of MTL and Timed Temporal Logic (TPTL) in which a freeze quantifier is used for

freezing and capturing particular real-time values for later use within the TPTL formula.

Time series constraints enable the specification of temporal properties of sequences of propositions with constraints on how data values change over time. They also enable the simulation and monitoring of such properties as stability, monotonicity, temporal average and sum values, and temporal min/max [6]. Note that time-series constraints differ from the freeze operator of [1] in that they capture and freeze data values, not time.

Remote Execution and Monitoring (REM) is a class of methods for tracking the temporal behavior of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complete formal requirements (written, e.g., in PLTL/MTL) for verification purposes. Recently, NASA used REM to verify the flight code for its Deep Impact project [8]. In addition, the U.S. Missile Defense Agency (MDA) is currently applying REM to verify its Ballistic Missile Defense System [9]. Both applications use the REM method described here.

In this paper we are particularly interested in on-line REM methods, where temporal rules are evaluated without storing an ever-growing and potentially unbounded history trace. In particular, we prove that for a large subset of temporal logic, the method requires only a polynomially sized implementation. In addition, we show the actual expected implementation size for several interesting assertions taken from the Kansas State specification patterns.

Recently, Thati and Rosu [18] have proven lower bounds for on-line REM of MTL assertions, showing that monitoring time grows exponentially to the size of the assertion being monitored when real-time constraint bounds are small, and double exponentially to the size of the assertion being monitored in the general case. Note that these lower bound refer to the amount of computation required during every cycle of a REM process.

Also, whereas Sistla and Wolfson investigated temporal rule checking in databases [20], our approach differs in the following two aspects: First, our implementation algorithm is based on executing and efficiently managing and reducing alternating finite automata (AFA) representations of temporal rules. Second, our AFA-based and/or tree reduction techniques enable on-line computations, whereas their algorithm is not on-line in that their requirement graph data structure grows with time. It should be noted, however, that in [19], Sistla and Wolfson provide an on-line (incremental) variant of their algorithm by limiting themselves to past-time temporal logic.

In [14] Kovacs, et-al, describe the use of PLTL formulas as runtime assertions in a parallel debugging environment. Their approach, like ours, distinguishes between transient evaluations of temporal assertions and “final” evaluations that cannot change in the future. They do not however support time-series within PLTL nor do they describe the details of their monitoring technique.

In [21] Tuzhilin describes Templar, a high-level simulation language based on temporal logic. The suggested interpreter is not on-line and relies on stored historical data. Also, Templar does not support MTL or time-series constraints.

In [12] Hevelund and Rosu describe the Java Path Explorer (JPaX), an on-line implementation method for PLTL based on a rewriting system using PLTL recurrence equations. The primary disadvantage of JPaX compared with the technique we

suggest in this paper is its inability to monitor MTL assertions or time-series constraints [6]. In addition, this paper provides analysis, growth metrics, and growth reduction techniques that have not been provided for in alternative techniques.

Tableau methods for PLTL (e.g., [22]), and for TPTL [1], are in effect on-line methods: They convert PLTL into exponential-sized non-deterministic finite automata (NFA), which are then executable and usable as a REM engine. Our method, however, uses AFA instead of NFA, yielding substantially smaller implementations. More importantly, the AFA method lends itself to the implementation of the sorts of PLTL extensions described here, such as real-time constraints (MTL) and time-series constraints, as well as counting operators [4].

Extended regular expressions are those that, as their name suggests, extend regular expressions with negation and conjunction. They have a direct automata theoretical representation using AFA, analogous to the way that non-deterministic automata represent standard regular expressions. For example, Perl and recent versions of Java support extended regular expressions in an off-line manner, in which the entire input string is stored in memory (e.g. as a string variable) and then checked for membership in a formal language defined by a given extended regular expression. Indeed, [15] presents a polynomial time off-line membership algorithm for extended regular expressions.

The DBRover and Temporal Rover REM tools described in [4-10] use the AFA-based implementation technique described in this paper. In addition to on-line processing they are [redundant] low-impact [7] in that they require only limited exposure of potentially confidential information on the monitored system, a useful property when monitoring financial or security-based systems.

The rest of this paper is organized in this way: Section 2 describes the syntax and finite sequence semantics for MTLS. Sections 3 and 4 outline our AFA-based REM method for PLTL. Sections 5 and 6 provide analysis and metrics for the growth of the data structures used by the suggested REM method, and Section 7 examines these growth functions for several specifications taken from the KSU specification pattern library. Section 8 describes an adaptation of the REM method for monitoring MTLS and past-time operators.

2 MTL with Time Series Constraints (MTLS): Syntax and Semantics

Conventionally, the semantics of PLTL and MTL are defined over infinite sequences. Run-time monitoring is, however, by definition a finite process. We therefore define MTLS semantics for finite sequences. We do so by using an automata-like approach, in which an MTLS formula evaluates (accepts or rejects) an input string, σ, over some alphabet, Σ. In addition, we extend the definition of a sequence, $\sigma=a_1 \ldots a_n$, to be a sequence of pairs from $\Sigma \times T$, i.e. $\forall_i$, $0 \le i \le n$, $a_i=(b_i, t_i)$, where $b_i \in \Sigma$ and t_i is an integer that represents the arrival time of b_i at the input. We denote the projections $a_i|\Sigma=b_i$ and $ai|T=t_i$. Clearly, also, $i \le j \rightarrow t_i \le t_j$.

Because input sequences are finite, we introduce the concept of the *finality* of an evaluation, which indicates whether the Boolean evaluation result has the potential of changing when using an input string, σ', that is an extension of σ. For example, p = □

(x>0) accepts a sequence σ of 100 time stamps with x>0 in each. The final value for this evaluation is *false* because x≤0 might occur in an extended sequence σ' that extends σ beyond time t=100. However, if x≤0 at some time t<*100*, then p is rejected and the final value is *true* because this rejection is unchangeable in the future. That is to say, every extension of the input sequence will not change the fact that □ (x>0) has been violated.

Let P be a set of Boolean propositions, V a set of variables called *frozen variables*, and V'={v'/v∈ V} a set of variables called *current variables*. Let Q be a set of time-series propositions defined as Boolean relations over arithmetic expressions of numeric constants and variables from V∪V'. For example, using V={v_1,v_2,v_3}, v_1+v_2'*0.9<2+v_3 is a legal time-series proposition. The formulae ϕ of MTLS are defined inductively as follows:

ϕ := p | false | $\phi_1 \vee \phi_2$ | ¬ϕ | Oϕ | $\Diamond_I$ $x_1,\ldots,x_m$ ϕ | ϕ_1 U_I $x_1,\ldots,x_m$ ϕ_2

for p∈ P∪Q and $\forall_i$, 0≤i≤m, x_i∈ V. As in [1], the subscript I is an interval of □ whose end points are natural number constants. Intervals may be open, half open, or closed, as well as empty, bounded, or unbounded. We use standard pseudo-arithmetic expressions to describe such intervals. For example, ≤ c_1 and > c_2 stand for the closed interval [0,c_1] and the open interval (c_2, ∝), respectively. An interval (c, ∝) or [c, ∝) is called right unbounded. We use the standard abbreviations: true = ¬false, $\phi_1 \wedge \phi_2$ = $\neg(\neg\phi_1 \vee \neg\phi_2)$, and $\Box_I$ $x_1,\ldots,x_m$ ϕ = $\neg\Diamond_I$ $x_1,\ldots,x_m$ ¬ϕ. For readability reasons we abbreviate $x_1,\ldots,x_m$ as $\underline{x}_m$ throughout the remainder of this paper.

An example of an MTLS formula is $\Diamond_{[10,500]}$x$ (x>30 ∧ $\Box_{\leq 20}$ x'>1.5*x), which states that sometime between 10 and 500 real-time units in the future it should hold that (i) the value of x is a number greater than 30, e.g. 35, and (ii) the value of x in every cycle within the 20 consecutive real-time units is greater than 1.5*35. Note how x, a frozen variable, denotes the initial value of x within a sub-sequence of cycles while x', a current variable, denotes values of the same variable x in the following cycles. Note also that the sub-formula $\Box_{\leq 20}$ x'>1.5*x has no frozen variables of its own.

Frozen and current variables are similar to free and bound instances of variables found in other languages and tools (e.g. [1]). Our definitions differ somewhat from the conventional definitions in the following respect. Frozen and current variables such as x and x' in the above example are defined under a particular temporal operator (◊ in the above example). The frozen variable x is frozen in the initial cycle of the sub-sequence for that operator (the first time between time 10 and 500 in which x is greater than 30) whereas the current variable x' refers to the value of x in following cycles. Note that the motivation for using the same variable name for frozen and current variables is that from a programmers standpoint they look like the same programming language variable x being observed in different snapshots.

We consider MTLS formulae as acceptors of sequences of an alphabet Σ whose letters are as follows. Each letter of Σ is a collection of functions, each being one of three types: a:P->Boolean, a:V→□, and a: V'→□. We distinguish between these functions using their argument, one takes a proposition as an argument while other two take a variable as their argument. As in [19], the semantics of MTLS is defined with respect to a time-stamped sequence, i.e., finite input sequence σ= $a_0 \ldots a_n$, where a letter a_i is interpreted as representing the following readings for the i^{th} cycle: (i) truth

assignments for Boolean propositions in P, and (ii) integer values of variables in V such as $a_i(v_1)=8$ and $a_i(v_2)=10$.

Let q be an expression in Q. We define the time series substitution of a variable in q with an integer value n, denoted q<v←*n*>, as the expression q where all instances of v are substituted with *n*. Because we have two types of variables, frozen and current, we define a similar substitution q<v←n_1, v'←n_2> in which both types of variables are substituted. For an input letter $a\in\Sigma$, q←$\$\underline{x}_m\$a$ is the expression q with all occurrences of x_i replaced by $a(x_i)$. Likewise, q←$\$\underline{x}_m\$$<a,b> is the expression q where $\forall i,\ 1\le i\le m$, all instances of a frozen variables x_i are substituted with $a(x_i)$, and all instances of a current variable x_i' are substituted with $b(x_i')$. In the following semantics we use the substitution q←$\$\underline{x}_m\$$<a_i,a_j> (where $i\le j$) in the following way: a_i and a_j are letters in Σ, where a_i is a reading of the input tape used for setting values of frozen variables, and a_j is a reading of a subsequent letter used for setting current variables.

For the purpose of creating simple semantics, we impose the following syntactic constrains: a variable x in the variable list ($\$\underline{x}_m\$$) of a formula cannot appear in the variable list of any sub-formula. For example, $\Diamond_{[10,500]}\$x,y\$$ (x>30 $\wedge$ $\Box_{\le 20}$ $\$y\$$ x'+y'>1.5*x+y) is illegal.

The finite-sequence semantics for an MTLS formula *p* is defined recursively with a standard evaluation (accept or reject) value, where p rejects a sequence of elements of Σ×T, $\sigma=a_1\ldots a_n$, if it does not accept it. We denote the acceptance relation as σ|=p for p accepting σ, and as σ|≠p for p rejecting σ.

- For a formula $p\in P$, $\sigma\models p$ iff $a_0|\Sigma(p)$ contains *p*.
- σ|=p ∨ q iff σ|=p or σ|=q.
- σ|=¬p iff σ|≠q.
- σ|=○ρ iff $a_{1\ldots}a_n$ |=p.
- σ|=$\Diamond_I\$\underline{x}_m\$$p iff $\exists i,\ 0\le i\le n$, such that: (i) $a_{i\ldots}a_n$ |= p←$\$\underline{x}_m\$$<a_0|Σ, a_i|Σ> and (ii) $a_i|T\in I$.
- σ|=ρ$U_I\$\underline{x}_m\$$q iff one or both of the following conditions hold:
  1. $\exists j,\ 0\le j\le n$, such that $a_{j\ldots}a_n$|=q←$\$\underline{x}_m\$$<a_0|Σ,a_j|Σ>, $a_j|T\in I$, and $\forall i,\ 0\le i<j$, $a_{i\ldots}a_n$|=p←<a_0|Σ, a_i|Σ >.
  2. *I* is right unbounded and $\forall i,\ 0\le i\le n$, $a_i|T\in I$ implies that $a_{i\ldots}a_n$|=p←$\$\underline{x}_m\$$<a_0|Σ, a_i|Σ>.

In addition, we define the Boolean finality qualifier, where the final value of an MTLS formula p, given an input sequence σ, is denoted f(p,σ). When f(p,σ) is true it means that the decision of whether p accepts σ is immutable.

- For a formula $p\in P$, f(p,σ)=*true*.
- f(p ∨ q)=*true* if at least one of the following conditions holds: (i) σ|=p and f(p,σ)=*true;* (ii) σ|=q and f(q,σ)=*true;* or (iii) σ|≠p and f(p,σ)=*true* and σ|≠q, and f(q,σ)=*true*.
- f(¬p, σ) = f (p,σ).
- f(○ρ, σ) = f(p,σ) iff |σ|>1.
- f($\Diamond_I\$\underline{x}_m\$$p, σ) is *true* if $\exists i,\ 0\le i\le n$, such that: $a_{i\ldots}a_n$ |= p←$\$\underline{x}_m\$$<a_0|Σ, a_i|Σ>, f(p←$\$\underline{x}_m\$$<a_0|Σ, a_i|Σ>, $a_{i\ldots}a_n$), and $a_i|T\in I$. Stated informally, f($\Diamond_I\$\underline{x}_m\$$p, σ) is

true if *p* with substituted variables is true in a *final* way sometime in the future.

$f(\Diamond_I \$\underline{x}_m \$p, \sigma)$ is also *true* if $\forall i$, $0 \le i \le n$, $a_i|_T \in I \rightarrow a_{i...}a_n| \neq p \leftarrow \$\underline{x}_m \$ < a_0|\Sigma, a_i|\Sigma >$ and $f(p \leftarrow \$\underline{x}_m \$ < a_0|\Sigma, a_i|\Sigma >, a_{i...}a_n)$. Otherwise $f(\Diamond_I \$\underline{x}_m \$p, \sigma)$ is *false*. Stated informally, $f(\Diamond_I \$\underline{x}_m \$p, \sigma)$ is true if *p* with substituted variables is false in a *final* way always in the future.

- $f(\rho U_I \$\underline{x}_m \$q, \sigma)$ is *true* if $\exists j$, $0 \le j \le n$, such that (i) $a_{j...}a_n| = q \leftarrow < a_0|\Sigma, a_j|\Sigma >$, $f(p \leftarrow < a_0|\Sigma, a_j|\Sigma >, a_{j...}a_n)$, $a_j|T \in I$, and (ii) $a_{i...}a_n| = p \leftarrow \$\underline{x}_m \$ < a_0|\Sigma, a_i|\Sigma >$ and $f(p \leftarrow \$\underline{x}_m \$ < a_0|\Sigma, a_i|\Sigma >, a_{i...}a_n)$. Stated informally, $f(\rho U_I \$\underline{x}_m \$q, \sigma)$ is true if q (with substituted variables) is true in a *final* way sometime in the future and all instances of p (with substituted variables) up to that point in time are true in a *final* way.

  $f(\rho U_I \$\underline{x}_m \$q, \sigma)$ is also *true* if *I* is right unbounded and $\forall i$, $0 \le i \le n$, $a_i|T \in I$ implies that $a_{i...}a_n| = p \leftarrow \$\underline{x}_m \$ < a_0|\Sigma, a_i|\Sigma >$ and $f(p \leftarrow \$\underline{x}_m \$ < a_0|\Sigma, a_i|\Sigma >, a_{i...}a_n)$.

  $f(\rho U_I \$\underline{x}_m \$q, \sigma)$ is otherwise *false*.

For practical implementation reasons beyond the scope of this paper our tools use the following *relaxed finality* definition for a formula $p \vee q$: $f(p \vee q, \sigma) = f(p, \sigma) \wedge f(q, \sigma)$. This relaxation induces a final value definition that is not complete, i.e., there can exist a PLTL formula and a sequence σ such that $f(p, \sigma)$ is true according to the formal definition but false according to the relaxed definition.

3 From Temporal Logic to AFA

An AFA is a finite automaton with two types of states, *and*-states and *or*-states. Whereas the computation of a deterministic or non-deterministic finite automaton is a *sequence* of states, a computation of an AFA is represented as a tree of *and-or* states. A *trace* is a sub-tree of the computation *and-or* tree such that every child of an *and*-state is in the trace and some child of an *or*-state is in the trace. A trace is *accepting* if all its leaves are final states. An AFA accepts a sequence, σ, if an accepting-trace for σ exists.

In this paper we consider AFA augmented with *zero-delay* transitions similar to ε-transitions of [13], i.e., transitions that are traversed without reading a symbol from the input tape. Unlike ε-transitions, however, zero-delay transitions must be traversed, i.e., they do not imply a non-deterministic possibility of being either traversed or not.

The temporal logic monitoring method described in this paper is based on executing equivalent AFA representations. A graph-oriented AFA-based technique was selected over string-based logic manipulation of recursive Boolean logic equations because of its visual appeal, and because the AFA method lends itself to extensions of PLTL like real-time constraints and time-series constraints.

Our (recursive) PLTL-to-AFA conversion method is based on the well-known recurrence equations:

- $\Box p = p \wedge \bigcirc \Box p$
- $\Diamond p = p \vee \bigcirc \Diamond p$
- $p \; U \; q = q \vee (p \wedge \bigcirc (p \; U \; q))$

Fig. 1 contains the recursive AFA construction procedure for PLTL based on these recurrence equations, and Fig. 2 contains two examples of PLTL to AFA translation. We use the following notation: Transitions labeled Σ are those that are enabled by all letters of the alphabet, and zero-delay transitions are visually depicted using dashed lines. Also, by convention, we consider the top-most state as the initial state. For clarity, AFA states are sometimes annotated with their PLTL logic symbols, using bubble callouts, as in Figs. 1c, 1d, and 1g. We refer to such a symbol as the type of the node. For example, the initial state in the AFA in Fig. 2b is an and-node whose type is □. Note that in Fig. 1 we assume that PLTL formulae contain no negations because, using conventional logic conversions and temporal logic conversions, negations can be pushed down to the proposition level.

4 Runtime Monitoring using AFA

This section describes an AFA based-technique for run-time monitoring of PLTL and MTLS.

4.1 Runtime Monitoring of PLTL using AFA

Given an PLTL formula p and its representative AFA, our REM method constructs and maintains an *and/or promise tree*, abbreviated as *p-tree*, which is an evolving AFA computation tree. The p-tree is the only composite state that is preserved between cycles by the REM method. No history trace of input information is stored. The p-tree performs the on-line evaluation of the logical value of p at any given time *t*. The p-tree then dynamically reconfigures itself based on the current on-line inputs, thereby generating a new p-tree that represents a Boolean function to be evaluated at time *t+1* (the *promise*). Fig. 3 illustrates the process for the AFA in Fig. 2b. We use a diagrammatic notation in which, for a node whose PLTL-type is *N* (e.g. a □ node), its *recursive copy*—namely a child node with the same PLTL type *N*—is the right hand-side child. For example, the □ root node of the AFA in Fig. 2b, whose corresponding p-tree is illustrated in Fig. 3a, has a recursive copy on its right-hand side child. Both □ nodes in the p-tree in Fig. 3a implement a □ node in the AFA: The first p-tree realizes the AFA node during the first visitation, and the second p-tree realizes the □ node as it is revisited after traversing the loop transition.

The p-tree is used for two purposes: One is for the evaluation of the acceptance and final values every cycle; the other is as a data structure that preserves the state of the evaluation for future cycles.

During the i^{th} cycle of performing REM of an PLTL formula p, $i \geq 0$, the p-tree is evaluated resulting in Boolean values for $\sigma|=p$ (acceptance) and $f(p,\sigma)$ (finality), where $\sigma= a_i \ldots a_n$. During p-tree evaluation, acceptance and final values are assigned in a bottom-up manner, where leaves hold the following values:

- A leaf holding a *Boolean* **T**(**F**) value, such as in Fig. 3b, has a *true* (*false*) acceptance value and a *true* final value.
- An unexpanded leaf (e.g. the unexpanded U_1 leaf in Fig. 3a) has an acceptance value that is *neutral* with respect to its least common Boolean parent. For example, in Fig. 3a, the parent of the U_1 leaf is an *and* node, so

the acceptance value of the U_1 node is *true*, whereas if the parent were an *or* node the leaf's acceptance value would be *false*. The final value of an unexpanded leaf is *false*, representing the fact that the node has yet to be expanded.

Internal p-tree nodes are assigned acceptance values as a straightforward Boolean function of the acceptance values of their children. An internal node is assigned a true final value if and only if all its children have been assigned a true final value. As stated earlier, we use a relaxed final value. Consequently the final value is incomplete; that is, an assigned final value might be false when, in fact, according to the formal semantics, it should be true.

Because a node with a true final value has an acceptance value that cannot change in the future, then the REM method replaces this node with a singleton **T** or F node representing the node's acceptance value.

After evaluation, the p-tree is modified to preserve a promise for the next cycle as follows: In Fig. 3, the original p-tree in Fig. 3a contains the computation that needs to be performed in the first cycle (time t=0). Note how the U1 leaf node is unexpanded. It will be expanded in the next cycle (Fig 3b) once the Σ transition is traversed, and therefore represents a promise for a computation in the future. Similarly, there exists an unexpanded □ node (a recursive copy of the root), which represents the loop in the original AFA in Fig. 2b. It will likewise be expanded in cycle 2 (Fig. 3b) once the Σ transition is traversed. Recall that solid line transitions consume one cycle of computation (i.e., they represent delays and conditions), whereas dashed transitions are traversed instantly (i.e., in the current cycle) and unconditionally. The p-tree in Fig. 3b represents the computation that needs to be performed in the second cycle (time t=1), assuming p was true at time t=0. Note, as well, the following: (i) instead of a solid transition labeled p in Fig. 3a, representing the fact that p needs to be evaluated in the next cycle, Fig. 3b contains the **T** singleton node, representing the fact that p has been evaluated already and found to be true; and (ii) transitions that have already been traversed are replaced with ε-transitions (dashed lines), representing conditions that have already been evaluated.

4.2 Runtime Monitoring of MTLS using AFA

A beneficial property of our REM method is its direct realization of the original temporal logic formula using *and-or* trees that are homomorphic to the original AFA, i.e., to the original PLTL formula. This property is helpful when implementing extensions and constraints for PLTL.

MTL includes real-time constraints that are associated with temporal operators. For example ψ = □(p => ◊<5 q), specifies that whenever p is true, then within five real-time units q must be true. Let N be the AFA node for the ◊ operator. To implement the real-time constraint, the following steps are taken:

1. Every object that realizes a p-tree for *N* is provided with a real-time counter that is updated from a designated real-time clock. which measures true real-time using native operating system calls, clock cycles, or simulated time. The real-time counter for a p-tree is initialized when the p-tree is created.

2. As long as the real-time counter has not crossed the constraint's lower bound (0, in our example), the p-tree's acceptance value is *false* (*true* for a □ node), and the final value is *false*.
3. Once the real-time counter has entered the range between the lower and upper bounds of the constraint, normal evaluation of the acceptance and final values begins.
4. If the real-time counter exceeds the constraint's upper bound, the final value for the p-tree is assigned *true*.

The time-series-constraint implementation technique also uses p-trees to store information. Consider, for example □x(p=>□$_{<5}$ x'>=2x), which states that whenever p is true at some time t, then for every cycle between t and t+4 the value of x should be at least twice its value at time t. Let N be the AFA node for the outer □ operator. Every p-tree for N is assigned storage space for x. This storage space is assigned the value of x at the time of construction of a p-tree for N. This value, as well as current values of x (referred to as x'), are then used in the expression x'>=2x.

The suggested REM method supports past time operators as follows: Consider, for example, the formula ○○[-]p. A special p-tree is created for every sub-formula headed by a past time operator, e.g., for [-]p. Rather than looking backward in time while evaluating [-]p, the REM method evaluates □p as of time 0, but it does not use the results of this evaluation until the delays introduced by the ○operators have elapsed. At this point, the □p p-tree is assigned a true final value and is used under the future time nodes as a basic proposition.

5 p-tree Growth Control and Analysis

Clearly, the method as shown thus far induces p-trees whose size increases over time, and are therefore not on-line. To achieve on-line capabilities, our REM method utilizes *collapse* optimization operations in which sub p-trees are replaced by smaller, logically equivalent, p-trees. Collapse operations are designed for all recursive future time temporal operators (□, ◊, and *U*). They identify and eliminate repetitive and redundant substructures in p-trees by identifying patterns of *isomorphic sub-p-trees.* Collapse operations are performed in a bottom-up manner so that collapsing lower level substructures increases the potential for identifying repetitions in higher-level substructures.

In this paper, we identify three classes of collapse operations. Some collapse operations are general; others are helpful for restricted types of PLTL formula, designated two-color and discussed in Section 6. The three classes are, simple, extended and special collapse operations. These classes are not necessarily mutually exclusive: a given temporal formula might have restricted sub-formulae and can therefore benefit from special collapse operations whereas other parts of the formula induce less efficient sub p-trees.

Fig. 4 illustrates simple collapse operations performed during REM of the PLTL formula □(p=>◊q), i.e., □ (¬p∨◊q), which is identical, except for the basic propositions, to the formula in Fig. 2a. As a result of the collapse operations, the p-tree in Fig. 4d is identical to the p-tree in Fig. 4b, the p-tree one cycle earlier. Hence,

in this example, the p-tree did not actually grow in size from cycle 1 to cycle 2. In the following sections we will analyze the growth of p-trees for various types of PLTL formulae.

Fig. 5 illustrates p-tree collapse patterns used for the ◊, □, and U operators, where the collapse patterns for the □ operator are analogous to those of the ◊ operator. Not illustrated in Fig. 5a are trivial collapse operations for ◊ (□) nodes when either A or A' is final, i.e., **F** or **T** singletons. Fig. 5a illustrates simple collapse operations where the two isomorphic sub-trees represent first-degree cousins; Fig. 5b illustrates extended collapse operation for ◊ (□) nodes, i.e., collapse operations where isomorphic sub-trees might be more remote than first-degree cousins. Extended collapse operations exist for U-based p-trees as well, namely, when referring to Fig. 5c, the pair of p-trees A', B' is not necessarily a first-degree cousin of the pair A, B, but is possibly an nth-degree cousin. For example, the logic representation of an instance of an extended collapse operation for a U node using n=3 is:
$q \vee (p \wedge (r \vee (s \wedge (q \vee (p \wedge \text{promise}))))) = q \vee (p \wedge (r \vee (s \wedge \text{promise})))$

Collapse patterns for Boolean nodes exist only for a restricted MTLS, as described in Section 5.

Note that collapse operations do not affect the acceptance and final values of the p-tree. Consider for example p-tree acceptance values: Let a and b be the Boolean acceptance values of the sub-trees labeled A and B, respectively; clearly $a \vee (a \vee b) = a \vee b$. Likewise, the final value is not affected by the removal of a redundant sub-tree.

6 Growth Analysis

We will use the following three restricted forms of MTLS to analyze the growth of the suggested REM method: (i) MTLS restricted to basic propositions and the ◊ and □ operators, designated *vanilla* MTLS; (ii) vanilla MTLS extended with Boolean operators, for which Boolean operators, when consecutively nested, are only permitted to be nested in a non-alternating manner, designated *two-color* MTLS; and (iii) MTLS with formulae in the form of *pUq,* where *p* and *q* are two-color formulae, designated *three-color* MTLS. Hence, for example, □◊p is in vanilla MTLS, whereas $\Box(\neg p \vee \Diamond q)$ is in two-color MTLS, as is $\Box \Diamond (\neg p \vee \Diamond (q_1 \wedge q_2))$. In contrast, $\Box((q_1 \wedge q_2) \vee \Diamond q)$ is not in two-color MTLS. Later, we will relax our definitions to enable ○ nodes anywhere within vanilla and two-color MTLS formulae.

In this section we consider two classes of p-trees: live and final. A final p-tree is a singleton, or a p-tree that is equivalent to one, i.e., it contains only propositions and Boolean nodes. A live tree is a non-final tree.

6.1 Growth Analysis for REM of Vanilla MTLS

It is well known for linear time temporal logic that □□p=□p and ◊◊p=◊p. Therefore, a vanilla MTLS formula either has no temporal operators or has alternating □ and ◊ operators, as in □◊p and ◊□◊p. We will show that after applying collapse operations, p-trees for vanilla MTLS formulae either are always final or assume one possible tree shape.

Lemma 1. During REM with simple collapse operations, p-trees for vanilla MTLS formulae exist in one of two forms: live and final.

Proof outline: Fig. 6a and 6b illustrate all possible p-trees for ◊p and □p vanilla MTLS formula, respectively, after a single execution cycle. Consider Fig. 6a for example: If p is false, the left p-tree results, whereas if p is true, the right p-tree results. The claim holds for these p-trees: the unique live p-tree for ◊p (□p) is of size 3, while the other p-tree for ◊p (□p) is a singleton, final (**T** or **F**) state. If the p-tree is live, then after one additional cycle the recursive copy assumes a form that is again one of the two forms in Fig. 6a (6b). If this form is a singleton the entire p-tree for ◊p (□p) persists as a singleton.

Consider now a deeper vanilla- MTLS formula such as □◊p. As illustrated in Fig. 6, let N denote the root node of the p-tree, and let left-N be the left-hand side child of N, i.e., the non-recursive child. Note that left-N is a p-tree for ◊p, i.e., we have already shown that it is either the **F** singleton or assumes one possible form as a live tree. Both possibilities are illustrated in Fig. 6b, while Fig. 6c illustrates the application of simple collapse operations, resulting in a single live p-tree for □◊p in both cases.

A full proof for this claim involves an inductive claim over the depth of the vanilla MTLS formula, while utilizing the fact that the □ and ◊ operators within the formula must be ordered in an alternating manner.

6.2 Growth Results for REM of Two-color and Three-Color MTLS

In this section, we provide the results used for the development of growth metrics of two-color and three-color MTLS, as described in the next section.

Along the lines of Lemma 1, Lemma 3 claims that during REM of two-color PLTL, the number of different p-trees is limited. We first state that when a p-tree A1 for a vanilla formula p becomes final then so must all p-trees for p that are created after A1.

Lemma 2. Let A1 and A2 be p-trees for a two-color (sub)formula p that are created during REM with simple collapse operations. If A1 is live during the period [t1,t2] and A2 is created within that period, then A2 must be live at time t2.

Using Lemma 3 below, our REM method limits the growth of p-trees for two-color PLTL formulae. To this end, we define new collapse operations, designated special collapse operations. Special collapse operations, illustrated in Fig. 7, perform optimization in the form of $(C \wedge \mathbf{T}) \vee (C \wedge D) \vee \text{promise} = C \vee \text{promise}$, where the inner logical or is the ◊ nodes' logical or, and promise represents any future extension of the currently un-extended leaf ◊ node. Similarly, for a logical Or B node (B is illustrated in Fig. 7) under a ◊ root, the collapse pattern represents the equation $(C \vee \mathbf{F}) \vee (C \vee D) \vee \text{promise} = (C \vee D) \vee \text{promise}$. Analogous collapse patterns exist under □ nodes.

Lemma 3. During REM with simple, extended, and special collapse operations, p-trees for two-color PLTL formulae exist in one of three forms: one live and two final. Also, three-color PLTL formulae are in the form of pUq, where p and q are in two-color PLTL.

Using Lemma 4, below, our REM method limits the growth of p-trees for three-color PLTL formulae.

Lemma 4. During REM with simple, extended, and special collapse operations, live p-trees for three-color PLTL formulae ψ=pUq must be either:

1. Of the form in Fig. 8a, where $\forall i$, $1 \le i \le n$, $\boldsymbol{P}_i$ is isomorphic to $\boldsymbol{P}_{i+1}$ and $n \le \max(2,d)$, where d is the syntactic nesting depth of q.
2. Of the form in Fig. 8b or Fig. 8c.

7 Growth Metrics

Two metrics measure the growth of p-trees for an PLTL formula ψ during REM:

1. *size*(ψ), an upper bound on the size of p-trees for ψ, and
2. *m*(ψ), an upper bound on the number of possible distinct p-trees that exist for ψ.

Let size2(ψ) and m2(ψ) denote the size(ψ) and m(ψ), respectively, for a two-color PLTL formula ψ.

Table 1 contains recurrence equations for these metrics based on the lemmas stated earlier, as well as Lemma 5 below. When calculating the size of hybrid PLTL formula ψ, such as $(p\ U\ (\neg p \wedge q))\ U\ (\neg p \vee \Diamond q)$, m_2 and $size_2$ are first computed for the two-color sub-formulae and are then used to compute *size* and *m*, respectively, for the whole of ψ. As suggested by Table 1, nesting of *U* operators is the primary reason for p-tree growth. In fact, *size*(ψ) is exponential in the nesting depth of *U* operators inside ψ as well as in the number of alternations of Boolean logic operators within ψ. However, nesting of *U* operators in PLTL is typically used in two primary ways:

- *Right-side nesting*, as in $\neg P\ U\ (P\ U\ (\neg P\ U\ (P\ U\ (\neg P\ U\ P))))$. Right-side nesting can often be replaced with more readable and more efficient counting operators [4].
- *Left-side nesting*, as in $(P \rightarrow (\neg R\ U\ (S \wedge \neg R)))\ U\ R$. Growth for left-side nesting is often limited, as Lemma 5 suggests.

Lemma 5. For a left nested formula ψ=ρUτ, (i) if τ is a propositional logic sub-formula, then size(ψ) ≤ m(ρ)*(size(ρ)+2) and m(ψ) ≤ m(ρ)!, and (ii) if τ is a vanilla PLTL sub-formula, then size(ψ) ≤ m(ρ)*(size(ρ)+size(τ)) and m(ψ) ≤ m(ρ)!*m(ρ).

Proof. First, consider a propositional logic τ, i.e., the p-tree for τ is always a singleton. The Boolean logic representation of any live p-tree for ψ must be in the form $e = \mathbf{F} \vee (P_1 \wedge (F \vee (P_2 \wedge \ldots (F \vee (P_n \wedge \text{promise})))$, where P_i's are logical representations of p-trees for ρ. There are at most m(ρ) distinct P_i's. Hence, after at most m(ρ) levels, extended collapse is enabled and consequently: size(ψ)≤m(ρ)*(size(ρ)+2) and m(ψ)≤m(ρ)!, representing all ordering possibilities for the P's. Similarly, for a vanilla τ, $e = Q_1 \vee (P_1 \wedge (Q_2 \vee (P_2 \wedge \ldots (Q_n \vee (P_n \wedge \text{promise})))$, where Q_i's are Boolean logic representations of p-trees for τ. From Lemmas 2 and 3, it follows that $\exists k$, $k>0$, such that $\forall j<k$, $Q_j=\mathbf{F}$, and $\forall j \ge k$. Also, Q_j is isomorphic to Q_k. (k=1 means all Q_j's are isomorphic to one other). Hence, after at most m(ρ) levels, extended collapse is enabled and consequently: size(ψ)≤m(ρ)*(size(ρ)+size(τ)) and m(ψ)≤m(ρ)!*m(ρ), where m(ρ)! represents all possible enumerations of the P_j's and an additional m(ρ) factor counts all possible k's.

In a three-color PLTL formula, $\psi=\rho U\tau$, when both ρ and τ are propositional, as in $\psi = (\neg R\ U\ (S \wedge \neg R))$, a p-tree for ψ is either final (**T** or **F** singletons) or assumes one possible live form, which is the initial live form for ψ. Hence, for the example above, $size(\psi)=7$ and $m(\psi)=3$. For similar reasons $size(\neg(S \wedge (\neg R) \wedge O(\neg R\ U\ (T \wedge \neg R))))=13$ and $m(\neg(S \wedge (\neg R) \wedge O(\neg R\ U\ (T \wedge \neg R))))=3$. Also, a formula like $\Box((Q \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ U\ R))$ enjoys properties of two-color PLTL, such as Lemmas 2 and 3, because, as in the two-color PLTL case, the sub-formula $(\neg P U R)$ has only one live form

PLTL Formula	Recurrence	Reasoning
ψ is $\Diamond\rho$ or $\Box\rho$	$size(\psi)\leq m(\rho)*size(\rho)$	Growth until collapse is enabled
ψ is $\Diamond\rho$ or $\Box\rho$	$size_2(\psi)\leq size_2(\rho) + 2$	Lemma 3
ψ is $\Diamond\rho$ or $\Box\rho$	$m(\psi)\leq m(\rho)!$	Ordering $m(\rho)$ objects
ψ is $\Diamond\rho$ or $\Box\rho$	$m_2(\psi)\leq 3$	Lemma 3
ψ is $\rho\ U\ \tau$	$size(\psi)\leq m(\rho)^2*m(\tau)^2*4$ $*size(\rho)*size(\tau)$	Growth until collapse is enabled
ψ is $\rho\ U\ \tau$	$m(\psi)\leq(m(\rho)^2*m(\tau)^2)!$	Ordering $m(\rho)^2*m(\tau)^2$ objects
ψ is $\rho\ U\ \tau$ (two-color ρ, τ)	$size_2(\psi)\leq size_2(\tau)*$ $(size_2(\rho)+size_2(\tau))$	Lemma 4: growth until collapse is enabled
ψ is $\rho\ U\ \tau$ (two-color ρ, τ)	$m_2(\psi)\leq m_2(\rho)*m_2(\tau)*$ $(size_2(\tau)+3)$	Cases III and V in the proof of Lemma 5, induce $m_2(\rho)*m_2(\tau)*size_2(\tau)$
ψ is $\rho\ U\ \tau$ (propositional τ)	$size(\psi)\leq m(\rho)*size(\rho)$	Lemma 5
ψ is $\rho\ U\ \tau$ (propositional τ)	$m(\psi)\leq m(\rho)! + 1$	Lemma 5
ψ is $\rho\ U\ \tau$ (vanilla τ)	$size(\psi)\leq m(\rho)*$ $(size(\rho)+size_2(\tau))$	Lemma 5
ψ is $\rho\ U\ \tau$ (vanilla τ)	$m(\psi)\leq m(\rho)!*m(\rho)$	Lemma 5
ψ is $\rho \wedge \tau$, $\psi=\rho \vee \tau$	$size(\psi)\leq size(\rho)+ size(\tau) + 1$	
ψ is $\rho \wedge \tau$, $\psi=\rho \vee \tau$	$size_2(\psi)\leq size_2(\rho)+size_2(\tau)+ 1$	
ψ is $\rho \wedge \tau$, $\psi=\rho \vee \tau$	$m(\psi)\leq(m(\rho)-1)*(m(\tau)-1)+1$	Cartesian product
ψ is $\rho \wedge \tau$, $\psi=\rho \vee \tau$	$m_2(\psi)\leq 3$	Lemma 3
ψ is basic proposition	$size(\psi)\leq 1$, $size_2(\psi)\leq 1$	A final node
ψ is basic proposition	$m(\psi)\leq 2$, $m_2(\psi)\leq 2$	**T** or **F** nodes

Table 1: Growth Metrics

As for the PLTL *next* (O) operator, it introduces transient delays and therefore does not affect the analysis performed so far, other than contributing a fixed number of additional states to the *size* and $size_2$ metrics. Moreover, O operators can also be expressed in MTL using ◊ and real-time constraints, where the real-time clock counts in cycles. For example, □OOρ is also expressible as $\Box\Diamond_{=2}\rho$. An MTL implementation is described in the next sub-section.

While it is convenient for the purposes of analysis to push negations down to the proposition level, this is rarely done in practice. Rather, tools typically use a special negation node that inverts the acceptance value of the p-tree hanging under it. With inner level negation, the definition of two-color PLTL changes, so that negations are not permitted to exist directly between two logical *and* nodes or directly between two logical *or* nodes.

KSU pattern ψ	English meaning	Alternative pattern	size(ψ)
$\Diamond R \rightarrow (\neg P \; U \; R)$	Absence of P before R	$\Diamond R \rightarrow (\neg P \; W \; R)$	9
$\Box ((Q \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \; U \; R))$	Absence of P after Q until R	$\Box ((Q \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \; W \; R))$	15
$\Diamond R \rightarrow ((\neg P \wedge \neg R) \; U \; (R \vee ((P \wedge \neg R) \; U \; (R \vee ((\neg P \wedge \neg R) \; U \; (R \vee ((P \wedge \neg R) \; U \; (R \vee (\neg P \; U \; R)))))))))$	2 transitions to P before R	$\Diamond R \rightarrow (\neg P \wedge \bigcirc P) \; RepeatedUntil_{\geq 2} \; R$	11
$\Box ((Q \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \; U \; (S \vee R)))$	S precedes P between Q, R	$\Box((Q \wedge \neg R \wedge \Diamond R \wedge \Diamond S) \rightarrow (\neg P \; W \; (S \vee R)))$	21
$\Box ((Q \wedge \neg R \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R \; U \; (S \wedge \neg R))) \; U \; R)$	Response S responds to P between Q and R	$\Box ((Q \wedge \neg R \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R \; U \; (S \wedge \neg R))) \; W \; R)$	576
$\Box(Q \rightarrow (\neg(S \wedge \neg R \wedge \bigcirc(\neg R \; U \; (T \wedge \neg R))) \; U \; (R \vee P) \vee \Box(\neg(S \wedge \bigcirc\Diamond T))))$	P precedes (S, T) after Q until R	Replace every $\alpha U \beta$ with $(\alpha W \beta) \wedge \Diamond\beta$	1575
$\Box (Q \wedge \neg R \rightarrow (\neg R \; W \; (P \wedge \neg R)))$	P exists between Q , R		13
$\Box (S \wedge \bigcirc\Diamond T \rightarrow \bigcirc(\Diamond (T \wedge \Diamond P)))$	2-stimulus, 1-response chain		17
$\Box (P \rightarrow \Diamond(S \wedge \neg Z \wedge \bigcirc(\neg Z U \; T)))$	P triggers S followed by T without Z in the given scope	$\Box (P \rightarrow \Diamond(S \wedge \neg Z \wedge \bigcirc((\neg Z W T) \wedge \Diamond T))$	20

Table 2: KSU Pattern Examples

8 Practical Examination using KSU Patterns

The KSU specification patterns list is a repository of patterns that occur commonly in the specification of concurrent and reactive systems. Table 2 contains examples of KSU patterns along with corresponding *size* metric, i.e., the upper bound on the size of their p-trees. Note that KSU patterns refer to the weak-until as *W*, and strong until as *U*, where $\rho U\tau = \rho W\tau \wedge \Diamond\tau$.

As an example for the analysis shown in Table 2, consider the sixth entry. Using the equations of Table 1 we get:

- *size*($\neg$R *W* (T $\wedge$ $\neg$R)) = 7; *size*($\Diamond$(T $\wedge$ $\neg$R))=5.
- *size*($\neg$R *U* (T $\wedge$ $\neg$R)) = *size* (($\neg$R *W* (T $\wedge$ $\neg$R)) $\wedge$ $\Diamond$(T $\wedge$ $\neg$R)) = 13; *size*($\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) = 19; *m*($\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) = 3.
- *size* of $\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *W* (R $\vee$ P) is therefore 3*19=57 and *size* of $\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *U* (R $\vee$ P) is 63.
- *m* of $\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *W* (R $\vee$ P) is 3! = 6 and therefore *m* of $\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *U* (R $\vee$ P) = (6-1)*(3-1)+1 = 11.
- $size_2$ of $\Box$($\neg$(S $\wedge$ $\bigcirc\Diamond$T)) is 9.
- m_2 of $\Box$($\neg$(S $\wedge$ $\bigcirc\Diamond$T)) is 3.
- *size* of ($\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *U* (R $\vee$ P) $\vee$ $\Box$($\neg$(S $\wedge$ $\bigcirc\Diamond$T))) is 63 + 9 + 1 = 73, and *size* of Q $\rightarrow$ ($\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *U* (R $\vee$ P) $\vee$ $\Box$($\neg$(S $\wedge$ $\bigcirc\Diamond$T))) is 75.
- *m* of ($\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *U* (R $\vee$ P) $\vee$ $\Box$($\neg$(S $\wedge$ $\bigcirc\Diamond$T))) = (11-1)*(3-1)+1 = 21, and *m* of Q $\rightarrow$ ($\neg$(S $\wedge$ $\neg$R $\wedge$ $\bigcirc$($\neg$R *U* (T $\wedge$ $\neg$R))) *U* (R $\vee$ P) $\vee$ $\Box$($\neg$(S $\wedge$ $\bigcirc\Diamond$T))) is 21.
- Therefore *size*(ψ) = 21*75 = 1575.

The REM traversal of an average p-tree node can be implemented with 20 lines of C or Java code. Hence, if we assume five instructions per C line of code, and execution on a 1-GHz CPU, ψ formulae for rows 2 and 6 can be evaluated at a rate of 60,000 and 6,000 cycles per second, respectively. Clearly, this estimation is for the largest possible p-tree for ψ, whereas for many inputs sequences, and for many cycles during each sequence, the actual size is possibly smaller.

9 Conclusion

We presented a technique for run-time monitoring of extended PLTL using dynamic and-or trees called p-trees based on AFA. This technique lends itself to extentions of PLTL in which sattelite information such as real-time and time-series measurements is stored in p-tree nodes. Further investigation into the suitablity of this technique to other extentions of PLTL is required.

We presented certain optimization techniques for capping the size of p-trees during on-line PLTL monitoring. Further investigation is required into more effixient optimization techniques.

References

[1] R. Alur and T. Henzinger, Realtime Logics: Complexity and Expressiveness, proc. LICS 1990.

[2] G.S. Avrunin - J. C. Corbett, M. B. Dwyer, Property Specification Patterns for Finite-State Verification, 2nd Workshop on Formal Methods in Software Practice, March, 1998.

[3] 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.

[4] D. Drusinsky, The Temporal Rover and ATG Rover. Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science, 1885, pp. 323-329.

[5] D. Drusinsky , Formal Specs Can Handle Exceptions, Embedded Developers Journal, Nov. 2001, pp., 10-14.

[6] D. Drusinsky, Monitoring Temporal Rules Combined with Time Series, Computer Aided Verification Conference 2003, pp. 114-117.

[7] 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, p. 345-348.

[8] D. Drusinsky and G. Watney, Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine, 28'th IEEE/NASA Software Engineering Workshop, 2003.

[9] D. Drusinsky, B. Michael and M. Shing, Behavioral Modeling and Run-Time Verification of System-of-Systems Architectural Requirements, CCCT 2004.

[10] D. Drusinsky and M. Shing, Verification of Timing Properties in Rapid System Prototyping, Proc. Rapid System Prototyping Conference 2003 (RSP'2003).

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

[12] K. Havelund, G. Rosu, Monitoring Programs using Rewriting. In Proc. IEEE Conf. on Automated Software Engineering (ASE), 2001.

[13] J. Hopcroft, R. Motwani, and J. Ullman, Introduction to Automata Theory Languages and Computation, Addison Wesley, second edition.

[14] J. Kovacs, G. Kusper, R. Lovas, and W. Schreiner, Integrating Temporal Assertions into a Parallel Debugger, Parallel processing. 8th international Euro-Par conference. Paderborn, 2002. Berlin, Springer-Verlag, 2002. pp. 113-120.

[15] O. Kupferman and S. Zuhovitzky, An Improved Algorithm for the Membership Problem for Extended Regular Expressions, Proc. 27th International Symposium on Mathematical Foundations of Computer Science, Springer LNCS 2420, p. 446, 2002.

[16] Z. Manna and A. Pnueli, Verification of Concurrent Programs: Temporal Proof Principles, Proc. of the Workshop on Logics of Programs, Springer LNCS 1981, pp. 200-252.

[17] A. Pnueli, The Temporal Logic of Programs, Proc.18th IEEE Symp.. on Foundations of Computer Science, pp. 46-57, 1977.

[18] G. Rosu and R. Thati, Monitoring Algorithms for Metric Temporal Logic Specifications, Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[19] 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.

[20] A. P. Sistla and O. Wolfson, Temporal Triggers in Active Databases, IEEE Transactions on Knowledge and Data Engineering (TKDE), June 1995 pp. 471-486.

[21] A. Tuzhilin, Extending Temporal Logic to Support High-Level Simulations, ACM Transactions on Modeling and Computer Simulation, vol. 5, no. 2, 1995.

[22] M. Vardi and P. Wolper, An Automata-theoretic Approach to Automatic Program Verification, In Proc. Symp. On Logic In Computer Science, pp. 322-331, Cambridge, June 1986.