URL: http://www.elsevier.nl/locate/entcs/volume89.html 16 pages

Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems

Kåre Jelling Kristoffersen [1,2] Christian Pedersen [3]
Henrik Reif Andersen [4]

Department of Innovation
IT University of Copenhagen
Copenhagen, 2400 NV, Denmark

Abstract

In this paper we present a new framework for runtime verification of properties of real time systems such as financial systems or backend databases. Such a systems has a semantics which resemples that of timed traces, namely a sequence of states where each state consists of predicates true in this state and then a *timestamp* explaining *when* the state is valid. We present a logic, LTL_t, which is an extension of LTL with time constraints and a freeze quantifier and show how formulae in this logic are able to express properties of bounded liveness and safety which are ideal for these systems. It is shown how a formula in LTL_t may be rewritten to a certain disjunctive normal form suitable for checking a real time system at runtime. The normal form captures the essential part of runtime verification by a set of mutually defined formula identifiers, each expressing two things: What should hold *now* and which formula identifiers that will need to hold in the *next* state. As part of the theoretical foundation for this work we propose a characterization of Runtime Verification and address the challenges in developing a method which is both sound and complete while at the same time efficient.

Keywords: Timed LTL, Disjunctive Normalized Equation Systems, Charaterization of Runtime Verification, Property Tranformation.

1 Introduction

The traditional field of application for formal methods and especially verification is within safety critical and embedded systems for which correctness is of

[1] This work is supported by the project NEXT which is a joint effort between Microsoft Business Solutions and the IT University of Copenhagen.

[2] Email: kjk@it-c.dk
[3] Email: cp@it-c.dk
[4] Email: hra@it-c.dk

©2003 Published by Elsevier Science B. V.

vital importance. Errors are either fatal or they are costly. Therefore a lot of energy has been put into developing tools for checking such systems at design time, that is, prior to execution i.e. SPIN [4]. As an alternative, attempts have been made to check a running java program in Java PathExplorer [1]. Common for these efforts is that what is verified is a *program*, and typically a program taking very little input, if any. This implies that the program under investigation can be regarded as a closed system, which may be checked alone.

Financial applications and ERP Systems [5] as well as traditional back end databases are a class of applications that hava attracted very little attention from the formal methods community. Such a system has a semantics which resemples that of timed traces, namely a sequence of states where each state consists of predicates true in this state and then a *timestamp* explaining *when* the state is valid. One reason that these systems have not gained very much interest from the verification communuty is that they are typically not of a safety–critical nature and hence correctness is of less importance compared with traffic control or production systems. At the same time these systems differ from the beforementioned in that they can (and should be able to) consume infinitely many different input data. As such they suffer from the well–known "State Explosion Problem" and consequently exhaustive verification is not feasible.

The IT University of Copenhagen is involved in establishing the foundation for the next generation of ERP systems to be developed by Microsoft Business Solutions, Denmark. One effort made here is the attempt to introduce formal methods and especially verification.

The fundamental idea is to regard the object of investigation as a *totality* of two things, an *ERP system* and its *user*. The behaviour of the user is highly unpredictable i.e. non–deterministic; a value of data may be inserted, updated or deleted at any given time. Not only non–determinism, but also the sheer amount of data, makes traditional verification efforts infeasible for ERP Systems.

In [2] it is demonstrated how the standard translation from LTL to Büchi Automata [3] can be used to perform runtime verification given that a subtle change of the accepting condition is made. The problem behind this subtelty is that the standard translation assumes that an explicit representation of the infinite trace to be verified can be provided, and the actual check amounts to a check for cycles in the synchronous product of the program trace and the automaton. In runtime verification, this construction is not possible, since here only the "current" state is available. In [1] this is overcome by substituting the rather heavy automata framework by a system for rewriting logics in which the LTL (alternatively past time LTL) specification is rewritten for each new state generated by the system under investigation.

We will claim that a *true* runtime verification procedure is characterized

[5] ERP: Enterprise Ressource Planning.

as follows. Given a timed trace $\sigma = \sigma_0\sigma_1 \ldots \sigma_i \ldots$, the procedure must obey that

(i) The σ_i's are used in sequential order.

(ii) Each σ_i is used only once.

Consequently, atomata and especially backtracking is not an option. For instance, Iif we want to verify that a timed trace satifies a disjunction containing temporal operators, we have to monitor both disjuncts in the future. This cannot be done with an atomata since it is only legal to be in one state at a time in an atomaton, and in this case it would demand the ability to stay in two states at one time. It can be argued that automata can be ade deterministic, but for real time it is not obvious to us how this should be done. However, as we shall soon see, by staing in the world of logics the desired feature is easily obtained.

We extend the existing work on LTL runtime verification by extending LTL with real time constructs embodied by a freeze quantifier together with atomic clock constraints (following [6]) making it possible to express real time logical properties of a system. This logic is called LTL_t. This logic is suitable for expressing bounded safety and liveness properties. An example of a bounded liveness property for a bank account could be: *If Balance is negative then within 10 days the Balance should be positive or zero.* In LTL_t this is written as follows

$$\Box(Balance < 0 \Rightarrow x.(t \leq x + 10\, U\, Balance \geq 0))$$

In the above expression the variable t is the global time, or more precisely the time of the observation trace generated by the system being monitored. In this paper we use states with discrete time. This corresponds to timestamps with real time in term of monitoring ERP-Systems. The semantics of the freeze formula $x.\phi$ is that when evaluated the value of x is replaced by the current value of t.

Moreover we introduce a normal form for formulae in LTL_t and show that all formulae may be written in this normal form. A formula on normal form consists of a collection of mutually dependent formula identifiers each stating:

(i) What should hold *now* (i.e. a *propositional* part).

(ii) What identifiers should hold in the *next* state (i.e. a *temporal* part).

Our algorithm for runtime verification is then a *property transformer* which works on sets of formula identifiers each defining a normal form expression. One step of the algorithm transforms one set of identifiers to another. Inherent in this methodology lies a danger of an exponential blow–up in the set of identifiers, but the application of heuristics may be used to keep the set small during program execution.

This work also has a relationship to [12] and [11] in terms of rewriting a specification with respect to state information. In [12] it is shown how to

build a (branching time) transition system into a modal logic specification by constructing a *quotient*. This method is used as an efficient verification procedure for *parallel* systems. In [11] the same exercise is repeated for real time systems. In both these works the quotient consists of a new (and *larger*) set of formulae whose number and size are sought to be kept small by applying a number of heuristics for minimization. Our work differs significantly from that of [12] and [11] in two substantial ways. First, by rewriting the specification to a normal form *prior* to verification we obtain a specification whose size is small (limited by the temporal depth of expressions) and whose size moreover is *fixed* throughout verification. Second, at runtime (corresponding to verification time in their work) we need only to maintain a workset of formula identifiers that will need to hold in the next state.

The paper is organized as follows. In Section 2 we present the logic LTL_t as well as its interpretation in timed traces. In Section 3 we present our normal form for formulae in LTL_t and show that any formula may be written in this normal form. In Section 4 we propose a charaterization of Runtime Verification. In Section 5 we present our algorithm and state that our method is sound and relatively complete with respect to satisfiability. In 6 we give some details on our implementation and address important scalability issues by investigating performance on a collection of examples. Finally in Section 7 we comment on the results obtained and give directions for further work.

2 Temporal Logics for Real Time

Let $AP = \{p, q, r, \ldots\}$ be a set of *atomic propositions* and let $t \in \mathbb{N}$ be a discrete time. We denote a *timed state* by a pair (s, t) where $s \subset AP$ whose meaning is that the propositions in s, and only these, are true at time t. The time component can be thought of as a time-stamp on a snapshot of a system's state.

Definition 2.1 A *(discretely) timed trace* over AP is an infinite sequence of states

$$\sigma = \sigma_0\sigma_1 \cdots \sigma_i \cdots,$$

where each $\sigma_i = (s_i, t_i)$ is a timed state. We require that $t_i < t_{i+1}$ for all i. A timed trace is *complete* if $t_0 = 0$ and for all i: $t_{i+1} = t_i + 1$. We denote by Σ the set of all complete timed traces.

We use the notations $s(\sigma_i)$ and $t(\sigma_i)$ for the propositional respectively timed part of a timed state. We use superscripting with i for the sub-sequence σ^i, which starts at σ_i, i.e., the sequence $\sigma^i = \sigma_i\sigma_{i+1}\cdots$. It is obvious that any timed trace can be made into a complete trace by replicating states, e.g., in the trace $\sigma = (\{p\}, 0)(\{q\}, 3)(\{p\}, 6)\sigma^3$ the timed state $(\{p\}, t)$ is added for $t \in \{1, 2\}$ and the timed state $(\{q\}, t)$ is added for $t \in \{4, 5\}$. The completed trace $\hat{\sigma}$ thus starts as: $(\{p\}, 0)(\{p\}, 1)(\{p\}, 2)(\{q\}, 3)(\{q\}, 4)(\{q\}, 5)(\{p\}, 6)$.

A completed trace thus corresponds to a more refined "sampling" of an

original trace in which all state changes are recorded at the point in time where they take place. In the subsequent developments, we work with completed traces. In section 7 we discuss how to improve performance of the resulting algorithm by relaxing this condition.

Definition 2.2 *Timed LTL*, LTL_t, is given by the following abstract syntax

$$\phi ::= p \mid \neg p \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid t \sim x + c \mid x.\phi \mid \phi_1 U \phi_2 \mid \phi_1 V \phi_2 \mid \bigcirc \phi$$

where $p \in AP$, t refers to the "current time" in a timed trace, x is a *discrete formula clock*, $c \in I\!N$ and $\sim \in \{<, \leq, =, >, \geq\}$.

The syntactic elements are: atomic propositions and negations of them, logical connectives, *time constraints*, the *freeze operator* which "records" the current time in the clock variable x, and the temporal operators *until*, *release* (the dual of until) and *next*. We shall use standard abbreviations such as $\Box\phi = false V \phi$ for the always operator, $\Diamond\phi = true U \phi$ for the eventually operator, and implication etc. as logical connectives.

The purpose of the standard temporal operators until (U) and release (V) is that they capture real-time properties: $\phi_1 U \phi_2$ holds if there is some time in the future t where ϕ_2 holds and on all time points up to (not including) this one, ϕ_1 holds. When working with discrete time and completed traces, this corresponds quite closely to the existence of an index i, such that σ^i satisfy ϕ_2 and for all smaller indices j, ϕ_1 holds. With discrete time and completed traces, the next operator also has a natural semantics: "next" refers to the next point in time, which is the time associated with the next timed state of the trace.

The resulting logic can be viewed as an extension of the normal untimed linear time temporal logic extended with the notion of clocks, which are capable of recording time (=positions) in the trace using the freeze operator and relate them to other times (=positions) in the trace using time constraints.

We do not have negation in the logic, but assume that the user can still freely use it since a negation may be pushed inside to the propositions using the standard procedure interchanging conjunction with disjunction and until with release (the operator V).

The specification language for our runtime verification will be that of closed LTL_t expressions. Neither we shall use the $\bigcirc$–operator nor the V–operator. However, the reason to include them in the logic is that they are needed when we rewrite specifications to *normal form*, see Section 3.

Given an environment for formula clocks ϵ as a partial mapping from the set of clocks to $I\!N$, we can define the semantics $[\![\phi]\!]\epsilon \subseteq \Sigma$ of a formula ϕ with

free clock variables in the domain of an enviroment ϵ, inductively as follows:

$$
\begin{aligned}
[\![p]\!]\epsilon &= \{\sigma \in \Sigma \mid p \in s(\sigma_0)\} \\
[\![\neg p]\!]\epsilon &= \{\sigma \in \Sigma \mid p \notin s(\sigma_0)\} \\
[\![\phi_1 \wedge \phi_2]\!]\epsilon &= [\![\phi_1]\!]\epsilon \ \cap \ [\![\phi_2]\!]\epsilon \\
[\![\phi_1 \vee \phi_2]\!]\epsilon &= [\![\phi_1]\!]\epsilon \ \cup \ [\![\phi_2]\!]\epsilon \\
[\![t \sim x + c]\!]\epsilon &= \{\sigma \in \Sigma \mid t(\sigma_0) \sim \epsilon(x) + c\} \\
[\![x.\phi]\!]\epsilon &= \{\sigma \in \Sigma \mid \sigma \in [\![\phi]\!]\epsilon[t(\sigma_0)/x]\} \\
[\![\phi_1 U \phi_2]\!]\epsilon &= \{\sigma \in \Sigma \mid \ \exists i.\ (\sigma^i \in [\![\phi_2]\!]\epsilon \text{ and } \forall j, 0 \leq j < i.\ \sigma^j \in [\![\phi_1]\!]\epsilon)\ \} \\
[\![\phi_1 V \phi_2]\!]\epsilon &= \{\sigma \in \Sigma \mid \ \forall i.\ \sigma^i \in [\![\phi_2]\!]\epsilon \text{ or} \\
&\qquad\qquad (\exists j.\ \sigma^j \in [\![\phi_1]\!]\epsilon \text{ and } \forall i, 0 \leq i \leq j.\ \sigma^i \in [\![\phi_2]\!]\epsilon)\ \} \\
[\![\bigcirc\phi]\!]\epsilon &= [\![\bigcirc]\!]([\![\phi]\!]\epsilon),
\end{aligned}
$$

where $[\![\bigcirc]\!](S) = \{\sigma \in \Sigma \mid \sigma^1 \in S\}$.

A complete timed trace σ is now said to satisfy a closed formula ϕ if $\sigma \in [\![\phi]\!]\epsilon$ for an environment ϵ where all clocks have value zero written as $\sigma \models \phi$. A timed trace σ satisfy a formula if the completion of it, $\hat{\sigma}$, satisfy the formula.

Using the next-operator, we can find an alternative characterization of the until and release operators in terms of fixed-points. We use the lattice 2^Σ of subsets of Σ ordered by set-inclusion. Then for any monotonic function $F : 2^\Sigma \to 2^\Sigma$ there will be a minimum fixed-point $\mu F \subseteq \Sigma$ and a maximum fixed-point $\nu F \subseteq \Sigma$ [13]. Observe, that the following functions are monotonic:

$$
\begin{aligned}
F_{\phi_1 U \phi_2, \epsilon}(S) &= ([\![\phi_1]\!]\epsilon \cap [\![\bigcirc]\!]S) \cup [\![\phi_2]\!]\epsilon \\
F_{\phi_1 V \phi_2, \epsilon}(S) &= ([\![\phi_1]\!]\epsilon \cup [\![\bigcirc]\!]S) \cap [\![\phi_2]\!]\epsilon
\end{aligned}
$$

The until and release operators are now given by a minimum and a maximum fixed-point:

Lemma 2.3 *For any formulae ϕ_1, ϕ_2 with free variables in the domain of ϵ, the following holds:*

$$
\begin{aligned}
[\![\phi_1 U \phi_2]\!]\epsilon &= \mu F_{\phi_1 U \phi_2, \epsilon} \\
[\![\phi_1 V \phi_2]\!]\epsilon &= \nu F_{\phi_1 V \phi_2, \epsilon}
\end{aligned}
$$

For convenience, fixed-points can be written as equations. We thus write for instance, $X =_\mu F(X)$ for an equation system with the solution $X = \mu F$, see for instance [12] for an example of how this is done in the modal μ-calculus. For runtime verification, satisfaction of a fixed-point formula, will depend only on whether it can be judged to be true or false within a finite number of unfoldings or not, and the distinction of minimum and maximum fixed-

points will be rather unimportant. We will therefore choose not to annotate equations as being either minimum or maximum fixed-points when writing down equation systems.

Using the lemma it is straightforward to write down a formula as an equivalent set of equations using only the temporal next-operator leaving out the until and the release operators. For convenience, the algorithm will work with equation systems.

3 Disjunctive Normalized Equation Systems

In this Section we present our notion of Disjunctive Normalized Equation Systems and argue why it forms a suitable basis for runtime verification of Timed LTL.

Definition 3.1 A Normal Form Equation System $\mathcal{D}$ over formula identifiers $\{X_1, \ldots, X_n\}$ and formula clocks $V = \{x_1, \ldots, x_m\}$ is a set of defining equations

$$X_1(\vec{x}_1) = \phi_1$$
$$\vdots$$
$$X_n(\vec{x}_n) = \phi_n$$

where for all $i \in \{1, \ldots, n\}$, $\vec{x}_i$ is a (possibly) empty vector of variables from V, such that $\vec{x}_i$ includes all the free variables in ϕ_i. The right-hand sides ϕ_i are each on the following form :

$$\phi_i ::= \vec{x}. \bigvee_{j \in J_i} (\psi_{ij} \wedge \bigcirc \bigwedge_{l \in L_{ij}} X_l(\vec{x}_l))$$

where J_i is an index set, $L_{ij} \subseteq \{1, \ldots, n\}$ is a subset of indices 1 to n, ψ_{ij} is a non-temporal formula composed of the propositional part of LTL_t: atomic propositions, clock constraints and Boolean connectives, and finally $\vec{x}$ is a (possibly empty) vector of formula clocks from V.

Notice, that the actual arguments to the formula identifiers are always exactly the same as the declared formal parameters. Some of the variables might be bound by the freeze operator and their values might be restricted by constraints in the propositional part, ϕ_i.

Intuitively the normal form means the following: After the value of the clock x is 'frozen' to the current time there is a set of possibilities each of which describes:

(i) What should hold *now* (i.e. a *propositional* part) and

(ii) What identifiers should hold in the *next* state (i.e. a *temporal* part).

This disjunctive normal form is particularily suitable for performing runtime verification, since here we are at all times faced with the job of 'processing'

one single state and this job really implies two things: Checking whether the current state is ok and remebering what should hold for the future.

The normal form introduces the $\bigcirc$–operator by application of the following identity $\phi_1 U \phi_2 \equiv (\phi_1 \wedge \bigcirc(\phi_1 U \phi_2)) \vee \phi_2$.

3.1 Translating to Disjunctive Normal Form

In the following we will describe how to rewrite formulae in LTL_t to normal form. For a LTL_t formulae ϕ we will let $\mathcal{D}_\phi$ denote the equation system for ϕ on normal form. For an equation system $\mathcal{D}$ with a formula identifier X, we use $\mathcal{D}(X)$ for the right-hand side of X.

$$\begin{aligned}
\mathcal{D}_p &= \{X_p() = p\} \\
\mathcal{D}_{\neg p} &= \{X_{\neg p}() = \neg p\} \\
\mathcal{D}_{t \sim x+c} &= \{X_{t \sim x+c}(x) = t \sim x + c\}
\end{aligned}$$

$$\mathcal{D}_{\phi_1 \vee \phi_2} = \mathcal{D}_{\phi_1} \cup \mathcal{D}_{\phi_2} \cup \{X_{\phi_1 \vee \phi_2}(\vec{x}) = \vec{y}_1 \vec{y}_2. \bigvee_{i \in I_1} \gamma_1^i \vee \bigvee_{j \in I_2} \gamma_2^j\}$$

where $\vec{x}$ is all the free variables in ϕ_1 and ϕ_2

$\mathcal{D}_{\phi_1}(X_{\phi_1}) = \vec{y}_1. \bigvee_{i \in I_1} \gamma_1^i$,

$\mathcal{D}_{\phi_2}(X_{\phi_2}) = \vec{y}_2. \bigvee_{j \in I_2} \gamma_2^j$, and

assuming that no free variables in γ_1^i and γ_2^i become bound

$$\mathcal{D}_{\phi_1 \wedge \phi_2} = \mathcal{D}_{\phi_1} \cup \mathcal{D}_{\phi_2} \cup \{X_{\phi_1 \wedge \phi_2}(\vec{x}) = \vec{y}_1 \vec{y}_2. \bigvee_{i \in I_1} \bigvee_{j \in I_2} \gamma_1^i \wedge \gamma_2^j\}$$

where $\vec{x}$ is all the free variables in ϕ_1 and ϕ_2,

$\mathcal{D}_{\phi_1}(X_{\phi_1}) = \vec{y}_1. \bigvee_{i \in I_1} \gamma_1^i$,

$\mathcal{D}_{\phi_2}(X_{\phi_2}) = \vec{y}_2. \bigvee_{j \in I_2} \gamma_2^j$, and

assuming that no free variables in γ_1^i and γ_2^i become bound

$$\mathcal{D}_{\phi_1 U \phi_2} = \mathcal{D}_{\phi_1} \cup \mathcal{D}_{\phi_2} \cup$$
$$\{X_{\phi_1 U \phi_2}(\vec{x}) = \vec{y}_1\vec{y}_2.((\bigvee_{i \in I_1} \psi_{i1} \wedge \bigcirc(X_{\phi_1 U \phi_2}(\vec{x}) \wedge \bigwedge_{l \in L_{i1}} X_l(\vec{x}_l))) \vee \bigvee_{j \in I_2} \gamma_2^j)\}$$

where $\vec{x}$ is all the free variables in ϕ_1 and ϕ_2,

$\mathcal{D}_{\phi_1}(X_{\phi_1}) = \vec{y}_1. \bigvee_{i \in I_1}(\psi_{i1} \wedge \bigcirc \bigwedge_{l \in L_{i1}} X_l(\vec{x}_l))$, and

$\mathcal{D}_{\phi_2}(X_{\phi_2}) = \vec{y}_2. \bigvee_{j \in I_2} \gamma_2^j$,

assuming that no free variables from ϕ_1 and ϕ_2 become bound.

$$\mathcal{D}_{\phi_1 V \phi_2} = \mathcal{D}_{\phi_1} \cup \mathcal{D}_{\phi_2} \cup$$
$$\{X_{\phi_1 V \phi_2}(\vec{x}) = \vec{y}_1\vec{y}_2.(\bigvee_{i \in I_1} \gamma_1^i \vee \bigvee_{j \in I_2} (\psi_{2j} \wedge \bigcirc(X_{\phi_1 V \phi_2}(\vec{x}) \wedge \bigwedge_{l \in L_{2j}} X_l(\vec{x}_l))))\}$$

where $\vec{x}$ is all the free variables in ϕ_1 and ϕ_2,

$\mathcal{D}_{\phi_1}(X_{\phi_1}) = \vec{y}_1. \bigvee_{i \in I_1} \gamma_1^i$,

$\mathcal{D}_{\phi_2}(X_{\phi_2}) = \vec{y}_2. \bigvee_{j \in I_2}(\psi_{2j} \wedge \bigcirc \bigwedge_{l \in L_{2j}} X_l(\vec{x}_l))$, and

assuming that no free variables from ϕ_1 and ϕ_2 become bound.

$$\mathcal{D}_{x.\phi} = \mathcal{D}_\phi \cup \{X_{x.\phi}(\vec{x}) = x.\mathcal{D}_\phi(X_\phi)\}$$

Example 3.2 Consider the closed LTL_t expression which states that q becomes true before the elapse of 5 time units and that p holds until then.

$$\phi = x.((p \wedge t < x + 5) \, U \, q)$$

Rewriting to disjunctive normal form using the procedure sketched above we obtain the following equation system $\mathcal{D}_\phi$:

$$X_0() = x.((p \wedge t < x + 5 \wedge \bigcirc X_1(x)) \vee q)$$
$$X_1(x) = (p \wedge t < x + 5 \wedge \bigcirc X_1(x)) \vee q$$

As can be seen we use numbers rather than formulae themselves as indexes on formula identifiers. This is more convenient. For a formula ϕ we shall be using X_0 instead of X_ϕ. Sometimes we shall be referring to X_0 as trhe top identifier. In the example above the equations for X_0 and X_1 are not the only ones generated, in fact also equations for e.g. the propostions p and q will be constructed. However, since these are not referred to either directly or indirectly by X_0 we shall omit these for clarity reasons. Further notice that the right-hand side of X_1 is repeated in the right-hand side for X_0 due to the rewriting into normal form. Of course, in actual implementations these parts can be shared.

Theorem 3.3 *Let ϕ be a formula in LTL_t. Then ϕ may be rewritten to a Normal Form Equation System $\mathcal{D}(\phi)$ such that for all (σ, t) it holds that*

$$(\sigma, t) \models \phi \iff (\sigma, t) \models \mathcal{D}_\phi(X_\phi)$$

4 Charaterizing Runtime Verification

Exhaustive verification techniques always give a definitive answer regarding satisfaction [6]. Verification of a running program is in a certain sense a much more difficult procedure, since no such thing as a complete trace (finite or infinite) exists. Rather, at all times only the *current state* of the program being monitored is present for the verification procedure. Based on this partial information an answer must be given. This answer cannot be guaranteed to be conclusive, instead it will be one of the following three $\{yes, no, maybe\}$. In case of *yes* or *no* verification may halt, but in the case of the third possibility, the answer will be that of a *new* requirement for the remaining part of the execution trace, in order for the complete trace to satisfy the original specification.

Runtime Verification is characterized by a *formula transformer FT* with the following property for a formula ϕ and a trace $\sigma = \sigma_0\sigma^1$:

$$\forall \sigma^1.(\sigma_0\sigma^1 \models \phi \Leftrightarrow \sigma^1 \models FT(\phi, \sigma_0))$$

We will say that a formula transformer exhibiting the property defined above is a Runtime Time Verification property transformer. In the next section we shall present one such property transformer, Ok, and argue that it is indeed a runtime verification property transformer.

5 Verification using Normal Form Equations

Our runtime verification procedure consists of the property transformer Ok defined below. We use $\sigma_i \vdash \psi$ for denoting whether the state σ_i fulfills the non-temporal formula ψ. It will thus be either true or false.

Definition 5.1 Let ϕ be an LTL_t formula, let $\mathcal{D}_\phi$ be the Normal Form Equation System for ϕ and let $\sigma = \sigma_0\sigma_1\ldots$ be a timed trace. Then the transfor-

[6] Provided that the program under investigation is finite–state.

mation of formulae are given as follows

$$Ok(\vec{x}. \bigvee_{j \in J_i} (\psi_{ij} \wedge \bigcirc \bigwedge_{l \in L_{ij}} X_l(\vec{x_l}), \sigma_0) = \bigvee_{j \in J_i} (\sigma_0 \vdash \psi_{ij}[t_0/\vec{x}] \wedge \bigwedge_{l \in L_i j} X_l(\vec{x_l})[t_o/\vec{x}])$$

$$Ok(\bigvee_{j \in J_i} \bigwedge_{l \in L_i j} X_l(\vec{a}), \sigma_0) = \bigvee_{j \in J_i} \bigwedge_{l \in L_{ij}} Ok(X_l(\vec{a}), \sigma_0)$$

$$Ok(X_l(\vec{a}), \sigma_0) = Ok(\mathcal{D}\phi(X_l(\vec{x}))[\vec{a}/\vec{x}], \sigma_0)$$

In the definition above $\vec{x}$ denotes the formal parameters $x_1, \ldots, x_n$ (formula clocks) while $\vec{a}$ denotes the corresponding actual parameters. Initially we will be using $X_0(0, \ldots, 0)$ as the top identifier, which is all right since any use of a formula clock x_i will be preceeded by a freeze of this clock.

Regarding correctness, if $Ok(\phi, \sigma_0) = true$ then it holds that for all suffixes σ^1 that $\sigma_0\sigma^1 \models \phi$. And if $Ok(\phi, \sigma_0) = false$ then for no suffix σ^1 it holds that $\sigma_0\sigma^1 \models \phi$. If $Ok(\phi, \sigma_0)$ gives a result ϕ' other than the constants $true$ and $false$ then for any suffix σ^1 it holds that $\sigma^1 \models \phi'$ if and only if $\sigma_0\sigma^1 \models \phi$. In other words, our property transformer is sound. Now ideally, a tautology ϕ' should be reduced to the constant value $true$, and similarly, an unsatisfiable formulae should preferably be reduced to the constant value $false$. But this demands the incorporation of a SAT solver which would seriously hamper the efficiency of our method. Thus, our method can be said to be relative complete. Moreover, formulae which are not tautologies or falsities may be bigger than necessary. However, by application of heuristics in the style of [8,12,11] it is possible to apply a range of formula reductions towards a smaller specification (possibly $true$ or $false$).

To fully describe and justify the runtime verification process we now need the following definition of the predicate Ok^i, which denotes the application of the predicate OK using the states of the prefix of length i of a given trace.

Definition 5.2 Let $\sigma = (x_0, t_0), (x_1, t_1), \ldots, (x_n, t_n)$ be a timed trace. Then we define

$$Ok^0(\phi, \sigma) = Ok(\phi, \sigma_0)$$

$$Ok^{i+1}(\phi, \sigma) = Ok(Ok^i(\phi, \sigma), \sigma_{i+1})$$

Theorem 5.3 *Let σ be a timed trace of length n and let ϕ be a formula in normal form. Then there exists an $i \leq n$ such that $Ok^i(\phi, \sigma)$ is valid if and only if $(\sigma, t_0) \models \phi$.*

Example 5.4 Let $\sigma = (\{p\}, 0)(\{p\}, 1)(\{p, q\}, 2)\sigma^3$ be a timed trace, and let ϕ be the formula $x.((p \wedge t < x + 5)\, U\, q)$ that states that q should hold before 5 time units and p should hold until then, see example 3.2. Clearly σ satisfies ϕ regardless of σ^3 and we shall now apply the Ok transformer to see this.

Therefore we first apply Ok on the pair $X_0() = x.((p \wedge t < x+5) \wedge \bigcirc X_1(x)) \vee q$ and $(\{p\}, 0)$.

$$
\begin{aligned}
& Ok(X_0(), (\{p\}, 0)) \\
= & Ok(\mathcal{D}\phi(X_0()), (\{p\}, 0)) \\
= & Ok(x.((p \wedge t < x + 5 \wedge \bigcirc X_1(x)) \vee q), (\{p\}, 0)) \\
= & ((\{p\}, 0) \vdash p \wedge t < 0 + 5) \wedge X_1(0)) \vee (\{p\}, 0) \vdash q \\
= & (true \wedge X_1(0)) \vee false \\
= & X_1(0)
\end{aligned}
$$

This means that in order for σ to satisfy $X_0()$ then σ^1 should satisfy $X_1(0)$. Consequently, we therefore apply Ok on $X_1(0)$ and the first state of σ^1 namely $(\{p\}, 1)$.

$$
\begin{aligned}
& Ok(X_1(0), (\{p\}, 1)) \\
= & Ok(\mathcal{D}\phi(X_1(0)), (\{p\}, 1)) \\
= & Ok((p \wedge t < 0 + 5 \wedge \bigcirc X_1(0) \vee q), (\{p\}, 1)) \\
= & (((\{p\}, 1) \vdash p \wedge t < 5) \wedge X_1(0)) \vee (\{p\}, 1) \vdash q \\
= & (true \wedge x_1(0)) \vee false \\
= & X_1(0)
\end{aligned}
$$

Still, no final conclusion may be drawn, so, we apply Ok on the first state of σ^2, i.e. $(\{p, q\}, 2)$

$$
\begin{aligned}
& Ok(X_1(0), (\{p, q\}, 2)) \\
= & Ok(\mathcal{D}\phi(X_1(0)), (\{p, q\}, 2)) \\
= & Ok((p \wedge t < 0 + 5 \wedge \bigcirc X_1(0)) \vee q, (\{p, q\}, 2)) \\
= & (((\{p, q\}, 2) \vdash p \wedge t < 5) \wedge X_1(0)) \vee (\{p, q\}, 2) \vdash q \\
= & (true \wedge X_1(0)) \vee true \\
= & true
\end{aligned}
$$

The result $true$ means that no matter what σ^3 might be, then we may now once and for all conclude that σ satisfies ϕ.

6 Implementation

We can divide the implementation into two separate parts. One part, the Formula System Builder, that given an abstract syntax tree for the expression we want to validate builds a formula system corresponding to the normal form we have described in section 3. And another part, The Condition Checker,

that given a formula system can tell whether a sequence of states fails or succeeds to satisfy the Timed LTL property described by the formula system.

6.1 *The Formula System Builder*

The intuition behind the Formula System Builder is as described in Section 3. The algorithm is a recursive method that traverses the syntax tree using a depth first search. When it hits a leaf in the tree (which always will be a proposition or an atomic timing constraint), it starts to produce a formula system of the propositional expression. After the leaves have been made the nodes above crunce the formulae together. Each time a formula is constructed it is saved in the formula system, so that later it is possible to refer to that formula. The datastructures are shown in Figure 1.

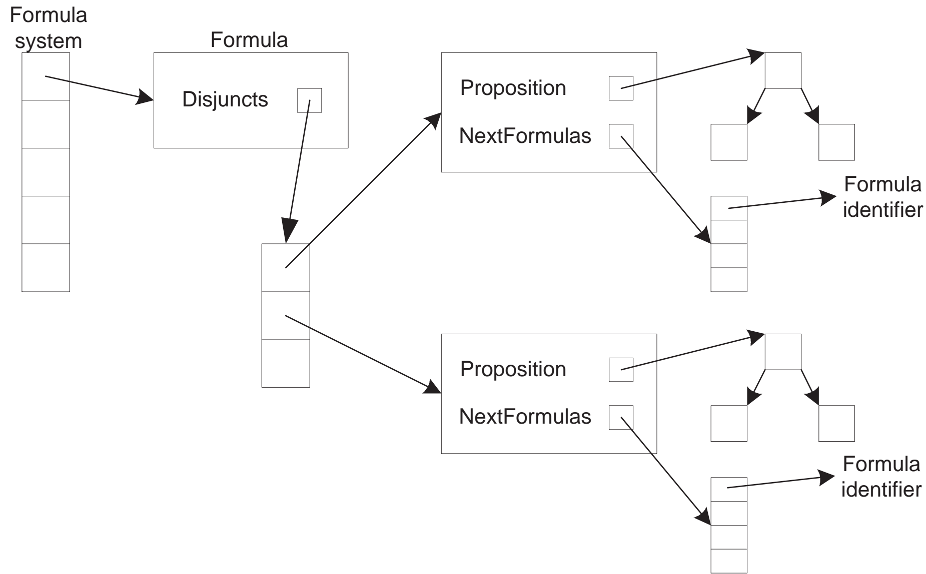

Fig. 1. *The datastructures for representing a Disjunctive Normalized Equation System.*

6.2 *The Condition Checker*

The Condition Checker, CD, is an implementation of the OK transformation in Section 5. The key element of the CD is the waiting list. It consists of a 3-dimensional list. A large list which is a disjunction of new lists that are a conjunction of formula pairs. The current implementation only takes care of one freeze variable x. Therefore each pair consists of an identifier, a formula, and a value which must be inserted for x in the identified formula. The waitinglist is shown in Figure 2. The intuition is that the state we try to verify must satisfy at least one of the disjuncts in the disjunction. If it does not we can say that the state fails to satisfy the condition given by the

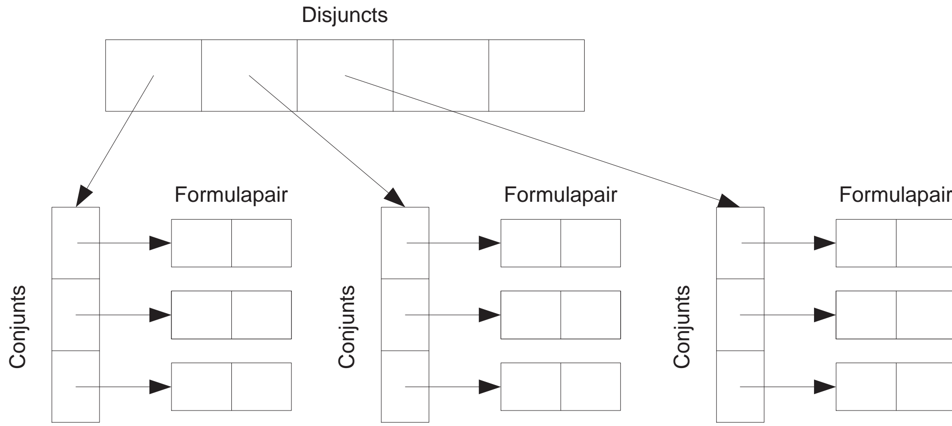

Fig. 2. *The waitinglist.*

formula system. To satisfy a given disjunct the state must satisfy the whole conjunction that is connected to that disjunct. This means that each conjunct in that conjunction must be satisfied. To satisfy a conjunct at least one of the disjuncts in the formula that the conjunct refers to must be satisfied.

When we have a combination of disjuncts from the different formulas in the conjunction, in which all propositions satisfy the state we are trying to verify, the next states from each of these disjuncts are put together in a new conjunction and placed in the waitinglist for the next state. If the result of one of these combinations is that nothing has to hold for the next state, then we can say that the trace we are trying to verify satisifies the temporal condition on which the formula system is based.

When we have traversed the waiting list we move the waitinglist's pointer to the next waiting list, and then we are ready to verify the next state.

7 Conclusion and Future Work

In this paper we have presented a framework for runtime verification of Timed LTL, a real time logic suitable for expressing time constrained safety and liveness properties of systems. This complements existing work on runtime verification of pure LTL and Past Time LTL. The core of our method is a disjunctive normal form and a corresponding property transformer, which given a state of a real–time system and a property to satisfy, returns the property to be satisfied for the remaining execution of the system. In cases where one of the constants $true$ or $false$ is returned, the verification process may stop and it can be concluded that the property is satisfied (or the opposite). Regarding scalability and completeness, we are still in the process of testing our prototype implementation and expect to invest a significant effort in experimenting with the introduction of heuristics for formula minimization as discussed in Section 5.

The assumption that timed traces must be complete, i.e. for every (discrete) instant in time the trace contains explicitely a state, see Section 2, may be relaxed. It will be possible to allow a more natural semantics of timed traces with the possibility of taking time steps of a size larger than one time unit, hereby improving performance by lowering the number of states that needs to be processed by the *Ok* transformer (and thus possibly keeping the size of the waiting list smaller). Our techniques would still be able to operate correctly in such a scenario given that a trace can be arbitrarily completed relative to a set of *relevant* points in time derived from the Timed LTL property under investigation. It is our plan to extend our framework to include this facility. In addition to this it will be interesting to re–think the whole scenario in a setting with a dense–time logic.

References

[1] K. Havelund, G. Ruso. Monitoring Java Programs with Java PathExplorer. First Workshop on Runtime Verification (RV'01), Paris, France, 23 July 2001. Electronic Notes in Theoretical Computer Science, Volume 55, Number 2, 2001

[2] D. Giannakopoulou, K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. Automated Software Engineering 2001 (ASE'01), San Diego, California, 26-29 November 2001, IEEE Computer Society.

[3] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple onthe -fly automatic verification of linear temporal logic. Proc. 15th International Symposium on Protocol Specification, Testing and Verification (PSTV XV), pp 318, Chapman and Hall, 1995.

[4] G. Holzmann. The Model Checker SPIN. IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

[5] Time-Rover Corp. Temporal Business Solutions. The DB Rover. 2000.

[6] R. Alur, T. Henzinger. Logics and Models of Real Time: A Survey. Real Time: Theory in Practice, Lecture Notes in Computer Science 600, Springer-Verlag, 1992, pp. 74-106.

[7] T. Henzinger. It's About Time: Real-Time Logics Reviewed. Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science 1466, Springer-Verlag, 1998, pp. 439-454.

[8] K. Havelund, G. Ruso. Monitoring Programs using Rewriting. Automated Software Engineering 2001 (ASE'01), San Diego, California, 26-29 November 2001, IEEE Computer Society.

[9] E. M. Clarke, O. Grumberg, D. A. Peled. Model Checking. The MIT Press 2001.

[10] J. S. David. Three Events that Defines an REA Methodology for Systems Analysis, Design and Implementation. 2001.

[11] K. G. Larsen, P. Pettersson and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. Proceedings of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, 5-7 December, 1995.

[12] H. R. Andersen. Partial Model Checking (Extended Abstract). Proceedings of LICS'95. La Jolla, San Diego, June 26-29, 1995, IEEE Computer Society Press, pp. 398-407.

[13] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific. J. Math., 55 (1955), pp. 285-309.