Checking Temporal Business Rules

Kåre J. Kristoffersen, Christian Pedersen, and Henrik R. Andersen

IT University of Copenhagen[*]
DK-2400 NV
Denmark
{kjk,cp,hra}@itu.dk

Abstract. In this paper we describe an event-based algorithm for runtime verification of timed linear temporal logic. The algorithm is based on a rewriting of the formula expressing a desired or undesired property of a timed system. Rewriting takes place, at discrete points in time, but only when there is a relevant state-change taking place in the timed system, or a deadline, determined by the formula, has been passed. By limiting the rewriting to only points in time where an event occurs, and not at all discrete time-points, makes the algorithm useful in situations where there are large data sets and large differences in the relevant time scales (ranging perhaps from milliseconds to months as in business software).

The algorithm works by rewriting, for each event, the timed LTL formula into a residual formula that takes into account the time and system state at the occurence of the event. The residual formula will be the requirement for the timed system in the future, to be further rewritten at the occurrence of the next event.

Keywords: Timed LTL, Disjunctive Normalized Equation Systems, Residual formula, Smallest Interesting Timepoint, Timed Based Fixed Point Reduction, Charaterization of Runtime Verification.

1 Introduction

Runtime verification is a branch of verification in which a running program is supervised by a concurrently running *verifier*. In this paper we shall look at *timed runtime verification*, in which time will be an important parameter in the task of the verifier. In [5] we suggested an algorithm which was capable of verifying complete timed traces which means that for each timepoint there is a corresponding state in the trace under investigation. In this paper our verifier will use a discrete notion of time but it is developed such that there will not be a discrete sampling of state at the choosen resolution but only at "relevant" points in time. The relevant points are given by events coming from state changes in the non-timed part of the timed system, and timeout defined by the formula. The verifier is capable of dealing with time even when the time scale spans from milliseconds to days or months. This is crucial for getting a useful verifier in systems with large amounts of data and events timed at milliseconds but with consequences that might be in days or months. This will be important in application areas such as business software (also

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

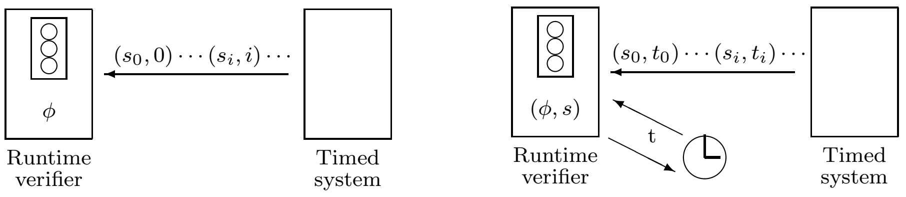


Fig. 1. A runtime verifier supervising a timed system. In the figure to the left, state information is computed and transmitted from the timed system at each discrete time point $0, 1, 2, \ldots$. The runtime verifier computes a new residual formula ϕ at each step, resulting in one of three situations, indicated by the three-coloured traffic light. "Red" corresponds to the situation where an error has been detected, i.e., the formula has reduced to false. "Green" corresponds to acceptance, i.e., the formula has reduced to true and the verifier can stop. "Yellow" corresponds to the situation where so far no error has been found but the verifier cannot yet stop. In the figure to the right the verifier only computes when an event happens, at the time points $t_0 < t_1 < \cdots$. An event is either a *state change* in the timed system or a timeout. Timeouts are set by the verifier and computed as what we shall refer to as the *smallest interesting timepoint* as given by the formula ϕ. A prototype, called TimeChecker, corresponding to the system at the right handside has been made.

sometimes commercially called Enterprise Resource Planning systems), where transactions occur at the millisecond level but for instance paying conditions might span weeks or months. Figure 1 illustrates the situation comparing a state-based "polling scheme" with an event-driven timed verifier.

Properties are formulated in a real-time logic called LTL_t. The logic LTL_t is an extension of LTL with real time constructs embodied by a freeze quantifier together with atomic clock constraints making it possible to express real time logical properties of a system (following [7]). This logic is suitable for expressing timed safety, liveness and fairness 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 could be written as follows (assuming the resolution and scale of time is days):

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

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 time stamps with real time at a given accuracy (for instance, milliseconds). The semantics of the *freeze* formula $x.\phi$ is that when evaluated, the value of x is replaced by the current time.

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 as a disjunctive normal form:

1. What should hold *now* (i.e. a *propositional* part).
2. What identifiers should hold in the *next* state (i.e. a *temporal* part).

Initially the property to be verified is a distinct identifier (the *top* identifier) of the equation system corresponding to the original formula. As the verifier proceeds this will be expanded to a disjunctive normal form of identifiers referred to as a residual formula. The number of equations in the normal form will be linear in the number of temporal operators of the original property. However, the total size might be exponential in the worst-case due to conversion of some sub-formulae into disjunctive-normal forms.

Related work

The traditional field of application for formal methods and especially verification is within safety critical and embedded systems for which correctness is of 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 input from a very restricted set of possibilities. This implies that the program under investigation can be regarded as a closed system, which may be checked alone.

Financial applications and business software (ERP Systems) as well as traditional back end databases are a class of applications that have attracted very little attention from the formal methods community. What needs to be verified in such a system is not the system software itself, but the data the system is managing. These data can be given 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 *time stamp* explaining *when* the state is measured. One reason that this have not gained very much interest from the verification community is that it is 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 ERP Systems differ from the beforementioned in that they can (and should be able to) consume unboundedly many different input data. As such they suffer severely from the well–known "State Explosion Problem" and consequently exhaustive verification is not feasible.

There could be several angles of attack on solving the timed run-time verification problem. One line could be to translate the timed LTL-formula into a timed automaton. The main challenge here is how to deal with the freeze operator. The logical clocks should correspond to clocks of the automaton, however, there is no simple bound of the number of needed clocks. Consider for instance a *punctuality* property like

$$\Box(p \Rightarrow x.\Diamond(q \wedge t = x + 1000)),$$

expressing that always, when p holds, then after exactly 1000 time units, q should hold. If p change as often as possible then it might change 500 times before the first time q should be verified to be true. All these time points must be remembered indicating that the automaton will need up to at least 500 clocks. It means that it would be impractical to try to compute a full automaton in advance and an on-the-fly construction employing a variation of alternating automaton seems to be a way to find a solution. In [11] this problem is avoided by limiting the expressive power of the timed logic to upper-bounds. They avoid having any acceptance condition and there is no blow-up in the number of neeeded clocks. We work with the full logic.

The approach taken here is that of computing a residual formula, taking into account the states seen sofar, and expressing the property to hold for the timed system in the future. This is a bit like computing "weakest pre-conditions" but in a forwards manner: a "strongest post-condition", as it has been classically done for program verification in the pioneering work of Dijkstra and Hoare.

The current approach also has a relationship to [16] and [15] in terms of rewriting a specification with respect to state information. In [16] 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 [15] 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 [16] and [15] in two substantial ways. First, by rewriting the specification to a normal form *prior* to verification we obtain a specification whose size is *fixed* throughout verification. Second, at runtime (corresponding to verification time in their work) we need only to maintain a formula which is a disjunctive normal form of 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. In Section 4 we present two algorithms for runtime verification. First **R**V-Simple for checking complete timed traces and second based on the notion of smallest interesting timepoint we present the algorithm **R**V-Advanced. In Section 5 we present a notion of time based fixed point reductions which improves the efficiency of our method, and in Section 5.1 we comment on how our techniques are implemented in our tool TimeChecker. Finally we conclude 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 $t \in I\!N$ 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 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 timed traces and we denote by $\hat{\Sigma}$ the set of all complete timed traces.*

We use superscripting with i for the sub-sequence σ^i, which starts at σ_i, i.e., the sequence $\sigma^i = \sigma_i\sigma_{i+1} \cdots$. An incomplete trace can always be completed and the verification algorithm proposed in [5] rested on an assumption that this was done. In this paper we will relax this condition, to allow verification of incomplete timed traces. As we shall see, it is indeed possible to check an incomplete trace directly provided that we take special care of inserting into the trace artificial states at certain relevant points in time.

Definition 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*, $c \in \mathbb{N}$ 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.

In the logic we only have negation at the level of propositions, 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. The semantics for the timed logic is available in Appendix A.

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. *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*

$$\begin{aligned} X_1(\boldsymbol{x}_1) &=_{\lambda_1} \phi_1 \\ &\vdots \\ X_n(\boldsymbol{x}_n) &=_{\lambda_n} \phi_n \end{aligned}$$

where for all $i \in \{1, \ldots, n\}$, $\boldsymbol{x}_i$ is a (possibly empty) vector of variables from V, such that $\boldsymbol{x}_i$ includes all the free variables in ϕ_i and each $\lambda_i \in \{\mu, \nu\}$ denotes either a minimal or a maximal fixed point. The right-hand sides ϕ_i are each on the following form :

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

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 $\boldsymbol{x}$ 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: There is a set of possibilities each of which describes:

1. What should hold *now* (i.e. a *propositional* part) and
2. Which 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: First, checking whether the current state is ok and second keeping track of what should hold for the future, i. e. what the residual formula should be. The normal form introduces the $\bigcirc$–operator as suggested by the functions $F_{\phi_1 U \phi_2,\epsilon}$ and $F_{\phi_1 V \phi_2,\epsilon}$, see Appendix A. We refer to [5] for a translation procedure from Timed LTL to Disjunctive Normal Form.

Example 4. 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 in [5] we obtain the following equation system $\mathcal{D}_\phi$:

$$\begin{aligned} X_0() &=_\mu x.(p \wedge t < x + 5 \wedge \bigcirc X_1(x)) \vee q \\ X_1(x) &=_\mu (p \wedge t < x + 5 \wedge \bigcirc X_1(x)) \vee q \end{aligned}$$

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 refer to X_0 as the 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. The number of identifiers in a normalform equation system will be equal to the number of temporal oprators in the formula plus 1.

4 Verification of Incomplete Timed Traces

In this section we present two algorithms for runtime verification of timed traces, RV-Simple and RV-Advanced. The first handles complete traces only, while the second treats incomplete traces. The core of both procedures is a residual formula construction '/' 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. *(Residual Formula) Let ϕ be an LTL_t formula, let $\mathcal{D}_\phi$ be the Normal Form Equation System for ϕ and let σ_0 be a timed state. Then the residual of ϕ with respect to σ_0 is obtained using the following transformations*

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

$$\left(\bigvee_{j\in J_i} \bigwedge_{l\in L_i j} X_l(\boldsymbol{a})\right)/\sigma_0 = \bigvee_{j\in J_i} \bigwedge_{l\in L_{ij}} X_l(\boldsymbol{a})/\sigma_0$$

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

In the definition above $\boldsymbol{x}$ denotes the formal parameters $x_1, \ldots, x_n$ (formula clocks) while $\boldsymbol{a}$ denotes the corresponding actual parameters. We will be using X_0 as the top identifier. The following theorem is the foundation for the algorithm **R**V-Simple, see Table 1.

Theorem 6. *Let $\sigma = \sigma_0\sigma_1 \ldots$ be a complete timed trace and let ϕ be a LTL_t formula in DNF. Then,*

$$\sigma \models \phi \textit{ iff } \sigma^1 \models \phi/\sigma_0$$

Proof: See Appendix C.

When dealing with an incomplete timed trace the algorithm RV-Simple does not apply anymore. By the simple approach of removing states one by one we run the risk of taking a timestep which is too large with respect to the property under consideration. To this end we introduce a notion of *smallest interesting timepoint* which exactly captues the deadlines imposed by a formula. In other words, the smallest interesting timepoint for a formula is the (future) point in time when the formula would change its truthhood. For instance, consider the time constraint $t < c$, where t is a discrete clock, which is moving forward, and c is a constant. Before time c this constraint is true. However, at time c it becomes false, and moreover, it will remain false. Therefore, if we for a moment assume that time is the only changing parameter in a timed trace, we may say that c is the *smallest interesting timepoint* for the time constraint $t < c$. The following definition of smallest interesting timepoint is a variant the "variable time advance" procedure [19].

Algorithm: RV-Simple
Let $\sigma = \sigma_0\sigma_1\ldots$ be a complete timed trace, and let ϕ be a DNF formula.

```
$\psi := \phi/\sigma_0$ $i := 1$
while $\psi \neq true \wedge \psi \neq false$ do
   $\psi := \psi/\sigma_i$
   $i{+}{+}$
end while
if $\psi == true$ then
   return "yes"
end if
if $\psi == false$ then
   return "no"
end if
```

Algorithm: RV-Advanced
Let $\sigma = \sigma_0\sigma_1\ldots$ be an incomplete timed trace, let ϕ be a DNF formula and let $exists(\sigma_i, c)$ be a predicate which is true exactly when σ contains σ_i and t_i=c.

```
$\psi := \phi/\sigma_0$ $i := 1$ $time := 1$ $sit := SIT(\psi)$
while $\psi \neq true \wedge \psi \neq false$ do
   if $exists(\sigma_i, time) \wedge s_i \neq s_{i-1}$ then
      $\psi := \psi/\sigma_i$
      $sit := SIT(\psi)$
      $i{+}{+}$
   end if
   if $time == sit$ then
      $\sigma_i := (s_{i-1}, j)$
      $\psi := \psi/\sigma_i$
      $sit := SIT(\psi)$
      $i{+}{+}$
   end if
   $time{+}{+}$
end while
if $\psi == true$ then
   return "yes"
end if
if $\psi == false$ then
   return "no"
end if
```

Table 1. Algorithms **R**V-Simple(left) and **R**V-Advanced.

Definition 7. *(Smallest Interesting Timepoint) Assume a normal form equation system over n identifiers $X_1, \ldots, X_n$. Let $J, L_j \in I\!N$, let f be a function mapping any pair (l, j) to a number in $\{1 \ldots n\}$ and (s, t) be a timed state.*

$$\begin{aligned}
SIT(\textstyle\bigvee_{j=1}^{J} \bigwedge_{l=1}^{L_j} X_{f(j,l)}(\boldsymbol{a}_{jl}), (s,t)) &= min\{SIT(D(X_{f(j,l)}(\boldsymbol{x}_{f(j,l)}))[\boldsymbol{a}_{jl}/\boldsymbol{x}_{f(j,l)}], (s,t)) \\
&\quad |1 \leq j \leq J, 1 \leq l \leq L_j\} \\
SIT(\textstyle\bigvee_{j=1}^{J} \phi_j, (s,t)) &= min\{SIT(\phi_j, (s,t)) | 1 \leq j \leq J\} \\
SIT(\boldsymbol{x}.(\psi \wedge \bigcirc \bigwedge_{i \in I} X_i(x)), (s,t)) &= \begin{cases} t+1 & \text{if } (s,t) \vdash \psi[t/\boldsymbol{x}] \text{ and } \boldsymbol{x} \text{ not empty} \\ \infty & \text{if } (s,t) \not\vdash \psi[t/\boldsymbol{x}] \text{ and } \boldsymbol{x} \text{ not empty} \\ SIT(\psi, (s,t) & \text{if } \boldsymbol{x} \text{ is empty} \end{cases} \\
SIT(\phi_1 \wedge \phi_2, (s,t)) &= SIT(\phi_1 \vee \phi_2, (s,t)) \\
&= min\{SIT(\phi_1, (s,t)), SIT(\phi_2, (s,t))\} \\
SIT(p, (s,t)) &= SIT(\neg p, (s,t)) = \infty \\
SIT(t > c, (s,t)) &= SIT(t \leq c, (s,t)) = c+1 \\
SIT(t = c, (s,t)) &= SIT(t < c, (s,t)) = SIT(t \geq c, (s,t)) = c
\end{aligned}$$

The only case where the smallest interesting timepoint will be as small as $t+1$, i.e. the time in the next state, is when (1) new clock variables are bound and (2) the propositional formula ψ is satisfied, since in this case new identifiers (previously hidden behind the $\bigcirc$–operator) are now instantiated in the residual formula. The reason why $SIT(p,(s,t)) = SIT(\neg p,(s,t)) = \infty$ is that no new clock variables have been bound, and henceforth no timevariables can timeout. The same argument can be used to explain why $SIT(\boldsymbol{x}.(\psi \wedge \bigcirc \bigwedge_{i \in I} X_i(x)),(s,t)) = \infty$ if $(s,t) \not\models \psi[t/\boldsymbol{x}]$ and $\boldsymbol{x}$ not empty .

The following theorem states that the residual computation need not take place for a timed state if it has the same propositional content as its predecessor and further that its time (i.e. the current time) is below the smallest interesting timepoint for the current residual. Moreover it prescribes that an artificial state has to be inserted when a smallest interesting timepoint (i.e. a deadline) is reached. The theorem is the foundation of algorithm RV-Advanced in Table 1.

Theorem 8. *Let $\sigma = \sigma_0 \sigma 1$ be a timed trace, let ϕ be a a LTL_t formula in disjunctive normal form, and let $sit = SIT(\phi/\sigma_0)$. Then,*

$$\begin{array}{l}
\textit{If } sit < t_1 \textit{ then, } \sigma \models \phi \textit{ iff } (s_0, sit)\sigma^1 \models \phi/\sigma_0 \\
\textit{If } sit = t_1 \textit{ then, } \sigma \models \phi \textit{ iff } \sigma^1 \models \phi/\sigma_0 \\
\textit{If } sit > t_1 \textit{ and } s_0 = s_1 \textit{ then, } \sigma \models \phi \textit{ iff } (s_1, sit)\sigma^1 \models \phi/\sigma_0 \\
\textit{If } sit > t_1 \textit{ and } s_0 \neq s_1 \textit{ then, } \sigma \models \phi \textit{ iff } \sigma^1 \models \phi/\sigma_0
\end{array}$$

Regarding correctness of both algorithms, if the answer is 'yes' then it holds that the trace satisfies the formula. Similarly, if the answer is 'no' then the trace does not satisfy the formula. Now ideally, for the algorithms to return a definite answer a tautology should be reduced to the constant value *true*, and similarly, an unsatisfiable formula should preferably be reduced to the constant value $false$. In general, this is not possible since satisfiability for logic considered is undecidable [7]. However, by application of heuristics in the style of [9, 16, 15] it is possible to apply a range of formula reductions towards a smaller specification (possibly *true* or $false$).

Example 9. Let us consider the punctuality property $\Box(p \Rightarrow x.(\Diamond(t = x + 10 \wedge q)))$ [7]. After translating to disjunctive normal form we get the following equation system:

$$\begin{array}{rl}
X_0 =_\nu & \neg p \wedge \bigcirc X_0 \\
& \vee x.(p \wedge true \wedge \bigcirc(X_1(x) \wedge X_0)) \\
& \vee x.(p \wedge q \wedge t = x + 10 \wedge \bigcirc X_0) \\
\\
X_1(x) =_\mu & \bigcirc X_1(x) \\
& \vee q \wedge t = x + 10
\end{array}$$

Checking the trace $\sigma = (\{p\},0)(\{\},1)(\{q\},11)\ldots$ we get that σ^1 must satisfy the residual formula $X_1(0) \wedge X_0$. Now, using Definition 7 we have that $SIT(X_1(0) \wedge X_0, (\{p\},0))$ is 1. But, since the trace already has a state where $t = 1$, we do not have to compute an artificial state here. When we compute the residual for $(\{\},1)$ the residual obtained is

$X_1(0) \wedge X_0$ and as a smallest interesting timepoint we get 10 since $SIT(X_1(0), (\{\}, 1)) = 10$ and $SIT(X_0, (\{\}, 1)) = \infty$. This means that since the next states t=11 we will have to insert an "artificial" state consisting of s_i from the previous state and the current sit which makes the trace look like this: $\sigma = (\{p\}, 0)(\{\}, 1)(\{\}, 10)(\{q\}, 11) \ldots$. Now since $(\{\}, 10) \not\models X_1(0) \wedge X_0$ we discover that σ fails to satisfy the condition, which we would not have discovered if we had just moved on verifying without inserting states.

5 Time Based Fixed Point Reductions

In exhaustive verification a property $\Diamond p$ (eventually p) will be concluded to be not satisfied if the state graph of the system being checked has an infinite p–free execution trace, i.e. a loop for which no states satisfies predicate p. In Runtime Verification, unfortunately, cycles go undetected and hence we are not able to reach such a conclusion. We are, however, able to do something else of great value, namely to exploit the passage of *time* as a basis of strengthening the completeness of our method. Consider for instance a formula $\Diamond(p \wedge t < 5)$ (p becomes true before time 5), which in disjunctive normal form becomes the minimal fixed point equation $X =_\mu \bigcirc X \vee (p \wedge t < 5)$. Obviously, if for a given trace, p is not satisfied at neither time $0, 1, 2, 3$ or 4 then the property is not satisfied no matter what the continuation might be. Unfortunately, our verification procedure does not quite capture this. For all timed states in which p does not hold, i.e. $(\{\}, i), i \in I\!N$ we get that $X/(\{\}, i) = X$ which means that after time 4 there is still a "hope", namely if X can be satisfied (but this is really a false hope!). Therefore, at time 5 we do need to *rewrite* the defining equation of X to $X =_\mu \bigcirc X \vee false$, which again is equivalent to $false$ since for this definition of X no finite number of unfoldings exists. In very much the same fashion we may conclude that time constraints with a lower bound (i.e. $t > c, t \geq c$) from certain timepoints (which in this particular case are $c + 1$ and c) becomes true and utilize this fact to rewrite maximal fixed point equations.

In the following we therefore introduce two things. First, a formula transformer tR (At Time t Reduction) whose purpose it is, given a (possibly closed) LTL_t formula ϕ and a timed state σ_0, to attempt to determine future truthhood (or falsehood) of timing constraints inside ϕ. And second, a fixed point reduction technique which attempts to use these rewritten equations to draw conclusions.

Definition 10. *(At Time t Reduction)*

Let $t_i \in N$ be a timepoint and let ϕ be the righthandside of an equation in Disjunctive Normal Form i.e. ϕ is on the form

$$\bigvee_{j \in J_i} \bigwedge_{l \in L_{ij}} X_{jl}(a_{lj})$$

Then the "At Time t reduction of ϕ" denoted $tR(\phi, t_i)$ is defined as follows:

$$
\begin{aligned}
tR(\psi_1 \vee \psi_2, t_i) &= tR(\psi_1, t_i) \vee tR(\psi_2, t_i)\\
tR(\psi_1 \wedge \psi_2, t_i) &= tR(\psi_1, t_i) \wedge tR(\psi_2, t_i)\\
tR(\bigcirc X, t_i) &= \bigcirc X\\
tR(x.\phi, t_i) &= x.\phi\\
tR(p, t_i) &= p\\
tR(\neg p, t_i) &= \neg p\\
tR(t < c, t_i) &= \begin{cases} \mathit{false} & \mathit{if}\ t_i \geq c\\ t < c & \mathit{otherwise}\end{cases}\\
tR(t \leq c, t_i) &= \begin{cases} \mathit{false} & \mathit{if}\ t_i > c\\ t \leq c & \mathit{otherwise}\end{cases}\\
tR(t = c, t_i) &= \begin{cases} \mathit{false} & \mathit{if}\ t_i > c\\ t = c & \mathit{otherwise}\end{cases}\\
tR(t > c, t_i) &= \begin{cases} \mathit{true} & \mathit{if}\ t_i > c\\ t > c & \mathit{otherwise}\end{cases}\\
tR(t \geq c, t_i) &= \begin{cases} \mathit{true} & \mathit{if}\ t_i \geq c\\ t \geq c & \mathit{otherwise}\end{cases}\\
tr(X, t_i) &= tR(D(x), t_i)
\end{aligned}
$$

Theorem 11. *Let $\sigma = \sigma_0\sigma^1$ be a timed trace, and let ϕ be a a LTL_t formula in disjunctive normal form. Then it holds that*

$$\sigma^1 \models \phi/\sigma_0 \ \mathit{iff}\ \sigma^1 \models tR(\phi/\sigma_0, t_0).$$

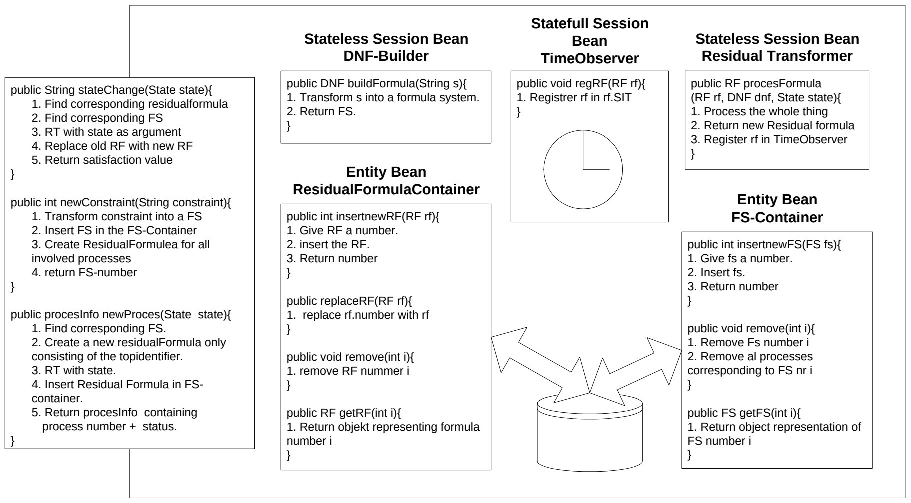
Proof: See Appendix B.

Returning to the example with the minimal fixed point equation $X =_\mu \bigcirc X \vee (p \wedge t < 5)$ we have that $tR(\bigcirc X \vee (p \wedge t < 5), i) = \bigcirc X \vee (p \wedge t < 5)$ for $i = 0, \ldots, 4$ which nicely resembles the fact that up to time 4 there is still a hope to satisfy the property. But $tR(\bigcirc X \vee (p \wedge t < 5), 5) = \bigcirc X \vee (p \wedge false) = \bigcirc X \vee false$ and thus the definition of X reduces to $X =_\mu \bigcirc X \vee false$ which does not have a solution.

In general we can say that if the righthandside of a minimal fixed point equation, $X =_\mu F(X)$, has no next–free disjuncts which are different from the constant $false$ then X is itself equivalent to false. And in our tool TimeChecker employ this to reduce the residual formula. A dual reduction for maximal fixed point equations exists: If the righthandside contains a disjunct for which the propositional requirement is not present the solution is all timed traces because the solution to $Y =_\nu \bigcirc Y$ is all timed traces.

5.1 Verificationserver

The verification server shown in Figure 2 is a j2EE implementation of the runtime verifier depicted in Figure 1. The server contains the original specifications to be checked as well as the dynamicaly computed residuals. It has an interface through which it gets information about updates of the ERP system being monitored. The program consists approximately of 10.000 lines of code.



```
public String stateChange(State state){
      1. Find corresponding residualformula
      2. Find corresponding FS
      3. RT with state as argument
      4. Replace old RF with new RF
      5. Return satisfaction value
}

public int newConstraint(String constraint){
      1. Transform constraint into a FS
      2. Insert FS in the FS-Container
      3. Create ResidualFormulea for all
      involved processes
      4. return FS-number
}

public procesInfo newProces(State  state){
      1. Find corresponding FS.
      2. Create a new residualFormula only
      consisting of the topidentifier.
      3. RT with state.
      4. Insert Residual Formula in FS-
      container.
      5. Return procesInfo  containing
         process number +  status.
}
```

```
public DNF buildFormula(String s){
1. Transform s into a formula system.
2. Return FS.
}
```

```
public void regRF(RF rf){
1. Registrer rf in rf.SIT
}
```

```
public RF procesFormula
(RF rf, DNF dnf, State state){
1. Process the whole thing
2. Return new Residual formula
3. Register rf in TimeObserver
}
```

```
public int insertnewRF(RF rf){
1. Give RF a number.
2. insert the RF.
3. Return number
}

public replaceRF(RF rf){
1.  replace rf.number with rf
}

public void remove(int i){
1. remove RF nummer i
}

public RF getRF(int i){
1. Return objekt representing formula
number i
}
```

```
public int insertnewFS(FS fs){
1. Give fs a number.
2. Insert fs.
3. Return number
}

public void remove(int i){
1. Remove Fs number i
2. Remove al processes
corresponding to FS nr i
}

public FS getFS(int i){
1. Return object representation of
FS number i
}
```

Fig. 2. The verification server.

The entire framework for runtime verification presented in this paper, is implemented in a tool called TimeChecker. The tool is a Java implementation of the righthandside of figure 1. The program is therefore two programs, a verifier and a small databasesystem, and is implementet as a client/server application. The program consists approximately of 10.000 lines of code.

The verification server is tested against a smaller databasesystem, which consists of a small database, and a instrumentation system where instrumentation objects about which verifiers the database system has to send messages to, if a database rows changes, is stored. When changes happens the instrumentation objects also contains the property of its atomic propositions, such that it can determine which of the atomic porpositions are true and which are false and write these in the message which it sends to its corresponding verifier.

The verifier is able to translate LTL_t formula into a disjunctive normalform. Besides that it also have a full implementation of the procedure sketched in [5] where the system, given a state and residual formula, is able construct a new residual formula with smallest interesting time point, and time based fixed point reduction.

For each object or row in the database system, there is a residual formula in the verifier. These residualformulas will have many common subtructures and hence a direct representation would consume an unnecessecarily large amount of storeage. Henceforth an effective storing procedure is necessary, and to accomodate for this we have used a hashcons procedure [18]. Experimental results have shown, that in case of 1000 residual formulas, a procedure where all the residual formulas is saved in normal tables takes 3 times as much space as the hashcons procedure. And as the sum of residual formulas increases, the factor of improvement also increases.

6 Conclusion and Future Work

In this paper we have improved our runtime verifier for timed LTL [5] from being state-based to become event-based, making the algorithm feasible for systems with big differences in the magnitudes of the relevant time points. We have expanded our previously presented framework for runtime verification of timed LTL, in such a way that we have improved both completness and performance. Completeness has been improved since we are able to detect timeouts in time constraints which is used in context of detecting if a minimial fixpoint formula can or cannot be satisfied within a limited number of formula unfoldings. The introduction of smallest interesting timepoint is a big performance improvement since we are now able to skip processing a lot of the statesequences, from the complete trace. Therefore it is also a big step towards a logic with dense time instead of discrete time.

In [11] an automaton-based solution is presented for the limitation of having only temporal operators with upper-bounds. It would be interesting to investigate to what extent restricting our logic to only upper bounds could allow for further improvements in the algorithm. It is for instance not difficult to see that if the residual formula is for instance $X_1(3) \wedge X_1(4) \wedge X_1(5)$ and there are only upper bounds in the equations, then this will be equivalent to $X_1(3)$ (because the two last will be implied by the first).

Another type of improvement to search for, could be in a better computation of the smallest interesting timepoint in the case of freeze-operators. In the current version this will sometimes simply be the next timepoint, which is the smallest possible step to take. This might happen with formulae as $\Box(p \Rightarrow x.\phi)$, where, whenever p holds, a new freeze must be performed. If p holds for a longer period, e.g., from timepoint 0 to 1000, many freeze-operations, being reflected as identifier-instantiations with different parameter values, will be performed. Here, it might be beneficial to instead use change-propositions expressing the change of a proposition like p. These would be $p \uparrow$ for p goes "high" and $p \downarrow$ for p goes "low" and express the properties with these instead. This would make the logic more like an interval logic (as e.g. duration calculus [12]) and the timing properties could be thought more of as properties of timing diagrams known from hardware design and analysis.

In general, timed LTL is undecidable ([7]), which means that it will not be possible to guarantee completeness in the sense that the verifier should in each step always be able to reduce the residual formula to false (if it is unsatisfiable) respectively true (if it is a tautology). However, for the fragments of timed LTL that are decidable, it should be possible to improve the verifier, by using the appropriate decision procedure.

Though our framework can be used in several contexts it is constructed with ERP-systems in mind. Therefore a future development will be J2EE implementation of the TimeChecker. This will make it capable of monitoring several processes concurrently, and automatically keep all the residualformulas as persistent data. In addition to this, we shall begin to address the important problem of how to *automatically instrument* an ERP system (or any other running program) with the ability to present its current timed state in a form which can be used by the TimeChecker.

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 on-the-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. K. Kristoffersen, C. Pedersen and H. R. Andersen. Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems. Appears in Preliminary Proceedings of Third International Workshop on Runtime Verification RV '03, Boulder, Colorado, USA. July 13, 2003. (Technical Report MS-CIS-03-21, University of Pensylvania). To appear in Issue 89.2 of Electronic Notes in Theoretical Computer Science.
6. Time-Rover Corp. Temporal Business Solutions. The DB Rover. 2000.
7. 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.
8. 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.
9. K. Havelund, G. Ruso. Monitoring Programs using Rewriting. Automated Software Engineering 2001 (ASE'01), San Diego, California, 26-29 November 2001, IEEE Computer Society.
10. M.C.W. Geilen, D.R. Dams. An on-the-Fly Tableau Construction for a Real-Time Temporal Logic. Proceedings of the Sixth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT2000, 20-22 September 2000, Lecture Notes in Computer Science Vol.1926 Pune, India, Ed. M. Joseph, pp. 276-290, Springer-Verlag, Berlin, 2000.
11. M.C.W. Geilen. An improved on-the-fly tableau construction for a real-time temporal logic. Computer Aided Verification, CAV 2003, Proceedings. Boulder, Colorado, USA, July 8 – 12, 2003.
12. Z. Chaochen, C.A.R. Hoare, A.P. Ravn: A Calculus of Durations, Information Processing Letter, 40, 5, pp. 269-276, 1991.
13. E. M. Clarke, O. Grumberg, D. A. Peled. Model Checking. The MIT Press 2001.
14. J. S. David. Three Events that Defines an REA Methodology for Systems Analysis, Design and Implementation. 2001.
15. 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.
16. 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.
17. A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific. J. Math., 55 (1955), pp. 285-309.
18. H. R. Andersen. An Introduction to Binary Decision Diagrams. Dept. of Information Technology, Technical University of Denmark, DK-2800, Denmark, 1997. pp.27.
19. G.S. Shedler. Regenerative Stochastic Simulation. Aca- demic Press, 1993.

A Semantics of Real Time Logics

Given an environment for formula clocks ϵ as a partial mapping from the set of clocks to $I\!N$, we can define the semantics $[\![\phi]\!]\epsilon \subseteq \hat{\Sigma}$ of a formula ϕ with free clock variables in the domain of an enviroment ϵ, inductively as follows:

$$
\begin{aligned}
[\![p]\!]\epsilon &= \{\sigma \in \hat{\Sigma} \mid p \in s(\sigma_0)\} \\
[\![\neg p]\!]\epsilon &= \{\sigma \in \hat{\Sigma} \mid p \not\in s(\sigma_0)\} \\
[\![\phi_1 \wedge \phi_2]\!]\epsilon &= [\![\phi_1]\!]\epsilon \ \cap \ [\![\phi_2]\!]\epsilon \\
[\![\phi_1 \vee \phi_2]\!]\epsilon &= [\![\phi_1]\!]\epsilon \ \cup \ [\![\phi_2]\!]\epsilon \\
[\![t \sim x + c]\!]\epsilon &= \{\sigma \in \hat{\Sigma} \mid t(\sigma_0) \sim \epsilon(x) + c\} \\
[\![x.\phi]\!]\epsilon &= \{\sigma \in \hat{\Sigma} \mid \sigma \in [\![\phi]\!]\epsilon[t(\sigma_0)/x]\} \\
[\![\phi_1 U \phi_2]\!]\epsilon &= \{\sigma \in \hat{\Sigma} \mid \exists i.\ (\sigma^i \in [\![\phi_2]\!]\epsilon \text{ and } \forall j, 0 \leq j < i.\ \sigma^j \in [\![\phi_1]\!]\epsilon)\ \} \\
[\![\phi_1 V \phi_2]\!]\epsilon &= \{\sigma \in \hat{\Sigma} \mid \forall i.\ \sigma^i \in [\![\phi_2]\!]\epsilon \text{ or} \\
&\qquad\qquad (\exists j.\ \sigma^j \in [\![\phi_1]\!]\epsilon \text{ and } \forall i, 0 \leq i \leq j.\ \sigma^i \in [\![\phi_2]\!]\epsilon)\ \} \\
[\![\bigcirc\phi]\!]\epsilon &= [\![\bigcirc]\!]([\![\phi]\!]\epsilon),
\end{aligned}
$$

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

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 $2^{\hat{\Sigma}}$ of subsets of $\hat{\Sigma}$ ordered by set-inclusion. Then for any monotonic function $F : 2^{\hat{\Sigma}} \rightarrow 2^{\hat{\Sigma}}$ there will be a minimum fixed-point $\mu F \subseteq \Sigma$ and a maximum fixed-point $\nu F \subseteq \Sigma$ [17]. Observe, that the following functions are monotonic:

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

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

Lemma 12. *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 [16] for an example of how this is done in the modal μ-calculus.

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.

Note that due to completeness of traces until corresponds to sometime in the future and next to the next time-instance only difference to untimed is the ability to record and make inequalities on time

B Proof of Theorem 11

Given that $\sigma^1 \models \phi/\sigma_0$, then we want to show that $\sigma^1 \models tR(\phi/\sigma_0, t_0)$

Method: Structural induction in LTL_t formulas.

Case p:

Assume that $\sigma^1 \models p$. Now we can see that $tR(p, t_0) = p$. So therefore we have $\sigma \models p$.

Case $\neg p$:

Assume that $\sigma^1 \models \neg p$. Now we can see that $tR(\neg p, t_0) = \neg p$. So therefore we have $\sigma \models \neg p$.

Case $t < c$:

Assume that $\sigma^1 \models t < c(\sigma^1 = (s_1, t_1)(s_2, t_2)\ldots)$. This must mean that $t_1 < c$. Now we consider $tR(t < c, t_0)$. Since that $t_0 < t_1 < c$ we must according to definition 10 get that $tR(t < c, t_0) = t < c$. Therefore we have that $\sigma^1 \models tR(t < c, t_0)$.

Case $t \leq c$:

Assume that $\sigma^1 \models t \leq c(\sigma^1 = (s_1, t_1)(s_2, t_2)\ldots)$. This must mean that $t_1 \leq c$. Now we consider $tR(t \leq c, t_0)$. Since that $t_0 < t_1 < c$ we must according to definition 10 get that $tR(t \leq c, t_0) = t \leq c$. Therefore we have that $\sigma^1 \models tR(t \leq c, t_0)$.

Case $t = c$:

Assume that $\sigma^1 \models t = c$. This must mean that $t_1 = c$. Now consider $tR(t = c, t_0)$. Since $t_0 < t_1 < c$ we get that $t_0 < c$. This gives according to definition 10 that $tR(t = c, t_0) = t = c$. Hence $\sigma^1 \models tR(t = c, t_0)$.

Case $t > c$:

Assume that $\sigma^1 \models t > c$. This means that $t_1 > c$. Now we consider $tR(t > c, t_0)$ We do not know $c's$ relation to (t_0). t_0 can be both larger and smaller than c. This means according to definition 10 that either $tR(t > c, t_0) = true$ or $tR(t > c, t_0) = t > c$. Now, if $tR(t > c, t_0) = true$ then it clearly holds that $\sigma^1 \models tR(t > c, t_0)$. But also if $tR(t > c, t_0) = t > c$ since σ^1 as stated earlier satisfy $t > c$.

Case $t \geq c$:

Assume that $\sigma^1 \models t \geq c$. This means that $t_1 \geq c$. Now we consider $tR(t \geq c, t_0)$ We do not know $c's$ relation to (t_0). t_0 can be both larger, smaller or equal to c. This means according to definition 10 that either $tR(t \geq c, t_0) = true$ or $tR(t \geq c, t_0) = t \geq c$. Now, if $tR(t \geq c, t_0) = true$ then it clearly holds that $\sigma^1 \models tR(t \geq c, t_0)$. But also if $tR(t \geq c, t_0) = t \geq c$ since σ^1 as stated earlier satisfy $t \geq c$.

Case $\bigcirc X$

$\bigcirc X$ is never reduced to anything else than it already is and is therefore trivially proved.

Case x.

x. is never reduced to anything else than it already is and is therefore trivially proved.

Case $\psi_1 \wedge psi_2$:

IH: $\sigma^1 \models \psi_1 \Leftarrow \sigma^1 \models tR(\psi_1, t_0)$ and $\sigma^1 \models \psi_2 \Leftarrow \sigma^1 \models tR(\psi_2, t_0)$

Assume that $\sigma^1 \models \psi_1 \wedge \psi_2$. This means that $\sigma^1 \models \psi_1$ and that $\sigma^1 \models \psi^2$. This gives according to our induction hypothesis that $\sigma^1 \models tR(\psi_1, t_0)$ and $\sigma^1 \models tR(\psi_1, t_0)$. According to definition 10 this gives us $\sigma^1 \models tR(\psi_1 \wedge \psi_2, t_0)$

Case $\psi_1 \vee \psi_2$:

IH: $\sigma^1 \models \psi_1 \Leftarrow \sigma^1 \models tR(\psi_1, t_0)$ or $\sigma^1 \models \psi_2 \Leftarrow \sigma^1 \models tR(\psi_2, t_0)$

Assume that $\sigma^1 \models \psi_1 \vee \psi_2$. This means that $\sigma^1 \models \psi_1$ or that $\sigma^1 \models \psi^2$. This gives according to our induction hypothesis that $\sigma^1 \models tR(\psi_1, t_0)$ or $\sigma^1 \models tR(\psi_1, t_0)$. According to definition 10 this gives us $\sigma^1 \models tR(\psi_1 \vee \psi_2, t_0)$

Case X:

Assume that $\sigma^1 \models X$. This means by definition that $\sigma^1 \models D(x)$. The other proofs so far has shown that if $\sigma^1 \models D(x)$ then we also get that $\sigma^1 \models tR(D(x), t_0)$. This gives us according to definition 10 $\sigma^1 \models tR(X, t_0)$

Given that $\sigma^1 \models tR(\phi/\sigma_0, t_0)$ then we want to show that $\sigma^1 \models \phi/\sigma_0$.

Method: Structural induction in LTL_t formulas.

Case p:

Assume that $\sigma^1 \models tR(p, t_0)$. We will show that $\sigma^1 \models p$ under this assumption. According to definition 10 $tR(p, t_0) = p$ so therefore does $\sigma^1 \models p$.

Case $\neg p$:

Assume that $\sigma^1 \models tR(\neg p, t_0)$. We will show that $\sigma^1 \models \neg p$ under this assumption. According to definition 10 $tR(\neg p, t_0) = p$ so therefore does $\sigma^1 \models \neg p$.

Case $t < c$:

Assume that $\sigma^1 \models tR(t < c, t_0)$. We will show that $\sigma^1 \models t < c$ under this assumption. Since that false is not satisfied with any trace (And therefore not σ^1 either), then we must have that $tR(t < c, t_0) = t < c$, so this gives us that $\sigma^1 \models t < c$.

Case $t \leq c$:

Assume that $\sigma^1 \models tR(t \leq c, t_0)$. We will show that $\sigma^1 \models t \leq c$ under this assumption. Since that false is not satisfied with any trace (And therefore not σ^1 either), then we must have that $tR(t \leq c, t_0) = t \leq c$, so this gives us that $\sigma^1 \models t \leq c$.

Case $t > c$

Assume that $\sigma^1 \models tR(t > c, t_0)$. We will show that $\sigma^1 \models t > c$ under this assumption. $tR(t > c, t_0)$ has two possible outcomes. If $tR(t > c, t_0) = t > c$, then it is trivially true that $\sigma^1 \models t > c$. If $tR(t > c, t_0) = true$ then, according to definition 10, it means that $t_0 > c$ which gives us that $t_1 > c$. When $t_1 > c$ where t_1 is the timestamp of $sigma_1$ (The first timed state of σ^1), then we must have that $\sigma^1 \models t > c$.

Case $t \geq c$

Assume that $\sigma^1 \models tR(t \geq c, t_0)$. We will show that $\sigma^1 \models t \geq c$ under this assumption. $tR(t \geq c, t_0)$ has two possible outcomes. If $tR(t \geq c, t_0) = t \geq c$, then it is trivially true that $\sigma^1 \models t \geq c$. If $tR(t \geq c, t_0) = true$ then, according to definition 10, it means that $t_0 \geq c$ which gives us that $t_1 > c$. When $t_1 > c$ where t_1 is the timestamp of $sigma_1$ (The first timed state of σ^1), then we must have that $\sigma^1 \models t \geq c$.

Case $\bigcirc X$

$\bigcirc X$ is never reduced to anything else than it already is and is therefore trivially proved.

Case x.

x. is never reduced to anything else than it already is and is therefore trivially proved.

Case $\psi_1 \wedge \psi_2$

Induction Hypothesis: $\sigma^1 \models tR(\psi_1\, t_0) \Rightarrow \sigma^1 \models \psi_1$ og $\sigma^1 \models tR(\psi_2\, t_0) \Rightarrow \sigma^1 \models \psi_2$

Assume that $\sigma^1 \models tR(\psi_1 \wedge \psi_2, t_0)$. We will show that $\sigma^1 \models \psi_1 \wedge \psi_2$ under this assumption. If $\sigma^1 \models tR(\psi_1 \wedge \psi_2, t_0)$ then it means that $\sigma^1 \models tR(\psi_1, t_0) \wedge tR(\psi_2, t_0)$

which also means that $\sigma^1 \models tR(\psi_1, t_0)$ and $\sigma^1 \models tR(\psi_2, t_0)$. According to our induction hypothesis this gives us that $\sigma^1 \models \psi_1$ and $\sigma^1 \models \psi_2$ which leads to $\sigma^1 \models \psi_1 \wedge \psi_2$.

Case $\psi_1 \vee \psi_2$

Induction Hypothesis: $\sigma^1 \models tR(\psi_1\, t_0) \Rightarrow \sigma^1 \models \psi_1$ og $\sigma^1 \models tR(\psi_2\, t_0) \Rightarrow \sigma^1 \models \psi_2$

Assume that $\sigma^1 \models tR(\psi_1 \vee \psi_2, t_0)$. We will show that $\sigma^1 \models \psi_1 \vee \psi_2$ under this assumption. If $\sigma^1 \models tR(\psi_1 \vee \psi_2, t_0)$ then it means that $\sigma^1 \models tR(\psi_1, t_0) \vee tR(\psi_2, t_0)$ which also means that $\sigma^1 \models tR(\psi_1, t_0)$ or $\sigma^1 \models tR(\psi_2, t_0)$. According to our induction hypothesis this gives us that $\sigma^1 \models \psi_1$ or $\sigma^1 \models \psi_2$ which leads to $\sigma^1 \models \psi_1 \vee \psi_2$.

Case X

Assume that $\sigma^1 \models tR(X, t_0)$. We will show that $\sigma^1 \models X$ under this assumption. $\sigma^1 \models tR(X, t_0)$ is according to definition 10 the same as $\sigma^1 \models tR(D(x), t_0)$. The previous cases has showed us that we can rewrite that to $\sigma^1 \models D(x)$ which according to definition 10 can be rewritten to $\sigma^1 \models X$

C Proof of Theorem 6

(Direction $\Rightarrow$) :

Assume that $\sigma \models \phi$ then we will show that $\sigma^1 \models \phi/\sigma_0$

Since ϕ is in disjunctive normal form we have that

$$\sigma \models \bigvee_{j \in J_i} x_i.(\psi_{ij} \wedge \bigcirc \bigwedge_{l \in L_{ij}} X_l(x_l))$$

This means that there is a j such that

$$\sigma \models \psi_{ij}[t_0/x_i] \text{ and that } \sigma^1 \models \bigwedge_{l \in L_{ij}} X_l(x_l)$$

since the $\bigcirc$–operator refers to the next timepoint. Now, this can be reduced to

$$\sigma^1 \models \sigma_0 \vdash \psi[t_0/x_i] \wedge \bigwedge_{J \in L_{ij}} X_l(x_l)$$

which may safely be rewritten to

$$\sigma^1 \models \bigvee_{j \in J_i} (\sigma_0 \vdash \psi[t_0/x_i] \wedge \bigwedge_{J \in L_{ij}} X_l(x_l))$$

which can be reduced to:

$$\sigma^1 \models \phi/\sigma_0$$

(Direction $\Leftarrow$) :

Assume that $\sigma^1 \models \phi/\sigma_0$. We will show that $\sigma_0\sigma^1 \models \phi$ First, we notice that ϕ/σ_0 is on the form:

$$\bigvee_{i \in J} \bigwedge_{l \in L_j} X_l(x_l).$$

Then for some J', ϕ must have been on the form:

$$\bigvee_{i \in J \cup J'} x_l.(\phi_j \wedge \bigcirc \bigwedge_{l \in L_j} X_l(x_l))$$

and there must have been a $j \in J \cup J'$ such that $\sigma_0 \vdash \psi_j[t_0/x]$. This gives us that:

$$\sigma_0\sigma^1 \models \bigvee_{j \in J \cup J'} x.(\psi_j \wedge \bigcirc \bigwedge_{l \in L_j} X_l(x_l))$$

which leads to $\sigma_0\sigma^1 \models \phi$.