Bounded Model-checking of Discrete Duration Calculus *

Quan Zu and Miaomiao Zhang
School of Software Engineering
Tongji University
Shanghai, China
{7quanzu,miaomiao}@tongji.edu.cn

Jiaqi Zhu and Naijun Zhan*
State Key Lab. of Comp. Sci.
Institute of Software, CAS
Beijing, China
{zhujq,znj}@ios.ac.cn

ABSTRACT

Fränzle and Hansen investigated the model-checking problem of the subset of Duration Calculus without individual variables and quantifications w.r.t. some approximation semantics by reduction to the decision problem of Presburger Arithmetic, thus obtained a model-checking algorithm with 4-fold exponential complexity [6, 7]. As an alternative, inspired by their work, we consider the bounded model-checking problem of the subset in the context of the standard discrete-time semantics in this paper. Based on our previous work [20], we reduce this problem to the reachability problem of timed automata. The complexity of our approach is singly exponential in the size of formulas and quadratic in the number of states of models. We implement our approach using UPPAAL and demonstrate its efficiency by some examples.

Keywords

Model Checking, Duration Calculus, Timed Automata

1. INTRODUCTION

In their seminal work [24], Zhou *et al* introduced the notion of durations of states into Interval Temporal Logic (ITL) [13] for specifying and reasoning about quantitative properties of real-time and hybrid systems and founded Duration Calculus (DC). DC is a very expressive interval-based logic for real-time and hybrid systems at a very abstract level, which is thought as a new trend in formal design of real-time and hybrid systems [8] and has been widely and successfully applied in practice [22]. However, because of its expressiveness, the dark side of DC is the high undecidability of its decision procedure and model-checking issues [23] in general, unless the notion of duration, the use of negation and chop (the only modality in DC), or the models considered are severely constrained, e.g. [23, 21, 4, 5, 9, 12, 15, 14, 17].

*The first two authors are funded by NSFC 61073022 and the Fundamental Research Funds for the Central Universities (No.2100219031), and the last two authors are funded by NSFC-91118007, NSFC-60970031 and National Science and Technology Major Project of China (No. 2012ZX03039-004).
*The corresponding author

Linear duration invariants (LDIs) form an important subset of DC, as many safety properties of real-time systems can be defined with them. For instance, in the gas burner example [16], it is easy to specify the requirement that for any observed interval of length greater than or equal to 60, the duration of the Leak is not greater than one twentieth by an LDI as

$$60 \leq \ell \Rightarrow (19\textstyle\int\text{Leak} - \int\neg\text{Leak}) \leq 0.$$

In [21], the model-checking problem of LDIs is reduced to the linear programming problem and therefore solvable, where models are given by *real-time automata*. While in [3, 11, 17, 19], the authors investigated the model-checking problem of LDIs over *timed automata*.

Whether it is possible to find out a larger subset of DC whose decision and/or model-checking problem are/is decidable is quite interesting. An obvious solution is to investigate the extension of LDIs with Boolean connectives and the chop (we call such extension ELDIs for short). However, unfortunately, according to the results given in [23], the decision problem of ELDIs becomes undecidable both in the discrete time and continuous time settings. Moreover, in [6, 7], Fränzle and Hansen pointed out that the model-checking problem of ELDIs over finite-state Kripke structures turns out to be undecidable also both in the discrete time and continuous time settings. Motivated by this observation, they proposed an approximation semantics for ELDIs, called *doubly situation based semantics*, and showed that its model-checking is decidable in the discrete time setting with cubic complexity in the number of states of the model and linear in the size of the formula. However, further observation indicates that the approximation semantics is too coarse to be useful in practice [7]. So, the authors refined the semantics to another approximation semantics called *counting semantics* and reduced the model-checking problem of ELDIs to Presburger Arithmetic with 4-fold exponential complexity. So, two obstacles hinder the application of their approach:

1. The first one is the approximation semantics. According to their approach, one can only prove/disprove those formulas that can be approximated to be **true/false** over the given model represented by a finite-state Kripke structure, but cannot say anything about other formulas;

2. The second one is the efficiency as explained above.

In this paper, motivated by Fränzle and Hansen's work, as an alternative, we give a more efficient algorithm for model-checking a subset of ELDIs over timed automata in the context of the standard discrete-time semantics. ELDI formulas considered here are of the form $a \leq \ell \leq b \Rightarrow \phi$, where ϕ is an ELDI formula defined in [5, 6] (see the definition given later), a is a natural number, and b is a natural number or ∞. When $a = 0$ and $b = \infty$, this case exactly corresponds to the subset of DC considered in [5, 6]. However, in this paper, we only focus on the case when b is bounded, which means all reference intervals should be with a bounded length. In other words, we just investigate "bounded model-checking" of ELDIs. The solution is based on the technique developed in our previous work [20]. The basic idea is as follows: For a given timed automaton $\mathcal{A}$ and an ELDI formula Φ, we first construct an auxiliary automaton $\mathcal{S}$ to count the observation time of $\mathcal{A}$, and meanwhile, to check whether Φ is satisfied or not at every integral time point whenever the observation time is within the scope; then for the product $\mathcal{A}\|\mathcal{S}$, we use a CTL formula to characterize all the failure states at which the checking procedure returns **false**. So, $\mathcal{A} \models \Phi$ is reduced to verifying that none of these failure states is reachable in $\mathcal{A}\|\mathcal{S}$, i.e., the formula is not satisfied by the product. The hardest part is to design an algorithm called *BMC-DC*, to check at every integral time point whether or not the given formula is satisfied on any reachable execution segment whose length is within the bound. *BMC-DC* is executed as an action when some transitions of the auxiliary automaton $\mathcal{S}$ happen. This allows us to easily implement our approach in the model checker UPPAAL, and we will demonstrate the efficiency of our approach by some examples.

The rest of the paper is organized as follows: Section 2 recalls some basic notions of timed automata and Duration Calculus. Section 3 explains the basic idea of our approach by some running examples, while the detail is given in Section 4. Section 5 reports the implementation and experimental results. Section 6 concludes this paper and discusses the future work.

2. PRELIMINARIES

In this section, we introduce timed automata with discrete time that will be used as models, and ELDIs that will be used as specification language for real-time systems.

For convenience, we fix a finite set of *state variables* $\mathcal{P}$, ranged by $P, Q, \cdots$, and let $\mathcal{L} = 2^{\mathcal{P}}$ in the rest of this paper.

2.1 Timed Automata with Discrete Time

A timed automaton [1] is a finite state machine equipped with a set of clocks. In our case, we use a subset of $\mathcal{P}$ to represent a state (location), and a set X of integer valued variables to represent the clocks. Let $\Delta(X)$ be the set of clock constraints on X, which are conjunctions of formulas of the form $x \leq c$ or $c \leq x$, where $x \in X$ and $c \in \mathbb{N}$. Formally,

Definition 1. A timed automaton $\mathcal{A}$ is a tuple $\mathcal{A} = (L, l_0, \Sigma, X, E, I)$, where $L \subseteq \mathcal{L}$ is a finite set of locations; $l_0 \in L$ is the initial location; Σ is a finite set of actions; X is a finite set of clocks; I is a mapping that assigns each location $l \in L$ with a clock constraint $I(l) \in \Delta(X)$ called the invariant at l; $E \subseteq L \times \Sigma \times \Delta(X) \times 2^X \times L$ is a relation among locations, whose elements are called edges labeled with an action, a guard and a set of clocks to be reset.

A clock interpretation ν for the set of clocks X is a mapping that assigns a natural number to each clock. For $t \in \mathbb{N}$, let $\nu + t$ denote the clock interpretation which maps each clock $x \in X$ to $\nu(x) + t$. For $\lambda \subseteq X$, let $\nu[\lambda = 0]$ denote the clock interpretation which assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clocks.

A state of automaton $\mathcal{A}$ is a pair (l, ν), where l is a location of $\mathcal{A}$ and ν is a clock interpretation which satisfies the invariant $I(l)$. State (l_0, ν_0) is the initial state, where $\nu_0(x) = 0$ for any clock $x \in X$.

Let $\mathcal{A}$ be a timed automaton,

1. A *run* or an *execution* r of $\mathcal{A}$ is an infinite sequence of the form

$$r \ : \ (l_0, \nu_0) \xrightarrow[t_1]{a_1} (l_1, \nu_1) \xrightarrow[t_2]{a_2} (l_2, \nu_2) \xrightarrow[t_3]{a_3} \cdots$$

with $l_i \in L$ and ν_i is a clock interpretation, for $i \geq 0$, satisfying the following requirements:

Initiation: (l_0, ν_0) is the initial state.

Consecution: for all $i > 0$, either there is an edge in E of the form $(l_{i-1}, a_i, \delta_i, \lambda_i, l_i)$ such that $(\nu_{i-1} + t_i - t_{i-1})$ satisfies δ_i and ν_i equals $(\nu_{i-1} + t_i - t_{i-1})[\lambda_i = 0]$; or $l_{i-1} = l_i$, $\nu_i = \nu_{i-1} + (t_i - t_{i-1})$, $a_i = t_i - t_{i-1}$ to denote an action to delay $t_i - t_{i-1}$ time units, and for any $0 \leq t < t_i - t_{i-1}$, $\nu_{i-1} + t$ satisfies $I(l_{i-1})$.

2. A *behaviour* ρ corresponding to the above run r, is the infinite sequence of timed locations

$$\rho \ : \ (l_0, t_0)(l_1, t_1) \cdots (l_n, t_n) \cdots$$

satisfying the following conditions: (1) $t_0 = 0$; (2) for any $T \in \mathbb{N}$, there is some $i \geq 0$ such that $t_i \geq T$, which guarantees **divergence** and **nonzeno**.

Note that t_i is the instant that $\mathcal{A}$ enters l_i, for all $i \geq 0$. This means that it stays in l_{i-1} for $t_i - t_{i-1}$ time units and then transits to l_i in the run. Also, in this paper we use a sequence of time-stamped locations to denote a behaviour instead of a sequence of time-stamped switches as in other papers.

The product of several timed automata is defined in a standard way (please refer to [1, 2]). As in UPPAAL [2], each component timed automaton can be associated with a priority in a product. Priorities among the component timed automata are specified on the system level using a partial order $<$ to indicate that the right component timed automaton has a higher priority.

2.2 Extended Linear Duration Invariants

ELDIs with $\mathcal{P}$ investigated in [5, 6] consist of three syntactic categories, which are state expressions S, linear duration formulas (LDFs) $\mathcal{D}$, and ELDI formulas ϕ. The BNFs for them are as follows:

$$\begin{aligned} S \ &::= \ 0 \mid P \mid \neg S \mid S_1 \vee S_2 \\ \mathcal{D} \ &::= \ \sum_{i \in \Omega} c_i \textstyle\int S_i \leq c \\ \phi \ &::= \ \mathcal{D} \mid \neg \phi \mid \phi_1 \vee \phi_2 \mid \phi_1 ; \phi_2 \end{aligned}$$

where c_is and c are integers, and Ω is a finite set of indices.

As the convention of DC, ℓ is defined as $\int 1$, denoting the length of the reference interval. The Boolean value **true**, denoted by $\top$, is defined by $\ell \geq 0$, falling in ELDIs. Obviously, each ELDI formula can be represented by the form $a \leq \ell \leq b \Rightarrow \phi$, where a is a natural number, b is either a natural number or ∞, and ϕ is defined as above. In this paper, we only focus on the case when b is bounded, and will represent an ELDI of this form by $\Phi, \Psi, \cdots$, possibly with superscript and subscript in the sequel.

Given a timed automaton $\mathcal{A}$, each of its behaviours ρ, defines an interpretation $\mathcal{I}_\rho$ of ELDIs in the following way:

State expressions: $\mathcal{I}_\rho(0)(t) = 0$ for any $t \in \mathbb{N}$;

$$\mathcal{I}_\rho(P)(t) = \begin{cases} 1 & \text{if } t_{i-1} \leq t < t_i \wedge P \in l_{i-1} \text{ for some } i > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\mathcal{I}_\rho(\neg S)(t) = 1 - \mathcal{I}_\rho(S)(t)$;

$\mathcal{I}_\rho(S_1 \vee S_2)(t) = \max\{\mathcal{I}_\rho(S_1)(t), \mathcal{I}_\rho(S_2)(t)\}$.

Durations: given an interval $[t_1, t_2]$, where $t_1, t_2 \in \mathbb{N}$ and $t_1 \leq t_2$, then $\int S$ is interpreted by $\int_{t_1}^{t_2} \mathcal{I}_\rho(S)(t)dt$.

Formulas: given an interval $[t_1, t_2]$ as above, an ELDI formula ϕ is interpreted by

$\mathcal{I}_\rho, [t_1, t_2] \models \sum_{i\in\Omega} c_i \int S_i \leq c$ iff $\sum_{i\in\Omega} c_i \mathcal{I}_\rho(\int S_i, [t_1, t_2]) \leq c$;

$\mathcal{I}_\rho, [t_1, t_2] \models \neg\phi$ iff $\mathcal{I}_\rho, [t_1, t_2] \not\models \phi$;

$\mathcal{I}_\rho, [t_1, t_2] \models \phi_1 \vee \phi_2$ iff $\mathcal{I}_\rho, [t_1, t_2] \models \phi_1$ or $\mathcal{I}_\rho, [t_1, t_2] \models \phi_2$;

$\mathcal{I}_\rho, [t_1, t_2] \models \phi_1 ; \phi_2$ iff $\mathcal{I}_\rho, [t_1, t] \models \phi_1$ and $\mathcal{I}_\rho, [t, t_2] \models \phi_2$ for some $t \in [t_1, t_2] \cap \mathbb{N}$.

In the above, $(\mathcal{I}_\rho, [t_1, t_2])$ is called an ELDI model of $\mathcal{A}$, and we denote $\mathcal{M}(\mathcal{A})$ the set of all ELDI models of $\mathcal{A}$. We say $\mathcal{A} \models \phi$ iff $M \models \phi$ for any $M \in \mathcal{M}(\mathcal{A})$.

Notice that using the axioms of DC [22], it is easy to transform each ELDI formula to an equivalent one in which all state expressions are of the form $P_1 \wedge \cdots \wedge P_n$, where $P_i \in \mathcal{P}$. Thus, hereafter, we assume all ELDIs are of this form unless otherwise stated.

3. BASIC IDEA AND RUNNING EXAMPLES

Given an ELDI $\Phi = (a \leq \ell \leq b \Rightarrow \phi)$ and a timed automaton $\mathcal{A}$, our approach for checking $\mathcal{A} \models \Phi$ is sketched as follows: Firstly, we construct an auxiliary automaton $\mathcal{S}$ that is parallel with $\mathcal{A}$ and can be triggered at any reachable state of $\mathcal{A}$ to check if ϕ is satisfied on any execution segment of $\mathcal{A}$ starting from the state whose length is in between a and b. Then we define a CTL formula to characterize the set of failure states at which the checking algorithm returns **false**. Thus, $\mathcal{A} \models \Phi$ iff the CTL formula is not satisfied by $\mathcal{A} \| \mathcal{S}$.

The auxiliary automaton $\mathcal{S}$ is given in Figure 1, in which

- there are three states: the initial state, `p0` and `p1`, and the invariants at `p0` and `p1` are both $x \leq 1$;

- there are five transitions: one from the initial state to `p0` with an action represented by the procedure *Init* to analyze the formula to be checked; one from `p0` to `p1` with an action represented by the checking algorithm *BMC-DC* from which the reference interval is counted and the checking algorithm is triggered; one self-transition at `p0` that keeps idle so that the checking algorithm can be triggered on arbitrary reference interval starting from any reachable state; and two self-transitions at `p1`. The first self transition labeled with *BMC-DC* keeps checking the formula at any integral time points of the reference interval whose length is still within the scope, while the second one does nothing whenever the reference interval is beyond the given scope. The details of *Init* and *BMC-DC* will be given in the next section.

- there are two clocks: gc is a local variable to record the length of the observed interval starting from a reachable state; x is a local clock variable with an initial value 1, to be reset to 0 whenever its value is 1, which is used to indicate only integral time points are observed, and the checking algorithm should be triggered and executed only at these points.

Obviously, the set of failure states $\mathcal{F}$ of $\mathcal{A} \| \mathcal{S}$ are the ones in which *BMC-DC* returns **false**, which can be characterized by a CTL formula $\psi \mathrel{\hat{=}} \texttt{E<>}\neg$*BMC-DC*$()$. When b is finite, it is easy to see that $\mathcal{A} \models \Phi$ iff $A \| \mathcal{S} \not\models \psi$.

In order to guarantee the elapse of clocks is never blocked by other actions in the product $\mathcal{A} \| \mathcal{S}$, it is required that $\mathcal{S}$ has a higher priority in $\mathcal{A} \| \mathcal{S}$, i.e., $\mathcal{S} < \mathcal{A}$.

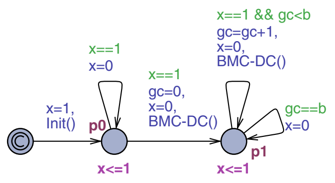


Figure 1: Auxiliary automaton $\mathcal{S}$

3.1 Running Examples

First of all, we use some examples to show how to use *BMC-DC* to check if a given ELDI formula is satisfied on a considered execution segment.

Let's consider an execution segment

$$\rho = (\{P_0\}, 0)(\{P_1\}, 1)(\{P_2\}, 2)(\{P_3\}, 3)(\{P_4\}, 4)(\{P_5\}, 5),$$

and five LDFs $\int P_0 - \int P_1 + \int P_2 + \int P_3 + \int P_4 \leq 0$, $2\int P_1 + \int P_2 - \int P_3 \leq 0$, $-\int P_0 + 2\int P_2 - 2\int P_4 \leq 0$, $\int P_0 \leq 0$ and $\int P_3 \leq 0$, denoted by $\mathcal{D}_1, \cdots, \mathcal{D}_5$ respectively. Moreover, for each $\mathcal{D}_i$, we introduce a variable d_i.

Firstly, consider to check the above segment against a simple formula $5 \leq \ell \leq 5 \Rightarrow \mathcal{D}_1 ; \mathcal{D}_2$. A natural way is to check its satisfiability by considering $\mathcal{D}_1$ on $[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5]$, correspondingly, $\mathcal{D}_2$ on $[0, 5], [1, 5], [2, 5], [3, 5], [4, 5], [5, 5]$, according to the semantics. Thus, the duration expressions of $\mathcal{D}_1$ and $\mathcal{D}_2$ have to be calculated on all the corresponding subintervals. Alternatively, we give a more efficient approach by using the notion of *optimal potential chop point* (OPCP). A time point is called *potential chop point* (PCP) if $\mathcal{D}_1$ is satisfied on the interval from the start point to this point, and a PCP is called *optimal* if the value of the duration expression of $\mathcal{D}_2$ on the interval from this PCP to the current point is minimal among all its values on these intervals from a PCP to the current point. We calculate the values of the duration expressions of $\mathcal{D}_1$ and $\mathcal{D}_2$ step by step from the beginning point to the end point of the execution segment, and record the value of the duration expression of $\mathcal{D}_1$ on the segment from the beginning time point to the current time point by d_1, and that of $\mathcal{D}_2$ on the segment from the OPCP to the current time point by d_2. Meanwhile, reset d_2 to 0 whenever the OPCP is updated. In this example, at the beginning, d_1 is set to 0, and d_2 to 11, which is large enough to guarantee that the duration expressions of $\mathcal{D}_1$ and $\mathcal{D}_2$ on the execution segment are always smaller than this value. At $t = 0$, obviously, $\mathcal{D}_1$ is satisfied as $d_1 \leq 0$, so this point is the current OPCP and the duration expression of $\mathcal{D}_2$ needs to be computed from scratch and hence d_2 is reset to 0. At $t = 1$, d_1 is increased by 1 (staying at $\{P_0\}$ for one time unit), so $t = 1$ cannot be a PCP. At $t = 2$, d_1 is changed to 0 and d_2 is updated to 2. Now, $t = 2$ becomes the new OPCP as $d_2 > 0$, and accordingly, d_2 is reset to 0 again. Repeat the above procedure until $t = 5$. Then we can conclude the formula is satisfied as $d_2 = 0 \leq 0$. In our approach, the duration expressions of $\mathcal{D}_1$ and $\mathcal{D}_2$ are just needed to be calculated one time on the execution segment.

However, in many cases, we cannot distinguish which is optimal among several PCPs. E.g., consider the above model against the formula

$$5 \leq \ell \leq 5 \Rightarrow \mathcal{D}_1 ; \neg(\neg(\mathcal{D}_2 ; \mathcal{D}_3) ; (\mathcal{D}_4 \wedge \mathcal{D}_5)).$$

Obviously, $t = 0$ and $t = 2$ are two PCPs w.r.t. the outmost chop, but we cannot tell which is optimal as we cannot guarantee that the values of the duration expression of $\mathcal{D}_4$ and $\mathcal{D}_5$ achieve *optimal* simultaneously. In this case, the two PCPs have to be checked according to the semantics separately. Thus, we duplicate d_2 at $t = 0$ and $t = 2$, and denote by d_2^0 and d_2^2 respectively. Obviously, d_2^0 is set to 0 at $t = 0$, while d_2^2 is set to 0 at $t = 2$. Moreover, according to the semantics, $\rho, [t_1, t_2] \models \neg(\neg(\mathcal{D}_2; \mathcal{D}_3); (\mathcal{D}_4 \wedge \mathcal{D}_5))$ iff $\rho, [t_1, t] \models \neg(\mathcal{D}_2; \mathcal{D}_3)$ implies $\rho, [t, t_2] \models \neg(\mathcal{D}_4 \wedge \mathcal{D}_5)$, for any $t \in [t_1, t_2]$. Checking the premise part can be done as above thanks to the existence of OPCPs. Accordingly, we also need to duplicate d_3 respectively corresponding to the two PCPs, and denote by d_3^0 and d_3^2 respectively. W.r.t. the PCP $t = 0$ of the outmost chop, it is easy to see that $t = 3$ and $t = 4$ are two PCPs of the second outmost chop as $\neg(\mathcal{D}_2; \mathcal{D}_3)$ is satisfied on both $[0, 3]$ and $[0, 4]$. Regarding the chop point $t = 4$ of the second outmost chop, $\mathcal{D}_4$ and $\mathcal{D}_5$ are both satisfied on $[4, 5]$. This indicates that firstly, $t = 0$ cannot be seen as a chop point of the outmost chop; secondly, the second outmost chop does not have "optimal" property, as we cannot guarantee the duration expressions of $\mathcal{D}_4$ and $\mathcal{D}_5$ achieve "optimal" simultaneously. An analogous analysis indicates the formula is satisfied if $t = 2$ is chosen as the chop point of the outmost chop, because only $t = 3$ is the PCP of the second outmost chop, and $\mathcal{D}_4$ is satisfied on $[3, 5]$, but $\mathcal{D}_5$ not.

4. ALGORITHMS

In this section, we focus on how to implement the procedures *Init* and *BMC-DC* for checking ELDIs.

Clearly, negation of an LDF or a logical combination of LDFs can be easily eliminated and we can obtain another equivalent LDF or logical combination of LDFs in which no negation occurs. Thus, for simplicity, as a convention, we just consider the ELDIs in which any $\neg$ is only applied to a subformula with ; as its outmost operator.

4.1 In a Nutshell

It is a natural way to check the satisfiability of an ELDI Φ against a timed automaton directly by the semantics, possibly with a complexity $O(n^b b^r)$, where n is the number of locations of the considered timed automaton, b is the upper bound of reference intervals, and r is the number of LDIs in Φ. Obviously, it is quite high. However, we found we can dramatically reduce the complexity by using the notion of *optimal potential chop point* (OPCP).

In order to define the notion of OPCP, we introduce some notations first. First of all, let's fix an ELDI $\Phi = a \leq \ell \leq b \Rightarrow \phi$ for the sequel discussions, $\mathcal{D}_1, \cdots, \mathcal{D}_r$ are all LDFs occurring ϕ, and all other formulas are subformulas of Φ unless otherwise stated. For each $\mathcal{D}_k$, we introduce a variable d_k to denote the value of the duration expression of $\mathcal{D}_k$ on the reference interval, and meanwhile, define an upper bound $d_{\max}$ of d_ks by $b \cdot \max\{ |c_{i_k,k}|, |c_k| \mid i_k \in \Omega_k \wedge k = 1, \ldots, r\} + 1$.

Definition 2. Given an ELDI formula $\phi'; (\mathcal{D}_1 \vee \cdots \vee \mathcal{D}_i)$ and an interval $[t_1, t_2]$, an integral time point t of $[t_1, t_2]$ is called *potential chop point* (PCP) if ϕ' is satisfied on $[t_1, t]$; a PCP t is called *optimal* up to t', if there is $1 \leq k \leq i$ such that

$$d_k([t, t']) = \min\{d_k([t^*, t']) \mid t^* \leq t' \text{ and } t^* \text{ is a PCP}\},$$

where $t_1 \leq t \leq t' \leq t_2$ and $t, t' \in \mathbb{N}$, and $d_k([b, e])$ denotes the value of d_k on $[b, e]$.

For the given $\phi'; (\mathcal{D}_1 \vee \cdots \vee \mathcal{D}_i)$ and $[t_1, t_2]$, we can show that at any time $t' \in [t_1, t_2]$, the formula is satisfied on $[t_1, t']$ iff there exists an OPCP t such that ϕ' is satisfied on $[t_1, t]$ and $\mathcal{D}_1 \vee \cdots \vee \mathcal{D}_i$ is satisfied on $[t, t']$ (see the proof for Theorem 1 in Appendix). Thus, we just need to visit all PCPs from t_1 to t_2, update the current OPCPs t, and keep the values of duration variables in ϕ' on $[t_1, t']$ and values of $d_1, \ldots, d_i$ on $[t, t']$, in order to check the satisfiability of the whole formula over $[t_1, t_2]$. We do not need to consider all possible values of variables on all subintervals of $[t_1, t_2]$. This indeed reduces the complexity of the checking so much. Similar idea is applicable to any ELDI formulas only with chop and disjunction of the form $\phi'; (\mathcal{D}_{1,1} \vee \cdots \vee \mathcal{D}_{1,i_1}); \cdots; (\mathcal{D}_{k,1} \vee \cdots \vee \mathcal{D}_{k,i_k})$ by simultaneously maintaining k interdependent OPCPs.

However, the idea is not applicable to the combinations of chop with conjunction or negation. For example, consider $\mathcal{D}_1; (\mathcal{D}_2 \wedge \mathcal{D}_3)$. It is impossible to guarantee the existence of OPCPs, because if it exists, the duration expressions of $\mathcal{D}_2$ and $\mathcal{D}_3$ both have to be optimal w.r.t. it. So, in order to check the satisfiability of $\mathcal{D}_1; (\mathcal{D}_2 \wedge \mathcal{D}_3)$, all the PCPs need to be maintained and checked for $\mathcal{D}_2 \wedge \mathcal{D}_3$ according to the semantics. Therefore, we duplicate the variable d_2 at each visited point (at most $b + 1$ times), denoted by $d_2^0, d_2^1, \cdots, d_2^b$, which respectively stand for the values of the duration expression of $\mathcal{D}_2$ on the interval from the corresponding time point to the end of the reference interval. Analogously for $\mathcal{D}_3$. In this case, we call the subformula $\mathcal{D}_2 \wedge \mathcal{D}_3$ *duplicated*. Similarly, for the negation of a subformula which is a right operand of chop, we also need to duplicate the corresponding variables for each LDF in the subformula at each visited time point. E.g., in $\mathcal{D}_1; \neg(\mathcal{D}_2; \mathcal{D}_3)$, $\neg(\mathcal{D}_2; \mathcal{D}_3)$ is a *duplicated subformula*. In order to improve the efficiency, we had better exploit the idea of OPCP as many as possible. So, we give a syntactical criterion to tell to which subformulas the OPCP-based approach is not applicable, called *duplicated formulas*.

Definition 3. Given a subformula φ of ϕ, we say φ is *duplicated*, if φ is of the form $\phi_1 \wedge \phi_2$ or $\neg\phi_1$, and $\psi; \varphi$ is a subformula of ϕ for some ψ. Duplicated subformulas can be nested and there is a largest nested depth for each ELDI formula. We use *D-Sub*(χ) to denote the set of duplicated subformulas of ϕ, of which χ is a proper subformula; furthermore denote the minimal one $\min($*D-Sub*$(\chi))$ by *MD*(χ).

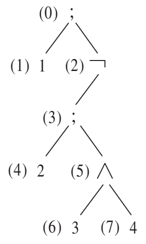
Example 1. In $\mathcal{D}_1; \neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4))$, $\neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4))$ is *duplicated*, and $\mathcal{D}_3 \wedge \mathcal{D}_4$ is *duplicated* too. So the largest nested depth of the formula is 2. *D-Sub*$(\mathcal{D}_3) = \{\mathcal{D}_3 \wedge \mathcal{D}_4, \neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4))\}$ and *MD*$(\mathcal{D}_3) = \mathcal{D}_3 \wedge \mathcal{D}_4$; *D-Sub*$(\mathcal{D}_3 \wedge \mathcal{D}_4) = \{\neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4))\}$ and *MD*$(\mathcal{D}_3 \wedge \mathcal{D}_4) = \neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4))$; while neither *D-Sub*$(\neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4)))$ nor *MD*$(\neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4)))$ exists.

According to Definition 3, we design a preprocessing procedure *PreTreat* in Algorithm 1 to analyze the syntactical structure of ϕ, including which subformulas have the "optimal property", which are duplicated, and the relation between them. To this end, ϕ will be represented as a binary syntactical tree, in which each node is organized as a tuple of the form $(\mathit{Nu}, \mathit{Op}, \mathit{Left}, \mathit{Right}, V, W, C, S, \mathit{tag})$, where *Nu* stands for the number of the subformula in the tree given by a mapping J, *Op* for its outmost operator, *Left* and *Right* for the numbers of the subformulas at the left and the right of *Op* respectively, V for the set of indices of the LDIs, W for the set of indices of the LDIs that are likely to be duplicated, C for the set of indices of the LDIs that have to be duplicated, S for the set of indices of the LDIs of the subformula whose duration expressions need to be initialized to 0 whenever the subformula is considered, and *tag* to indicate if the subformula itself is *duplicated*. Note that if a subformula is an LDF $\mathcal{D}_k$, then its *Op* will be set to its index k, and its *Left* and *Right* will be set to -1; if a subformula is of the form $\neg\psi$, *Left* will be set to the number of ψ, while *Right* will be set to -1. The numbering mapping J follows the convention that all of ϕ's subformulas are numbered by consecutive integers from 0, and different subformulas with different numbers. In addition, ϕ itself is numbered as 0 and the number of a formula is always less than those of its subformulas. We use an array A of the tuple to represent the syntax tree, and each subformula corresponds to an element of A whose index is the same as the subformula's number *Nu*.

Algorithm 1 *PreTreat*()

```
Input: $\phi, flag$
1: $n := J(\phi)$;
2: if $\phi = \mathcal{D}_k$ then
3:     $A[n].(Nu, Op, tag, Left, Right) := (n, k, \mathbf{false}, -1, -1)$;
4:     $A[n].(V, W, C, S) := (\{k\}, \emptyset, \emptyset, \{k\})$;
5: if $\phi = \phi_1 \vee \phi_2$ then
6:     $PreTreat(\phi_1, flag)$; $PreTreat(\phi_2, flag)$;
7:     $A[n].(Nu, Op, tag, Left, Right) := (n, \vee, \mathbf{false}, J(\phi_1), J(\phi_2))$;
8:     $A[n].(V, W, C, S) :=$
9:             $(A[J(\phi_1)].V \cup A[J(\phi_2)].V, A[J(\phi_1)].W \cup A[J(\phi_2)].W,$
10:            $A[J(\phi_1)].C \cup A[J(\phi_2)].C, A[J(\phi_1)].S \cup A[J(\phi_2)].S)$;
11: if $\phi = \phi_1 \wedge \phi_2$ then
12:     $PreTreat(\phi_1, \mathbf{false})$; $PreTreat(\phi_2, \mathbf{false})$;
13:     $A[n].(Nu, Op, tag, Left, Right) := (n, \wedge, flag, J(\phi_1), J(\phi_2))$;
14:     $A[n].(V, W, C, S) :=$
15:            $(A[J(\phi_1)].V \cup A[J(\phi_2)].V, A[J(\phi_1)].V \cup A[J(\phi_2)].V,$
16:            $A[J(\phi_1)].C \cup A[J(\phi_2)].C, A[J(\phi_1)].S \cup A[J(\phi_2)].S)$;
17: if $\phi = \neg\phi_1$ then
18:     $PreTreat(\phi_1, \mathbf{false})$;
19:     $A[n].(Nu, Op, tag, Left, Right) := (n, \neg, flag, J(\phi_1), -1)$;
20:     $A[n].(V, W, C, S) := A[J(\phi_1)].(V, V, C, S)$;
21: if $\phi = \phi_1; \phi_2$ then
22:     $PreTreat(\phi_1, flag)$; $PreTreat(\phi_2, \mathbf{true})$;
23:     $A[n].(Nu, Op, tag, Left, Right) := (n, ;, \mathbf{false}, J(\phi_1), J(\phi_2))$;
24:     $A[n].(V, W, C, S) := (A[J(\phi_1)].V \cup A[J(\phi_2)].V, A[J(\phi_1)].W,$
25:            $A[J(\phi_1)].C \cup A[J(\phi_2)].C \cup A[J(\phi_2)].W, A[J(\phi_1)].S)$
```

Example 2. Still consider the formula above $\mathcal{D}_1; \neg(\mathcal{D}_2; (\mathcal{D}_3 \wedge \mathcal{D}_4))$. Thus, four d_k $(k = 1, \dots, 4)$ are introduced. The syntactic tree just with the number and operator of each subformula is shown in Figure 2, and the array represented the syntactic tree is shown in the table. Using *PreTreat*, except that *tag* is computed from top to bottom, other information is computed from bottom to top. E.g., 3 and 4 are added to $A[3].C$ as they are contained in $A[5].W$, and $A[5].tag$ is true as node 5 is the right child of node 3 and its operator is $\wedge$.



Nu	V	W	C	S	tag
0	{1,2,3,4}	{ }	{2,3,4}	{1}	false
1	{1}	{ }	{ }	{1}	false
2	{2,3,4}	{2,3,4}	{3,4}	{2}	true
3	{2,3,4}	{ }	{3,4}	{2}	false
4	{2}	{ }	{ }	{2}	false
5	{3,4}	{3,4}	{ }	{3,4}	true
6	{3}	{ }	{ }	{3}	false
7	{4}	{ }	{ }	{4}	false

Figure 2: The syntactic tree and the information on the tuples

So, the procedure *Init* in the auxiliary automaton $\mathcal{S}$ of Figure 1 is implemented in Algorithm 2. It first analyzes the syntactical information of ϕ and records it in A by calling *PreTreat*. Then it initializes the corresponding introduced variables.

Then, we implement the action *BMC-DC* in the auxiliary automaton by calling two subroutines after updating the values of the d_is and their duplicates, where *Reset* for resetting the values of the corresponding duration expressions if the considered time point is a new OPCP, and *Sat* for calculating the return value indicating whether a formula is satisfied on the current reference interval. We will explain these two subroutines in detail later.

Algorithm 2 *Init*()

```
Input: $\phi$
1: $PreTreat(\phi, \mathbf{false})$;
2: for all $k \in A[0].V - A[0].C$ do
3:     $d_k := d_{\max}$;
4: for all $k \in A[0].C$ do
5:     for $i := 0$ to $b$ do
6:         $d_k^i := d_{\max}$;
7: for all $k \in A[0].S$ do
8:     $d_k := 0$;
```

Algorithm 3 *BMC-DC*()

```
1: for all $k \in A[0].V - A[0].C$ do
2:     for all $i \in \Omega_k$ do
3:         if $d_k \neq d_{\max}$ and $S_{i,k}$ is satisfied at the current location[1] and
           $gc > 0$ then
4:             $d_k := d_k + c_{i,k}$;
5: for all $k \in A[0].C$ do
6:     for $i := 0$ to $b$ do
7:         for all $i \in \Omega_k$ do
8:             if $d_k^i \neq d_{\max}$ and $S_{i,k}$ is satisfied at the current location and
               $gc > 0$ then
9:                 $d_k^i := d_k^i + c_{i,k}$;
10: $Reset(0, \mathbf{false})$;
11: if $gc \geq a \wedge \neg Sat(0, -1, -1)$ then
12:     return false;
13: return true;
```

4.2 The Subroutine *Reset*

The functions of *Reset* include the following two points:

1. Whenever a PCP is visited, we need to see whether to update the OPCPs, and accordingly reset the values of the duration expressions listed in the set S of the right operand of the chop;

2. For any two immediately nested chops in a nested duplicated subformula, maintain a correspondence relation between the chop points of the outer chop and the ones of the inner chop. That is recorded in a mapping relation T, which is empty at the beginning.

Reset takes as parameters the number n of a formula to be considered and a Boolean variable *flag* to indicate whether the formula is inside a duplicated subformula. For an LDF, *Reset* does nothing but return; for logical connectives, we recursively call the subroutine with its subformulas and the recalculated *flag* as parameters; for the chop, it is needed to reset the values of some duration expressions in the right operand of the chop, which is executed between the recursive invocations of the subroutines to its two operands. That can be categorized into the following two cases:

¬*flag* In this case, the d_ks corresponding to the LDFs of the left operand $A[n].Left$ do not need to be duplicated, therefore there is no need to keep a correspondence with the chop points of any outer chops. The current time point is possibly optimal if $A[n].Left$ is satisfied, and then the d_ks corresponding to the LDFs listed in the set S of the right operand $A[n].Right$ should be reset by case analysis:

- If the index of a d_k is in $A[A[n].Right].S$, but not in $A[A[n].Right].W$, then if its value is greater than 0, it should be reset to 0 as the current point is indeed

[1]This can be easily implemented by checking if each state variable in $S_{i,k}$ occurs in the current location.

optimal; otherwise unchanged, as the previous optimal point is better than the current one.

- If the index of a d_k is in both $A[n]$.*Right*'s S and W, a new duplicated subformula is encountered, so we just simply reset its duplicate at the current point to 0.

flag In this case, the OPCPs of the left operand $A[n]$. *Left* are related to the PCPs of the immediate outer chop. Thus, the current time point is possibly optimal w.r.t. some outer chop point i if the right operand of the outer chop has been calculated from i (indicated by $d_h^i \neq d_{\max}$) and $A[n]$.*Left* is satisfied w.r.t. i, so the respective duplicates of the d_ks corresponding to the LDFs listed in the set S of the right operand $A[n]$.*Right* should be reset by case analysis:

- If the index of a d_k is in $A[A[n].Right].S$, but not in $A[A[n].Right].W$, then if the duplicate d_k^i is greater than 0, it should be reset to 0 as the current point is indeed optimal w.r.t. i; otherwise unchanged.
- If the index of a d_k is in both $A[n]$.*Right*'s S and W, we just simply reset its duplicate at the current point to 0. Meanwhile, the correspondence between the chop point of the outer chop at i and the chop point of the inner chop at gc is recorded in T.

Algorithm 4 *Reset*()

```
Input: $n, flag$
1: if $A[n].Op \in \mathbb{N}$ then
2:     return ;
3: if $A[n].Op == \vee$ or $A[n].Op == \wedge$ then
4:     $Reset(A[n].Left, flag \vee A[n].tag)$;
       $Reset(A[n].Right, flag \vee A[n].tag)$;
5: if $A[n].Op == \neg$ then
6:     $Reset(A[n].Left, flag \vee A[n].tag)$;
7: if $A[n].Op ==;$ then
8:     $Reset(A[n].Left, flag)$;
9:     if $\neg flag$ then
10:        if $Sat(A[n].Left, -1, -1)$ then
11:            for all $h \in A[A[n].Right].S - A[A[n].Right].W$ do
12:                $d_h := \min(0, d_h)$;
13:            for all $h \in A[A[n].Right].S \cap A[A[n].Right].W$ do
14:                $d_h^{gc} := 0$;
15:    else
16:        choose $k \in A[n].S$;
17:        for all $i \in [0, gc]$ do
18:            if $d_k^i \neq d_{\max} \wedge Sat(A[n].Left, k, i)$ then
19:                for all $h \in A[A[n].Right].S - A[A[n].Right].W$ do
20:                    $d_h^i := \min(0, d_h^i)$;
21:                for all $h \in A[A[n].Right].S \cap A[A[n].Right].W$ do
22:                    $d_h^{gc} := 0$;
23:                    for all $k \in A[n].S$ do
24:                        $T(d_k^i) := T(d_k^i) \cup \{d_h^{gc}\}$;
25:    $Reset(A[n].Right, flag)$;
```

4.3 The Subroutine *Sat*

The subroutine *Sat* is to determine whether a considered formula is satisfied on the reference interval, which is used as the return condition of *BMC-DC* and also as resetting conditions in *Reset*. *Sat* takes three parameters: n is the number corresponding to the formula to be checked; if the minimal duplicated subformula strictly containing n exists, say m, i.e., $m = MD(n)$, then k is the index of some d_k in $A[m].S$ and i is the specific point to duplicate d_k; otherwise, k and i are both assigned with -1.

We compute the satisfiability in a recursive way. For an LDF $\mathcal{D}_k$, the return value is calculated according to the semantics depending on whether $MD(n)$ exists. If it does not exist, the value of d_k is used; otherwise, the value of d_k's i-th duplicate, i.e., d_k^i, is used. ";" and "$\vee$" can be handled in a standard way as they do not need to be duplicated. Regarding "$\neg$" and "$\wedge$", if the considered formula is not duplicated, then it is handled in a standard way; otherwise, the satisfiability of the formula should be discussed on all possible reference subintervals indicated by the duplicates of a d_h from its S according to the following two cases:

1. the first is when the formula itself is not strictly contained in another duplicated subformula, indicated by $i == -1$, i.e., *MD*(n) does not exist. Then, we just need to check if the formula is satisfied on some of the considered subintervals (indicated by $d_h^j \neq d_{\max}$);

2. the other is when the formula itself is strictly contained in another duplicated subformula, indicated by $i \neq -1$. Thus, we need to check if the formula is satisfied on some of the subintervals corresponding to the chop point of the outer chop (indicated by $d_h^i \in T(d_k^i)$).

Algorithm 5 Boolean *Sat*()

```
Input: $n, k, i$
1: if $A[n].Op \in \mathbb{N}$ then
2:     if $i == -1$ then
3:         return $d_{A[n].Op} \leq c_{A[n].Op}$;
4:     else
5:         return $d^i_{A[n].Op} \leq c_{A[n].Op}$;
6: if $A[n].Op == \vee$ then
7:     return $Sat(A[n].Left, k, i) \vee Sat(A[n].Right, k, i)$;
8: if $A[n].Op == \wedge$ and $A[n].tag ==$ false then
9:     return $Sat(A[n].Left, k, i) \wedge Sat(A[n].Right, k, i)$;
10: if $A[n].Op == \wedge$ and $A[n].tag ==$ true then
11:     choose a $h \in A[n].S$;
12:     for $j := 0$ to $gc$ do
13:         if $i == -1 \wedge d_h^j \neq d_{\max} \wedge Sat(A[n].Left, h, j) \wedge$
              $Sat(A[n].Right, h, j)$ then
14:             return true;
15:         if $i \neq -1 \wedge d_h^j \in T(d_k^i) \wedge Sat(A[n].Left, h, j) \wedge$
              $Sat(A[n].Right, h, j)$ then
16:             return true;
17:     return false;
18: if $A[n].Op == \neg$ and $A[n].tag ==$ false then
19:     return $\neg Sat(A[n].Left, k, i)$;
20: if $A[n].Op == \neg$ and $A[n].tag ==$ true then
21:     choose $h \in A[n].S$;
22:     for $j := 0$ to $gc$ do
23:         if $i == -1 \wedge d_h^j \neq d_{\max} \wedge \neg Sat(A[n].Left, h, j)$ then
24:             return true;
25:         if $i \neq -1 \wedge d_h^j \in T(d_k^i) \wedge \neg Sat(A[n].Left, h, j)$ then
26:             return true;
27:     return false;
28: if $A[n].Op ==;$ then
29:     return $Sat(A[n].Right, k, i)$;
```

4.4 Correctness and Complexity

The correctness of our approach is guaranteed by the following theorem. The proof is given in the appendix.

THEOREM 1. *Our approach is correct. That is,*

Termination: *Our approach is certain to terminate.*

Soundness: *If none of the failure states is reachable in $\mathcal{A}||\mathcal{S}$, then the given ELDI formula Φ is satisfied by $\mathcal{A}$.*

Completeness: *If some of the failure states are reachable in $\mathcal{A}\|\mathcal{S}$, then the given ELDI formula Φ is not satisfied by $\mathcal{A}$.*

In order to implement our algorithms on UPPAAL, given a timed automaton $\mathcal{A}$, we reformulate it by replacing each location with a label and using a labeling function f to map each label to the set of state variables corresponding to the location, denoted by $\mathcal{G}$.

As we implement the checking of the reachability of the failure states in the composed automaton $\mathcal{G}\|\mathcal{S}$ using UPPAAL, the complexity of our approach depends on the implementation of UPPAAL, in which on-the-fly checking on the whole system is applied. Here the state space is the product of the locations of the automaton and the values of introduced variables. Regarding the number of variables, first of all, we need to introduce a duration variable d_k for each LDF $\mathcal{D}_k$ ($k = 1, \ldots, r$); moreover, each d_k could be duplicated at most $b + 1$ times; finally, the mapping T is implemented as a $(b + 1)r \times (b + 1)r$ matrix, in which each entry is a Boolean variable that indicates whether there is a correspondence between the two corresponding introduced variables. So, the total number of introduced variables is at most $(b + 1)r \cdot (br + r + 1)$. In addition, in our approach we have to take the cost for executing the action *BMC-DC* on the corresponding transitions into account.

The checking procedure consists of three phases. The first phase is when $\mathcal{S}$ stays in `p0`, in which all the introduced variables as well as gc keep unchanged. So, this phase contains at most $|S_\mathcal{A}|$ state changes and $|S_\mathcal{A}|^2$ transitions.

The second phase is to do the actual checking in which gc keeps increased from 0 to b. Let the transition from `p0` to `p1` happen at a specific state s. Each execution of *BMC-DC* consists of the following three steps: *updating*, *resetting* and *checking*. Obviously, the cost for updating is $O(br)$. Resetting may change the values of the introduced variables and the entries of the mapping matrix, and recursively call *Reset* and *Sat* many times. Each d_k possibly with a superscript i is reset at most once, so the time cost of reset operations is $O(br)$. Meanwhile, applying *Sat* to a duplicated subformula n could result in recursive invocations of *Sat* of at most $(2(b + 1))^h$ times, and each execution of *BMC-DC* checks each node (subformula) of the ELDI ϕ at most once, so the cost of the satisfiability checking in each execution of *BMC-DC* is $O(|\phi|b^h)$, where h is the largest nested depth of duplicated subformulas of ϕ.

Let x be the maximal number of outgoing transitions from the locations of $\mathcal{A}$. In each execution of *BMC-DC*, gc increases by 1, and there are at most $x + 1$ possibilities for the next location, which results in at most $2(x + 1)^{(b+1)}$ possible states. As shown above, each transition costs time $O(br + |\phi|b^h)$, so the cost for checking intervals starting from s is $O((br + |\phi|b^h)x^b)$. Thus, the cost of the second phase is totally $O(|S_\mathcal{A}|(br + |\phi|b^h)x^b)$.

The third phase is when gc is equal to b and the self transition at `p1` is executed. In this phase, the length of the current execution segment has exceeded b, and *BMC-DC* will not be invoked any more, therefore we can conclude that none of the failure states will be reached. Thus, the cost of this phase is zero.

In summary, the number of transitions in the system handled by UPPAAL is $O(|S_\mathcal{A}|^2 + |S_\mathcal{A}|x^b)$ and the time complexity of our approach is $O(|S_\mathcal{A}|^2 + |S_\mathcal{A}|(br + |\phi|b^h)x^b)$. We can see that the largest nested depth of duplicated subformulas directly affects the complexity of the algorithm. As it is generally very small, compared with the 4-fold exponential approximation algorithm in [7], we have an essential improvement subject to the constraint of the finite observation time.

5. IMPLEMENTATION AND EXPERIMENTS

Using C++, we develop a tool that can be integrated with UPPAAL to check whether a given system $\mathcal{A}$ satisfies an ELDI Φ (see the dotted box in Figure 3). The input of the tool is Φ and an *XML* file representing $\mathcal{A}$. Then it automatically generates $\mathcal{G}$ in terms of these information. While the generation of $\mathcal{S}$, whose crucial part is the procedure *BMC-DC*, is only dependent on Φ. At last, the composed automaton $\mathcal{G}\|\mathcal{S}$, as well as the CTL formula defined in Section 3 is the input to the model checker UPPAAL.

We now use the following four examples to show the efficiency of our approach, the first three of which are taken from [10] and the fourth is the complex one given in Section 3.1. All the experiments are conducted on a laptop with the Intel Core2 Duo T6400 processor and 2 GB RAM using the operating system Windows 7. Due to the space limitation, we here skip the detailed description of models and model transformation, while only compare the results with those in paper [10].

1. We first check the automaton $\mathcal{A}_N$ obtained by N iterations of the automaton depicted in Figure 4(a) with respect to the property also shown in the figure. The automaton $\mathcal{A}_N$ is constructed by combining N automata $M_1, M_2, \ldots, M_N$, so that there are edges from D_i to A_{i+1}, for $1 \leq i < N$, and edges from D_i to A_j , for $1 \leq j \leq i \leq N$. With regard to different values of N, the checking times (t_1) using our approach are given in the second column of Figure 4(b). Clearly, for each N, the time is less than those listed in the third and the fourth columns (t_2 and t_3), which are needed by two different approaches in [10].

2. The second problem is taken from the experiment 2 of [10][2], where k_1, k_2 and k_3 are coefficients that appear in an ELDI to be checked. We need to solve the problem that for given values of k_1 and k_2, the smallest value of k_3 should be found, so that the ELDI is satisfied. The results of the experiment demonstrate that for each case of different values of k_1 and k_2 given in [10], it takes less time to find the smallest value of k_3 using our approach. For instance, in the first case, it takes $6.3s$, while the two methods proposed in [10] respectively need $38.9s$ and $9.0s$.

3. As discussed in [10], the approximation algorithm fails to check the automaton with respect to the property shown in Figure 4(c) even when the observation interval is finite. While our tool is able to give definite results. For instance, it takes $0.2s$ to verify the satisfiability of the property for $\ell = 4$.

4. The tool can handle complex ELDI formulas as well. For example, for the simple and the complex ones in Section 3.1, it respectively takes $0.16s$ and $0.22s$ to verify the satisfiability of the formulas for the given execution segment.

6. CONCLUSION

In this paper, inspired by Fränzle and Hansen's work [6, 7, 10], we investigated bounded model-checking of ELDIs, which is a very expressive subset of DC. Compared with their work, the advantages of our approach include:

1. instead of using approximation semantics of DC, the standard discrete time semantics of DC is adopted;

2. our approach is much more efficient by case studies. As analyzed in Section 4.4, the complexity of our approach is much

[2]This example actually comes from the full version of [10] thanks to Prof. Michael Hansen for his courtesy of this example.

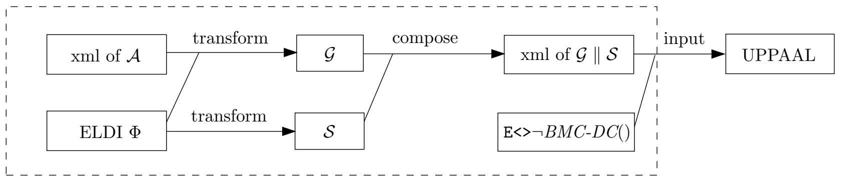
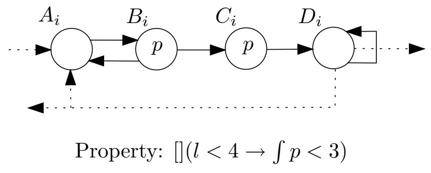


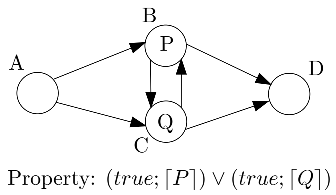
Figure 3: ELDI checking tool



(a)

N	$t_1(s)$	$t_2(s)$	$t_3(s)$
3	0.59	5.7	5.7
4	0.64	16.5	15.8
5	0.75	27.9	26.2
6	0.84	55.4	−−

(b)



B
P
A
D
Q
C
Property: $(true; \lceil P \rceil) \vee (true; \lceil Q \rceil)$

(c)

Figure 4: (a) Models in experiment 1 (b) Execution time of experiment 1 (c) Models in experiment 3

lower than theirs subject to the constraint of the finite observation time.

We implemented the approach using UPPAAL, and showed its efficiency by some examples.

The disadvantage of our approach is that all reference intervals are constrained to be bounded. So, our main future work is to consider how to weaken this constraint.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *SFM'04*, pages 200–236, 2004.

[3] V. A. Braberman and D. V. Hung. On checking timed automata for linear duration invariants. In *RTSS'98*, pages 264–273. IEEE Computer Society, 1998.

[4] M. Fränzle. Model-checking dense-time duration calculus. *Formal Aspects of Computing*, 16(2):121–139, 2004.

[5] M. Fränzle and M. R. Hansen. Deciding an interval logic with accumulated durations. In *TACAS'07*, pages 201–215, 2007.

[6] M. Fränzle and M. R. Hansen. Efficient model checking for duration calculus based on branching-time approximations. In *SEFM'08*, pages 63–72, 2008.

[7] M. Fränzle and M. R. Hansen. Efficient model checking for duration calculus. *International Journal of Software and Informatics*, 3(2-3):171–196, 2009.

[8] V. Goranko, A. Montanari, and G. Sciavicco. A road map of interval temporal logics and duration calculi. *Journal of Applied Non-Classical Logics*, 14(1-2):9–54, 2004.

[9] M. R. Hansen. Model-checking discrete duration calculus. *Formal Aspects of Computing*, 6(6):826–845, 1994.

[10] M. R. Hansen and A. W. Brekling. On tool support for duration calculus on the basis of presburger arithmetic. In *TIME'11*, pages 115–122, 2011.

[11] X. Li and D. V. Hung. Checking linear duration invariants by linear programming. In *ASIAN'96*, pages 321–332, 1996.

[12] R. Meyer, J. Faber, J. Hoenicke, and A. Rybalchenko. Model checking duration calculus: a practical approach. *Formal Aspects of Computing*, 20(4-5):481–505, 2008.

[13] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 18(2):10–19, February 1985.

[14] P. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. In *RT-TOOLS'01*, 2001.

[15] B. Sharma, P. K. Pandya, and S. Chakraborty. Bounded validity checking of interval duration logic. In *TACAS'05*, pages 301–316, 2005.

[16] E. V. Sorensen, A. P. Ravn, and H. Rischel. Control program for a gas burnew: Part 1: Informal requirements, ProCoS case study 1. Technical report, Department of Computer Science, Technical University of Denmark, 1990.

[17] P. H. Thai and D. V. Hung. Verifying linear duration constraints of timed automata. In *ICTAC'04*, pages 295–309, 2004.

[18] N. Zhan and M. E. Majster-Cederbaum. On hierarchically developing reactive systems. *Inf. Comput.*, 208(9):997–1019, 2010.

[19] M. Zhang, D. V. Hung, and Z. Liu. Verification of linear duration invariants by model checking CTL properties. In *ICTAC'08*, pages 395–409, 2008.

[20] M. Zhang, Z. Liu, and N. Zhan. Model checking linear duration invariants of networks of automata. In *FSEN'09*, pages 244–259, 2009.

[21] C. Zhou. Linear duration invariants. In *FTRTFT'94*, pages 86–109, 1994.

[22] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer, 2004.

[23] C. Zhou, M. R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In *STACS'93*, pages 58–68, 1993.

[24] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

Appendix: Proof of Theorem 1

In order to prove Theorem 1, we need the following three lemmas. Lemma 1 certifies the existence of OPCPs for ELDIs of the form $\mathcal{D}_1;\mathcal{D}_2$, which will be a basic case in the proof of the correctness of *BMC-DC*. Based on that, Lemma 2 shows the correctness of the subroutine *Sat*, and Lemma 3 shows the correctness of the procedure *BMC-DC* in general cases.

LEMMA 1. *Let $\rho = (l'_0, t'_0)(l_1, t_1)\cdots(l_{gc}, t_{gc})$ be the current execution segment in the automaton $\mathcal{A}$ and Φ be $(a \le \ell \le b \Rightarrow \mathcal{D}_1; \mathcal{D}_2)$. Then, the model checking algorithm BMC-DC is correct w.r.t. ρ and Φ, i.e.,*

1. *The algorithm is certain to terminate.*
2. *If the algorithm returns **true**, then Φ is satisfied by ρ.*
3. *If the algorithm returns **false**, then Φ is not satisfied by ρ.*

PROOF. When checking $\mathcal{D}_1;\mathcal{D}_2$ subject to $a \le \ell \le b$, the effect of *Reset*$(0, \textbf{false})$ is same as

```
if $d_1 \le c_1$ then
    $d_2 := \min(0, d_2)$;
```

while the return condition by calling *Sat*$(0, -1, -1)$ is $d_2 \le c_2$. So, 1 is obvious.

Regarding to 2, if the algorithm returns **true**, then $gc < a$ or $(gc \ge a \wedge d_2 \le c_2)$ holds. Φ holds trivially for the former case. For the latter case, since $d_2 \le c_2 \ne d_{\max}$, it must have been reset to 0 at some point. Assume the latest reset statement happens at point $t = t_i$ ($0 \le i \le gc$). As the condition of resetting is $d_1 \le c_1$, $\rho, [t_0, t_i] \models \mathcal{D}_1$ holds. Also, $t = t_i$ is the latest resetting point, so d_2 is exactly the value of the duration expression of $\mathcal{D}_2$ on the interval $[t_i, t_{gc}]$, hence $\rho, [t_i, t_{gc}] \models \mathcal{D}_2$. Therefore, Φ is satisfied by ρ.

As for 3, if the algorithm returns **false**, then both $gc \ge a$ and $d_2 > c_2$ hold. By contraposition, suppose $\rho, [t_0, t_{gc}] \models \mathcal{D}_1;\mathcal{D}_2$, i.e., there exists t_j such that $\rho, [t_0, t_j] \models \mathcal{D}_1$ and $\rho, [t_j, t_{gc}] \models \mathcal{D}_2$. So d_2 must have been reset to 0 at some point. Also, assume the latest reset happens at $t = t_i$ ($0 \le i \le gc$). Now, we make a case analysis on the relation between t_i and t_j. Firstly, if $t_i = t_j$, then $d_2 \le c_2$ simply holds. Secondly, if $t_i < t_j$, as $t = t_i$ is the latest resetting point and no reset operation happens at point $t = t_j$, the duration expression of $\mathcal{D}_2$ on the subinterval $[t_i, t_j]$ is less than 0. Besides, the duration expression of $\mathcal{D}_2$ on the subinterval $[t_j, t_{gc}]$ is less than or equal to c_2 as $\rho, [t_j, t_{gc}] \models \mathcal{D}_2$ holds. So, d_2 is equal to the summation of the above two parts and thus also less than or equal to c_2. Thirdly, if $t_i > t_j$, the duration expression of $\mathcal{D}_2$ is larger than 0 on $[t_j, t_i]$ as $t = t_i$ is a resetting point. So the duration expression of $\mathcal{D}_2$ is less than or equal to c_2 on $[j, gc]$ implies that the proposition also holds on $[t_i, t_{gc}]$. In summary, in any case the duration expression of $\mathcal{D}_2$ is always less than or equal to c_2 on $[t_i, t_{gc}]$, which contradicts to $d_2 > c_2$ at $t = t_{gc}$. Hence, ϕ is not satisfied by ρ. $\square$

In the following, we give some definitions that will be used in the proof.

Definition 4. We say a subformula ϕ is a *left formula*, if $A[J(\phi)].S \cap A[J(MD(\phi))].S \ne \emptyset$. That is, in *MD*$(\phi)$, ϕ does not occur as the right operand of any chop operator. Hereafter, conceptually, if *D-Sub*(ϕ) is $\emptyset$, we set *MD*(ϕ) to be the whole formula.

Regarding left formulas, we have

LEMMA 2. *Let $\rho = (l'_0, t'_0)(l_1, t_1)\cdots(l_{gc}, t_{gc})$ be the current execution segment, ϕ be a left formula with $k \in A[J(MD(\phi))].S$ and $J(\phi) = n$, and $i \ge 0$. Then,*

*(1) Sat$(\phi, -1, -1)$ returns **true** iff $\rho, [t_0, t_{gc}] \models \phi$; and*

*(2) Sat(ϕ, k, i) returns **true** iff $\rho, [t_i, t_{gc}] \models \phi$.*

In order to prove the lemma by induction on formulas, as in [18], we need to define a well-founded order on the formulas of ELDIs, denoted by $<$. To this end, we first define a partial order, denoted by $\prec$ over ELDIs as: $(\phi_1;\phi_2) \prec (\psi_1;\psi_2)$ iff $\phi_1;\phi_2 \Leftrightarrow \psi_1;\psi_2$ and ψ_1 is a proper subformula of ϕ_1. In other words, we assume the left association of ; has a higher precedence. Then, we say $\phi < \psi$ iff either ϕ is a proper subformula of ψ, or $\phi \prec \psi$. It can be proved that $<$ is well-founded, referring to [18] for the detail.

PROOF. By induction on the structure of ϕ w.r.t "$<$".

Base case $\phi = \mathcal{D}_h$. Then for (1)

Sat$(\phi, -1, -1)$ returns **true**
iff $d_h \le c_h$ (line 2 of *Sat*)
iff $\rho, [t_0, t_{gc}] \models \mathcal{D}_h$ ($h \in A[n].S$ and d_h is initialized to 0)

For (2)

Sat(ϕ, h, i) returns **true**
iff $d^i_h \le c_h$ (line 4 of *Sat*)
iff $\rho, [i, gc] \models \mathcal{D}_h$ ($h \in A[n].S$ so is reset to 0 at t_i)

Induction Hypothesis (IH): For any ψ, if $\psi < \phi$, then (1) and (2) hold for ψ.

Inductive steps: We just prove (1) if (2) can be proved similarly for the considered case.

- $\phi = \phi_1 \vee \phi_2$
  Sat$(n, -1, -1)$ returns **true**
  iff *Sat*$(\phi_1, -1, -1)$ returns **true** or *Sat*$(\phi_2, -1, -1)$ returns **true** (line 6)
  iff $\rho, [t_0, t_{gc}] \models \phi_1$ or $\rho, [t_0, t_{gc}] \models \phi_2$ (IH, as ϕ_1 and ϕ_2 are both left formulas)
- $\phi = \phi_1 \wedge \phi_2$. Because ϕ is a left formula, $A[n]$.*tag* is false. So,
  Sat$(n, -1, -1)$ returns **true**
  iff *Sat*$(\phi_1, -1, -1)$ returns **true** and *Sat*$(\phi_2, -1, -1)$ returns **true** (line 8)
  iff $\rho, [t_0, t_{gc}] \models \phi_1$ and $\rho, [t_0, t_{gc}] \models \phi_2$ (IH, as ϕ_1 and ϕ_2 are both left formulas)
- $\phi = \neg\phi_1$. Because ϕ is a left formula, $A[n]$.*tag* is false. Thus,
  Sat$(n, -1, -1)$ returns **true**
  iff *Sat*$(\phi_1, -1, -1)$ returns **false** (line 18)
  iff $\rho, [t_0, t_{gc}] \models \neg\phi_1$ (IH, as ϕ_1 is a left formula)
- $\phi = \phi_1;\phi_2$. By induction on the structure of ϕ_2.
  - $\phi = \phi_1;\mathcal{D}_h$
    Sat$(n, -1, -1)$ returns **true**
    iff *Sat*$(\phi_2, -1, -1)$ returns **true** (line 28)
    iff $d_h \le c_h$ (line 2)
    iff $\exists t \bullet (\rho, [t_0, t] \models \phi_1$ and $\rho, [t, t_{gc}] \models \mathcal{D}_h)$ (d_h has been reset to 0 at t by line 11 of *Reset*, and the property of OPCPs (Lemma 1))
    iff $\exists t \bullet (\rho, [t_0, t] \models \phi_1 \wedge \rho, [t, t_{gc}] \models \mathcal{D}_h)$ (IH, as ϕ_1 is a left formula)
    iff $\rho, [t_0, t_{gc}] \models \phi_1;\phi_2$
  - $\phi = \phi_1;(\phi_2 \vee \phi_3)$
    Sat$(\phi, -1, -1)$ returns **true**
    iff *Sat*$(\phi_1;\phi_2, -1, -1)$ returns **true** or *Sat*$(\phi_1;\phi_3, -1, -1)$ returns **true** (Distributivity of chop over disjunction)
    iff $\rho, [t_0, t_{gc}] \models \phi_1;\phi_2$ or $\rho, [t_0, t_{gc}] \models \phi_1;\phi_3$ (IH)
    iff $\rho, [t_0, t_{gc}] \models \phi_1;(\phi_2 \vee \phi_3)$

- $\phi = \phi_1; (\phi_2 \wedge \phi_3)$. Here $A[A[n].\mathit{Right}].\mathit{tag}$ is true as the conjunction occurs as the right operand of the outmost chop operator. For (1)

$\mathit{Sat}(\phi, -1, -1)$ returns **true**
iff $\mathit{Sat}(\phi_2 \wedge \phi_3, -1, -1)$ returns **true** (line 28)
iff $\exists j \in [0, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \neq d_{\max} \wedge \mathit{Sat}(\phi_2, h, j) \wedge \mathit{Sat}(\phi_3, h, j))$ (line 13)
iff $\exists j \in [0, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \neq d_{\max}, \rho, [t_j, t_{gc}] \models \phi_2$ and $\rho, [t_j, t_{gc}] \models \phi_3)$
(IH, as ϕ_2 and ϕ_3 are both left formulas)
iff $\exists j \in [0, gc] \bullet (\rho, [t_0, t_j] \models \phi_1$,
$\rho, [t_j, t_{gc}] \models \phi_2$ and $\rho, [t_j, t_{gc}] \models \phi_3)$
(d_h^j is reset to 0 at t_j, and line 13 of *Reset*)
iff $\exists j \in [0, gc] \bullet (\rho, [t_0, t_j] \models \phi_1$,
$\rho, [t_j, t_{gc}] \models \phi_2$ and $\rho, [t_j, t_{gc}] \models \phi_3)$
(IH, as ϕ_1 is a left formula)
iff $\rho, [t_0, t_{gc}] \models \phi_1; (\phi_2 \wedge \phi_3)$
For (2)
$\mathit{Sat}(\phi, k, i)$ returns **true**
iff $\mathit{Sat}(\phi_2 \wedge \phi_3, k, i)$ returns **true** (line 28)
iff $\exists j \in [i, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \in T(d_k^i) \wedge \mathit{Sat}(\phi_2, h, j) \wedge \mathit{Sat}(\phi_3, h, j))$ (line 15)
iff $\exists j \in [i, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \in T(d_k^i)$ and $\rho, [j, gc] \models \phi_2$ and $\rho, [j, gc] \models \phi_3)$
(IH, as ϕ_2 and ϕ_3 are both left formulas)
iff $\exists j \in [i, gc] \bullet (\rho, [t_0, t_j] \models \phi_1$
$\rho, [t_j, t_{gc}] \models \phi_2$ and $\rho, [t_j, t_{gc}] \models \phi_3)$
(d_h^j is reset to 0 at t_j, and line 21 of *Reset*)
iff $\exists j \in [i, gc] \bullet (\rho, [t_i, t_j] \models \phi_1$,
$\rho, [t_j, t_{gc}] \models \phi_2$ and $\rho, [t_j, t_{gc}] \models \phi_3)$
(IH, as ϕ_1 is a left formula)
iff $\rho, [t_i, t_{gc}] \models \phi_1; (\phi_2 \wedge \phi_3)$

- $\phi = \phi_1; (\neg\phi_2)$. Here $A[A[n].\mathit{Right}].\mathit{tag}$ is also true as the negation occurs as the right operand of the outmost chop operator. For (1)

$\mathit{Sat}(\phi, -1, -1)$ returns **true**
iff $\mathit{Sat}(\neg\phi_2, -1, -1)$ returns **true** (line 28)
iff $\exists j \in [0, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \neq d_{\max} \wedge \neg\mathit{Sat}(\phi_2, h, j))$ (line 23)
iff $\exists j \in [0, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \neq d_{\max}$ and $\rho, [t_j, t_{gc}] \models \neg\phi_2)$
(IH, as ϕ_2 is a left formula)
iff $\exists j \in [0, gc] \bullet (\rho, [t_0, t_j] \models \phi_1$ and
$\rho, [t_j, t_{gc}] \models \neg\phi_2)$
(d_h^j has to be reset to 0 at t_j by line 13 of *Reset*)
iff $\exists j \in [0, gc] \bullet (\rho, [t_0, t_j] \models \phi_1$ and $\rho, [t_j, t_{gc}] \models \neg\phi_2)$
(IH, as ϕ_1 is a left formula)
iff $\rho, [t_0, t_{gc}] \models \phi_1; \neg\phi_2$
For (2)
$\mathit{Sat}(\phi, k, i)$ returns **true**
iff $\mathit{Sat}(\neg\phi_2, k, i)$ returns **true** (line 28)
iff $\exists j \in [i, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \in T(d_k^i) \wedge \neg\mathit{Sat}(\phi_2, h, j))$ (line 25)
iff $\exists j \in [i, gc], h \in A[A[n].\mathit{Right}].S \bullet$
$(d_h^j \in T(d_k^i)$ and $\rho, [j, gc] \models \neg\phi_2)$
(IH, as ϕ_2 is a left formula)
iff $\exists j \in [i, gc] \bullet (\rho, [t_0, t_j] \models \phi_1$ and $\rho, [t_j, t_{gc}] \models \neg\phi_2)$
(d_h^j has to be reset to 0 at t_j by line 21 of *Reset*)
iff $\exists j \in [i, gc] \bullet (\rho, [t_i, t_j] \models \phi_1$ and $\rho, [t_j, t_{gc}] \models \neg\phi_2)$
(IH, as ϕ_1 is a left formula)
iff $\rho, [t_i, t_{gc}] \models \phi_1; \neg\phi_2$

- $\phi = \phi_1; (\phi_2; \phi_3)$.

$\mathit{Sat}(\phi, -1, -1)$ returns **true**
iff $\mathit{Sat}((\phi_1; \phi_2); \phi_3, -1, -1)$ returns **true**
iff $\rho, [t_0, t_{gc}] \models (\phi_1; \phi_2); \phi_3$
(IH as $(\phi_1; \phi_2); \phi_3 \prec \phi_1; (\phi_2; \phi_3)$)
iff $\rho, [t_0, t_{gc}] \models \phi_1; (\phi_2; \phi_3)$ □

From this lemma, we can conclude the correctness of *BMC-DC* w.r.t. a given execution.

LEMMA 3. *Let $\rho = (l'_0, t'_0)(l_1, t_1) \cdots (l_{gc}, t_{gc})$ be the current execution segment in the automaton $\mathcal{A}$ and Φ be $(a \leq l \leq b \Rightarrow \phi)$. The model checking algorithm BMC-DC is correct, that is*

1. *Termination: The algorithm is certain to terminate.*
2. *Soundness: If the algorithm returns **true**, the ELDI formula Φ is satisfied by ρ.*
3. *Completeness: If the algorithm returns **false**, the ELDI formula Φ is not satisfied by ρ.*

PROOF. From the complexity analysis, we can see that in each execution of *BMC-DC*, the reset operation by *Reset* is executed at most $O(br)$ times, and the basic satisfiability checking by *Sat* is executed at most $O(|\phi|b^h)$ times, so each *BMC-DC* is certain to terminate.

The soundness and the completeness can be directly obtained from Lemma 2 as follows.

The algorithm returns **false**
iff $t_{gc} \geq a \wedge \neg\mathit{Sat}(n, -1, -1)$ (line 11 of *BMC-DC*)
iff $t_{gc} \geq a$ and $\rho, [t_0, t_{gc}] \models \neg\phi$ (Lemma 2(1))
iff $\rho, [t_0, t_{gc}] \not\models (a \leq \ell \leq b \Rightarrow \phi)$ ($t_{gc} \leq b$ always holds)
iff $\rho, [t_0, t_{gc}] \not\models \Phi$ □

Now we can prove Theorem 1, which guarantees the correctness of our approach.

PROOF FOR THEOREM 1.

Termination From the complexity analysis, we can see that the on-the-fly procedure in UPPAAL has at most $O(|S_{\mathcal{A}}|^2 + |S_{\mathcal{A}}|\, x^b)$ transitions to handle. Moreover, no action is needed for any transition in the first phase, and each transition in the second phase is contained in an execution of *BMC-DC*, which is certain to terminate (Lemma 3(1)). Thus, our approach is also certain to terminate.

Soundness If none of the failure states is reachable, *BMC-DC* always returns **true**. The first phase of the on-the-fly checking can stay on any location of the original automaton, and the second phase checks each execution segment from that location with the length bounded by b, so all the ELDI models of $\mathcal{A}$ have been considered and they all satisfy the formula Φ according to Lemma 3(2). Therefore, Φ is satisfied by the original automaton, which guarantees the soundness of our approach.

Completeness If some failure states are reachable, *BMC-DC* returns **false** at some time, so according to Lemma 3(3), the current execution segment does not satisfy Φ and a counterexample is found. As the current execution segment is an ELDI model of the original automaton, Φ is not satisfied by the original automaton. That guarantees the completeness of our approach. □