Computing and Informatics, Vol. 31, 2012, 507–530

VERIFICATION OF SYSTEMS WITH DEGRADATION

Jiří Barnat, Ivana Černá, Jana Tůmová

Masaryk University, Faculty of Informatics
Botanická 68a, 602 00, Brno, Czech Republic
e-mail: {barnat, cerna, xtumova}@fi.muni.cz

Abstract. We focus on systems that naturally incorporate a degrading quality, such as electronic devices with degrading electric charge or broadcasting networks with decreasing power or quality of a transmitted signal. For such systems, we introduce an extension of linear temporal logic (Linear Temporal Logic with Degradation Constraints, or DLTL for short) that provides a user-friendly formalism for specifying properties involving quantitative requirements on the level of degradation. We investigate the possibility of translating DLTL verification problem for systems with degradation into previously solved MITL verification problem for timed automata, and we show that through the translation, DLTL model checking problem can be solved with limited, yet arbitrary, precision. For a specific subclass of DLTL formulas, we present a full precision verification technique based on translation of DLTL formulas into a specification formalism called Büchi Automata with Degradation Constraints (BADCs) developed earlier.

Keywords: Systems with degradation, linear temporal logic, quantitative model checking, automata-based approach to verification, timed automata

1 INTRODUCTION

Model checking [6] has been recognized as one of the successful formal verification techniques that, if employed during the software development cycle, may bring significant reduction in total development cost or time-to-market [15, 14]. Recently, we have shown how the automata-based verification procedure, as used for model checking of non-deterministic systems, may be extended to systems with *degradation* [7]. Degradation is a natural phenomenon present in many systems we encounter regularly in our everyday lives. For example, data stored in memory are subject to bit

rot, value of money degrades with time due to inflation, signal strength degrades with the distance from the transmitter, capacity of a recharging battery pack degrades with every charging cycle, and many others. There is no doubt that a number of software systems produced must take the degradation phenomenon into account. Verification of worst-case degradation scenarios that a software system under development must survive or designing a strategy to avoid degradation below a given threshold are examples of problems that can be addressed with the model checking approach.

In our previous work, we have introduced two formalisms to capture the verification problem for systems with degradation [7]. Those were Transition Systems with Degradation (TSD) used to describe the degradation aspects in the behavior of the system under development, and Büchi Automata with Degradation Constraints (BADC) used to capture their properties. Given a TSD and a BADC specification of an undesired system behavior we have shown how to decide whether the system exhibits the behavior or not.

A drawback of the designed verification framework is the necessity to express the undesired behavior of the system with use of an automaton. According to our experience, constructing a BADC from a natural language description of a property is far more complicated than in standard non-degradation case, let alone the necessity of negation of the degradation specification. We address this issue by introducing an easy-to-use specification formalism, called Linear Temporal Logic with Degradation Constraints (DLTL) that is capable of expressing quantitative properties of systems with degradation.

This paper aims at verification of systems with degradation against DLTL formulas and it is based on preliminary results presented in [8]. We show that DLTL verification problem can be translated into a verification problem for real-time systems. In particular, we present how a system with degradation can be interpreted as a Timed Automaton (TA) and a DLTL formula as a formula of Metric Interval Temporal Logic (MITL) [2]. Using this approach, the verification problem for systems with degradation and DLTL fomulas can be solved up to chosen precision. Furthermore, we show that a formula from a specific DLTL subclass, which we call *half-bounded DLTL*, can be translated into a BADC and thus we obtain a method for verification of the DLTL subclass with full precision.

The contribution of our paper can be seen in the context of the work aimed at specification and verification of quantitative temporal logic properties. The current research is focused mainly on probabilistic and timed aspects inherently present in systems under investigation. Namely, in the area of probabilistic systems, probabilistic versions of CTL [13], LTL [17], and CTL* [5] have been developed and widely used both for specification and verification of systems behavior. In fact, probability can be viewed as a degrading phenomenon and DLTL as a probabilistic logic. We have investigated the relation between DLTL and PCTL* in [8] and we have proven that their expressivness is incomparable. For timed systems, a number of specification formalisms have been developed as well. Let us mention at least the temporal logic for the specification of real-time systems [3, 4]. In [10, 11] the authors consider

model checking of quantitative LTL (qLTL) properties. Their approach is different than ours as they interpret qLTL over structures whose atomic propositions have values in $[0,1]$ rather than in $\{0,1\}$. In [9] the authors introduce Discounted CTL (DCTL). DCTL formulas are not interpreted as true or false, their value is a real number in interval $[0,1]$. Works of Koymans and Alur et al. ([16, 1]) focus on metric temporal logic (MTL), and parametric temporal logic (PLTL), respectively, which permit limiting the scope of temporal operators. Rather than the level of degradation, the number of steps within which the given formula is satisfied, is limited. To our best knowledge, none of the existing works focuses on systems with a quality that degrades relatively, not absolutely, along a run of a system.

The rest of the paper is organized as follows. Section 2 reviews timed automata and MITL. In Section 3, we review transition systems with degradation and introduce DLTL. Sections 4 and 5 aim at verification of DLTL formulas and its half-bounded subclass, respectively. Finally, in Section 6 we conclude and outline possible future directions.

2 PRELIMINARIES

A timed automaton is an automaton equipped with a finite set of real-valued clock variables (*clocks*) that can be intuitivelly viewed as stopwatches allowing us to reason about timed properties of real-time systems.

A *clock constraint* γ over finite set of clocks X is a finite expression constructed according to the grammar $\gamma ::= x \bowtie c \mid \gamma \wedge \gamma$, where $\bowtie \in \{<, \leq, >, \geq\}$, $x \in X$, and $c \in \mathbb{N}$. Let $CC(X)$ denote the set of all clock constraints over X. A *clock valuation* ν is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ assigning to each clock $x \in X$ its current value $\nu(x)$. We use $\nu + d$ to denote valuation ν', where $\nu'(x) = \nu(x) + d$ for each $x \in X$.

Definition 1 (Timed Automaton (TA))**.** A *timed automaton* is a tuple $\mathcal{A} = (Q, \Sigma, X, \delta, Q_{init}, Inv, AP, L)$, where Q is a finite set of states, Σ is a finite set of actions, X is a finite set of clocks, $\delta \subseteq Q \times CC(X) \times \Sigma \times 2^X \times Q$ is a transition relation, $Q_{init} \subseteq Q$ is a set of initial states, $Inv : Q \rightarrow CC(C)$ is an invariant-assignment function, AP is a set of atomic propositions, and $L : Q \rightarrow 2^{AP}$ is a labeling function.

A 5-tuple $(q_1, \gamma, \sigma, R, q_2) \in \delta$ corresponds to a transition from state q_1 to q_2 labeled with σ that is enabled if constraint γ is satisfied. R denotes the subset of clock variables that are reset to zero when the transition is executed. Time can progress (i.e. the value of clock can increase) in states, whereas transitions between states always take zero time. Function Inv assigns to each state an invariant that gives a limit on how much time can be spent in that state. There are two possible ways how a TA can evolve: via *discrete* transitions, i.e., those between states, and *delay* transition, i.e., staying in a state with letting time pass.

A run of a timed automaton is a sequence $\rho = (q_0, \nu_0) \xrightarrow{d_0} (q_0, \nu'_0) \xrightarrow{\sigma_0} (q_1, \nu_1) \xrightarrow{d_1} (q_1, \nu'_1) \xrightarrow{\sigma_1} (q_2, \nu_2) \ldots$, such that $q_0 \in Q_{init}$, $\forall x \in X : \nu_0(x) = 0$, and $\forall i \in \mathbb{N}$:

- $\big((q_i,\nu_i),d_i,(q_i,\nu_i')\big)$ if and only if $d_i \in \mathbb{R}_{\geq 0}$, $\nu_i'(x) = \nu_i(x) + d_i$ for all $x \in X$, and ν_i' satisfies $Inv(q_i)$, and
- $\big((q_i,\nu_i'),\sigma_i,(q_{i+1},\nu_{i+1})\big)$ if and only if there exists $(q_i,\gamma_i,\sigma_i,R_i,q_{i+1}) \in \delta$, such that $\nu_i' \models \gamma_i$, $\nu_{i+1}(x) = \nu_i'(x)$ for all $x \in X \setminus R_i$, $\nu_{i+1}(x) = 0$ for all $x \in X \cap R_i$, and ν_{i+1} satisfies $Inv(q_{i+1})$.

A *position* on the run ρ is defined as any state that may appear during the run, i.e., a tuple (q_i,ν), where $\nu = \nu_i + d$, and $d \leq d_i$. A *time duration* $\mathbb{T}_\rho(q_i,\nu)$ up to position (q_i,ν) is the sum of the delays up to this position $\mathbb{T}_\rho(q_i,\nu) = \sum_{j=0}^{i-1} d_i + d$. We denote by $\rho(t)$ and ρ^t a position (q,ν) on ρ, such that $\mathbb{T}_\rho(q,\nu) = t$ and suffix of run ρ initialized in $\rho(t)$, respectively. Run ρ produces a *word* $w = \big(L(q_0),\mathbb{T}_\rho(q_0,\nu_0')\big)\big(L(q_1),\mathbb{T}_\rho(q_1,\nu_1')\big)\ldots$. A *language* $\mathcal{L}(\mathcal{A})$ of a timed automaton $\mathcal{A}$ is a set of all words produced by all runs of $\mathcal{A}$.

Metric Interval Temporal Logic (MITL) is a specification logic for real-time systems. MITL formulas are interpreted over runs of timed automata.

Definition 2 (MITL Syntax)**.** The syntax of a MITL formula over the set of atomic propositions AP is given as follows: $\varphi ::= tt \mid \alpha \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathsf{U}_I \varphi$, where $\alpha \in AP$, and I is a non-singular[1] interval with integer end-points (I may be also unbounded).

Definition 3 (MITL Semantics)**.** Given a MITL formula φ and a run ρ of a timed automaton $\mathcal{A}$, the satisfaction relation $\rho \models \varphi$ is for formulas φ of form $tt \mid \alpha \mid \neg\varphi \mid \varphi \wedge \varphi$ given analogously as for LTL [2, 6]. Furthermore, $\rho \models \phi \mathsf{U}_I \psi \Leftrightarrow \exists t \in \mathbb{R}_{\geq 0}$, such that $(\rho^t \models \psi \wedge t \in I \wedge \forall 0 \leq t' < t.(\rho^{t'} \models \phi))$.

Each MITL formula φ defines a language $\mathcal{L}(\varphi)$ of all words produced by all runs satisfying φ. Note that MITL formulas do not contain next operator, because the time domain is dense. Boolean operators $\vee$, and $\Rightarrow$ are defined in the usual way. Besides that, we define temporal operators $\mathsf{F}_I\varphi \equiv tt\mathsf{U}_I\varphi$ (eventually), $\mathsf{G}_I\varphi \equiv \neg\mathsf{F}_I\neg\varphi$ (globally), and $\phi\mathsf{R}_I\varphi \equiv \neg(\neg\phi\mathsf{U}_I\neg\varphi)$ (release).

Given a timed automaton $\mathcal{A}$ and a MITL formula φ, the model checking question whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\varphi)$ can be solved using automata-based approach. First, formula φ is negated and translated into a timed automaton $\mathcal{B}_{\neg\varphi}$. Then, a product timed automaton $\mathcal{A} \times \mathcal{B}_{\neg\phi}$ is built, such that $\mathcal{L}(\mathcal{A} \times \mathcal{B}_{\neg\varphi}) = \mathcal{L}(A) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi})$. Finally, by checking emptiness of $\mathcal{L}(\mathcal{A} \times \mathcal{B}_{\neg\varphi})$, the answer to model checking problem is obtained. MITL model checking is EXPSPACE-complete [2].

The translation process from a MITL formula $\neg\varphi$ into timed automaton $\mathcal{B}_{\neg\varphi}$ requires the intervals appearing in $\neg\varphi$ to have integer bounds. Although this might seem restrictive, there is a simple way how to extend the results to deal with intervals with rational bounds. The "trick" is to pick a suitable constant $p \in \mathbb{Q}_{>0}$ and multiply all the interval bounds appearing in $\neg\varphi$ with p in order to get integer interval bounds. All the constants that appear in the model checked timed automaton $\mathcal{A}$ have to be multiplied with p as well.

[1] Singular intervals are those of $[t,t]$ form.

3 SPECIFICATION OF SYSTEMS WITH DEGRADATION

3.1 Modeling Systems with Degradation

In this section we review a modeling formalism for systems with degradation introduced in our earlier work [7]. A Transition System with Degradation is a labeled transition system that is enhanced with a rational degradation constant associated with every transition.

Definition 4 (Transition System with Degradation (TSD)). A transition system with degradation is a tuple $\mathcal{T} = (S, Act, T, D, S_{init}, AP, L)$, where

- S is a finite set of states,
- Act is a finite set of actions,
- $T \subseteq S \times Act \times S$ is a transition relation,
- $D : T \to (0, 1]$ is a degradation relation,
- $S_{init} \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions,
- $L : S \to 2^{AP}$ is a labeling function.

Transition $t = (s_1, a, s_2) \in T$ represents that the system can make a transition from state s_1 to state s_2 under action a. The degradation constant $D(t)$ determines to what fraction the level of quality degrades when the transition t is executed. If $D(t) = 1$ the level of quality is unchanged, if $D(t) = 0.75$ the level of quality is decreased to 75% of the level of quality at the moment before the transition was executed. In other words, if the level of degradation is l at state s_1, then after the execution t, the level of degradation at state s_2 is $l \cdot D(t)$.

A *run* of a TSD $\mathcal{T} = (S, Act, T, D, S_{init}, AP, L)$ is an infinite sequence $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots$, where $s_i \in S$ and $t_i = (s_i, a_i, s_{i+1}) \in T$ for all $i \geq 0$. We denote by $\pi(i)$ and π^i the (i+1)-st state of the run π (i.e., s_i) and the suffix beginning in $\pi(i)$, respectively. A *level of degradation* $\mathbb{D}_\pi(i)$ on run π up to state $\pi(i)$ is defined as a product of all degradation constants associated with transitions along this state $\mathbb{D}_\pi(i) = \prod_{j=0}^{i-1} D(t_j)$.

Example 1. An example of a system with degradation is illustrated in Figure 1.

3.2 Temporal Logic for Systems with Degradation

In our previous work [7], we have shown that systems with degradation may be model checked if the property to be verified (its negation to be more precise) is described by a so-called Büchi automaton with degradation constraints (BADC). This is, however, the major drawback of the method as specifying properties (or

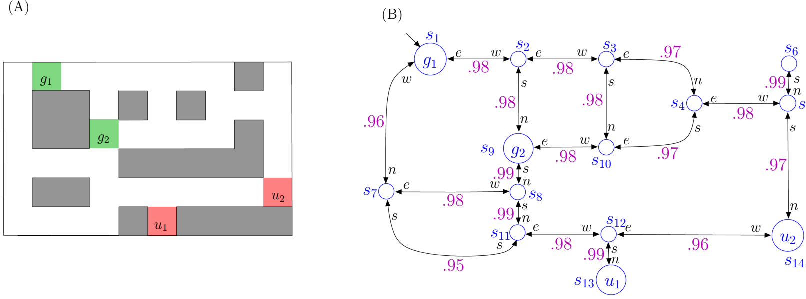

Fig. 1. An example of a data gathering robot system. The left figure (A) illustrates an environment with two data gather locations (g_1, and g_2) and two data upload locations (u_1, and u_2). The set $AP = \{g_1, g_2, u_1, u_2\}$ is the set of atomic propositions. Initially, the robot is placed in g_1. In intersections, one of the actions n (go north), s (go south), w (go west), or e (go east) can be applied. When an action is chosen, the robot turns the corresponding direction and further simply follows the path until another intersection, a data gather location, or a data upload location is reached. The robot's task is to periodically gather data in data gather locations and upload them in data upload locations. The robot is equipped with a fast, cheap, volatile memory that stores data in capacitors within an integrated circuit. Since the capacitors leak charge, the data gathered in data gather locations gradually decay. We can capture the data quality degradation through labeling of transitions with degradation constants. The traveling times between regions differ and thus the degradation constants differ for different transitions, too. The right figure (B) illustrates the corresponding TSD. For simplicity, we illustrate two transitions representing connection of two regions as a bidirectional arrow. Each bidirectional arrow is thus labeled with two actions, one for each direction. In each state, the closer one of the two labels represents the action that can be applied in that state. For instance, $s_1 \xrightarrow{e} s_2$, and $s_2 \xrightarrow{w} s_1$. Each transition is also labeled with a degradation constant. States are labeled with sets of atomic propositions. Namely, $L(s_1) = \{g_1\}$, $L(s_9) = \{g_2\}$, $L(s_2) = \emptyset$, etc.

their negations) directly as BADCs is not a user-friendly process. On the other hand, expressing properties by means of a temporal logic can be viewed as quite intuitive process with some resemblance to natural language.

We propose Linear Temporal Logic with Degradation Constraints (DLTL) that allows for specification of quantitative properties of systems with degradation. The syntax of DLTL resembles syntax of MITL, however the logics differ in their semantics as they are interpreted over significantly different models.

Definition 5 (DLTL Syntax)**.** Let $\alpha \in AP$, and I be an interval within $(0, 1]$. The syntax of a DLTL formula over the set of atomic propositions AP is given according to the following rules:

$$\varphi ::= tt \mid \alpha \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}_I \varphi \mid \varphi \mathsf{U}_I \varphi.$$

Definition 6 (DLTL Semantics)**.** Let π a run of a TSD $\mathcal{T}$. DLTL semantics is defined through the satisfaction relation $\models$.

- $\pi \models tt$ always
- $\pi \models a \iff a \in L(s_0)$
- $\pi \models \neg\varphi \iff \pi \not\models \varphi$
- $\pi \models \varphi \wedge \psi \iff \pi \models \varphi \wedge \pi \models \psi$
- $\pi \models \mathsf{X}_I\varphi \iff \pi^1 \models \varphi \wedge \mathbb{D}_\pi(1) \in I$
- $\pi \models \varphi \mathsf{U}_I \psi \iff \exists j.(\pi^j \models \psi \wedge \mathbb{D}_\pi(j) \in I \wedge \forall 0 \leq i < j.(\pi^i \models \varphi))$

The standard LTL operators X, and U are included in DLTL as $\mathsf{X}_{(0,1]}$, and $\mathsf{U}_{(0,1]}$, respectively. Other Boolean operators such as $\vee$ (disjunction), and $\Rightarrow$ (implication) are defined in the expected way. In addition to that, we also define three useful temporal operators $\mathsf{F}_I\varphi \equiv tt\mathsf{U}_I\varphi$ (eventually), $\mathsf{G}_I\varphi \equiv \neg\mathsf{F}_I\neg\varphi$ (globally), and $\phi\mathsf{R}_I\psi \equiv \neg(\neg\phi\mathsf{U}_I\neg\psi)$ (release).

Similarly as LTL formulas, DLTL formulas can be *normalized*, i.e. transformed into a form, where all negations are applied only directly to atomic propositions.

Example 2. In our robot data gathering system (Example 1), a property "Data are gathered in location g_1. The data are moved to an upload location before their integrity drops bellow 90% and meanwhile, no data are gathered in location g_2." can be expressed as a DLTL formula $g_1 \wedge \neg g_2 \mathsf{U}_{\geq 0.9}(u_1 \vee u_2)$. A more complicated property for our robot data gathering system is, for instance, "Periodically visit both data gathering locations. Always upload the data in one of the upload locations before their integrity drops bellow 90%." A DLTL formula expressing this behavior is $\mathsf{GF}g_1 \wedge \mathsf{GF}g_2 \wedge \mathsf{G}\big(g_1 \vee g_2 \rightarrow \mathsf{F}_{\geq 0.9}(u_1 \vee u_2)\big)$.

4 TIMED AUTOMATA APPROACH TO DLTL MODEL CHECKING

The verification question we would like to answer is, whether all runs of a given TSD satisfy a given DLTL formula. We approach this problem via its conversion into verification problem for timed automata and MITL formulas. During the conversion process, two major differences have to be overcome: (1) in systems with degradation, the degradation decreases along the *transitions*, whereas in timed systems, the time passes in the *states*, and (2) the degradation constants are meant to be *multiplied*, whereas time passes in *additive* fashion. We address the first one by modelling transitions of a TSD as states of a timed automaton and the second one by applying logarithm to the degradation constants. We build on the fact that $\log a \cdot b = \log a + \log b$.

Assume that the given DLTL formula φ sastisfies two additional assumptions:

1. the intervals that appear in φ are non-singular, and
2. φ does not contain next operator.

These restrictions allow us to translate φ into a MITL formula. We discuss how to deal with full DLTL later.

First, we preprocess the given TSD $\mathcal{T} = (S, Act, T, D, S_{init}, AP, L)$ into a TSD $\mathcal{T}' = (S', Act', T', D', S_{init}, AP', L')$ this way:

- $S' = S \cup S \times T \cup T \times S$,
- $Act' = Act \cup \{\epsilon\}$, where $\epsilon \notin Act$,
- $T' = \{\big(s_1, \sigma, (s_1, t)\big), \big((s_1, t), \epsilon, (t, s_2)\big), \big((t, s_2), \epsilon, s_2\big) \mid t = (s_1, \sigma, s_2) \in T\}$,
- $D'(t) = 1$ for all transitions leading from and to some $s \in S$, and $D'\big((s_1, t), \epsilon, (t, s_2)\big) = D(t)$ for the rest of the transitions,
- $AP' = AP \cup \{\alpha_\epsilon\}$, where $\alpha_\epsilon \notin AP$,
- $L'(s) = L(s)$ for all $s \in S$, and $L'(s) = \{\alpha_\epsilon\}$ for all $s \in S' \setminus S$.

Second, we convert the given normalized DLTL formula φ into φ' by replacing each *non-negated* occurrence of atomic proposition α with $\alpha_\epsilon \mathsf{U} \alpha$ and each *negated* occurrence of α with $\alpha_\epsilon \mathsf{U} \neg\alpha$. This way, we "ignore" the states corresponding to the transitions of $\mathcal{T}$.

Lemma 1. $\mathcal{T} \models \varphi \iff \mathcal{T}' \models \varphi'$.

Proof: (Sketch.) Each run π producing word $\alpha_0\alpha_1\alpha_2\ldots$ in $\mathcal{T}$ maps to a single run π' producing word $\alpha_0\alpha_\epsilon\alpha_\epsilon\alpha_1\alpha_\epsilon\alpha_\epsilon\alpha_2\ldots$ in $\mathcal{T}'$. It is easy to show by induction that for all $i \geq 2$ it holds that $\pi^i \models \varphi$ if and only if $\pi'^{3i-4} \models \varphi' \wedge \pi'^{3i-3} \models \varphi' \wedge \pi'^{3i-2} \models \varphi'$. Finally, we get that $\pi \models \varphi \iff \pi' \models \varphi'$. □

Given TSD $\mathcal{T}$ and the corresponding TSD $\mathcal{T}'$, we build a timed automaton $\mathcal{A} = (S \cup T, Act', \{x\}, \delta, S_{init}, Inv, AP', L_{\mathcal{A}})$, where

- $\delta = \{\big(s_1, x = 0, \sigma, \emptyset, t\big), \big(t, x = \log D(t), \epsilon, \{x\}, s_2\big) \mid t = (s_1, \sigma, s_2) \in T\}$,
- $Inv(s) = x \leq 0$ for all $s \in S$, and $Inv(t) = x \leq \log_k D(t)$ for all $t \in T$,
- $L_{\mathcal{A}}(s) = L(s)$ for all $s \in S$, and $L_{\mathcal{A}}(t) = \{\alpha_\epsilon\}$ for all $t \in T$,

such that k is a logarithm base arbitrarily picked from $(0, 1)$. Note that $\log_k x > 0$ for all $x \in (0, 1]$. For simplicity, let log denote $\log_k$ in further text. A DLTL formula φ' is transformed into a MITL formula ϑ as follows: Each occurrence of interval (a, b) is replaced with $(\log b, \log a)$, and analogously, each occurrence of $(a, b]$, $[a, b]$, and $[a, b)$ is replaced with $[\log b, \log a)$, $[\log b, \log a]$, and $(\log b, \log a]$, respectively. In case $a = 0$, we use ∞ instead of $\log a$. The rest of the formula remains the same.

Lemma 2. $\mathcal{T}' \models \varphi' \iff \mathcal{A} \models \vartheta$.

Proof: Follows directly from the structure of $\mathcal{T}'$, construction of $\mathcal{A}$, and the fact that $log(a \cdot b) = \log a + \log b$, and $0 < a \leq c \leq b \leq 1 \Rightarrow 0 \leq \log b \leq \log c \leq \log a < \infty$. □

Corollary 1. $\mathcal{T} \models \varphi \iff \mathcal{A} \models \vartheta$.

Example 3. An example of an edge transformation of a TSD $\mathcal{T}$ into a TSD $\mathcal{T}'$ and into a timed automaton $\mathcal{A}$ is illustrated in Figure 2.

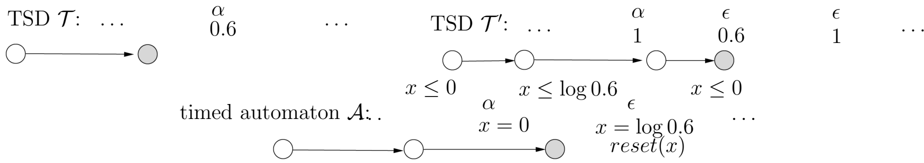

Fig. 2. An example of edge transformation of a TSD $\mathcal{T}$

The remaining task is to check emptiness of $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\neg\vartheta)$. Without loss of generality, assume that $\neg\vartheta$ is normalized from now on. If $\log c$ is a rational number for all constants c that appear in formula $\neg\vartheta$, then we proceed as follows. We pick a suitable constant $p \in \mathbb{Q}_{>0}$ and multiply all the constants both in $\mathcal{A}$ and $\neg\vartheta$ with p in order to make all the interval bounds appearing in formula $\neg\vartheta$ integer. Formula $\neg\vartheta$ can now be translated into a timed automaton $\mathcal{B}_{\neg\vartheta}$. The rest is well-known checking of language emptiness for timed automaton $\mathcal{A} \times \mathcal{B}_{\neg\vartheta}$. However, in many cases it is not possible to find k, such that $\log_k c$ is a rational number for each constant c that appears in $\neg\vartheta$. Therefore, necessarily, some kind of approximation is needed.

Lemma 3. Consider intervals I and I', such that interval I' is within I. For any run ρ, it holds that $\rho \models \varphi \mathsf{U}_{I'} \psi \Rightarrow \varphi \mathsf{U}_I \psi$, and dually, $\rho \models \varphi \mathsf{R}_I \psi \Rightarrow \varphi \mathsf{R}_{I'} \psi$.

Proof: Directly from expanding definition of U_I and R_I, respectively. □

Based on Corollary 1 and Lemma 3, the model checking procedure can be summarized as follows:

1. Transform TSD $\mathcal{T}$ into timed automaton $\mathcal{A}$ and DLTL formula φ into MITL formula ϑ. Obtain $\neg\vartheta$ as normalized negation of ϑ.
2. Pick a precision constant $p \in \mathbb{Q}_{>0}$, and multiply all constants in $\mathcal{A}$ and in $\neg\vartheta$ with p.
3. In each U_I operator in $\neg\vartheta$ replace left bound a and right bound b of interval I with $\lceil a \rceil$ and $\lfloor b \rfloor$, respectively. Dually for each R_I operator.
4. Translate $\neg\vartheta$ into $\mathcal{B}_{\neg\vartheta}$ and check, whether $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}_{\neg\vartheta}) = \emptyset$. If yes, then $\mathcal{T} \models \varphi$, otherwise continue on line 5.
5. In each U_I operator in of $\neg\vartheta$ replace left bound a and right bound b of interval I with $\lfloor a \rfloor$ and $\lceil b \rceil$, respectively. Dually for each R_I operator.
6. Translate $\neg\vartheta$ into $\mathcal{B}_{\neg\vartheta}$ and check, whether $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}_{\neg\vartheta}) = \emptyset$. If no, then $\mathcal{T} \not\models \varphi$, otherwise pick a precision constant $p' > p$ and repeat the procedure from line 3.

If the outlined procedure provides an answer, the answer is correct. On the other hand, the termination is not guaranteed and thus we can only answer model checking question with limited, yet arbitrary precision. The price paid for increasing precision is rapidly increasing computational demands. The size of timed automaton $\mathcal{B}_{\neg\vartheta}$ is $O(2^{N\cdot K})$ with $N \cdot K$ clocks, where N is the number of atomic propositions, Boolean, and temporal operators in $\neg\vartheta$ and $K-1$ is the largest integer constant in $\neg\vartheta$ [2]. Higher precision causes increase of the constant K, and hence also significant increase of the size of $\mathcal{B}_{\neg\vartheta}$.

Remark 1. For the sake of presentation simplicity, we assumed continuous semantics of timed automata, although the dynamics of a TSD is purely discrete. Therefore we had to restrict DLTL formulas not to contain next operators and singular intervals. In order to solve the model checking problem for full DLTL, one can consider discrete semantics of timed automata and Metric Temporal Logic (which is MITL including singular intervals). The approach is analogous, but approximation is needed not only in the formulas, but in the timed automaton as well. The complexity of emptiness checking of $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\neg\vartheta)$ remains EXPSPACE-complete and dependent on the size of constants appearing in $\neg\vartheta$ [2].

5 VERIFICATION OF HALF-BOUNDED DLTL FORMULAS

The timed automata approach summarized in the previous section gives answer to the DLTL verification question with limited precision only. In this section, we focus on a different approach to DLTL model checking problem providing a full precision verification algorithm for a special subclass of DLTL formulas with half-bounded intervals.

Definition 7 (Half-Bounded DLTL Formula)**.** A half-bounded DLTL formula is a DLTL formula given according to the grammar $\varphi ::= tt \mid \alpha \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}_I\varphi \mid \varphi \mathsf{U}_I \varphi$, where $\alpha \in AP$, and I is an interval of form $(0,d], (0,d), (d,1]$, or $[d,1]$, where $d \in (0,1]$. For simplicity, we use $\bowtie d$, where $\bowtie \in \{\leq, <, \geq, >\}$ to denote interval I. In particular, we use $\leq d, < d, \geq d$, and $> d$ to denote interval $(0,d], (0,d), (d,1]$, and $[d,1]$, respectively.

Unlike in the previous section dealing with full DLTL, in this section we follow directly the automata-based approach to LTL model checking [18], which consists of four steps. First, a system is modeled as a transition system, and a specification is expressed as an LTL formula. Second, the negation of the LTL formula is translated into an automaton over infinite words. Third, a product automaton is constructed accepting the runs of the system that satisfy the negation of the specification. Finally, the product automaton graph is analyzed with the use of standard graph techniques to answer whether there exists a run satisfying the negation of the specification, i.e., violating the original LTL formula.

In our earlier work [7] we developed an automata-like formalism for specification of systems with degradation, called Büchi Automata with Degradation Constraints

(BADC), and an algorithm for verification of TSDs against specification given as a BADC. Hence, in this section, we only focus on the second step of the above outline and we present translation of a half-bounded DLTL formula into a BADC. Thus we complete a technique for verification of TSDs against half-bounded DLTL formulas. Note that the algorithm provides answer to the verification question with full precision.

5.1 Büchi Automata with Degradation Constraints

Büchi Automata with Degradation Constraints [7] are the standard Büchi automata that are enriched with a finite set of rational-domain variables, the so-called *degradation variables*, used to capture the level of degradation during the system execution. Furthermore, transitions of BADCs are enriched with the so-called *resets* of degradation variables and they are guarded with degradation constraints. Note that the range of every degradation variable is $(0, 1]$. Informally, the role of a degradation variable is to measure the level of degradation since the last time the variable was reset. Likewise the standard Büchi automata, BADCs recognize sets of traces over a given set of atomic propositions. They can be thus used to express desired or invalid properties of runs of TSDs.

In order to define BADCs formally, we first give definition of degradation constraints. A *degradation constraint* γ *over* the set of degradation variables X is constructed according to the following rules: $\gamma ::= x \bowtie d \mid d \bowtie d \mid \gamma \wedge \gamma$, where $\bowtie \in \{<, \leq, >, \geq\}$, $x \in X$, and $d \in (0, 1]$ is rational. *Degradation valuation* ν is a function that assigns every degradation variable a value from $(0, 1]$, i.e. $\nu : X \to (0, 1]$. The set of all possible degradation valuations and the set of all degradation constraints is denoted by $Eval(X)$ and $DC(X)$, respectively. We also use $\gg$ to denote $>$ or $\geq$, and $\ll$ to denote $<$ or $\leq$.

Definition 8 (BADC)**.** A Büchi Automaton with Degradation Constraints (BADC) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, X, q_{init}, F)$, where

- Q is a finite, nonempty set of states,
- Σ is a finite alphabet,
- $\delta \subseteq Q \times \Sigma \times DC(X) \times 2^X \times Q$ is a transition relation,
- X is a finite set of degradation variables,
- $q_{init} \in Q$ is an initial state,
- $F \subseteq Q$ is a finite set of states (so-called Büchi accepting condition).

A 5-tuple $(q, \alpha, \gamma, R, q') \in \delta$ corresponds to a transition from state q to q' labeled with α that is enabled if constraint γ is satisfied. R then denotes the subset of degradation variables that are reset to 1 when the transition is executed.

The semantics of BADCs are defined over infinite input words from $(\Sigma \times (0, 1])^\omega$, i.e., when $\Sigma = 2^{AP}$ then the semantics are defined over traces of a TSD.

Definition 9 (BADC Semantics)**.** The semantics of a BADC $\mathcal{A} = (Q, \Sigma, \delta, X, q_{init}, F)$ is given by a (possibly infinite) labeled transition system $\mathcal{S} = (S, Act, T, S_{init})$, where

- $S = Q \times Eval(X)$
- $Act = \Sigma \times (0, 1]$
- $T \subseteq S \times Act \times S$. $\big((q_1, \nu_1), (\alpha, d), (q_2, \nu_2)\big) \in T$, i.e., $(q_1, \nu_1) \xrightarrow{\alpha,d} (q_2, \nu_2)$, if and only if there is a transition $(q_1, \alpha, \gamma, R, q_2) \in \delta$, such that
  - ν_1 satisfies constraint γ
  - $\nu_2(x) = \begin{cases} d, & \text{if} x \in R \\ \nu_1(x) \cdot d & \text{otherwise} \end{cases}$
- $S_{init} = \{(q_{init}, \nu_{init}) \mid \nu_{init}(x) = 1 \text{ for all } x \in X\}$.

A run of a BADC over a word $\sigma = (\alpha_0, d_0)(\alpha_1, d_1) \ldots \in (\Sigma \times (0, 1])^\omega$ is an infinite sequence $\rho = (q_0, \nu_0)(q_1, \nu_1)(q_2, \nu_2) \ldots$ such that $(q_0, \nu_0) \in S_{init}$ and $(q_i, \nu_i) \xrightarrow{\alpha_i,d_i} (q_{i+1}, \nu_{i+1})$ for all $i \geq 0$. A run $\rho = (q_0, \nu_0)(q_1, \nu_1) \ldots$ is accepting if $q_i \in F$ for infinitely many indices i. Language of all words accepted by BADC $\mathcal{A}$ is $L_\omega(\mathcal{A}) = \{\sigma \in (\Sigma \times (0, 1])^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{A}\}$.

Example 4. For our robot data gathering system from Example 1, Figure 3 illustrates a BADC that captures the following property: "Data are periodically gathered in location g_1 and uploaded in an upload location before their integrity drops bellow 90%."

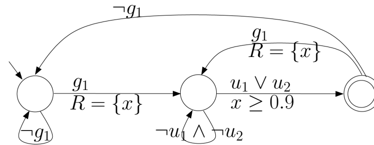

Fig. 3. A BADC for a robot data gathering system

5.2 Translation of DLTL Formulas into BADCs

The algorithm we present for translation of DLTL formulas into BADCs is based on the standard algorithm for translation of LTL formulas into Büchi automata as given in [12]. However, several nontrivial issues have to be overcome to deal with the degradation constraints.

5.2.1 Scheme of Translation

The whole process of translation consists of several succeeding steps. These are summarized below.

- First, a linear temporal logic formula φ is rewritten into such a form that the formula contains only the standard and constrained temporal operators U (until), R (release), and X (next), and at the same time all the $\neg$ (negation) operators are applied to propositional variables only.
- Second, a formula graph is built with a tableau-like procedure. Each graph node is defined by several sets of formulas. The key idea of the procedure employs several equalities such as the following one. Formula $\phi \mathsf{U} \psi$ is equal to $\psi \vee (\phi \wedge \mathsf{X}(\phi \mathsf{U} \psi))$. This equality allows to define what are the requirements on the current state of the run and what are the requirements on the next state of the run. For $\phi \mathsf{U} \psi$ subformula there are two decomposition options: either ψ has to hold true in the current state and there are no requirements regarding the next state, or ϕ has to hold true in the current state and $\phi \mathsf{U} \psi$ has to hold in the next state.
- Third, using the formula graph, a generalized Büchi automaton (GBA) is built.
- Finally, the GBA is transformed into a standard Büchi automaton accepting exactly the words satisfying the formula φ.

5.2.2 Normal Form

DLTL formula φ is *normalized* if it contains only temporal operators U, X, R, $\mathsf{U}_{\bowtie d}$, $\mathsf{X}_{\bowtie d}$, and $\mathsf{R}_{\bowtie d}$ and all negation operators apply to atomic propositions. Any DLTL formula can be normalized using the standard syntax equivalences and equivalences defined earlier in Section 3.2. Thus, without loss of generality, we may assume that from now on the given DLTL formula φ is normalized.

5.2.3 Formula Graph

Let us first informally present key ideas used in the construction of the formula graph for a normalized DLTL formula φ. The full algorithm is presented later in this section.

The basic data structure of the formula graph is called a *node*. Each node contains three important set of formulas, which are subformulas of the original formula φ. Intuitively, these describe temporal properties of suffices of runs of TSDs. The three sets are:

- *Old* is a set of formulas, that must hold in the node and have been already processed.
- *New* is a set of formulas, that must hold in the node and have not been processed yet.
- *Next* is a set of formulas that must hold in all immediate successors of nodes satisfying properties in *Old*.

Initially, the construction starts with a node, whose *Old* and *Next* are empty and *New* contains the formula φ. The nodes are iteratively expanded. Every time,

a node q and a formula $\psi \in New(q)$ are chosen to be processed. The processed node is replaced with a new node (or several nodes). The sets *Old*, *New* and *Next* are inherited from the processed node, but ψ is moved from *New* to *Old* and *New* and *Next* are updated depending on the formula ψ. When there are no other formulas in $New(q)$, a new node p with $New(p) = Next(q)$ is created together with an edge leading from q to p.

The key idea in the translation process of DLTL formulas is that constraints on the level of degradation given in the formula φ can be captured through degradation constraints and resets associated with the edges between nodes in the formula graph. More precisely, we associate a degradation variable with every unary, or binary operator $\mathsf{O}_{\bowtie d}$ when a formula $\mathsf{O}_{\bowtie d}\psi$, or $\phi\mathsf{O}_{\bowtie d}\psi$, respectively, is moved from *New* to *Old*. Let us now explain in more details how each of formulas $\mathsf{X}_{\bowtie d}\psi$, $\phi\mathsf{U}_{\bowtie d}\psi$, and $\phi\mathsf{R}_{\bowtie d}\psi$ is processed.

Let q and $\mathsf{X}_{\bowtie d}\psi \in New(q)$ be the currently processed node and formula, respectively. For each trace π satisfying $\mathsf{X}_{\bowtie d}\psi$ it holds that the level of degradation between $\pi(0)$ and $\pi(1)$ has to satisfy $\bowtie d$. Therefore, we associate a degradation variable x with the node q and reset x on each edge incoming to the node q. All outgoing edges from the current node are labeled with constraint $x \bowtie d$. Finally, we replace q with a new node q', such that formula the $\mathsf{X}_{\bowtie d}\psi$ is moved from $New(q')$ to $Old(q')$ and ψ is added to $Next(q')$. See Figure 4.

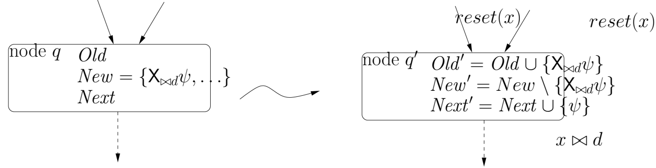

Fig. 4. Expansion of a node containing a constrained next operator in its *New* set

Let q and $\phi\mathsf{U}_{\bowtie d}\psi \in New(q)$ be the currently processed node and formula, respectively. Similarly as in the previous case, we need to measure degradation from the current node q and we do it with degradation variable x. We reset variable x on each edge incoming to the node q. There are two options: Either ψ is satisfied in the current node and the current degradation (i.e., the value of x) satisfies $\bowtie d$ or ϕ is satisfied in the current node and in the next node it holds that ϕ is satisfied until the moment when ψ is satisfied and the current degradation (i.e., the value of x) in that moment satisfies $\bowtie d$. As we have to remember that degradation variable x is already initialized in the node q, we define an until operator with fixed degradation variable $\mathsf{U}^x_{\bowtie d}$:

$$\phi\mathsf{U}^x_{\bowtie d}\psi \equiv (x \bowtie d \wedge \psi) \vee \mathsf{X}(\phi\mathsf{U}^x_{\bowtie d}\psi).$$

The difference between $\mathsf{U}_{\bowtie d}$ and $\mathsf{U}^x_{\bowtie d}$ operators is as follows: When $\phi\mathsf{U}_{\bowtie d}\psi$ is to be satisfied in the node q, we have to associate a variable x with this formula and reset it on all edges incoming to q. When $\phi\mathsf{U}^x_{\bowtie d}\psi$ is to be satisfied in the node q,

it is already associated with a degradation variable x. Variable x cannot be reset, because it "measures" the degradation from some of the predecessors of q. The operator $\mathsf{U}_{\bowtie d}$ can be expressed as:

$$\phi \mathsf{U}_{\bowtie d} \psi \equiv (x \bowtie d \wedge \psi) \vee \mathsf{X}(\phi \mathsf{U}^{x}_{\bowtie d} \psi),$$

where x is a degradation variable with value set to 1.

However, in some cases (such as in translation of formula $\mathsf{G}(\phi \mathsf{U}_{\bowtie d} \psi)$) this approach would lead into an *infinite* formula graph with an *infinite* number of degradation variables. Hence, we present ideas that allow us to introduce only finite number of degradation variables in the formula graph.

1. Let φ be a translated formula and $\phi \mathsf{U}_{\gg d} \psi$ its subformula. We allocate exactly one degradation variable x with the subformula $\phi \mathsf{U}_{\gg d} \psi$ and whenever $\phi \mathsf{U}_{\gg d} \psi$ appears in $New(q)$ of a node q, we associate it with this x. The question is, how to proceed when variable x is needed to be reset in two different nodes as illustrated in Figure 5.

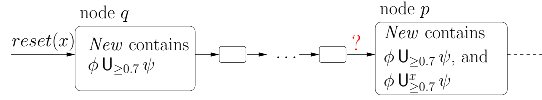

Fig. 5. We cannot reset x at the edge labeled with "?" as we would lose information about the degradation for the first occurrence of the until formula. At the same time we have to reset x there to measure the degradation for the second occurrence of the until formula.

   We know that if $\phi \mathsf{U}^{x}_{\gg d} \psi$ is satisfied in a node q, then also $\phi \mathsf{U}_{\gg d} \psi$ is satisfied there. This is because the degradation is monotonically decreasing. Hence, whenever both $\phi \mathsf{U}_{\gg d} \psi \in New(q)$ and $\phi \mathsf{U}^{x}_{\gg d} \psi \in New(q)$, we remove $\phi \mathsf{U}_{\gg d} \psi$ from $New(q)$ completely. The satisfaction of $\phi \mathsf{U}^{x}_{\gg d} \psi$ will ensure satisfaction of $\phi \mathsf{U}_{\gg d} \psi$.

2. A bit more complicated situation arises when we consider subformulas $\phi \mathsf{U}_{\ll d} \psi$.

   We know that if $\phi \mathsf{U}_{\ll d} \psi$ is satisfied in a node q, then also $\phi \mathsf{U}^{x}_{\ll d} \psi$ is satisfied there (because the degradation is monotonically decreasing). However, when both $\phi \mathsf{U}_{\ll d} \psi \in New(q)$ and $\phi \mathsf{U}^{x}_{\ll d} \psi \in New(q)$, we cannot simply ignore the formula $\phi \mathsf{U}^{x}_{\ll d} \psi$ and reset the variable x. This is due to the fact that the complete satisfaction (meaning reaching ψ and satisfying $x \ll d$) of the formula $\phi \mathsf{U}^{x}_{\ll d} \psi$ can be reached earlier than satisfaction of $\phi \mathsf{U}_{\ll d} \psi$. This may cause that completing the satisfaction of $\phi \mathsf{U}_{\ll d} \psi$ formula will be permanently postponed and degradation constraints in the formula graph will not be satisfied even if they should be, as illustrated in Figure 6.

   Therefore, we allocate two degradation variables x and y with subformula $\phi \mathsf{U}_{\ll d} \psi$.

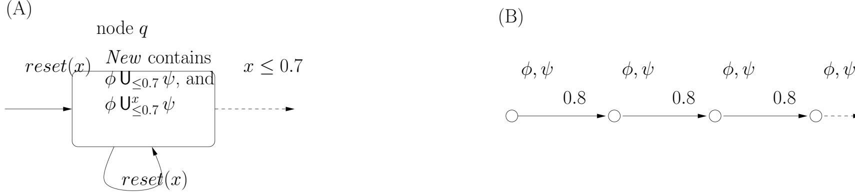

Fig. 6. When translating a formula $\mathsf{G}(\phi\mathsf{U}_{\le 0.7}\psi)$, the node q illustrated in (A) would never be left until the value of degradation variable x drops bellow 0.7. At the same time, x would be reset in every step. That means that reading the trace as produced by the run in (B) would keep the formula graph in the node q forever. Satisfaction of ψ would never been checked and the trace would not be accepted. However, the run obviously satisfies the formula $\mathsf{G}(\phi\mathsf{U}{\le}\ 0.7\psi)$.

Instead of operator $\mathsf{U}^{x}_{\bowtie d}$, we have $\mathsf{U}^{xy}_{\ll d}$, which is defined analogously:

$$\phi\mathsf{U}^{xy}_{\ll d}\psi \equiv (x \ll d \wedge \psi) \vee \mathsf{X}(\phi\mathsf{U}^{xy}_{\ll d}\psi).$$

Similarly, $\phi\mathsf{U}_{\ll d}\psi$ can be then expanded as

$$\phi\mathsf{U}_{\ll d}\psi \equiv (x \ll d \wedge \psi) \vee \mathsf{X}(\phi\mathsf{U}^{xy}_{\ll d}\psi),$$

where x and y are degradation variables with value set to 1.

Assume that variables x and y are reset in node q (i.e., $\phi\mathsf{U}_{\ll d}\psi \in New(q)$, but $\phi\mathsf{U}^{xy}_{\ll d}\psi \notin New(q)$). Consider a node p, such that $\phi\mathsf{U}_{\ll d}\psi \in New(p)$ and $\phi\mathsf{U}^{xy}_{\ll d}\psi \in New(p)$. Instead of the variable x we reset the variable y. This way, the variable x captures the degradation from the node q, whereas y from the node p. When another node r appears, such that both $\phi\mathsf{U}_{\ll d}\psi \in New(r)$ and $\phi\mathsf{U}^{xy}_{\ll d}\psi \in New(r)$, we again reset y and thus x still captures the degradation from the node q, but y now captures the degradation from the node r. Then, when constraint $x \ll d$ is satisfied and ψ holds, the variables x and y interchange their roles until $y \ll d$ is satisfied. Now, when constraint $y \ll d$ is satisfied and ψ holds, we know that $\phi\mathsf{U}_{\ll d}\psi$ is satisfied not only for r, but also for p. This is illustrated in Figure 7.

Very similar ideas as in the case of the constrained until operator $\mathsf{U}_{\bowtie d}$ are followed in the case of the constrained release operator $\mathsf{R}_{\bowtie d}$.

Lemma 4. Given a formula φ, only a finite number of degradation variables is needed to build a formula graph for the formula.

Proof: Follows from the description above. □

Lemma 5. The expansion of each node of the formula graph corresponds to the semantics of the expanded subformula.

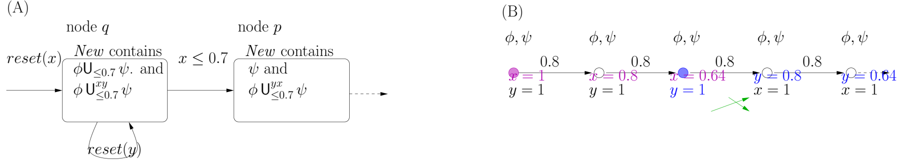

Fig. 7. Illustration of the $\phi \mathsf{U}_{\ll d}\psi$ case. The value of x is not reset until $x \leq 0.7$. After then, its value becomes unimportant and the variable x can be reused to measure degradation from a different point. The value of y is reset whenever we need to start measuring degradation. In the fifth state of the run, we find out that $\phi \mathsf{U}_{\leq 0.7}\psi$ is satisfied for the third (blue) state. Therefore, it is satisfied also for all states preceding this one.

Proof: The proof follows from the following idendities for $\mathsf{U}_{\bowtie d}$ and $\mathsf{R}_{\bowtie d}$ operators. Let π be a run and x be a degradation variable, such that $\nu(x)$ in the state $\pi(i)$ is $\mathbb{D}^{\pi(i)}_{\pi(0)}$ for all i. The semantics of operator $\mathsf{U}_{\bowtie d}$ is defined as follows. $\pi^k \models \varphi \mathsf{U}^x_{\bowtie d}\psi \iff \exists j \geq k.(\mathbb{D}^{\pi(j)}_{\pi(0)} \bowtie d \wedge \pi^j \models \psi \wedge \forall k \leq i < j.(\pi^i \models \varphi))$. Simiarly, $\pi^k \models \varphi \mathsf{R}^x_{\bowtie d}\psi \iff \forall j.(\mathbb{D}^{\pi(j)}_{\pi(0)} \bowtie d \Rightarrow (\pi^j \models \psi \vee \exists 0 \leq i < j.(\pi^i \models \varphi))$. Then $\pi \models \varphi \mathsf{U}_{\bowtie d}\psi \iff \pi \models (x \bowtie d \wedge \psi) \vee \pi \models \varphi \wedge \mathsf{X}(\varphi \mathsf{U}^x_{\bowtie d}\psi)$, which follows from the semantics of operators $\mathsf{U}_{\bowtie d}$ and $\mathsf{U}^x_{\bowtie d}$, and $\pi \models \varphi \mathsf{R}_{\bowtie d}\psi \iff \pi \models \varphi \wedge (x \bowtie d \Rightarrow \psi) \vee \pi \models (x \bowtie d \Rightarrow \psi) \wedge \mathsf{X}(\varphi \mathsf{R}^x_{\bowtie d}\psi)$, which follows from the semantics of operators $\mathsf{R}_{\bowtie d}$ and $\mathsf{R}^x_{\bowtie d}$. □

The details of the translation algorithm are given in the Appendix.

5.2.4 Formula Graph to Generalized BADC

Definition 10. A generalized BADC is a tuple $\mathcal{A}_G = (Q, \Sigma, \delta, X, q_{init}, F_G)$, where Q, Σ, δ, X and q_{init} are defined as in BADC, and $F_G \subseteq 2^{2^Q}$ is an acceptance condition. $F_G = \{F_1, \ldots, F_n\}$ is a set of sets of accepting states.

A run is defined exactly the same as for a BADC. An *accepting run* of a generalized BADC is a run $\rho = (q_0, \nu_0)(q_1, \nu_1)\ldots$, such that for each acceptance set $F \in F_G$ it holds that $q_i \in F$ for infinitely many indices i.

A generalized BADC is obtained from the formula graph as follows:

- The set of states Q is the set of nodes with a new initial node called *init*
- $\Sigma = 2^{AP}$
- $t = (q, \alpha, \gamma, R, q') \in T$ if there exists an edge e between q and q' labeled with input $\alpha \in \Sigma$ which satisfies restriction given by the set $((AP \cup \neg AP) \cap \mathit{Old}(q'))$. γ and R are the degradation constraints and resets associated with the edge e, respectively.
- $F = \{F_1, \ldots, F_n, F'_1, \ldots, F'_n\}$. For each $\varphi \mathsf{U}\psi$ subformula we define $F_i = \{q \in Q \mid \varphi U \psi \notin \mathit{Old}(q) \vee \psi \in \mathit{Old}(q)\}$, For each $\varphi \mathsf{U}_{\bowtie d}\psi$ subformula associated with degradation variable x (or variables x, y) we define $F'_i = \{q \in Q \mid$ none of $\varphi U_{\bowtie d}\psi$, $\varphi U^x_{\bowtie d}\psi$, $\varphi U^{xy}_{\bowtie d}\psi$, $\varphi U^{yx}_{\bowtie d}\psi$ belongs to $\mathit{Old}(q)$, or $\psi \in \mathit{Old}(q)\}$

5.2.5 Generalized BADC to BADC

The translation from the generalized BADC to BADC follows the same principles as the translation from the generalized BA to BA. The key idea is to make a separate copy of the BADC for each set acceptance $F \in F_G$. In i^{th} copy, only the set F_i is accepting. When a state from F_i is reached, the computation is moved to the $(i+1)^{\text{th}}$ copy (or, more precisely $((i+1) \mod |F_G|)^{\text{th}}$ copy). This way we force the resulting Büchi automaton to visit each of the set $F \in F_G$ infinitely many times.

Theorem 1. The automaton $\mathcal{A}$ constructed for a property φ over a set of atomic propositions AP accepts exactly the set of words that satisfy φ.

Proof: Follows from the correctness of construction in [12], proof of Lemma 4, and proof of Lemma 5. □

Note that because the number of degradation variables needed to capture a single degradation constraint is constant, the worst case complexity of the overall translation remains exponential as in [12].

6 CONCLUSIONS AND FUTURE WORK

In this paper we aim at quantitative properties of systems with degradation. We introduce a version of linear temporal logic that allows for specification of requirements on the level of degradation of individual system runs. We show a connection between systems with degradation and timed automata and we use MITL model checking algorithm to solve DLTL model checking problem. The solution suffers from two major drawbacks. First, the verification problem can be answered only with limited precision, and second, higher precision causes rapidly higher computational demands. We partially address these issues in this paper by introducing a verification algorithm for half-bounded fragment of DLTL.

In our future work, we plan to overcome the mentioned issues fully by introducing a direct translation process from all DLTL formulas into BADCs. Another future focus of ours is on control strategy synthesis for systems with degradation from DLTL specifications, on continuous and hybrid systems with degradation, and also on a case study.

Acknowledgement

This work has been partially supported by grants No. GAP202/11/0312, LH11065, and GD102/09/H042. Jana Tůmová is the corresponding author.

REFERENCES

[1] Alur, R.—Etessami, K.—La Torre, S.—Peled, D.: Parametric Temporal Logic for "Model Measuring". ACM Trans. Comput. Log., Vol. 2, 2001, No. 3, pp. 388–407.

[2] Alur, R.—Feder, T.—Henzinger, T. A.: The Benefits of Relaxing Punctuality. Journal of the ACM, Vol. 43, 1996, pp. 116–146.

[3] Alur, R.—Henzinger, T. A.: Real-Time Logics: Complexity and Expressiveness. Inf. Comput., Vol. 104, 1993, No. 1, pp. 35–77.

[4] Alur, R.—Henzinger, T. A..: A Really Temporal Logic. Journal of the ACM, Vol. 41, January 1994, pp. 181–203.

[5] Aziz, A.—Singhal, V.—Balarin, F.—Brayton, R. K.—Sangiovanni-Vincentelli, A. L.: It Usually Works: the Temporal Logic of Stochastic Systems. In Proceedings of the 7th International Conference on Computer Aided Verification, Vol. 195, pp. 155–165.

[6] Baier, C.—Katoen, J. P.: Principles of Model Checking. The MIT Press, 2008.

[7] Barnat, J.–Černá, I.—Tůmová, J.: Quantitative Model Checking of Systems With Degradation. In QEST '09, pp. 21–30.

[8] Barnat, J.—Černá, I.—Tůmová, J.: Timed Automata Approach to Verification of Systems With Degradation. In MEMICS '11, LNCS Volume 7119, 2011, pp. 86–95.

[9] De Alfaro, L.—Faella, M.—Henzinger, T. A.–Majumdar, R.—Stoelinga, M.: Model Checking Discounted Temporal Properties. Theor. Comput. Sci., Vol. 345, 2005, No. 1, pp. 139—170.

[10] De Alfaro, L.—Faella, M.—Stoelinga, M.: Linear and Branching Metrics for Quantitative Transition Systems. In ICALP '04, 2004, pp. 97–109.

[11] Faella, M.—Legay, A.—Stoelinga, M.: Model Checking Quantitative Linear Time Logic. Electr. Notes Theor. Comput. Sci., Vol. 220, 2008, No. 3, pp. 61–77.

[12] Gerth, R.—Peled, D.—Vardi, M. Y.—Wolper, P.: Simple On-the-Fly Automatic Verification of Linear Temporal Logic. In Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, 1996, pp. 3–18.

[13] Hansson, H.—Jonsson, B.: A Framework for Reasoning about Time and Reliability. In IEEE Real-Time Systems Symposium, 1989, pp. 102–111.

[14] iFEST homepage. `http://www.artemis-ifest.eu` (Jan 2011).

[15] Kaivola, R.—Ghughal, R.—Narasimhan, N.—Telfer, A.—Whittemore, J.—Pandav, S.—Slobodová, A.—Taylor, C.—Frolov, V.–Reeber, E.—Naik, A.: Replacing Testing With Formal Verification in Intel r Coretm i7 Processor Execution Engine Validation. In CAV '09, 2009, pp. 414–429.

[16] Koymans, R.: Specifying Real-Time Properties With Metric Temporal Logic. Real-Time Systems, Vol. 2, 1990, No. 4, pp. 255–299.

[17] Pnueli, A.: The Temporal Logic of Programs. In Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press 1977, pp. 46–57.

[18] Vardi, M. Y.—Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. Logic in Computer Science 1986, pp. 322–331.

7 APPENDIX – TRANSLATION ALGORITHM DETAILS

In the algorithm presentation, we follow the notation given by authors in [12]. We define negation of a degradation constraint $\neg(x \bowtie d)$ as follows: $\neg(x \leq d) = x > d$, $\neg(x < d) = x \geq d$, $\neg(x \geq d) = x < d$, and $\neg(x > d) = x \leq d$.

Data structures

Id: unique name of the node,

Inc: structure coding edges of the graph, i.e., set of triples **(id,constraint,reset)**, such that each triple codes an incoming edge from the *Father* node (id) to the current node. When $id = init$, then the current node is an initial node ($init$ is not a name of a node).

Old: set of formulas, that must hold in the node and have been already processed

New: set of formulas, that must hold in the node and have not yet been processed

Next: set of formulas that must hold in all immediate successors of nodes satisfying properties in *Old*

Out: constraint and reset that must be on an outgoing edge **(constraint,reset)**

A *node* is a structure **(Id, Inc, New, Old, Next, Out)**.The set of all nodes is denoted by *Nodes*.

List of Functions

$expand(n, Nodes)$: returns a set of nodes after processing node n

$new_ID()$: returns a fresh id

$get_Var(\varphi)$: returns a set of degradation variables associated with subformula φ

$Neg(_)$: returns a set of negated formulas for a given set of formulas

Pseudo-code

```
proc create_graph(φ)
  q = (new_ID(), {(init, tt, ∅)}, {φ}, ∅, ∅, (tt, ∅))
  returnexpand(q, ∅)
end
proc expand(q, Nodes)
  if New(q) == ∅
    then if (∃r ∈ Nodes with Old(r) = Old(q) ∧ Next(r) = Next(q)∧
              Out(r) = Out(q))
              then Inc(r) = Inc(r) ∪ Inc(q)
```

```
            returnNodes
        else N = (new_ID(), {Id(q)} × {Out(q)}, Next(q), ∅, ∅, (∅, tt))
            return expand(N, Nodes ∪ {q})          /*q is a new node */
    fi
else
    switch

        case φU_{≫d}ψ ∈ New(q) and φU^x_{≫d}ψ ∈ New(q) ∪ Old(q)
        New(q) = New(q) \ {φU_{≫d}ψ}
        return expand(q, Nodes)

        case φU_{≪d}ψ ∈ New(q) and φU^{xy}_{≪d}ψ ∈ New(q) ∪ Old(q)
        New(q) = New(q) \ {φU_{≪d}ψ, φU^{xy}_{≪d}ψ}
        N1 = (new_ID(), inc1, New(q) ∪ {ψ}, Old(q) ∪ {η}, Next(q), Out(q)),
              where inc1 = {(id, c ∧ x ≪ d ∧ 1 ≪ d, r) | (id, c, r) ∈ Inc(q)}
        N2 = (new_ID(), inc2, New(q) ∪ {φ, ψ}, Old(q) ∪ {η}, Next(q)∪
              {φU^{yx}_{≪d}ψ}, Out(q)), where
              inc2 = {(id, c ∧ x ≪ d, r ∪ {y}) | (id, c, r) ∈ Inc(q)}
        N3 = (new_ID(), inc3, New(q) ∪ {φ}, Old(q) ∪ {η}, Next(q)∪
              {φU^{xy}_{≪d}ψ}, Out(q)), where
              inc3 = {(id, c, r ∪ {y}) | (id, c, r) ∈ Inc(q)}
        return expand(N1, expand(N2, expand(N3, Nodes)))

        case φR_{≪d}ψ ∈ New(q) and φR^x_{≪d}ψ ∈ New(q) ∪ Old(q)
        New(q) = New(q) \ {φR_{≪d}ψ}
        return expand(q, Nodes)

        case φR_{≫d}ψ ∈ New(q) and φR^x_{≫d}ψ ∈ New(q) ∪ Old(q)
        New(q) = New(q) \ {φR^x_{≫d}ψ}
        return expand(q, Nodes)

        otherwise
        let η be the longest formula in New(q)
        New(q) = New(q) \ {η}
        switch(η)    /* according to the shape of η*/

        case(η ∈ (AP ∪ Neg(AP) ∪ {tt, ¬tt}))
        if (η == False ∨ Neg(η) ∈ Old(q))       /*q contains a contradiction */
          then returnNodes
           else N = (new_ID(), Inc(q), New(q), Old(q) ∪ {η}, Next(q), Out(q))
                return expand(N, Nodes)
        fi

        case(η ≡ φ ∨ ψ)
```

$\mathsf{N1} = (new_ID(), Inc(q), New(q) \cup \{\varphi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$
$\mathsf{N2} = (new_ID(), Inc(q), New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$
$\mathsf{return} expand(\mathsf{N2}, expand(\mathsf{N1}, \mathsf{Nodes}))$

$\mathsf{case}(\eta \equiv \varphi \wedge \psi)$
$\mathsf{N} = (new_ID(), Inc(q), New(q) \cup \{\varphi, \psi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$
$\mathsf{return} expand(\mathsf{N}, \mathsf{Nodes})$

$\mathsf{case}(\eta \equiv \mathsf{X}_{\bowtie d}\varphi)$
$x = get_Var(\mathsf{X}_{\bowtie d}\varphi)$
$\mathsf{N} = (new_ID(), inc, New(q), Old(q) \cup \{\eta\}, Next(q) \cup \{\varphi\}, out)$, where
$inc = \{(id, c, r \cup \{x\}) \mid (id, c, r) \in Inc(q)\}$,
$out = \{(c \wedge x \bowtie d, r) \mid (c, r) \in Out(q)\}$
$\mathsf{return} expand(\mathsf{N}, \mathsf{Nodes})$

$\mathsf{case}(\eta \equiv \varphi \mathsf{U}_{\gg d}\psi)$
$x = get_Var(\varphi \mathsf{U}_{\gg d}\psi)$
$\mathsf{N1} = (new_ID(), inc1, New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$,
where $inc1 = \{(id, c \wedge 1 \gg d, r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N2} = (new_ID(), inc2, New(q) \cup \{\varphi\}, Old(q) \cup \{\eta\}, Next(q)$
$\cup \{\varphi \mathsf{U}^{x}_{\gg d}\psi\}, Out(q))$, where
$inc2 = \{(id, c, r \cup \{x\}) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{return} expand(\mathsf{N1}, expand(\mathsf{N2}, \mathsf{Nodes}))$

$\mathsf{case}(\eta \equiv \varphi \mathsf{U}^{x}_{\gg d}\psi)$
$\mathsf{N1} = (new_ID(), inc1, New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$,
where $inc1 = \{(id, c \wedge x \gg d, r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N2} = (new_ID(), Inc(q), New(q) \cup \{\varphi\}, Old(q) \cup \{\eta\}, Next(q) \cup$
$\{\varphi \mathsf{U}^{x}_{\gg d}\psi\}, Out(q))$
$\mathsf{return} expand(\mathsf{N1}, expand(\mathsf{N2}, \mathsf{Nodes}))$

$\mathsf{case}(\eta \equiv \varphi \mathsf{U}_{\ll d}\psi)$
$(x, y) = get_Var(\varphi \mathsf{U}_{\ll d}\psi)$
$\mathsf{N1} = (new_ID(), inc1, New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$,
where $inc1 = \{(id, c \wedge 1 \ll d, r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N2} = (new_ID(), inc2, New(q) \cup \{\varphi\}, Old(q) \cup \{\eta\}, Next(q) \cup$
$\{\varphi \mathsf{U}^{xy}_{\ll d}\psi\}, Out(q))$, where
$inc2 = \{(id, c, r \cup \{x, y\}) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{return} expand(\mathsf{N1}, expand(\mathsf{N2}, \mathsf{Nodes}))$

$\mathsf{case}(\eta \equiv \varphi \mathsf{U}^{xy}_{\ll d}\psi)$
$\mathsf{N1} = (new_ID(), inc1, New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$,
where $inc1 = \{(id, c \wedge x \ll d \wedge y \ll d, r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N2} = (new_ID(), inc2, New(q) \cup \{\varphi, \psi\}, Old(q) \cup \{\eta\}, Next(q) \cup$

$\{\varphi \mathsf{U}^{yx}_{\ll d}\psi\}, Out(q))$, where
$inc2 = \{(id, c \wedge x \ll d \wedge, r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N3} = (new_ID(), inc3, New(q) \cup \{\varphi\}, Old(q) \cup \{\eta\}, Next(q) \cup$
$\{\varphi \mathsf{U}^{xy}_{\ll d}\psi\}, Out(q))$, where
$inc3 = \{(id, c, r) \mid (id, c, r) \in Inc(q)\}$
return$expand(\mathsf{N1}, expand(\mathsf{N2}, expand(\mathsf{N3}, \mathsf{Nodes})))$

case$(\eta \equiv \varphi \mathsf{R}_{\bowtie d}\psi)$
$x = get_Var(\varphi \mathsf{R}_{\bowtie d}\psi)$
$\mathsf{N1} = (new_ID(), inc1, New(q) \cup \{\phi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$,
where $inc1 = \{(id, c \wedge \neg(1 \bowtie d), r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N2} = (new_ID(), Inc(q), New(q) \cup \{\varphi \wedge \psi\}, Old(q) \cup \{\eta\}, Next(q),$
$Out(q))$, where
$\mathsf{N3} = (new_ID(), inc3, New(q), Old(q) \cup \{\eta\}, Next(q) \cup \{\varphi \mathsf{R}^{x}_{\bowtie d}\psi\},$
$Out(q))$, where
$inc3 = \{(id, c \wedge \neg(1 \bowtie d), r \cup \{x\}) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N4} = (new_ID(), inc4, New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q) \cup \{\varphi \mathsf{R}^{x}_{\gg d}\psi\},$
$Out(q))$, where
$inc4 = \{(id, c, r \cup \{x\}) \mid (id, c, r) \in Inc(q)\}$
return$expand(\mathsf{N3}, expand(\mathsf{N2}, expand(\mathsf{N1}, \mathsf{Nodes})))$

case$(\eta \equiv \varphi \mathsf{R}^{x}_{\bowtie d}\psi)$
$x = get_Var(\varphi \mathsf{R}_{\bowtie d}\psi)$
$\mathsf{N1} = (new_ID(), inc1, New(q) \cup \{\phi\}, Old(q) \cup \{\eta\}, Next(q), Out(q))$,
where $inc1 = \{(id, c \wedge \neg(x \bowtie d), r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N2} = (new_ID(), Inc(q), New(q) \cup \{\varphi \wedge \psi\}, Old(q) \cup \{\eta\}, Next(q),$
$Out(q))$, where
$\mathsf{N3} = (new_ID(), inc3, New(q), Old(q) \cup \{\eta\}, Next(q) \cup \{\varphi \mathsf{R}^{x}_{\bowtie d}\psi\},$
$Out(q))$, where
$inc3 = \{(id, c \wedge \neg(x \bowtie d), r) \mid (id, c, r) \in Inc(q)\}$
$\mathsf{N4} = (new_ID(), Inc(q), New(q) \cup \{\psi\}, Old(q) \cup \{\eta\}, Next(q) \cup$
$\{\varphi \mathsf{R}^{x}_{\gg d}\psi\}, Out(q))$, where
return$expand(\mathsf{N3}, expand(\mathsf{N2}, expand(\mathsf{N1}, \mathsf{Nodes})))$
end
fi
end

Jiří Barnat is an Associate Professor at Faculty of Informatics, Masaryk University, Czech Republic. He has received the M. Sc. and Ph. D. degrees in computer science from Masaryk University, Czech Republic, in 2000 and 2005, respectively. His research interests include parallel algorithms, parallel and distributed methods in formal verification, and platform dependent algorithm engineering. Prof. Barnat is currently leading the ParaDiSe research group at the Faculty of Informatics. He is a co-author of parallel and distributed verification tool DiVinE.

Ivana Černá is a Professor at Faculty of Informatics, Masaryk University, Czech Republic. She has received the M. Sc. and Ph. D. degrees in computer science from Comenius University, Slovak Republic, in 1986 and 1992, respectively. Her research interests include theory of communicating and parallel systems, formal verification and verification tools, algorithm design and analysis. She is a co-author of algorithms implemented in a parallel and distributed verification tool DiVinE.

Jana Tůmová received the B. Sc. and M. Sc. degrees in applied computer science and computer science from Masaryk University, Brno, Czech Republic, in 2006 and 2009, respectively. She is currently a Ph. D. student in computer science at the same university. Her research interests include formal methods, temporal logics, model checking and control strategy synthesis.