Efficient algorithms and tools for MITL model-checking and synthesis (short paper)

Thomas Brihaye, *UMons*
Gilles Geeraerts, *ULB*
Hsi-Ming Ho, *Cambridge U.*
Arthur Milchior, *ULB*
Benjamin Monmege, *Aix Marseille Univ*

***Abstract*—Metric Interval Temporal Logic (MITL) is an extension of the classical Linear Time Logic (LTL) that can be used to characterise real-time properties of computer systems. While the practical interest of MITL is undeniable, there is still today a remarkable lack of tool support for this logic. In this short paper, we report on our on-going work effort to complete the theoretical knowledge about MITL. We also report on our recently introduced tool MightyL, which translates MITL formulæ into timed automata, enabling efficient model-checking of this logic. Finally, we sketch the future directions of our current line of research, which will be to extend MightyL to support reactive synthesis of MITL properties.**

I. Introduction

The development of reactive and critical *real-time* systems is known to be a difficult problem, since the safety of those systems does not only rely on their correct interaction with their environment, but also on the *precise timings* of these interactions. Two major approaches have been advocated by the formal methods community to ensure strong guarantees about the behaviour of such systems, namely: *model-checking* and *synthesis*. Roughly speaking, model-checking allows one to check whether some model of the system satisfies a given property. Synthesis extends the scope of computer-assisted design of hardware/software systems beyond the capabilities of model-checking: given a model of the environment, synthesis amounts to *computing* (if possible) a controller for the environment that enforces a given property.

In the untimed settings, theoretical and practical contributions on these two problems abound, and many tools are available to perform model-checking and synthesis when the system and the environment are specified as (Büchi) automata, and the properties are given in Linear Temporal Logic (LTL for short) [18]. See for example [3], [8] for tools that allow to perform LTL model-checking and synthesis respectively.

In the real-time setting, timed automata [1] are now established as the prime automaton model, and several tools such as Uppaal [15], Uppaal TiGa or LTSMin (with the Opaal plug-in) [13] support them. However, these tools are essentially limited to verifying and synthesising *untimed* properties (for example, Uppaal supports mainly CTL properties with some restricted TCTL queries), which limits their range of application.

The most natural real-time extension of LTL is arguably the metric temporal logic (MTL) [14]. In MTL, one can write properties such as $\Box(a \to \Diamond_{[x,y]} b)$, where $[x, y]$ is an interval, and meaning 'every a should eventually be followed by a b after at least x and at most y time units'. Unfortunately, MTL is mostly undecidable [17], a limitation which has prompted Alur *et al.* to introduce the metric interval temporal logic (MITL) as a fragment of MTL where punctual (i.e singular) intervals are disallowed [2]. They have showed that MITL satisfiability is decidable (in EXPSPACE) and that MITL formulæ can be translated into timed automata thereby enabling automata-based model-checking. MITL thus seems to be an excellent *real-time* counterpart to LTL. Unfortunately, and albeit the works of Alur *et al.* have appeared 20 years ago, tool support for MITL is still lacking.

In this short paper, we present our ongoing research effort, that aims at providing the formal methods community with (hopefully efficient) tools to perform MITL model-checking and synthesis. During the past decade, we have been working to extend the theory about MITL (we have, for instance, characterised the decidability of synthesis, and proposed new translation techniques to timed automata), and to develop tools based on these new results. The cornerstone of our contribution is our recently introduced tool MightyL, which performs efficient translation of MITL into timed automata [6], [7], thereby enabling efficient automata-based model-checking of MITL (thanks to efficient model-checking engines such as Uppaal [15] or LTSMin [13] with Opaal plug-in as back-ends). We plan to continue this line of research by extending MightyL to perform efficient *synthesis* of MITL, based on the techniques presented in [11] for LTL.

II. The Metric Interval Temporal Logic

We start by defining the syntax of MITL. Let Σ be a finite *alphabet* that models the possible actions of the system. Then, MITL formulae over Σ are generated by the grammar: $\varphi := \top \mid \bot \mid \sigma \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi \mathcal{U}_I \varphi$, where $\sigma \in \Sigma$ and I is a non-singular interval with endpoints in $\mathbb{N}_{\geq 0} \cup \{\infty\}$. As in the case of LTL, the $\mathcal{U}_I$ operator is read 'until'. So intuitively (see below for a formal definition), $p\mathcal{U}_I q$ means 'p should hold *until* q holds, and this happens after t time units, with $t \in I$'. As is usual, also in the case of LTL, we rely on useful shorthands such as the 'eventually' operator $\Diamond_I \varphi \equiv \top \mathcal{U}_I \varphi$ and the 'globally' operator $\Box_I \varphi \equiv \neg\Diamond_I \neg\varphi$. For example, $\Box(req \to (\Diamond_{(0,3)} grant))$ is an MITL formula, while $p\mathcal{U}_{[1,1]} q$ is not, because the $[1, 1]$ interval is singular.

Let us now discuss the *semantics* of MITL. In this short paper, we consider the so-called *pointwise* semantics only. That is, formulæ of MITL will be interpreted over *timed words* which are sequences of pairs of the form (action, timestamp),

where the action is an element from Σ, and the sequence of timestamps along the timed word is non-decreasing, in order to model faithfully the elapsing of time. Formally, a timed word over some alphabet Σ is an infinite sequence $\rho = (\sigma_0, \tau_0)(\sigma_1, \tau_1)\cdots$ over $\Sigma \times \mathbb{R}^+$ with $(\tau_i)_{i\in\mathbb{N}}$ a non-decreasing sequence of non-negative real numbers. We let $T\Sigma^\omega$ be the set of timed words over Σ. A *timed language* is a subset of $T\Sigma^\omega$.

We can now define the (pointwise) semantics of MITL as a timed language. We say that a timed word $\rho = (\sigma_0, \tau_0)(\sigma_1, \tau_1)\cdots$ over Σ *satisfies* φ at position i (written $\rho, i \models \varphi$) if ρ, i and φ respect one of the inductive rules:

- $\rho, i \models \top$ and $\rho, i \not\models \bot$;
- $\rho, i \models \sigma$ if $\sigma = \sigma_i$ and $\rho, i \models \neg\phi$ if $\rho, i \not\models \phi$;
- $\rho, i \models \phi_1 \wedge \phi_2$ if $\rho, i \models \phi_1$ and $\rho, i \models \phi_2$;
- $\rho, i \models \phi_1 \vee \phi_2$ if $\rho, i \models \phi_1$ or $\rho, i \models \phi_2$;
- $\rho, i \models \phi_1 \mathcal{U}_I \phi_2$ if there exists $j > i$, $\rho, j \models \phi_2$, $\tau_j - \tau_i \in I$, and, for all $i < k < j$, $\rho, k \models \phi_1$.

We write $\rho \models \varphi$ when $\rho, 0 \models \varphi$. We denote by $L(\varphi)$ the set of all timed words ρ s.t. $\rho \models \varphi$, and call $L(\varphi)$ the *language* of φ. For example, $\Box(req \rightarrow (\Diamond_{(0,3)} grant))$ is satisfied by $\rho = (req, 1)(req, 1.5)(grant, 3.6)(\emptyset, 5)(\emptyset, 6)(\emptyset, 7)\cdots$ because the (unique) *grant* event occurs in the words 2.6 and 2.1 time units after the respective *req* events.

Observe that MITL can be interpreted on other models than infinite timed words, namely: *finite* timed words or finite and infinite *signals* (in the so-called continuous semantics) [7] but we omit these alternative definitions in this short paper.

III. Model-checking MITL and the MightyL tool

The first main problem we are interested in about MITL is the well-known *model-checking* problem: model-checking allows one to *prove* in a mathematical sense that a model of a computer system is correct wrt a given specification, or to expose bugs. The family of models that we rely on in this work are known as *timed automata* (TA for short) [1]. Roughly speaking, timed automata extend classical (non-deterministic, Büchi) automata by a finite set X of real-valued variables called *clocks*. The values of those clocks increase spontaneously with time elapsing, all at the same rate. Then, discrete transitions in a TA can *constrain* the value of the clocks (by a condition called a *guard*) and *reset* some clocks.

Formally, let X be a finite set of real-valued clocks. The set $\mathcal{G}(X)$ of *clock constraints* g over X is defined by $g := \top \mid g \wedge g \mid x \bowtie c$, where $\bowtie \in \{\leq, <, \geq, >\}$, $x \in X$ and $c \in \mathbb{N}$. A *valuation* over X is a mapping $v\colon X \rightarrow \mathbb{R}^+$. We denote by $\mathbf{0}$ the valuation that maps every clock to 0. The satisfaction of a constraint g by a valuation v is defined in the usual way and noted $v \models g$. For $t \in \mathbb{R}^+$, we let $v + t$ be the valuation defined by $(v+t)(x) = v(x)+t$ for all $x \in X$. For $R \subseteq X$, we let $v[R \leftarrow 0]$ be the valuation defined by $(v[R \leftarrow 0])(x) = 0$ if $x \in R$, and $(v[R \leftarrow 0])(x) = v(x)$ otherwise.

Then, a *timed automaton* (with Büchi acceptance condition) on the alphabet Σ is a tuple $A = (L, \ell_0, T, F)$ where: (i) L is a finite set of locations; (ii) $\ell_0 \in L$ is the initial location; (iii) $T \subseteq L \times \Sigma \times \mathcal{G}(X) \times 2^X \times L$ is the transition relation, where each transition consists of the source location, the letter to be read, the clock constraint (guard) to be checked, the clocks to be reset, and the target location; (iv) $F \subseteq L$ is a set of *accepting* locations.

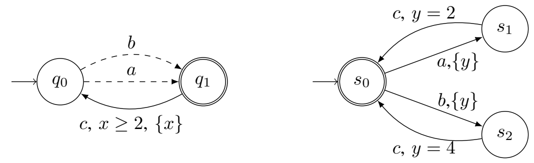

Fig. 1. Two example timed automata: a plant and a controller for this plant.

For example, Fig. 1 displays two TAs (ignore the dashed arrows for the moment). The right one has three states, where s_0 is both initial and accepting ($s_0 \in F$). The transition from s_0 to s_1 has label a, and resets clock y, while the transition from s_1 to s_2 has label c, and can be taken only when $y = 2$.

As expected, timed automata define timed languages. Formally, a *configuration* of a TA $A = (L, \ell_0, T, F)$ is a pair (ℓ, v) where $\ell \in L$ is the current location of the automaton, and v is a valuation of the clocks in X. Given a configuration (ℓ, v), two sorts of transitions can occur. First, a *continuous* transition corresponds to the elapsing of δ time units and affects the valuation of the clocks only. Formally, for all configurations (ℓ, v), for all $\delta \in R^+$, we let $(\ell, v) \xrightarrow{\delta} (\ell, v + \delta)$. Second, a *discrete* transition corresponds to a change of location of the automaton, and affects both the current location and the clock valuation (through the resets). Formally, for a configuration (ℓ, v) and a letter $\sigma \in \Sigma$, we have a discrete transition $(\ell, v) \xrightarrow{\sigma} (\ell', v')$ iff there exists a transition $t = (\ell, \sigma, g, R, \ell') \in T$ s.t. $v \models g$ and $v' = v[R \leftarrow 0]$. We write $(\ell, v) \xrightarrow{\delta,\sigma} (\ell', v')$ whenever there is v'' s.t. $(\ell, v) \xrightarrow{\delta} (\ell, v'') \xrightarrow{\sigma} (\ell', v')$.

Then, a run (of A) on a timed word $\rho = (\sigma_0, \tau_0)(\sigma_1, \tau_1)\cdots$ over Σ is an infinite sequence of transitions of the form $\pi = (\ell_0, \mathbf{0}) \xrightarrow{\delta_0,\sigma_0} (\ell_1, v_1) \xrightarrow{\delta_1,\sigma_1} (\ell_2, v_2)\cdots$ s.t. $\tau_i = \sum_{j=0}^{i} \delta_j$ for all $i \geq 0$. We consider Büchi acceptance conditions: a word ρ is accepted by a TA A iff A admits a run π on ρ s.t. at least one accepting location $\ell \in F$ occurs infinitely often along π. We denote by $L(A)$ the timed language made up of all words accepted by A. As an example, the timed language of the TA on the left of Fig. 1 accepts all timed words of the form $(\alpha_0, \tau_0)(c, \tau_0')(\alpha_1, \tau_1)(c, \tau_1')\cdots$ where each α_i is either a or b, and $\tau_i - \tau_{i-1} \geq 2$ for all $i \geq 0$ (assuming $\tau_{-1} = 0$).

Finally, we recall that, given two TAs A_1 and A_2, one can build a TA $A_1 \times A_2$, called the *synchronous product* of A_1 and A_2, which is s.t. $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$ [1].

Now that we have fixed the model of timed automata, we can define the first main problem we are interested in, namely the model-checking problem. Given a TA M (modelling some computer system) and an MITL formula φ (modelling some requirement on the actions of the system) on the same alphabet Σ, the model-checking problem asks whether all executions of M satisfy φ, i.e. whether $L(A) \subseteq L(\varphi)$.

A widely used technique (also in the untimed case) is

automata-based model-checking, which consists in expressing all inputs of the problem in terms of automata, and relying on automata-based algorithms to check for the language inclusion. In our case, this amounts to: first, computing $\neg\varphi$ (which thus represents all the *undesired* behaviours of the system); then, to *translate* $\neg\varphi$ into a timed automaton $A_{\neg\varphi}$ s.t. $L(A_{\neg\varphi}) = L(\neg\varphi)$; and finally to check whether M and $A_{\neg\varphi}$ accept a common word, i.e. whether $L(A_{\neg\varphi}) \cap L(M) = L(A_{\neg\varphi} \times M) = \emptyset$ (observe that the model-checking problem is thus reduced to a language emptiness problem for TAs). The answer to the model-checking problem is positive iff $L(A_{\neg\varphi}) \cap L(M) = \emptyset$, i.e. no execution of M violates φ.

As can be seen from this brief summary, a key ingredient to automata-based model-checking is the ability to translate MITL formulæ to equivalent TAs. Such a technique can already be found (for the continuous semantics) in the seminal paper of Alur *et al.* [2], and several other proposals have been made since then [4], [16], including from some of the authors of the present paper. We will now discuss our recently introduced tool MightyL [6], which translates efficiently MITL formulæ into equivalent timed automata[1]. To the best of our knowledge, MightyL is the first such tool that is freely available, although several translation algorithms have been proposed in the literature during the past 20 years. The main features of MightyL can be summarised as follows:

1) As explained, MightyL performs the *translation* of the MITL specification into a corresponding TA representation. An automaton equivalent to the MITL formula is output in the Uppaal XML format [15]. This allows one to use a broad range of model-checkers as the back-end to perform the actual verification, such as Uppaal (in the finite words semantics) or LTSMin [13].
2) The translation algorithm implemented in MightyL is *compositional*. Instead of building a single monolithic timed automaton corresponding to the formula, MightyL builds a network of so-called *component* timed automata, one for each sub-formula[2]. Hence, the network of TAs retains the *structure* of the original MITL formula. The actual composition of these component TAs is computed by the model-checker. This can thus be performed on-the-fly, and benefit from any optimisation that the model-checker implements. In future works, we plan to study heuristics that could be incorporated in the model-checking engine and exploit the structure of the TAs that are produced by MightyL.
3) The synchronisation between the component TAs is designed in a way to mitigate the state explosion that could occur during the composition. To achieve this, we have drawn inspiration from results on *very weak* one-clock alternating timed automata [12].
4) The total number of clocks needed by the resulting component TAs is reduced by half at most, compared to previous works [4]. This optimisation is important, as it is well-known that model-checking tools such as Uppaal are very sensitive to the number of clocks.

[1]Note that MightyL can also be used to perform MITL model-checking in the continuous semantics [7], by reducing it to a language inclusion problem in the pointwise semantics, thereby allowing to rely on the same model-checking back-end as in the pointwise case.

[2]Note that this construction subsumes the construction implemented in LTL2BA for LTL [12].

TABLE I
SOME RUNNING TIMES OF MIGHTYL ON THE SATISFIABILITY PROBLEM.

Formula	MightyL	LTSMin
$p_1\mathcal{U}_{[0,5]}(p_2\mathcal{U}_{[0,5]}(p_3\mathcal{U}_{[0,5]}(p_4\mathcal{U}_{[0,5]}(p_5\mathcal{U}_{[0,5]}p_6))))$	< 1 s.	1.12 s.
$((\Box\Diamond p_1) \to \Box(q \to \Diamond_{[0,5]} r))$	< 1 s.	1 s.
$\big((\Box\Diamond p_1 \wedge \Box\Diamond p_2) \to \Box(q \to \Diamond_{[0,5]} r)\big)$	< 1 s.	1.2 s.

MightyL has been written in OCaml and is freely available for download on the web at `http://www.ulb.ac.be/di/verif/mightyl`. The website also allows one to test MightyL online: one can submit MITL formulæ, obtain the corresponding network of TAs in the Uppaal XML format, and even run Uppaal or LTSMin online to check for satisfiability of the formulæ. We are currently working on refactoring the code of MightyL to make it more readable, and easier to extend.

Our experiments show very promising results in practice, see [6] for more details. As a benchmark, we check satisfiability of MITL formulæ. i.e. whether $L(\varphi) = L(A_\varphi) \neq \emptyset$ for a given MITL formula φ. Satisfiability is thus a good benchmark for MightyL, since it can be solved by translating the MITL formula into an equivalent TA, and then checking for language emptiness, as in the case of model-checking, our target application. We report on some results in TABLE I. In general, the model-checking times are more tractable with formulæ from the $\mathsf{MITL}_{[0,\infty]}$ fragment, that consists of formulæ where all intervals either start with 0 or end with $+\infty$.

IV. TOWARDS EFFICIENT SYNTHESIS

As already argued in the introduction, model-checking is a very powerful tool, but it offers very few help to the designer when a bug is found. While model-checking tools can produce *error traces* (i.e. executions of the model that violate the specification), they do not offer clue as how to modify the system (and its model) in order to avoid them.

A much more powerful approach has been advocated by the formal methods community in the past 20 years: synthesis. In our setting, the *reactive synthesis problem* asks, given a (deterministic) TA P called the *plant*, and an MITL formula φ (the specification), to find, if possible, a deterministic TA[3] C, called the controller, such that, when C *controls* the plant P, the whole system respects φ, i.e. $L(C \times P) \subseteq L(\varphi)$.

Of course, not any deterministic TA is admissible as a controller. A proper theoretical framework for the reactive synthesis problem in the real-time setting has been introduced by D'Souza *et al.* in [10]; let us highlight their main ideas. First, the alphabet Σ of the plant P is partitioned into *controllable* and *uncontrollable actions*, i.e. those on which the controller can and cannot act, respectively. Then, the conditions on the

[3]with some additional restrictions, see below.

controller are that: (i) it cannot reset the clocks of the plant, but it can observe them, and can have its own set of clocks that it can test and reset at will; (ii) it cannot restrict the uncontrollable moves of the plant: whenever such a move is possible in the plant, it should also be possible under the controller (i.e. in $P \times C$); and (iii) it must be non-blocking, i.e. it cannot introduce deadlocks.

For example, consider again the automata in Fig. 1. The left one is an example of plant P, where the uncontrollable actions are a and b (dashed arrow), and only c is controllable. Let us fix the MITL requirement $\Box(a \rightarrow \Diamond_{[2,3]}c) \wedge \Box(b \rightarrow \Diamond_{[4,5]}c)$. That is, the goal of the controller is to ensure the proper timing of the (controllable) c actions that follow each of the (uncontrollable) a's and b's: when an a has occurred, the c must take place after at least 2 and at most 3 time units (and similarly for the b: between 4 and 5 time units). Then, the TA on the right is a possible controller for this requirement. Note that it does not restrict the uncontrollable actions of the plant (which can still perform a's and b's at will), but controls the moment at which the (controllable) c occurs. To do so, it needs an extra clock y that is reset whenever an a or b occurs. This clock is necessary because the x clock in P does not measure the time since the last a or b, but since the last c.

Unfortunately, MITL synthesis has been shown undecidable in [9], a result recently refined in [5]. More precisely, MITL synthesis is undecidable in the general case both on finite and infinite words, and even for very restricted fragments of the logic. Decidability can be recovered, however, when we *bound a priori the resources that the controller can use*. In our setting the resources mean: (i) the number of *clocks* that the controller can use; and (ii) the maximal constant that can appear in the guards of the controller. The reader familiar with timed automata will understand that these restrictions allow to bound *a priori* the number of *regions* that any potential controller can have. We call this new problem the *bounded resources reactive synthesis* problem (BRRS for short). A 3EXPTIME solution to BRRS is readily obtained by combining the algorithm of D'Souza and Madhusudan [10] with any procedure to translate MITL formulæ into timed automata (such as the one implemented in MightyL). Unfortunately, this complexity (together with the fact that one needs to build the full region automaton of the potential controller) makes this algorithm a very poor candidate for practical implementation. The *next step in our research* will thus be to look for a more feasible solution. Our aim is to adapt the efficient techniques of Jin *et al.* [11] (for LTL synthesis), as we are about to explain now.

As in the case of LTL, our plan is to obtain an *iterative algorithm* that, given an instance (P, φ) of BRRS (assuming fixed resources), creates a finite sequence $G_1, G_2, \ldots, G_n$ of *games* played between some protagonist (that can take all the possible actions that a potential controller could chose) against the plant. Those games have the following properties: (i) if there is a winning strategy for the protagonist in some game G_i, then, this winning strategy can be readily translated into a correct controller for the specification φ; (ii) if no such controller exists, then, the protagonist has no winning strategy in any game of the sequence. Moreover, the size of those games increases along the sequence, with the first ones G_1, $G_2\ldots$ being of relatively small sizes, hence very easy to analyse using standard tools. In some sense, each G_i can be seen as an approximation of the full instance, each step in the sequence being a refinement of the approximation. In the LTL case, experimental evidence shows that, on most instances, a controller can be computed from the coarsest approximations G_1 or G_2, which yields a very efficient algorithm in practice.

Concretely, those games G_i could be: (i) either *untimed* (Büchi) games that can be solved using classical algorithms. This is the most straightforward approach, however, to obtain such untimed games we will need to rely on the region construction, which will probably yield an undesired state explosion; (ii) or *timed* games, such as the one that can be solved by Uppaal TiGa (in the finite words setting). Relying on such tools will allow us, as in the case of model-checking, to benefit from their efficient implementation (in the case of Uppaal TiGa, this means relying on its on-the-fly zone-based algorithm for solving timed games). We will investigate the relative merits of these different techniques, and plan to implement them as an extension of our tool MightyL.

References

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

[2] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[3] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.F. Raskin. Acacia+, a tool for LTL synthesis. In *CAV 12*, LNCS 7358. Springer, 2012.

[4] T. Brihaye, M. Estiévenart, and G. Geeraerts. On MITL and alternating timed automata of infinite words. In *FORMATS 14* LNCS 8711. Springer, 2014.

[5] T. Brihaye, M. Estiévenart, G. Geeraerts, H. Ho, B. Monmege, and N. Sznajder. Real-time synthesis is hard! In *FORMATS 16*, LNCS 9884. Springer, 2016.

[6] T. Brihaye, G. Geeraerts, H. Ho, and B. Monmege. Mightyl: A compositional translation from MITL to timed automata. In *CAV 17*, LNCS 10426. Springer, 2017.

[7] T. Brihaye, G. Geeraerts, H. Ho, and B. Monmege. Timed-automata-based verification of MITL over signals. In *TIME 17*, *LIPIcs* 90. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[8] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: An opensource tool for symbolic model checking. In *CAV' 02*, LNCS 2404. Springer, 2002.

[9] L. Doyen, G. Geeraerts, J.-F. Raskin, and J. Reichert. Realizability of real-time logics. In *FORMATS 09*, LNCS 5813. Springer, 2009.

[10] D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *STACS 02*, LNCS 2285. Springer, 2002.

[11] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *CAV 09*, LNCS 5643. Springer, 2009.

[12] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV 01*, LNCS 2102. Springer, 2001.

[13] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-performance language-independent model checking. In *TACAS 15*, LNCS 9035. Springer, 2015.

[14] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[15] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. J. STTT*, 1(1-2):134–152, 1997.

[16] O. Maler, D. Nickovic, and A. Pnueli. From MITL to timed automata. In *FORMATS 06*, LNCS 4202. Springer, 2006.

[17] J. Ouaknine and J. Worrell. On the decidability of metric temporal logic. In *LICS 05*. IEEE, 2005.

[18] A. Pnueli. The temporal logic of programs. In *FOCS 77*. IEEE, 1977.