Reachability in Parametric Timed Automaton with Two Parametric clocks and One Parameter

Hilaire Mathieu
Goller Stefan, LSV

August 24, 2018

Le contexte général

Il s'agit ici d'étudier un problème d'accessibilité dans les automates temporisés paramétriques. Ces automates ont commencé à être étudié afin de modéliser des processus informatiques munis d'une dimension temporelle ayant une grande importance, comme les systèmes embarqués. Le problème de l'accessibilité dans les automates temporisés à trois horloges paramétriques a été démontré comme étant indécidable, tandis que des bornes supérieures et inférieures ont été démontrée dans le cas de deux et une horloges paramétriques.

Le problème étudié

La question abordée ici est celle de la complexité du problème de l'accessibilité dans les automates temporisés paramétriques à deux horloges paramétriques et un paramètre. Le problème d'accessibilité permet de modéliser des conditions de sécurité, en démontrant par exemple qu'un état dangereux n'est pas accessible, condition importante pour qu'un système embarqué soit fiable.

Ce problème n'est pas nouveau, et des bornes de complexité existaient déjà. Nous pensions que ces bornes pouvaient être améliorées, et nous voulions également proposer des méthodes qui puissent être généralisées.

La contribution proposée

Notre principale contribution a consisté à mettre au point un langage de programmation simulable par automate temporisé, et adapté à une nouvelle preuve de borne inférieure de complexité. Nous avons également mis en place une preuve qui permette d'obtenir une borne supérieure.

Les arguments en faveur de sa validité

La technique utilisée dans la preuve de la borne inférieure a déjà démontré son utilité : celle-ci a déjà été utilisée pour des preuves de complexités sur les problèmes de vérification, d'accessibilité et de bisimulation sur des automates à compteurs. Le langage de programmation mis en place semble robuste, et nous pensons qu'il peux servir à d'autres applications. Enfin, vis à vis de la borne supérieure, le résultat dépends encore de certaines hypothèses de travail, notamment vis à vis des quantités exponentielles servant à définir ce qu'est une "grande" valeur.

Le bilan et les perspectives

Dans le cadre la borne inférieure, nous avons adapté une technique qui a déjà été utilisé pour résoudre d'autres problèmes. Notre contribution fait donc parti d'un cadre plus général. Nous pensons également que notre approche sur la borne supérieure peut être étendue au cas d'automates temporisés paramétriques avec plus d'un paramètre. Notre contribution a apporté non seulement un raffinement des bornes de complexité, mais également a démontré l'utilité de notre technique en procurant un outil de travail utile aux preuves de bornes inférieures. Pour aller plus loin, nous pourrions nous intéresser aux problèmes comportant plus d'un paramètre, mais aussi aux problèmes d'automates temporisés autres que celui de l'accessibilité: en étudiant la vérification de formules ou encore la bisimulation. La prochaine question est de savoir si la technique utilisée peut être étendue pour obtenir des résultats sur ces autres problèmes.

Internship Report

Hilaire Mathieu

August 21, 2018

Abstract

We investigate the complexity of the problem of reachability for parametric timed automaton. More specifically, we consider automaton with one parameter and only two parametric clocks, for which we design a framework for proofs of lower bound in complexity. We also study the case of the upper bound.

1 Introduction

Automaton have long been used to model computer systems and other kinds of program executions. However, for some computer systems, such as real time systems and embedded computer systems, the issue of time is an important one. Timed automaton are a successful formalism to tackle modelisation and verification of real time systems. Timed automaton are automaton extended with clocks evolving at the same rate, that can be reset, and that can be checked against clock expressions in order to enable or not certain transitions [14]. Perhaps the most important problem for timed automaton is the reachability problem: given an initial state, can the automaton perform a sequence of transitions in order to reach a certain final state?

However, timing constraints in real time systems are sometimes dependent on the environment and factors hard to predict, therefore leaving some constraints under-specified in the timed automata model allows to check properties about environments, for instance in which kind of environment the system behaves as required. Such considerations led to the study of parametric timed automaton, introduced in [1], where some number of parametric clocks can be checked against expressions containing parameters of unspecified values. Here the problem of reachability can enable to decide in which environment (i.e. for which parameters) some bad state is reachable, which is the kind of verification of safety properties we want to be able to prove with this model.

1.1 State of the art

The reachability problem for parametric timed automata is in general undecidable. Indeed it is already for only three parametric clocks with one parameter [1]. On the other end, the complexity of the reachability problem for one parametric clock and one parameter has been studied and given a lower bound NEXP-hard and an upper bound 2NEXP [5]. In this report we concern ourselves with the reachability problem in the case where the automaton has two parametric clocks (and any number of non-parametric clocks). In this case, the upper bound is only Decidability [5] and only in the case of one parameter, while the lower bound is a result of $\text{PSPACE}^{\text{NEXP}}$-hardness. Our goal in this internship was to try to improve our understanding of this problem.

1.2 Motivations

We consider the reachability problem of parametric timed automata with two parametric clocks and one parameter. The technique we use is a general framework also for showing EXPSPACE-hardness of model checking succinct one counter automata against fixed CTL formulas [2], and the reachability problem for succinct one counter automata [15] along with the bisimulation equivalence problem for succinct one counter automata [16]. The technique works due to a combination of two things : the fact that PSPACE is AC^0-serializable [17] along with the fact that we can transform a number in Chinese remainder representation into binary representation in logspace [3].

We also aimed to find a way to improve the knowledge about the upper bound of the reachability problem for one-parameter . Our approach is linked to the reduction from timed automaton to one counter automaton result [5] and therefore we will mainly study parametric bounded one-counter automaton here. The aim is to find an upper bound for parametric one counter automaton that can then provide an upper bound for parametric two clock timed automaton, both with only one parameter.

1.3 Our contribution

For the lower bound, we worked to adapt a proof from [2] that provides an EXPSPACE lower bound for CTL-model checking for succinct one counter automaton. The proof uses serializability. The idea is to look at the leafs of some computation of a complexity class C (C being PSPACE or LOGSPACE or any other complexity class) viewed as a tree. For instance, a computation tree of a normalized NP Turing machine, normalized here meaning that there exists some polynomial p such that every computation on input of length n has length of exactly $p(n)$, with a binary choice in each step of the computation tree. Then we can study the leafs of this computation tree, ordered lexicographically: there are $2^{p(n)}$ such leafs, which can be associated with 0 or 1 whether the final states are rejecting or accepting. The leafs are there associated to a word in $\{0,1\}^*$, and we can check whether this word belongs or not to some regular languages. We therefore obtain a new computation process where a word w is accepted for some NP computation and some regular language L if the leafs of the computation tree of the NP-computation on w form a word in L.

For a NP-computation and checking whether the word induced by the leafs belongs to the language 1^+, we are actually checking universal quantification: an input is accepted if all run of the machine are accepting, while the regular language $0^*1\{0+1\}^*$ corresponds to existential quantification: an input is accepted if and only if there is at least one accepting run. More complicated leaf languages can and do corresponds to higher and more complex complexity class. Exponential serialization works on a similar concept, with length of computation an exponential of some polynomial, instead of some polynomial p.

It has been shown in [4, 17] that any language in PSPACE is AC^0-serializable and any language in EXPSPACE is exponentially LOGSPACE-serializable. The idea of the proof, which is detailed in section 6, is to use the parametric timed automaton to both compute the run leading to each leafs, and to recognize the leaf language, thus providing a lower bound.

Our main contribution is the design of a suitable programing language whose halting problem we show to be polynomial time reducible to the reachability problem of parametric timed automaton with one parameter and two parametric clocks. The design of this programing language is tailored towards a possible reduction to obtain an EXPSPACE lower bound, as we will see in the Applications Section.

1.4 Definition

Definition 1.1. *A* ***transition system*** *is a tuple $T = (S, \rightarrow)$, where S is a set of* ***states*** *and $\rightarrow \subseteq S \times S$ is a* ***transition relation****.*

Definition 1.2. *An* ***automaton*** *is a tuple $\mathcal{A} = (S, s_0, \Delta, F, Op, \lambda)$ where :*

- *S is the set of states,*
- *$s_0 \in S$ is the initial state,*
- *$\Delta \subseteq S \times S$ is the set of edges,*
- *F is the set of final states, and*

- *$\lambda : \Delta \to Op$ is a function assigning a label from set Op to every edge.*

When defining a class of automata, we specify the set Op of allowed labels and their respective semantics.

Timed automaton are automaton extended with a finite set of clocks C that all progress at the same rate and that can be individually reset to zero. Moreover, every transition is labeled by an expression (called guard) that is a conjunction of comparisons $c \bowtie d$ where d is a constant, c a clock, and $\bowtie \in \{\leq, =, \geq\}$. We denote the set of guards over C by $G(C)$.

Definition 1.3. *A **timed automaton** $\mathcal{A}$ over the set of clocks C is an automaton with :*
$Op = 2^C \times G(C)$

The element in 2^C being a function from C to $\{0,1\}$ who assign 1 to $c \in C$ iff the edge of the automaton reset the clock c to zero.

Definition 1.4. *A **parametric timed automaton** $\mathcal{A}$ over the set of clocks C and parameters P is an automaton with :*
$Op = 2^C \times G(C,P)$

*Where $G(C,P)$ is the set of guards extended with $d \in \mathbb{N} \cup P$. A **parameter valuation** $\gamma : P \to \mathbb{N}$ is a function who assigns a natural value to each parameter. We denote $\mathcal{A}^\gamma$ the **timed automaton induced by** γ obtained from parametric timed automaton $\mathcal{A}$ by replacing for every $p \in P$, the instances of p in Op by the value $\gamma(p)$.*

Definition 1.5. *The **size** $|\mathcal{A}|$ of $\mathcal{A}$ is defined as $|\mathcal{A}| = |S| + |P| + \Sigma_{(s,s') \in \Delta} |\lambda(q,q')|$, where for all $\theta \in Op$ we define*

$$|\theta| = \begin{cases} \lceil \log_2(c) \rceil & \text{if } o \in \{+c, -c, \equiv 0 \mod c, \leq c, \geq c, +[0,c]\} \text{ for some } c \in \mathbb{N} \\ 1 & \text{otherwise} \end{cases}$$

That is, we present constants appearing in $\mathcal{A}$ in binary.

Definition 1.6. *A **clock-assignment u** for a given set of clocks C is a function from C to $\mathbb{N}$.*

Definition 1.7. *A **configuration** of a timed automaton $\mathcal{A} = (S, s_0, \Delta, F, Op, \lambda)$ is a pair $\langle s, u \rangle$ composed of a state $s \in S$ and a clock-assignment u of the set of clocks C of the automaton $\mathcal{A}$.*

Definition 1.8. *A **path** in an automaton $\mathcal{A}$ is a sequence of transitions such that the destination of every transition is the source of the next one; it can be written as: $\pi = e_1 e_2 \dots e_n$*

Definition 1.9. *A **run** of a timed automaton $\mathcal{A} = (S, s_0, \Delta, F, Op, \lambda)$ with initial configuration $\langle s_0, u_0 \rangle$ is a sequence of configurations $(\langle s_m, u_m \rangle)_{m \in \mathbb{N}}$, where $s_m \in S$ and u_m is a clock-assignment, satisfying the condition that, for every $m \in \mathbb{N}$, either $u_m = u_{m+1}$ and $(s_m, s_{m+1}) \in \Delta$ such that u_m satisfies the guard in $\lambda((s_m, s_{m+1}))$, or $s_m = s_{m+1}$ and the clock-assignment u_{m+1} is obtained from u_m by adding the same amount of time, in (N), to every clocks.*

This means that any timed automaton $\mathcal{A}$ induces a certain transition systems $T_\mathcal{A} = (S_\mathcal{A}, \to_\mathcal{A})$ where $S_\mathcal{A}$ consists of configurations of the timed automaton, and $\to_\mathcal{A}$ corresponds to the binary relation between two successive configurations in a run.

Definition 1.10. *A **Turing machine** is a tuple $M = \langle Q, q_0, \Gamma, b, \Sigma, \mu, F \rangle$ where : Q is a finite set of states with an initial state q_0 and a set of final states F, Γ is a finite set of tape symbols with a blank symbol b (the only symbol allowed to occur on the tape infinitely often at any step during the computation) with $\Sigma \subseteq \Gamma \setminus \{b\}$ the set of input symbols, that is, the set of symbols allowed to appear in the reading tape contents. Turing machines in our models have a reading tape and a working tape, with transitions of the form μ from $Q \times \Gamma \times \Gamma$ to $Q \times \{$move left, stay, move right$\} \times \{$move left, stay, move right$\} \times \Gamma$ with $\mu(s, read, work) = (s', move\_read, move\_work, work')$.*

Definition 1.11. *A Turing machine working in PSPACE is a Turing machine such that there exists a polynomial $p(n)$ such that for any input on the reading tape of size n, the size of the working tape of the Turing machine will not exceed $p(n)$.*

Definition 1.12. *A Turing machine working in LOGSPACE is a Turing machine such that for any input on the reading tape of size n, the size of the working tape of the Turing machine will not exceed $o(log(n))$.*

For any constant d we use the notation $\tilde{d}$ to denote any function from a set $\mathcal{D}$ to a set containing d that assign every elements of $\mathcal{D}$ to d.

For any function $f : A \to B$, $a_0 \in A$, $b_0 \in B$, we use the notation $f[a_0 \leftarrow b_0]$ to denote the function one obtains from f by assigning a_0 to b_0, formally, for all $a \in A$:

$$f[a_0 \leftarrow b_0](a) = \begin{cases} b_0 & \text{if } a = a_0 \\ f(a) & \text{otherwise} \end{cases}$$

Reachability Problem for Timed Automaton :
Input: A timed automaton $\mathcal{A} = (Q, \Delta, F, Op, \lambda)$ and state q_F.
Output: Does there exists a run of the automaton ending in a configuration whose state is q_F ?

Reachability Problem for Parametric Timed Automaton :
Input: A parametric timed automaton $\mathcal{A} = (Q, \Delta, F, Op, \lambda)$ and state q_F.
Output: Does there exist some parameter valuation $\gamma : P \to \mathbb{N}$ such that there exists a run of the induced time automaton $\mathcal{A}^\gamma$ ending in a configuration whose state is q_F ?

A **positive instance** of the Reachability Problem for timed automaton (resp. parametric timed automaton) is a timed automaton (resp. parametric timed automaton) $\mathcal{A} = (Q, \Delta, F, Op, \lambda)$ and state q_F such that the output of the reachability problem on this input consists of a positive answer.

1.5 Example

Automata $\mathcal{A}_1$:

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[r, "x > 3"] & q_1 \arrow[r, "x \leq p"] \arrow[l, bend left, "x := 0"] & q_f
\end{tikzcd}$$

For this parametric timed automaton, for any valuation γ where $\gamma(p) \geq 3 + 1$, there exists a run starting in configuration $\langle q_0, 0\rangle$ and ending in configuration $\langle q_f, \gamma(p)\rangle$, thus the reachability problem indeed have some solution. $\mathcal{A}_1$, q_f is a positive instance of the reachability problem for parametric timed automaton.

Automata $\mathcal{A}_2$:

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[r, "x > p"] & q_1 \arrow[r, "x < 1"] \arrow[l, bend left, "x := 0"] & q_f
\end{tikzcd}$$

For this second parametric timed automaton, there is no valuation $\gamma : P \to \mathbb{N}$ such that there exists a run starting in configuration $\langle q_0, 0\rangle$ and ending in configuration whose first component is q_f, because upon entering q_1, x is strictly greater than 0 at best, but need to be strictly less than 1 (without reset in between) in order to reach q_f, which is not possible as long as we deal with time in (N).

The goal of this report is to prove the following complexity Theorem about the reachability problem:

Theorem 1.13. *The Reachability problem for parametric timed automaton restricted to parametric timed automaton with one parameter and two parametric clocks is EXPSPACE-complete.*

2 Syntax

In order to explain what kind of programs the programing language should be able to execute, and thus, what kind of programs we want to be able to implement and modelize, we first introduce the syntax of our programing language.

Let us consider a set of "space variables" $\mathcal{X} =\{ X_0, X_1, \dots X_k \}$, a set of "boolean variables" $\mathcal{Y} =\{ Y_1,\dots,Y_{k'} \}$ and, finally, a variable B.

Those basic operations will be programs:

- $B := B \pm X$
- $X_i := f_1(X_j)$ where f_1 is a function $PSPACE$-computable going from $\{0,1\}^n$ to $\{0,1\}^n$.
- $X_i := f_2(B, X_j)$ where f_2 is a function $LOGSPACE$-computable going from $\{0,1\}^{2^n+n}$ to $\{0,1\}^n$.
- $Y := g_1(X)$ where g_1 is a function $PSPACE$-computable going from $\{0,1\}^n$ to $\{0,1\}$.
- $Y := g_2(B, X)$ where g_2 is a function $LOGSPACE$-computable going from $\{0,1\}^{2^n+n}$ to $\{0,1\}$.

for $X, X_i, X_j \in \mathcal{X}$ and for $Y \in \mathcal{Y}$.

We also have the following closure properties: If π_1, π_2, π are programs, then the following are programs too :

- if Y then π_1 else π_2
- π_1; π_2
- while Y do π

3 Semantics

In order to express the semantics of our programing language, we introduce the notion of n-assignment. This will enable us to describe in which ways the programs modify the variables.

Definition 3.1. *An n-assignment $\boldsymbol{\sigma}$ is a tuple (α, β, z) where*

- $\alpha : \mathcal{X} \to \{0,1\}^n$
- $\beta : \mathcal{Y} \to \{0,1\}$
- $z \in \{0,1\}^{2^n}$

We define $\boldsymbol{\sigma}_w$, where w is a word of $\{0,1\}^n$, to be the n-assignment where α is the function who assigns all variables from $\mathcal{X}$ to 0^n except for X_0 that is set to w, where β is the function who assigns all variables from $\mathcal{Y}$ to 0 and where z is 0^{2^n}.

The semantic of a program is seen as a partial function from any set of n-assignments towards the same set of n-assignments.

Rules for the semantics :

Basic operations : for all $X, X_i, X_j \in \mathcal{X}$ and for all $Y \in \mathcal{Y}$ we have :

- $[\![B := B \pm X]\!](\alpha, \beta, z) = (\alpha, \beta, z \pm \alpha(X))$

  Where the addition (resp. subtraction) of two numbers written in binary as string of $\{0,1\}^n$ is considered as bit by bit addition (resp. subtraction) modulo 2^n, and addition (resp. subtraction) of a string of $\{0,1\}^n$ to a string of $\{0,1\}^{2^n}$ is also done bit by bit but this time with regards to modulo 2^{2^n}.

- $[\![X_i := f_1(X_j)]\!](\alpha, \beta, z) = (\alpha[X_i \leftarrow f_1(\alpha(X_j))], \beta, z)$
- $[\![X_i := f_2(B, X_j)]\!](\alpha, \beta, z) = (\alpha[X_i \leftarrow f_2(z, \alpha(X_j))], \beta, z)$
- $[\![Y := g_1(X)]\!](\alpha, \beta, z) = (\alpha, \beta[Y \leftarrow g_1(\alpha(X))], z)$
- $[\![Y := g_2(B, X)]\!](\alpha, \beta, z) = (\alpha, \beta[Y \leftarrow g_2(z, \alpha(X))], z)$.

Combination of operations :

- $[\![\text{ if } Y \text{ then } \pi_1 \text{ else } \pi_2]\!](\sigma) = \begin{cases} [\![\pi_1]\!](\alpha, \beta, z) & \text{if } \beta(Y) = 1 \\ [\![\pi_2]\!](\alpha, \beta, z) & \text{otherwise} \end{cases}$
- $[\![$ while α_i do $\pi]\!](\sigma) = \rho$ iff there exists a sequence of assignments λ_1, λ_2, ... λ_t such that $\sigma = \lambda^1$, $\lambda^t = \rho$, with $\rho_A(\alpha^i) = 0$ (i.e. False) and for every $0 < j < t$ $\lambda^j [\![\pi]\!]\ \lambda^{j+1}$ and $\lambda^j_A(\alpha_i) = 1$ (i.e. True)
- $[\![\ \pi_1\ ;\ \pi_2]\!](\sigma) = [\![\pi_2]\!]([\![\pi_1]\!](\sigma))$

4 Interface

Let us consider **interpretations** of an automaton $\mathcal{A}$ that, given a state s_m and a clock-assignment u_m for the clocks C of an automaton, provides an n-assignment σ. Our interpretation ν will be detailed during the implementation section (see section 5.1).

For every program π we consider the following problem :

Halt$_\pi$ Problem :

Input: A word $w \in \{0,1\}^n$

Output: Is $[\![\pi]\!](w)$ defined ? (in other words does the program π holds on input w ?)

This problem is a halting problem because the semantic of the program is defined if and only if the "while" loops are finite, i.e. if and only if the program terminates. It is important to note that, once the semantic of the program on a word is defined, we can use this semantic as some sort of result sent by the program.

The other problem we consider is the reachability problem for parametric timed automaton (as seen in section 1) with one parameter and two parametric clocks. Our aim is to link the complexity of those problems, and to express how reachability on timed automaton enable us to simulate our programing language. For every fixed π there exists a polynomial-time algorithm that, given an n-assignment $\boldsymbol{\sigma} = (\alpha, \beta, z)$ finite (i.e. such that both α and β have a finite domain), output a parametric timed automaton $\mathcal{A} = A_n(\pi, \sigma)$ whose reachability problem is equivalent to the halting problem. The way the automaton will work will simulate the program, and in the end will mimic its semantic.

Theorem 4.1. *For every program π the following is computable in polynomial time :*
INPUT: an n-assignment σ for the program π.
OUTPUT: a parametric timed automaton with one parameter and two parametric clocks $A_n(\pi,\sigma)$, and state s_f such that the following two statements are equivalent:

- *$A_n(\pi,\sigma)$, s_f is a positive instance of the reachability problem for parametric timed automaton.*
- *the program π terminates on w.*

Based on this, we can reduce the halt$_\pi$ problem to the reachability problem for parametric timed automaton. From an instance w of the halt$_\pi$ problem, we can build an automaton $A_{|w|}(\pi,\sigma_w)$, instance of the reachability problem for parametric timed automaton, such that the execution of π on w halt if and only if there is a run of the automaton which ends by a state $s_f \in F$.

5 Implementation

This section consists of the proof of Theorem 4.1. Actually what we are doing is a little bit stronger as we not only simulate the halting problem of π, but we will at each step modelise it exact semantic. We are going to explain how, from a program π and a word w of size n, we are going to build a two parametric-clocks timed-automaton implementing the program on w. First, we explain how we are going to modelize the storage of the variables.

5.1 Implementation of the memory

The variable B, that is going to be able to store big numbers, is going to be stored as the difference between the two parametric clocks b_+ and b_-. The two clocks will count modulo the one parameter p, and we will show how to enforce that the parameter p is 2^{2^n}. This way, we can store B inside $b_+ - b_-$ that can reach double exponential value, but not higher. What we mean here, and later, by *counting modulo some c*, is that, for each state of the automaton, there is an edge from it towards itself which check that the clock is equal to the modulo value and resets it. We also add to every other transitions, not there to enforce modulo reset, guards guaranteeing that the clocks are all strictly less than their associated modulo. This way we are sure that the value contained in a clock is in the domain we want, and that its value is counted in regards to the right modulo.

The boolean variables $\mathcal{Y}$ ={ Y_1,...,$Y_{k'}$ } will each be stored as the difference between two clocks $(Y_i)_+$ and $(Y_i)_-$. When the two clocks are different, this is interpreted as the corresponding boolean variable having value 0. Respectively, when the two clocks have the same value, this is interpreted as the corresponding boolean variable having value 1. Those clocks will also need to count with modulo, and will count modulo 2.

The "space" variables $\mathcal{X}$ ={ $X_0, X_1,$ ... X_k } will be stored, bit by bit, similarly than the boolean variables. The first bit of X_i will be stored as the difference between two clocks, and so will the $j-th$ bit of X_i, for every j between 1 and n. This mean that, for each "space" variable, we will have $2n$ non-parametric clocks. The number of clocks here stays polynomial, and we remark that it wouldn't if we tried to do the same thing to store B.

Accessing bits of X_i, or Y_i, is then done by a guard that checks the value of the two corresponding non-parametric clocks: if $(Y_i)_+ = 1$ and $(Y_i)_- = 0$ or $(Y_i)_+ = 0$ and $(Y_i)_- = 1$, then the value is 0. If $(Y_i)_+ = 1$ and $(Y_i)_+ = 1$ or $(Y_i)_+ = 0$ and $(Y_i)_- = 0$ then the value is 1. Modifying bits of X_i, or Y_i, depends : in order to change it to 1, we reset both clocks at the same time, and in order to change it to 0, we reset one clocks when the other is at value 1.

We note that there are other possibles ways to modelize the memory (for instance, we could store each "space" variable as the difference between two clocks that count modulo 2^n, the same way as with B) on which we could also build our implementation (we then need proper gadgets to access the $j-th$ bit of X_i, but adding ± 1 becomes simpler), but we choose to focus on this particular implementation of the memory.

Our interpretation of the clock assignment in terms of n-assignment follow from the construction of this model : given a clock assignment u_m, our interpretation look at the differences between the clocks $(X_j)_+$ and $(X_j)_-$, Y_+ and Y_-, and b_+ and b_-, and conclude from this the values the n assignment is supposed to grant to X, Y and B.

Definition 5.1. *For any clock assignment u_m of our automaton, we define our **interpretation** $\nu_{\mathcal{X},\mathcal{Y},B}(u_m)$ to be the n-assignment (α, β, z) such that:*

- $\alpha(X) = \Sigma_{0 \leq i \leq n-1}((u_m((X_i)_+) - u_m((X_i)_-) \ modulo\ 2) * 2^i)$
- $\beta(Y) = u_m(Y_+) - u_m(Y_-)\ modulo\ 2$
- $z = u_m(b_+) - u_m(b_-)\ modulo\ 2^{2^n}$

Now that we defined our implementation of the memory, we are going to explain how we are going to implement the basic operations :

The first basic operation is $B := B \pm X$. What we want to do is add (resp. remove) the value stored in X from the one stored in B.

Lemma 5.2. *The following is computable in polynomial time:*
INPUT: An n-assignment (α, β, z).
OUTPUT: A parametric timed automaton $P_n(\sigma)$ where there exists a valuation $\gamma : P \to \mathbb{N}$ for which we can reach a configuration $\langle s_f \in F, u_f \rangle$ in $P_n(\sigma)^\gamma$ if and only if $\nu_{\mathcal{X},\mathcal{Y},B}\ (\langle s_f, u_f \rangle) = (\alpha, \beta, z \pm \alpha(X))$.

Where ν is the interpretation of the clock assignments as seen in the implementation of the storage of the variables.

Proof: The different variables used by the program, X and B, are going to be stored as seen in the implementation of the memory (see section 5.1). The automaton's clocks here are the clocks needed to store the variables, plus some additional clock w. We start by noting that, because B is stored as $b_+ - b_-$, if we reset b_+ when it is equal to some value v, then in regards to the stored variables, we removed v from B. Respectively, if we reset b_- when it is equal to some value v, we add v to B. In order to add (resp. remove) the value stored in X from B, we will wait until b_- (resp. b_+) is equal to zero (has been reset due to counting modulo p). Then we will wait a time equal to the value stored in X so that b_- (resp. b_+) is equal to the value stored in X.

To do that we use a gadget (See Fig.1) in which, for every $i-th$ bit of X, we check if the bit is equal to 1, at the same time check with additional clock w that we waited time 2^i, and, in parallel, check if the bit is equal to 0 and at the same time check with additional clock w that we waited time 0. In the end of the gadget, we waited time X, and thus the value of b_- (resp. b_+) is equal to the value of X, and resetting this clock performs the required operation.

We remark that the implementation is consistent with the corresponding semantic: $[\![B := B \pm X]\!](\alpha, \beta, z) = (\alpha, \beta, z \pm \alpha(X))$

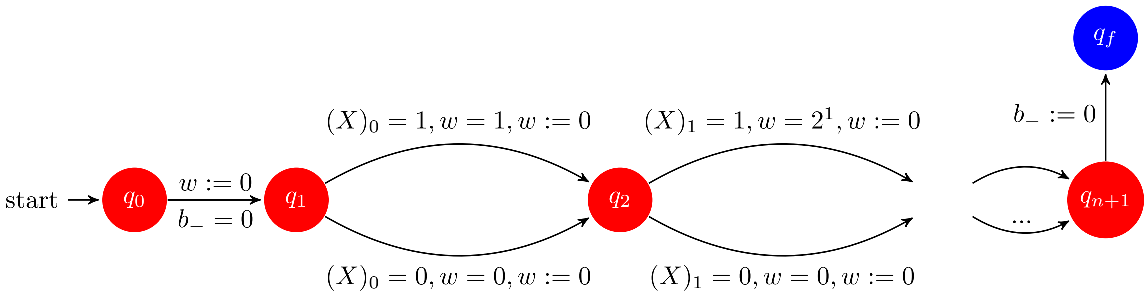
start
q_0
$w := 0$
$b_- = 0$
q_1
$(X)_0 = 1, w = 1, w := 0$
$(X)_0 = 0, w = 0, w := 0$
q_2
$(X)_1 = 1, w = 2^1, w := 0$
$(X)_1 = 0, w = 0, w := 0$
...
q_{n+1}
$b_- := 0$
q_f

Fig. 1 : Timed automaton waiting some time stored in X. The bits $(X)_{0\leq i\leq n-1}$ represent $X = \Sigma_i(X)_i 2^i$ in binary. An auxiliary clock w counts to the powers of 2. The gadget is entered on the left. One traversal through the gadget takes X time units. The case of the subtraction is obtained by replacing the reset of b_- by the reset of b_+.

Secondly, we give implementation for those two basic operations $X_i := f_1(X_j)$ and $Y := g_1(X)$.

Lemma 5.3. *The following is computable in polynomial time:*

INPUT: an n-assignment (α, β, z) and function f_1 (resp. g_1) from $\{0,1\}^n$ to $\{0,1\}^n$ (resp. $\{0,1\}$), that can be computed with a Turing machine working in PSPACE.

OUTPUT: a parametric timed automaton $PF_n(f_1, \sigma)$ (resp. $PG_n(g_1, \sigma)$) where there exists a valuation $\gamma : P \to \mathbb{N}$ for which we can reach a configuration $\langle s_f \in F, u_f\rangle$ in $PF_n(f_1, \sigma)^\gamma$ (resp. $PG_n(g_1, \sigma)^\gamma$) if and only if $\nu_{\mathcal{X},\mathcal{Y},B}(\langle s_f, u_f\rangle) = (\alpha[X_i \leftarrow f_1(\alpha(X_j))], \beta, z)$ (resp. $\nu_{\mathcal{X},\mathcal{Y},B}(\langle s_f, u_f\rangle) = (\alpha, \beta[Y \leftarrow g_1(\alpha(X))], z)$).

where ν is the interpretation of the clock assignments as seen in the implementation of the storage of the variables.

Proof: For every such operation in the program, we consider a deterministic Turing machine working in PSPACE and a polynomial $p(n)$ that computes the function f_1 (resp. g_1) while having a working tape of size less than $p(n)$, and whose working tape at the end of computation is of size n (resp. of any size less than $p(n)$). We will attribute a "ghost" variable for the function, X_{f_1} (resp. X_{g_1}), stored like a "space" variable (but of size $p(n)$), that will serve to store the working tape for the function. The automaton will also have clocks to store X_j (resp. Y) and X_i (resp. X). Now we consider the case $X_i := f_1(X_j)$, as the other scenario is implemented in a similar way.

As in [2], we want to simulate in some timed automata the behavior of the Turing machine. To do that, we simulate each step of the Turing machine from a state q, reading position r, writing position l, read letter a and written letter b by some steps in the automaton.

To do that, we build an automaton where states are tuples of $S \times \{0, ..., n-1\} \times \{0, ..., n-1\} \times \{0, 1, \$, \rhd, \lhd\} \times \{0, 1, \rhd, \lhd\}$, and where, for every transition $\mu(s, a, b) = (s', \delta_1, \delta_2, d)$ in the Turing Machine, we have in the automaton the gadget seen in Fig. 2.

We write each transition of the Turing machine $\mu(s, a, b) = (s', \delta_1, \delta_2, d)$ with δ_1 and δ_2 interpreted as shifts: -1 if equal to *move left*, $+0$ if equal to *stay* and $+1$ if equal to *move right*. For every such transition, we allow transition from state (s, i, h, a, b) to a state $(s', i + \delta_1, h + \delta_2, a', b')$ for any a', b', as those are not specified in the transition of the Turing machine. The guards on those transitions will have two purpose : first, to check whether a' and b' are indeed the values of the $i + \delta_1$-th (resp. $h + \delta_2$-th) bits of the reading (resp. working) tape, which can be done just by accessing the bits as seen in the implementation of the memory. The second purpose will be to rewrite the h-th bit of the working tape from b to d, an operation also addressed in the implementation of the memory. Fig. 2 shows a gadget in order to perform this transition.

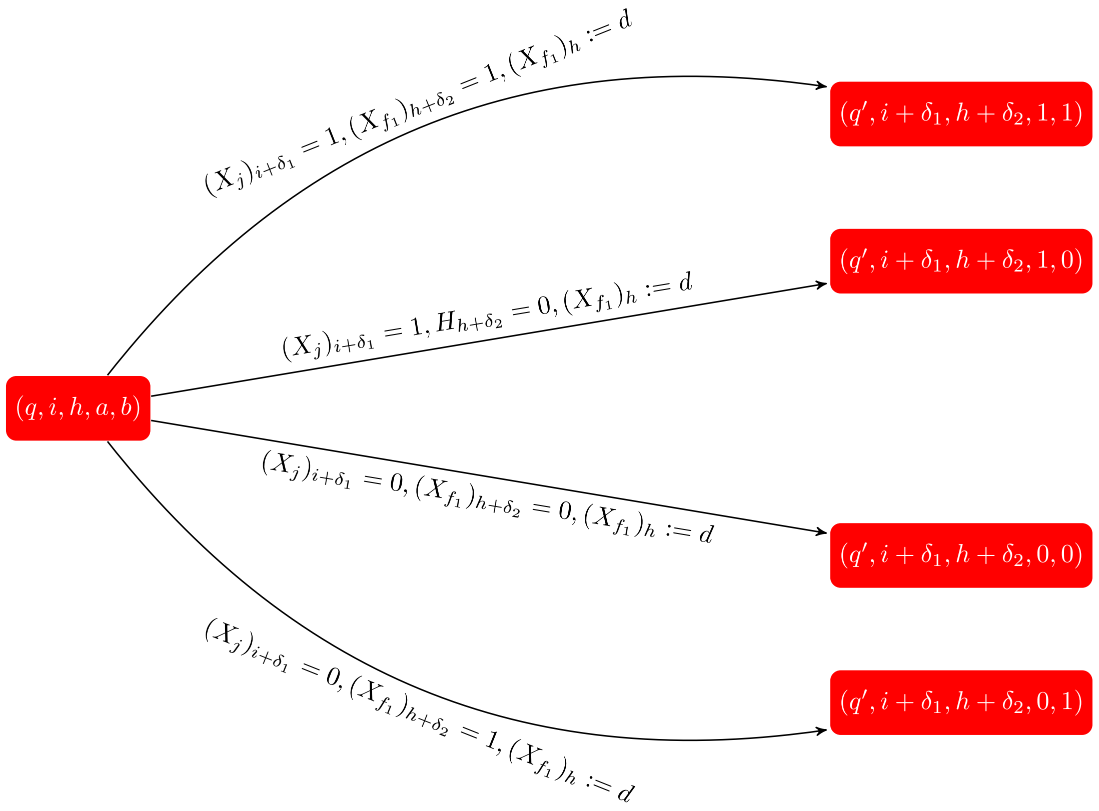
(q, i, h, a, b)
$(q', i+\delta_1, h+\delta_2, 1, 1)$
$(q', i+\delta_1, h+\delta_2, 1, 0)$
$(q', i+\delta_1, h+\delta_2, 0, 0)$
$(q', i+\delta_1, h+\delta_2, 0, 1)$
$(X_j)_{i+\delta_1} = 1, (X_{f_1})_{h+\delta_2} = 1, (X_{f_1})_h := d$
$(X_j)_{i+\delta_1} = 1, H_{h+\delta_2} = 0, (X_{f_1})_h := d$
$(X_j)_{i+\delta_1} = 0, (X_{f_1})_{h+\delta_2} = 0, (X_{f_1})_h := d$
$(X_j)_{i+\delta_1} = 0, (X_{f_1})_{h+\delta_2} = 1, (X_{f_1})_h := d$

Fig. 2. For performing a transition of the Turing machine from state q, reading tape a and working tape b, to state q', rewriting d in the working tape X_{f_1}. We move the reading and working tape and check their contents in order to chose the next state, then rewrite the bit on the working tape from b to d.

We use the gadget for each transition to simulate the way the machine operates on its working tape. Then from the states (q, i, h, a, b) of the automaton where q is a final state of the Turing machine, we leave the simulation and rewrite X_i such that it becomes the final value of the working tape X_{f_1} (we only rewrite X_i into the n first bits of X_{f_1} as the result of f_1 is of size n). Respectively in the case of the operation $Y := g_1(X)$ we just modify the boolean according to whether the final state of the Turing machine is accepting or not.

Again this is consistent with the semantics :
$[\![X_i := f_1(X_j)]\!](\alpha, \beta, z) = (\alpha[X_i \leftarrow f_1(\alpha(X_j))], \beta, z)$
$[\![Y := g_1(X)]\!](\alpha, \beta, z) = (\alpha, \beta[Y \leftarrow g_1(\alpha(X))], z)$

Third, we need to explain how deal with the two remaining basic operations.

Lemma 5.4. *The following is computable in polynomial time:*
INPUT: an n-assignment (α, β, z) and function f_2 (resp. g_2) from $\{0,1\}^{2^n+n}$ to $\{0,1\}^n$ (resp. $\{0,1\}$), that can be computed with a Turing machine working in LOGSPACE.
OUTPUT: a parametric timed automaton $EXF_n(f_2, \sigma)$ (resp. $EXG_n(g_2, \sigma)$) where there exists a valuation $\gamma : P \to \mathbb{N}$ for which we can reach a configuration $\langle s_f \in F, u_f \rangle$ in $EXF_n(f_2, \sigma)^\gamma$ (resp. $EXG_n(g_2, \sigma)^\gamma$) if and only if $\nu_{\mathcal{X},\mathcal{Y},B}(\langle s_f, u_f \rangle) = (\alpha[X_i \leftarrow f_2(z, \alpha(X_j))], \beta, z)$
(resp. $\nu_{\mathcal{X},\mathcal{Y},B}(\langle s_f, u_f \rangle) = (\alpha, \beta[Y \leftarrow g_2(z, \alpha(X))], z)$).

where ν is the interpretation of the clock assignments as seen in the implementation of the storage of the variables.

Proof: First, we need to introduce some additional notion:

Definition 5.5. ***Chinese remainder representation****: For every m, $M \in \mathbb{N}$ we denote by $CRR_m(M)$ the Chinese Remainder Representation of M as the boolean tuple $(b_{i,c})_{i\in[1,m],0\leq c<p_i}$, where $b_{i,c} = 1$ if M mod $p_i = c$ and 0 otherwise, and p_i denotes the i-th prime number.*

The following theorem tells us that in logarithmic space we can compute the binary representation of a natural number that is given in Chinese remainder representation.

Theorem 5.6. *([3] Theorem 3.3) The following problem is in LOGSPACE:*
INPUT: $CRR_m(M)$, $j \in [1, m]$, $b \in \{0, 1\}$
QUESTION: $bit_j(M \ mod\ 2^m) = b$?

A corollary to this theorem is that, if there exists a machine working on the binary representation of its input in LOGSPACE, there exists one working on the Chinese remainder representation of said input in LOGSPACE with the same result. It is due to the fact that the composition of two LOGSPACE computation can be done by a LOGSPACE computation.

We are going to consider the deterministic Turing machine working with the Chinese remainder representation on the reading tape. We are going to operate in a way similar to how we performs computations in PSPACE, except now we can no longer store the reading tape in some X_j the same way. Indeed, the reading tape is here of exponential size, and we cannot access the $i-th$ bit as easily as before. The way we are going to solve this problem is by storing only the Chinese remainder we are currently reading in the Chinese remainder representation. This one can be stored in polynomial space by some "ghost" variable X_{crr} along with some other "ghost" variable X_{index} storing its position in the Chinese remainder representation of B. When we need to read another Chinese remainder, we are going to compute it on the fly and store it in X_{crr}. First we store the X_{index}-th prime in X_{crr} (by using a PSPACE computation *ith_prime* it corresponds to performing $X_{crr} := ith\_prime(X_{index})$). Then, similarly as in Fig. 1. we wait some precise amount of time X_{crr}, which enable us to check divisibility: by non-deterministically waiting N amounts of time X_{crr} and comparing with the value of B, we can obtain the residual class of B modulo X_{crr}, then replace the X_{index}-th prime in X_{crr} by the residual class.

What remains of the execution, that is, times when we are not exceeding the boundaries of the reading positions of some particular remainder, we are performing the same way as when doing a PSPACE computation (see Fig. 2). We use the same gadgets to simulate the way the machine operates on its working tape stored in a variable X_{f_2} or X_{g_2}. Then from the states (q, i, h, a, b) of the automaton where q is a final state of the Turing machine, we leave the simulation and rewrite X_i such that it becomes the final value of the working tape X_{f_2} (of size n as seen in the conditions of the functions) OR just modify the boolean Y according to whether the final state of the Turing machine is accepting or not, depending on whether we are in the case of computing f_2 or g_2. Again this is consistent with the semantics.

Lemma 5.7. *Given an n-assignment (α, β, z) three programs π, π_1 and π_2, and a boolean variable Y, we can build in polynomial time the timed automaton $IF_n(Y, \pi_1, \pi_2, \sigma)$, $WHILE_n(Y, \pi, \sigma)$ and $THEN_n(\pi_1, \pi_2, \sigma)$ where there exists a valuation $\gamma : P \to \mathbb{N}$ for which we can reach a final state $s_f \in F$ with a clock assignment u_f if and only if :*

in IF^γ : $\nu_{\mathcal{X},\mathcal{Y}\cup Y,B}(\langle s_f, u_f\rangle) = \begin{cases} [\![\pi_1]\!](\alpha, \beta, z) & \text{if } \beta(Y) = 1 \\ [\![\pi_2]\!](\alpha, \beta, z) & \text{otherwise} \end{cases}$

in $WHILE^\gamma$: $\nu_{\mathcal{X}_\pi,\mathcal{Y}_\pi,B}(\langle s_f, u_f\rangle) = \rho$ iff there exists a sequence of assignments λ_1, λ_2, ... λ_t such that $\sigma = \lambda^1$, $\lambda^t = \rho$, with $\rho_A(\alpha^i) = 0$ and for every $0 < j < t$, $[\![\pi]\!](\lambda^j) = \lambda^{j+1}$ and $\lambda^j_A(\alpha_i) = 1$

in $THEN^\gamma$: $\nu_{\mathcal{X},\mathcal{Y},B}(\langle s_f, u_f\rangle) = [\![\pi_2]\!]([\![\pi_1]\!](\sigma))$.

where $\nu_{\mathcal{X},\mathcal{Y},B}$ is the interpretation of the clock assignments as seen in the implementation of the storage of the variables, with $\mathcal{X},\mathcal{Y}, andB$ being the unions of the corresponding set of variables for π_1 and π_2.

Proof: *if Y then π_1 else π_2*: $IF_n(Y,\pi_1,\pi_2,\sigma)$ can be done by building the timed automatons A_1 (resp. A_2) performing π_1 (resp. π_2) and having an initial state q_0 where we check the value of Y with guards, with a transition with a guard corresponding to $Y = 1$ going to the initial state of A_1 and a transition with a guard corresponding to $Y = 0$ going to the initial state of A_2, with final states being the final states of both A_1 and A_2. The clocks of this automaton being the union of the clocks of A_1 and A_2.

π_1; π_2: $THEN_n(\pi_1,\pi_2,\sigma)$ can be done by building the timed automatons A_1 (resp. A_2) performing π_1 (resp. π_2) and for every final state of A_1 have a transition leading to the initial state of A_2, with initial state being the initial state of A_1 and final states being the final states of A_2. The clocks of this automaton being the union of the clocks of A_1 and A_2.

while Y do π: $WHILE_n(Y,\pi,\sigma)$ and can be done by building the timed automatons A performing π and having an initial state q_0 where we check the value of Y with guards, with a transition with a guard corresponding to $Y = 1$ going to the initial state of A and a transition with a guard corresponding to $Y = 0$ going to a new (unique) final state, while also adding transition for every final states of A going to q_0.

Now in order to prove that p is larger than 2^{2^n} (and thus that we can actually store the value B) we performs several divisibility tests in a pre-computation. They function similarly then the implementation of the first basic operation (see Fig. 1) : by waiting a nondeterministically number of time the exact value M for one of the parametric clocks starting at 0, then checking whether it is equal to p, we actually are able to advance further in the automaton if and only if M divides p. Based on this, along with the fact that all residues in 2^{2^n}'s Chinese remainder representation can be computed on the fly with PSPACE-computations, we know that it is possible as a pre-computation to check whether p is equal or not to 2^{2^n}.

The combinations of lemmas 5.2, 5.3, 5.4 and 5.7 leads to the proof of Theorem 4.1. by structural induction on the syntax of our programing language.

6 Application of the programming language

Here we are going to show some application of the simulation of this programing language by parametric timed automaton with one parameter and two parametric clocks. We use it to provide a proof of the lower bound adapted from paper [2]. This is the first part of the proof of Theorem 1.13, namely, the lower bound consisting of EXPSPACE-hardness for the reachability problem for parametric timed automaton with one parameter and two parametric clocks. Before going into more details about this proof, we need to introduce formally the notion of serializability.

For a language $L \subseteq \Sigma^*$ let $\chi_L : \Sigma^* \to \{0,1\}$ denote the **characteristic function of L**, i.e. $\chi_L(x) = 1$ if $x \in L$ and $\chi_L(x) = 0$ otherwise, for each $x \in \Sigma^*$.

Definition 6.1. *Let C be a complexity class. We say a language L is **C-serializable** via some language $R \subseteq \{0,1\}^*$ if there is some polynomial p(n), and some language $U \in C$ such that for all $x \in \{0,1\}^n$:*

$$x \in L \iff \chi_U(x, 0^{p(n)}) \dots \chi_U(x, 1^{p(n)}) \in R$$

where '$\dots$' refer to $\leq_{p(n)}$ in each of the second components

Definition 6.2. *Let C be a complexity class. A language of words over an alphabet Σ is said to be in C if the following problem is in C :*

INPUT: a word w in Σ^*

QUESTION: does w belongs to L ?

Theorem 6.3. *(theorem 22 in [4])*

For every L in PSPACE there is some regular language R such that L is logspace-uniformly AC^0-serializable via R.

Corollary: For every L in PSPACE there is some regular language R such that L is LOGSPACE-serializable via R.

Definition 6.4. *Let C be a complexity class. We say a language L is* ***exponentially C-serializable*** *via some language $R \subseteq \{0,1\}^*$ if there is some polynomial p(n), and some language $U \in C$ such that for all $x \in \{0,1\}^n$:*

$$x \in L \iff \chi_U(x, 0^{2^{p(n)}}) \dots \chi_U(x, 1^{2^{p(n)}}) \in R$$

where '... ' refer to $\leq_{p(n)}$ in each of the second components

Theorem 6.5. *For every language L in EXPSPACE there is some regular language R such that L is exponentially LOGSPACE-serializable via R.*

For a language in EXPSPACE, let $A = (Q, \{0,1\}, q_0, \delta, F)$ be some finite deterministic automaton such that $L(A) = R$. Then, to decide whether some x_0 is in the language, we can apply the following program:

```
$q \in Q; q := q_0;$
$B \in \mathbb{N}; B := 0;$
$b \in \{0,1\};$
while $B \neq 2^{2^n}$ loop
$b := \chi_U(x_0, bin_{2^N}(B));$
$q := \delta(q, b);$
$B := B + 1;$
endloop
return $q \in F;$
```

This program can be done by the programing language as $b := \chi_U(x_0, bin_{2^N}(B))$ and $B \neq 2^{2^n}$ are LOG operations on exponential size variables, $q := \delta(q, b)$; is PSPACE-computable and $B := B + 1$ belongs in one of the basics operations.

Let us consider then the following program :
X_0 stores x_0
X_q stores q, initially 0
Y_b stores the value $b \in \{0,1\}$;, initially 0
Y_B stores the value $B \neq 2^{2^n}$, initially 0
X_1 stores the value 1, initially 0
B stores the value $B \in \mathbb{N}$;, initially 0

```
$X_q := q$ ; $X_1 := bin_n(1)$ ;
$Y_B := B \neq 2^{2^n}$;
while $Y_B$ do
$Y_b := \chi_U(x_0, bin_{2^n}(B))$ ;
if $Y_b$ then $X_q := \delta(X_q, 1)$ else $X_q := \delta(X_q, 0)$ ;
$B := B + X_1$ ; $Y_B := B \neq 2^{2^n}$ ;
endloop ;
```

This program recognize some language in EXPSPACE and its execution can be simulated via a parametric timed automaton with one parameter and two parametric clocks as expressed in Theorem 4.1. Therefore we have proven an EXPSPACE lower bound in complexity for the Reachability Problem for Parametric Timed Automaton with one parameter and two parametric clocks, which proves the lower bound for Theorem 1.13.

7 Finding an Upper Bound

In this section we concern ourselves with the search for an upper bound, in order to prove wholly the Theorem 1.13. We consider parametric bounded one counter automaton $\mathcal{C}$ obtained from a timed automaton $\mathcal{A}$ as seen in [5] (a similar construction can be found in [12]) such that the reachability problem for both automaton is equivalent. We want to show an upper bound for the reachability problem for those kinds of automaton, in a way that would enable us to deduce an upper bound for the reachability problem for parametric timed automaton with one parameter and two parametric clocks.

Our idea is to try to show that if there exists a path in $\mathcal{C}$ for a value $\gamma(p)$ of the parameter p, then there exists a path for a "small enough" value of p. With a number of "small" values of p less than some exponential, it means that we only need to compute the reachability problem for one counter automaton for every valuation with "small enough" p, which can then be done in PSPACE in regards to the size of automaton $\mathcal{C}$.

In order to do that, we want to show that if $\gamma(p)$ is sufficiently large, that is, larger than some exponential in the constants present in the automaton, we can change the value of the parameter by a new value $\gamma(p) - \Delta$ and still obtain a path in the automaton $\mathcal{C}$. What we are going to do now is look at a path π in $\mathcal{C}$ with value of the parameter $\gamma(p)$, give a value for Δ, and see if we can adapt the path into a path that is valid for the new value $\gamma(p) - \Delta$ of the parameter. This process can then lead to an upper bound proof, and complete the theorem 1.13.

In the following section we have $\Delta = LCM(\{b | b \in \text{"congruent mod b" transitions}\} \cup \{1, 2, ..., |Q|\})$.

Definition 7.1. *A **0/1 parametric bounded one counter automaton** is an automaton with:*
$OP = \{\pm 1, \pm p, = p, \geq p, \leq p, = 0 \ mod\ c, +[0, p] : c \in \mathbb{N}, p \in P\}$
where P is the set of parameters.

7.1 Bellow p

We look at a time before the counter reach the value of p for the first time.

Thus, in the interval we consider, there is no $\pm p$ transition, and $+v \in [0, p]$ transitions are not of value $\gamma(p)$. Also, all tests in the path is either a modulo test or a $\leq p$ test (due to the construction of C).

We replace the value of the parameter by $\gamma(p) - \Delta$. Some transitions $+[0, p]$ can now no longer be valid because they can reach higher than $\gamma(p) - \Delta$ and be followed by a test that thus would become illegal.

Definition 7.2. *A Drop is:*
$D(i) = min_{j>i}(value\_of\_the\_counter\_at\_step\_i - value\_of\_the\_counter\_at\_step\_j)$

First step : we "massage" the path.

Let us assume at some point we have a $+v \in [0, p]$ transition in the path, going from step $i - 1$ to step i, then if $v > (Q + 1) * \Delta$ and $D(i) > (Q + 1) * \Delta$, we will do the following :

- find k such that, in the region of the path after i, there is a state q that repeat itself between some step j and some step $j + k * \Delta$ (holds due to the pigeonhole principle applied on configurations)
- remove this portion/cycle from the path
- replace the value of the transition by $v - k * \Delta$

We keep doing this until there is no more transition $+[0, p]$ at step $i - 1$ with both its value and its drop $D(i)$ greater than $(Q + 1) * \Delta$.

Second Step : if there is a big $+[0, p]$ transition (thus with a 'small' drop)

Now there is no more transition $+[0, p]$ with both its value and its drop greater than $(Q+1)*\Delta$. Which mean that if there is a big $+[0, p]$ transition from step $i-1$ to step i, its drop $D(i)$ is 'small'.

For $+v \in [0, p]$ with $v > (Q + 2) * \Delta$ The difference between v and the drop is greater than Δ, so, removing Δ from v, the transitions stays valid because we do not reach bellow zero.

Third Step : if there is no big $+v \in [0, p]$ transition,

If there is no $+v \in [0, p]$ with $v > (Q + 2) * \Delta$, but there is still many $+v \in [0, p]$ transitions that in the end makes the counter reach higher than $\gamma(p) - \Delta$, we are going to create one big transition and then go back to the second step.

In the sequence of $+v \in [0, p]$ transitions, we have more than n such transitions because $\gamma(p) >> n * (Q + 2) * \Delta > n * v$, as assumed at the beginning when we assumed that $\gamma(p)$ was "big". This mean that one state q appears more than once in the sequence :

$q - [+v] \to q' \to .... \to q - [+v'] \to q'' \to ...$

with v 'small' and v' 'small', i.e. smaller than $(Q + 2) * \Delta$

We replace the path with :

$q - [+v + ... + v'] \to q'' \to ...$

now we have a big transition, and we go back to the Second step.

Fourth Step : just looking at ± 1 transitions

Now we are assured that the $+v \in [0, p]$ transitions will not enable the counter to reach higher than $\gamma(p) - \Delta$. However, after the last $+v \in [0, p]$ transition at position j in the path, there is still the possibility to reach higher than $\gamma(p) - \Delta$ before the last 'below p' step ;

If $+1$ transitions makes the counter go higher than $\gamma(p) - \Delta$, then we are going to perform the following operations on the path :

- find k such that there is a state q that repeat itself between some step j and some step $j + k * \Delta$
- then remove the corresponding part of the path
- then pump one of the $+1$ cycle with $+(k - 1) * \Delta$

7.2 Above p :

First, we have to make some distinction right now :

either the counter of the initial path went higher than $\gamma(p) - \Delta$, and the counter in the new path is, after some time, below the counter of the initial path by a margin Δ

either it is not the case

In both case, we go from a step i with value of the counter below $\gamma(p) - \Delta$ to a step $i+1$ with value of the counter above $\gamma(p) - \Delta$.

Now we need to take care of the $\geq p$ tests and we can have $\pm p$ transitions. We also have, however, $= p$ tests.

Because of the $= p$ tests, we want the value of the counter at steps where we perform an $= p$ test to be, in the new path, equal to $\gamma(p) - \Delta$, as it is the new value for the parameter p.

In the case where after some time (below p), the counter of the new path becomes below the counter of the initial path by a margin Δ, and then reach $\gamma(p) - \Delta$ instead of $\gamma(p)$, the equality test would stay valid.

In the other case, we have to consider how we go from below p to above p. Here, because we did not have to shift the new path by a margin Δ, we know that the initial path's counter stays below $\gamma(p) - \Delta$ before taking a step making its counter go above $\gamma(p)$.

We are now interested in this particular step :

- it is not a $+1$ transition because otherwise the initial path's counter would be at $\gamma(p) - 1$ before taking this transition, which contradicts the hypothesis (and step four).
- if it is by a $+v \in [0, p]$ transition, we note that the value of v needs to be higher than Δ as, if we note the value of the counter z, we have $z < \gamma(p) - \Delta$ and $z + v > \gamma(p)$. We can therefore modify the value of the transition into $v - \Delta$ if need be.
- if it is by a $+p$ transition, the initial path's counter is augmented by $\gamma(p)$ while the new path's counter is augmented by $\gamma(p) - \Delta$.

When we stay around p we can therefore enforce that equality tests remain valid. Also when we are strictly above p we do some smoothing up, kind of the same way as "below p", in order to make the $+v \in [0, p]$ transitions work out.

From this process, we deduce that if $\gamma(p)$ is "big", that is, bigger than some exponential $exp >> n * (Q + 2) * \Delta$, we can find a new valuation which assigns p to $\gamma(p) - \Delta$ for which we also have a path and therefore a positive answer to the reachability problem. Then, if $\gamma(p) - \Delta$ is also big enough, we can repeat this process, until we obtain a valuation which assigns p to a small value, for which there exists a path.

Therefore a parametric one counter automata as described in [5] (part 6.4) is a positive instance of the reachability problem if and only if there exists a valuation with a "small" value assigned to p. It means that we only need to compute the reachability problem for one counter automaton $\mathcal{C}$ for every valuation with "small enough" p, which can then be done in PSPACE in regards to the size of automaton $\mathcal{C}$. As the size of $\mathcal{C}$ is an exponential in the size of $\mathcal{A}$, it leads to the fact that the reachability problem on parametric timed automaton with one parameter and two parametric clocks is in EXPSPACE, which completes the proof of Theorem 1.13.

8 Conclusion

In this report, we have improved the knowledge of the complexity of the reachability problem for parametric timed automaton with one parameter and two parametric clocks. Our proof for the lower bound provides a nice framework that can be useful in regards to other proofs, and has already been used elsewhere, most notably in regards to succinct one counter automaton. Such techniques, we believe, can be of interest for providing lower bound in other decision problems concerning infinite state systems.

We also provided a process for proving an upper bound, one that we believe can be expended in order to prove upper bound to parametric timed automaton with two parametric clocks and more than one parameter.

An interesting aspect of future work could also be to go into more detailed aspects of complexity and size in the upper bound proof as it could be used to obtain a more constructive approach. We could also expand on the study of the complexity of problems concerning one counter automaton, as those are deeply tied with timed automaton.

Bibliography

1. R. Alur, T.A.Henziger, and M.Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th Annual Symposium on Theory of Computing*, 1993.

2. S. Göller, C. Haase, J. Ouaknine, and J. Worrell. Model checking succinct and parametric one-counter automata. In *ICALP13*, volume 6199 of Lecture Notes in Computer Science. Springer, 2010.

3. Andrew Chiu, George Davida, and Bruce Litow. Division in logspace-uniform NC^1. *Theoretical Informatics and Applications. Informatique Théorique et Applications*, 35(3):259275, 2001.

4. Stefan Göller and Markus Lohrey. Branchning-time model checking of one-counter processes. Technical report, arXiv.org, 2009. http://arxiv.org/abs/0909.1102.

5. Daniel Bundala, Joël Ouaknine. On Parametric Timed Automata and One-Counter Machines.

6. John Fearnley and Marcin Jurdziǹski. Reachability in Two-Clock Timed Automata is PSPACE-complete.

7. Daniel Bundala and Joël Ouaknine. Advances in Parametric Real-Time Reasoning. Daniel Bundala and Joël Ouaknine

8. T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of Lecture Notes in Computer Science. 2001.

9. A. Jovanovic, D. Lime, and O. H. Roux. Integer parameter synthesis for timed automata. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, 2013.

10. L. Doyen. Robust parametric reachability for timed automata. *Information Processing Letters*, 102(5):208 213, 2007.

11. J. S. Miller. Decidability and complexity results for timed automata and semi-linear hybrid automata. In *Hybrid Systems: Computation and Control, volume 1790 of Lecture Notes in Computer Science.* 2000.

12. C. Haase, J. Ouaknine, and J. Worrell. On the relationship between reachability problems in timed and counter automata. In *RP*, volume 7550 of *Lecture Notes in Computer Science.* Springer, 2012.

13. Rajeev Alur and David L. Dill. A theory of timed automata.

14. Étienne André. An Inverse Method for the Synthesis of Timing Parameters in Concurrent Systems.

15. P. Hunter. Reachability in Succinct One-Counter Games. In: Bojanczyk M., Lasota S., Potapov I. (eds) Reachability Problems. RP 2015. Lecture Notes in Computer Science, vol 9328. Springer, Cham (2015).

16. P. Jančar, A. Kučera, F. Moller, and Z. Sawa. DP lower bounds for equivalence-checking and model-checking of one-counter automata. *Information Computation*, 188(1):119, 2004.

17. U. Hertrampf, C. Lautemann, T. Schwentick, H. Vollmer, and K. W. Wagner. On the power of polynomial time bit-reductions. In *Proc. 8th Annual Structure in Complexity Theory Conference*, 200207. IEEE Computer Society Press, 1993.