Reachability in Two-Clock Timed Automata is PSPACE-complete☆

John Fearnley[a,*], Marcin Jurdziński[b]

Abstract

Recently, Haase, Ouaknine, and Worrell have shown that reachability in two-clock timed automata is log-space equivalent to reachability in bounded one-counter automata. We show that reachability in bounded one-counter automata is PSPACE-complete.

Keywords: Timed automata, counter automata, PSPACE-complete.

1. Introduction

Timed automata [1] are a successful and widely used formalism, which are used in the analysis and verification of real time systems. A timed automaton is a non-deterministic finite automaton that is equipped with a number of real-valued *clocks*, which allow the automaton to measure the passage of time.

Perhaps the most fundamental problem for timed automata is the *reachability* problem: given an initial state, can we perform a sequence of transitions in order to reach a specified target state? In their foundational paper on timed automata [1], Alur and Dill showed that this problem is PSPACE-complete. To show hardness for PSPACE, their proof starts with a linear bounded automaton (LBA), which is a non-deterministic Turing machine with a finite tape of length n. They produced a timed automaton with $2n + 1$ clocks, and showed that the timed automaton can reach a specified state if and only if the LBA halts.

☆This work was supported by EPSRC grants EP/H046623/1 *Synthesis and Verification in Markov Game Structures*, EP/L011018/1 *Algorithms for Finding Approximate Nash Equilibria*, and EP/D063191/1 *The Centre for Discrete Mathematics and its Applications (DIMAP)*.

Email addresses: john.fearnley@liv.ac.uk (John Fearnley), marcin.jurdzinski@dcs.warwick.ac.uk (Marcin Jurdziński)

However, the work of Alur and Dill did not address the case where the number of clocks is small. This was rectified by Courcoubetis and Yannakakis [2], who showed that reachability in timed automata with only three clocks is still PSPACE-complete. Their proof cleverly encodes the tape of an LBA in a single clock, and then uses the two additional clocks to perform all necessary operations on the encoded tape. In contrast to this, Laroussinie et al. have shown that reachability in one-clock timed automata is complete for NLOGSPACE, and therefore no more difficult than computing reachability in directed graphs [3].

The complexity of reachability in two-clock timed automata has been left open. So far, the best lower bound was given by Laroussinie et al., who gave a proof that the problem is NP-hard via a very natural reduction from subset-sum [3]. Moreover, the problem lies in PSPACE, because reachability in two-clock timed automata is no harder than reachability in three-clock timed automata. However, the PSPACE-hardness proof of Courcoubetis and Yannakakis seems to fundamentally require three clocks, and does not naturally extend to the two-clock case. Naves [4] has shown that several extensions to two-clock timed automata lead to PSPACE-completeness, but his work does not advance upon the NP-hardness result for unextended two-clock timed automata.

In a recent paper, Haase et al. have shown a link between reachability in timed automata and reachability in *bounded counter automata* [5]. A bounded counter automaton is a non-deterministic finite automaton equipped with a set of counters, and the transitions of the automaton may add or subtract arbitrary integer constants to the counters. The state space of each counter is bounded by some natural number b, so the counter may only take values in the range $[0, b]$. Moreover, transitions may only be taken if they do not increase or decrease a counter beyond the allowable bounds. This gives these seemingly simple automata a surprising amount of power, because the bounds can be used to implement inequality tests against the counters.

Haase et al. show that reachability in two-clock timed automata is log-space equivalent to reachability in bounded *one*-counter automata. Reachability in bounded one-counter automata has also been studied in the context of one-clock timed automata with energy constraints [6], where it was shown that the problem lies in PSPACE and is NP-hard. It has also been shown that the reachability problem for *unbounded* one-counter automata is NP-complete [7], but the NP membership proof does not seem to generalise to bounded one-counter automata. Haase et al. also showed that reachability in bounded *two*-counter automata is log-space equivalent to reachability in *three*-clock timed automata, and that therefore, for any $k > 1$, reachability in bounded k-counter automata is PSPACE-complete [5].

Our contribution. We show that reachability in bounded one-counter automata is PSPACE-complete. Therefore, we resolve the complexity of reachability in two-clock timed automata. Our reduction uses two intermediate steps: *subset-sum games* and *safe counter-stack automata*.

Counter automata are naturally suited for solving subset-sum problems, so our reduction starts with a quantified version of subset-sum, which we call subset-sum games. One interpretation of satisfiability for quantified boolean formulas is to view the problem as a game between an *existential* player and a *universal* player. The players take turns to set their propositions to true or false, and the existential player wins if and only if the boolean formula is satisfied. Subset-sum games follow the same pattern, but apply it to subset-sum: the two players alternate in choosing numbers from sets, and the existential player wins if and only if the chosen numbers sum to a given target. Previous work by Travers can be applied to show that subset-sum games are PSPACE-complete [8].

We reduce subset-sum games to reachability in bounded one-counter automata. However, we will not do this directly. Instead, we introduce safe counter-stack automata, which are able to store multiple counters, but have a stack-like restriction on how these counters may be accessed. These automata are a convenient intermediate step, because having access to multiple counters makes it easier for us to implement subset-sum games. Moreover, the stack based restrictions mean that it is relatively straightforward to show that reachability in safe counter-stack automata is reducible, in logarithmic space, to reachability in bounded one-counter automata, which completes our result.

2. Subset-sum games

A subset-sum game is played between an *existential* player and a *universal* player. The game is specified by a pair (ψ, T), where $T \in \mathbb{N}$, and ψ is a list:

$$\forall \{A_1, B_1\} \; \exists \{E_1, F_1\} \; \ldots \; \forall \{A_n, B_n\} \; \exists \{E_n, F_n\},$$

where A_i, B_i, E_i, and F_i, are all natural numbers encoded in binary.

The game is played in rounds. In the first round, the universal player chooses an element from $\{A_1, B_1\}$, and the existential player responds by choosing an element from $\{E_1, F_1\}$. In the second round, the universal player chooses an element from $\{A_2, B_2\}$, and the existential player responds by choosing an element from $\{E_2, F_2\}$. This pattern repeats for rounds 3 through n. Thus, at the end of the game, the players will have constructed a sequence of numbers, and the existential player wins if and only if the sum of these numbers is T.

Formally, the set of *plays* of the game is the set:

$$\mathcal{P} = \prod_{1 \leq j \leq n} \{A_j, B_j\} \times \{E_j, F_j\}.$$

A play $P \in \mathcal{P}$ is winning for the existential player if and only if $\sum P = T$.

A strategy for the existential player consists of a list of functions $\mathsf{s} = (s_1, s_2, \ldots, s_n)$, where each function s_i dictates how the existential player should play in the ith round of the game. Thus, each function s_i is of the form:

$$s_i : \prod_{1 \leq j \leq i} \{A_j, B_j\} \to \{E_i, F_i\}.$$

This means that the function s_i maps the first i moves of the universal player to a decision for the existential player in the ith round. Note that the function s_i does not need to take the previous moves of existential player as inputs, because these moves are entirely determined by the previous moves of the universal player and the functions s_j with $j < i$.

A play P conforms to a strategy s if the decisions made by the existential player in P always agree with s. More formally, if $P = p_1 p_2 \ldots p_{2n}$ is a play, and $\mathsf{s} = (s_1, s_2, \ldots, s_n)$ is a strategy, then P conforms to s if and only if we have $s_i(p_1, p_3, \ldots, p_{2i-1}) = p_{2i}$ for all i. Given a strategy s, we define $\text{Plays}(\mathsf{s})$ be the set of plays that conform to s. A strategy s is *winning* if every play $P \in \text{Plays}(\mathsf{s})$ is winning for the existential player. The *subset-sum game problem* is to decide, for a given SSG instance (ψ, T), whether the existential player has a winning strategy for (ψ, T). Travers has shown that this problem is PSPACE-complete [8].

Lemma 1 ([8], Sec. 3)**.** *The subset-sum game problem is PSPACE-complete.*

3. Bounded one-counter automata

Outline. A bounded one-counter automaton has a single counter that can store values between 0 and some bound $b \in \mathbb{N}$. The automaton may add or subtract values from the counter, so long as the bounds of 0 and b are not overstepped. This property can be used to test inequalities against the counter. For example, let $n \in \mathbb{N}$ be a number, and suppose that we want to test whether the counter is smaller-than or equal to n. We first attempt to add $b - n$ to the counter, then, if that works, we subtract $b - n$ from the counter. This creates a sequence of two transitions which can be taken if and only if the counter is smaller-than or equal to n. A similar construction can be given for greater-than tests. For the sake of convenience, we will include explicit inequality testing in our formal definition, with the understanding that this is not actually necessary.

Formal definition. For two integers $a, b \in \mathbb{Z}$ we define $[a, b] = \{n \in \mathbb{Z} : a \leq n \leq b\}$ to be the subset of integers between a and b. A bounded one-counter automaton is defined by a tuple (L, b, Δ, l_0), where L is a finite set of locations, $b \in \mathbb{N}$ is a global counter bound, Δ specifies the set of transitions, and $l_0 \in L$ is the initial location. Each transition in Δ has the form (l, p, g_1, g_2, l'), where l and l' are locations, $p \in [-b, b]$ specifies how the counter should be modified, and $g_1, g_2 \in [0, b]$ give lower and upper guards for the transition. All numbers used in the specification of a bounded one-counter automaton are encoded in binary.

Each state of the automaton consists of a location $l \in L$ along with a counter value c. Thus, we define the set of states S to be $L \times [0, b]$. A transition exists between a state $(l, c) \in S$, and a state $(l', c') \in S$ if there is a transition $(l, p, g_1, g_2, l') \in \Delta$, where $g_1 \leq c \leq g_2$, and $c' = c + p$.

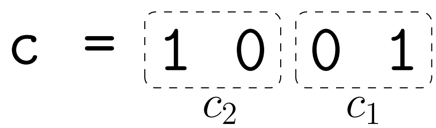
The reachability problem for bounded one-counter automata is specified as follows. An input to the problem is a pair $(\mathcal{B}, t)$, where $\mathcal{B}$ is a bounded one-counter automaton, and t is a target location. To solve the problem, we must decide whether there is a sequence of transitions between state $(l_0, 0)$ and the state $(t, 0)$. In a recent result, Haase, Ouaknine, and Worrell have shown that the reachability problem for bounded one-counter automata is equivalent to the reachability problem for two-clock timed automata.

Theorem 2 ([5])**.** *Reachability in bounded one-counter automata is log-space equivalent to reachability in two-clock timed automata.*

4. Counter-Stack Automata

Outline. In this section we consider the following question: can we use a bounded one-counter automaton to store multiple counters? The answer is yes, but doing so forces some interesting restrictions on the way in which the counters are modified. By the end of this section, we will have formalised these restrictions as *counter-stack* automata.

Suppose that we have a bounded one-counter automaton with counter c and bound $b = 15$. Hence, the width of the counter is 4 bits. Now suppose that we wish to store two 2-bit counters c_1 and c_2 in c. We can do this as follows:



We allocate the top two bits of c to store c_2, and the bottom two bits to store c_1. We can write to both counters: if we want to increment c_2 then we add 4 to c, and if we want to increment c_1 then we add 1 to c. However, note that if we increment c_1 too many times, then we will eventually cause an overflow, which will inadvertently modify the value of c_2. To deal with this issue, we will introduce the notion of *safe* counter-stack automata, which never overflow in this way.

If we want to test equality, then things become more interesting. It is easy to test equality against c_2: if we want to test whether $c_2 = 2$, then we test whether $8 \leq c \leq 11$ holds. But, we cannot easily test whether $c_1 = 2$ because we would have to test whether c is 2, 6, 10, or 14, and this list grows exponentially as the counters get wider. However, if we know that $c_2 = 1$, then we only need to test whether $c = 6$. Thus, we arrive at the following guiding principle: if you want to test equality against c_i, then you must know the values of c_j for all $j > i$. Counter-stack automata are a formalisation of this principle.

Counter-stack automata. A counter-stack automaton has a set of k distinct counters, which are referred to as c_1 through c_k. In contrast to our definitions for bounded one-counter automata, we will allow the counters to take all values from $\mathbb{N}$. Later, this will be refined by the concept of *safe* counter-stack automata. The defining feature of a counter-stack automaton is that the counters are arranged in a stack-like fashion:

- All counters may be increased at any time.
- c_i may only be tested for equality if the values of c_{i+1} through c_k are known.
- c_i may only be reset if the values of c_i through c_k are known.

When the automaton increases a counter, it adds a specified number $n \in \mathbb{N}$ to that counter. The automaton has the ability to perform equality tests against a counter, but the stack-based restrictions must be respected. An example of a valid equality test is $c_k = 3 \land c_{k-1} = 10$, because we are only required to test whether $c_{k-1} = 10$ in the case where $c_k = 3$ is known to hold. Conversely, the test $c_{k-1} = 10$ by itself is invalid, because it places no restrictions on the value of c_k.

The automaton may also reset a counter, but the stack-based restrictions still apply. Counter c_i may only be reset by a transition if that transition tests equality against the values of c_i through c_k. For example, c_{k-1} may only be reset if the transition is guarded by a test of the form $c_{k-1} = n_1 \land c_{k-2} = n_2$.

Formal definition. A counter-stack automaton is defined by a four-tuple (L, C, Δ, l_0), where L is a finite set of locations, $C = [1, k]$ is a set of counter indexes, $l_0 \in L$ is an initial location, and Δ specifies the transition relation. Each transition in Δ has the form (l, E, I, R, l') where:

- $l, l' \in L$ is a pair of locations.
- E is a partial function from C to $\mathbb{N}$ which specifies the equality tests. If $E(i)$ is defined for some i, then $E(j)$ must be defined for all $j \in C$ with $j > i$.
- $I \in \mathbb{N}^k$ specifies how the counters must be increased.
- $R \subseteq C$ specifies the set of counters that must be reset. It is required that $E(r)$ is defined for every $r \in R$.

All numbers used in the specification of a counter-stack automaton are encoded in binary.

Each state of the automaton is a location annotated with values for each of the k counters. That is, the state space of the automaton is $L \times \mathbb{N}^k$. A state $(l, c_1, c_2, \ldots, c_k)$ can transition to a state $(l', c'_1, c'_2, \ldots, c'_k)$ if and only if there exists a transition $(l, E, I, R, l') \in \Delta$, where the following conditions hold:

- For every i for which $E(i)$ is defined, we must have $c_i = E(i)$.
- For every $i \in R$, we must have $c'_i = 0$.
- For every $i \notin R$, we must have $c'_i = c_i + I_i$.

A *run* is a sequence of states $s_0, s_1, \dots, s_n$, where each s_i can transition to s_{i+1}.

Safe counter-stack automata. So far, we have allowed the counters to take any value from $\mathbb{N}$, but we now refine this by introducing the concept of safety. A counter-stack automaton is *b-safe*, for some $b \in \mathbb{N}$, if it is impossible for the automaton to increase a counter beyond b. Formally, this condition requires that, for every state $(l, c_1, c_2, \dots, c_k)$ that can be reached by a run from $(l_0, 0, 0, \dots, 0)$, we have $c_i \leq b$ for all i. If a counter-stack automaton is b-safe for some $b \in \mathbb{N}$, then we say that it is *safe*.

Note that the notion of safety is fundamentally different from the notion of boundedness that we used in bounded one-counter automata. In a bounded one-counter automaton, the bound b is given as part of the input, and the transitions of the automaton ensure that the counter is never increased beyond b. In contrast to this, it is easy to construct a counter-stack automaton that allows some counter to be increased arbitrarily many times, and therefore not all counter-stack automata are safe. Instead, safety is a property that some counter-stack automata happen to possess. In this paper, we will only consider reachability in counter-stack automata that are known to be b-safe for some b. In formal terms, this means that we are considering a promise problem. Goldreich's survey paper [9] provides an excellent introduction on the topic of promise problems, although no prior knowledge should be necessary in order to understand our result.

The reachability problem for safe counter-stack automata takes, as input, a triple $(\mathcal{S}, b, t)$, where $\mathcal{S}$ is a counter-stack automaton, $b \in \mathbb{N}$ is a natural number, and t is a target location. The promise is that $\mathcal{S}$ is a b-safe counter-stack automaton. To solve this problem, we must decide whether there is a sequence of transitions from $(l_0, 0, 0, \dots, 0)$ to $(t, 0, 0, \dots, 0)$ in $\mathcal{S}$.

Reduction to bounded one-counter automata. We now show that the reachability problem for safe counter-stack automata can be reduced, in logarithmic space, to the reachability problem in bounded one-counter automata. Let $(\mathcal{S}, b, t)$ be an instance of the reachability problem for safe counter stack automata, and suppose that $\mathcal{S}$ is b-safe. Our reduction will produce an instance $(\mathcal{B}, t)$ of the reachability problem for bounded one-counter automata.

Note that since $\mathcal{S}$ is b-safe, it must also be b' safe for every $b' \geq b$. Therefore, we can assume without loss of generality that $b = 2^n - 1$, for some $n \in \mathbb{N}$. This means that each counter in $\mathcal{S}$ is exactly n bits wide. We will construct a bounded one-counter automaton $\mathcal{B} = (L', b', \Delta', l'_0)$ that simulates $\mathcal{S}$. We will refer to the counters of $\mathcal{S}$ as c_1 through c_k, and the counter of $\mathcal{B}$ as c.

We follow the approach laid out at the start of this section. That is, we will set the bound $b' = 2^{k \cdot n} - 1$ so that c is $k \cdot n$ bits wide. We then partition these bits in order to implement the counters c_1 through c_k. The counter c_k will use the n most significant bits, the counter c_{k-1} will use the next n most significant bits, and so on.

We introduce some notation to formalise this encoding. Let $x \in [0, b]$ be a counter value for counter c_i. We define $\text{Enc}(x, i) = x \cdot 2^{(i-1) \cdot n}$. To understand this definition, note that for $i = 1$, we have $\text{Enc}(x, i) = x$. Then, for $i = 2$, we have that $\text{Enc}(x, i)$ is the value of x bit-shifted to the left n times. Thus, this definition simply translates x to the correct position in c.

We can now define the translation. We set $L' = L$ and $l'_0 = l_0$, which means that both automata have the same set of locations, and the same start location. We will use the transitions in Δ' to simulate S. For each transition $\delta = (l, E, I, R, l') \in \Delta$, we construct a transition $\delta' = (l, p, g_1, g_2, l') \in \Delta'$ between the same pair of locations. We will show the following property: if δ can be used to move from $(l, c_1, c_2, \ldots, c_k)$ to $(l', c'_1, c'_2, \ldots, c'_k)$, then δ' can be used to move from $(l, \sum_i \text{Enc}(c_i, i))$ to $(l', \sum_i \text{Enc}(c'_i, i))$.

We begin by defining p. We set:

$$p = \sum_{i \notin R} \text{Enc}(I_i, i) - \sum_{i \in R} \text{Enc}(E(i), i).$$

In other words, for each counter $i \notin R$ that is not to be reset, we add $\text{Enc}(I_i, i)$ to c, which correctly adds I_i to c_i. The assumption that $\mathcal{S}$ is b-safe is crucial here, because it ensures that the counters can never overflow their alloted space. For the counters $i \in R$, we subtract $E(i)$ from c_i. Recall that $E(i)$ must always be defined for the indices $i \in R$. This means that the transition may only be taken if $c_i = E(i)$. Thus, subtracting $E(i)$ from c_i will correctly set it to 0. These properties ensure that, if δ can be used to move from $(l, c_1, c_2, \ldots, c_k)$ to $(l', c'_1, c'_2, \ldots, c'_k)$, then $\sum_i \text{Enc}(c_i, i) + p = \sum_i \text{Enc}(c'_i, i)$.

Next we define the inequality tests. Let j be the smallest index for which $E(j)$ is defined, and recall that, by definition, $E_{j'}$ must be defined for every $j' \geq j$. Our guards are:

$$g_1 = \sum_{i \geq j} \text{Enc}(E(i), i),$$
$$g_2 = \sum_{i \geq j} \text{Enc}(E(i), i) + \text{Enc}(1, j) - 1.$$

It is straightforward to show that, if $c = \sum_i \text{Enc}(c_i, i)$, then we have $c_i = E(i)$ for all $i \geq j$ if and only if $g_1 \leq c \leq g_2$. This completes the argument that if δ can be used to move from $(l, c_1, c_2, \ldots, c_k)$ to $(l', c'_1, c'_2, \ldots, c'_k)$, then δ' can be used to move from $(l, \sum_i \text{Enc}(c_i, i))$ to $(l', \sum_i \text{Enc}(c'_i, i))$. We can now use this property to argue that $(t, 0, 0, \ldots, 0)$ can be reached from $(l_0, 0, 0, \ldots, 0)$ in $\mathcal{S}$ if and only if $(t, 0)$ can be reached from $(l'_0, 0)$ in $\mathcal{B}$. It is also clear that this reduction can be carried out in logarithmic space. Thus, we have shown the following lemma.

Lemma 3. *Let $\mathcal{S}$ be a counter-stack automaton. If $\mathcal{S}$ is b-safe, then each reachability problem $(\mathcal{S}, b, t)$ for $\mathcal{S}$, can be reduced, in logarithmic space, to a bounded one-counter reachability problem $(\mathcal{B}, t)$.*

5. Reachability in counter-stack automata is PSPACE-complete: outline

Our goal is to show that solving subset-sum games can be reduced to reachability in safe counter-stack automata. In Section 6, we will give a formal proof of the result, but in this section, we give an overview of the ideas behind our construction using the following two-round subset-sum game.

$$\big(\, \forall \, \{A_1, B_1\} \; \exists \, \{E_1, F_1\} \; \forall \, \{A_2, B_2\} \; \exists \, \{E_2, F_2\}, \; T\big).$$

For brevity, we will refer to this instance as (ψ, T) for the rest of this section. The construction is split into two parts: the *play* gadget and the *reset* gadget.

The play gadget. The play gadget is shown in Figure 1. The construction uses nine counters. The locations are represented by circles and the transitions are represented by edges. The annotations on the transitions describe the increments, resets, and equality tests: the notation $c_i + n$ indicates that

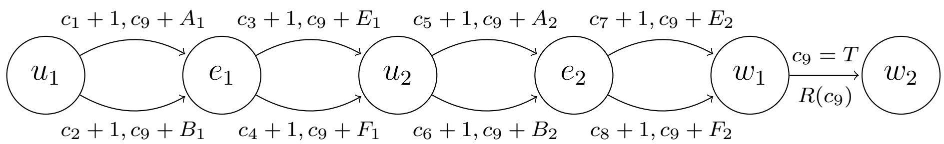


Figure 1: The play gadget

n is added to counter i, the notation $R(c_i)$ indicates that counter i is reset to 0, and the notation $c_i = n$ indicates that the transition may only be taken when $c_i = n$ is satisfied.

This gadget allows the automaton to implement a play of the SSG. The locations u_1 and u_2 allow the automaton to choose the first and second moves of the universal player, while the locations e_1 and e_2 allow the automaton to choose the first and second moves for the existential player. As the play is constructed, a running total is stored in c_9, which is the top counter on the stack. The final transition between w_1 and w_2 checks whether the existential player wins the play, and then resets c_9. Thus, the set of runs between u_1 and w_2 corresponds precisely to the set of plays won by the existential player in the SSG.

In addition to this, each outgoing transition from u_i or e_i comes equipped with its own counter. This counter is incremented if and only if the corresponding edge is used during the play, and this allows us to check precisely which play was chosen. These counters will be used by the reset gadget. The idea behind our construction is to force the automaton to pass through the play gadget multiple times. Each time we pass through the play gadget, we will check a different play, and our goal is to check a set of plays that verify whether the existential player has a winning strategy for the SSG.

The set of plays that must be checked. In our example, we must check four plays. The format of these plays is shown in Table 1. The table shows four different plays, which cover every possible strategy choice of the universal player. Clearly, if the existential player does have a winning strategy, then that strategy should be able to win against all strategy choices of the universal player. The plays are given in a very particular order: the first two plays contain A_1, while the second two plays contain B_1. Moreover, we always check A_2, before moving on to B_2.

We want to force the decisions made at e_1 and e_2 to form a consistent

Play	u_1	e_1	u_2	e_2
1	A_1	E_1 or F_1	A_2	E_2 or F_2
2	A_1	Unchanged	B_2	E_2 or F_2
3	B_1	E_1 or F_1	A_2	E_2 or F_2
4	B_1	Unchanged	B_2	E_2 or F_2

Table 1: The set of plays that the automaton will check

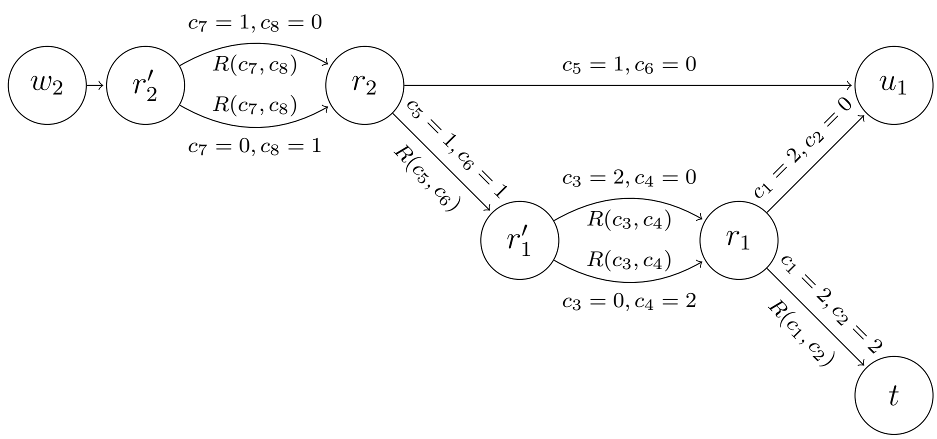


Figure 2: The reset gadget

strategy for the existential player. In this game, a strategy for the existential player is a pair $\mathsf{s} = (s_1, s_2)$, where s_i describes the move that should be made at e_i. It is critical to note that s_1 only knows whether A_1 or B_1 was chosen at u_1. This restriction is shown in the table: the automaton may choose freely between E_1 and F_1 in the first play. However, in the second play, the automaton must make the same choice as it did in the first play. The same relationship holds between the third and fourth plays. These restrictions ensure that the plays shown in Table 1 are a description of a strategy for the existential player.

The reset gadget. The reset gadget, shown in Figure 2, enforces the constraints shown in Table 1. The locations w_2 and u_1 represent the same locations as they did in Figure 1. To simplify the diagram, we have only included non-trivial equality tests. Whenever we omit a required equality test, it should be assumed that the counter is 0. For example, the outgoing transitions from r_2 implicitly include the requirement that c_7, c_8, and c_9 are all 0.

We consider the following reachability problem: can $(t, 0, 0, \ldots, 0)$ be reached from $(u_1, 0, 0, \ldots, 0)$? The structure of the reset gadget places restrictions on the runs that reach t. All such runs pass through the reset gadget exactly four times, and the following table describes each pass:

Pass	Path
1	$w_2 \to r'_2 \to r_2 \to u_1$
2	$w_2 \to r'_2 \to r_2 \to r'_1 \to r_1 \to u_1$
3	$w_2 \to r'_2 \to r_2 \to u_1$
4	$w_2 \to r'_2 \to r_2 \to r'_1 \to r_1 \to t$

To see why these paths must be taken, observe that, for every $i \in \{1, 3, 5, 7\}$, each pass through the play gadget increments either c_i or c_{i+1}, but not both. So, the first time that we arrive at r_2, we must take the transition directly to u_1, because the guard on the transition to r'_1 cannot possibly be satisfied after a single pass through the play gadget. When we arrive at r_2 on the second pass, we are forced to take the transition to r'_1, because we cannot have $c_5 = 1$ and $c_6 = 0$ after two passes through the play gadget. This transition resets both c_5 and c_6, so the pattern can repeat again on the third and fourth visits to r_2. The location r_1 behaves in the same way as r_2, but the equality tests are scaled up, because r_1 is only visited on every second pass through the reset gadget.

This explains why all strategies of the universal player must be considered. The transition between r_2 and u_1 forces the play gadget to increment c_5, and therefore the first and third plays must include A_2. Similarly, the transition between r_2 and r'_1 forces the second and fourth plays to include B_2. Meanwhile, the transition between r_1 and u_1 forces the first and second plays to include A_1, and the transition between r_1 and t forces the third and fourth plays to include B_1. Thus, we select the universal player strategies exactly as Table 1 prescribes.

The transitions between r'_1 and r_1 check that the existential player is playing a consistent strategy. When the automaton arrives at r'_1 during the second pass, it verifies that either E_1 was included in the first and second plays, or that F_1 was included in the first and second plays. If this is not the case, then the automaton gets stuck. The counters c_3 and c_4 are reset when moving to r_1, which allows the same check to occur during the fourth pass. For the sake of completeness, we have included the transitions between

r'_2 and r_2, which perform the same check for E_2 and F_2. However, since the existential player is allowed to change this decision on every pass, the automaton can never get stuck at r'_2.

These properties ensure that location t can be reached if and only if the existential player has a winning strategy for (ψ, T). As we will show in the next section, the construction extends to arbitrarily large SSGs, which then leads to a proof that the problem of solving subset-sum games can be reduced to reachability in counter-stack automata. Moreover, our counter-stack automata are guaranteed to be safe: c_9 may never exceed the maximum value that can be achieved by a play of the SSG, and reset gadget ensures that no other counter may exceed 4. Thus, we will then be able to apply Lemma 3 to prove PSPACE-hardness of reachability in bounded one-counter automata, and we can then invoke Theorem 2 to prove PSPACE-hardness of reachability in two-clock timed automata.

6. Reachability in counter-stack automata is PSPACE-complete: formal proof

In this section, we give a formal description of our reduction from subset-sum games to reachability in safe counter-stack automata.

Sequential strategies for SSGs. We start by formalising the ideas behind Table 1. Recall that the table gives a strategy for the existential player in the form of a list of plays. Moreover, the table gave a very specific ordering in which these plays must appear. We now formalise this ordering.

We start by dividing the integers in the interval $[1, 2^n]$ into *i-blocks*. The 1-blocks partition the interval into two equally sized blocks. The first 1-block consists of the range $[1, 2^{n-1}]$, and the second 1-block consists of the range $[2^{n-1} + 1, 2^n]$. There are four 2-blocks, which partition the 1-blocks into two equally sized sub-ranges. This pattern continues until we reach the n-blocks.

Formally, for each $i \in \{1, 2, \ldots, n\}$, there are 2^i distinct i-blocks. The set of i-blocks can be generated by considering the intervals $[k + 1, k + 2^{n-i}]$ for the first 2^i numbers $k \geq 0$ that satisfy $k \bmod 2^{n-i} = 0$. An i-block is *even* if k is an even multiple of 2^{n-i}, and it is *odd* if k is an odd multiple of 2^{n-i}.

The ordering of the plays in Table 1 can be described using blocks. There are four 2-blocks, and A_2 appears only in even 2-blocks, while B_2 only appears in odd 2-blocks. Similarly, A_1 only appears in the even 1-block, while B_1 only appears in the odd 1-block. The restrictions on the existential player can also be described using blocks: the existential player's strategy may not change between E_i and F_i during an i-block. We formalise this idea in the following definition.

Definition 4 (Sequential strategy)**.** *A sequential strategy for the existential player in* (ψ, T) *is a list of* 2^n *plays* $\mathcal{S} = P_1, P_2, \dots, P_{2^n}$*, where for every* i *in the range* $1 \leq i \leq n$*, and every* i*-block* L *we have:*

- *If* L *is an even* i*-block, then* P_j *must contain* A_i *for all* $j \in L$*.*
- *If* L *is an odd* i*-block, then* P_j *must contain* B_i *for all* $j \in L$*.*
- *We either have* $E_i \in P_j$ *for all* $j \in L$*, or we have* $F_i \in P_j$ *for all* $j \in L$*.*

An alternative way of viewing this definition is by inspecting the bits used to represent j. The first two conditions state that P_j must contain A_i if the ith bit of j is 0, and it must contain B_i if the ith bit of j is 1. The third condition states that if the first i bits of j are the same as the first i bits of k, then P_j and P_k must agree on the first i choices for the existential player.

We say that $\mathcal{S}$ is winning for the existential player if $\sum P_j = T$ for every $P_j \in \mathcal{S}$. The following lemma shows an equivalence between strategies and sequential strategies. This property is fairly obvious, because one can always turn a strategy into a sequential strategy by listing all plays that conform to that strategy, and one can always turn a sequential strategy into a strategy because, by the third condition of Definition 4, in each round of the game all plays that are consistent with the moves made so far have the same strategy choice for the existential player. Nevertheless, for the sake of completeness, we give a formal proof this fact below.

Lemma 5. *The existential player has a winning strategy if and only if the existential player has a sequential winning strategy.*

Proof. Let $\mathsf{s} = (s_1, s_2, \dots, s_n)$ be a winning strategy for the existential player. We define a sequential winning strategy as follows. Note that, since the universal player makes n different choices, we have that Plays(s) contains exactly 2^n plays[1]. We argue that these plays can be ordered so that they

[1] If there exists an i such that $A_i = B_i$ or $E_i = F_i$, then Plays(s) may actually contain fewer than 2^n plays, because some plays will be repeated. In this case, we annotate each play with the corresponding strategy choices, and we define Plays(s) to be the set of annotated plays. This ensures that Plays(s) contains exactly 2^n plays. This technical detail does not affect our argument

form a sequential strategy. We give an iterative procedure that achieves this task: the first step of the procedure will ensure that the 1-blocks contain the correct plays, the second step will ensure that the 2-blocks contain the correct plays, and so on. In the first step, we observe that exactly 2^{n-1} of the plays contain A_1, while exactly 2^{n-1} of the plays contain B_1, so we can order the plays so that the even 1-block contains all plays containing A_1.

Now suppose that we have finished processing the i-blocks. We observe that each i-block L has exactly $2^{n-(i+1)}$ plays that contain A_{i+1}. Therefore, for each i-block L, we can order the plays in L so that the even $(i+1)$-block has all plays that contain A_{i+1}, and the odd $(i+1)$-block has all plays that contain B_{i+1}.

After we have finished processing the n-blocks, we will have a list of plays $\mathcal{S} = P_1, P_2, \ldots, P_{2^n}$ where:

- P_j contains A_i whenever j is in an even i-block.
- P_j contains B_i whenever j is in an odd i-block.

So $\mathcal{S}$ satisfies the first two conditions of Definition 4. We argue that $\mathcal{S}$ also satisfies the third condition. Let L_i be an i-block. By definition, for every $j \leq i$, there is a unique j-block L_j such that $L_i \subseteq L_j$. We define a play prefix $F \in \Pi_{1 \leq j \leq i}\{A_i, B_i\}$ so that for each $j \leq i$ we have $A_j \in F$ if and only if $A_j \in P$ for all $P \in L_j$. Note that, by construction, for each play $P \in L_i$, we have $F \subseteq P$. Since $\mathcal{S}$ is a reordering of $\text{Plays}(\mathsf{s})$, we must have $s_i(F) \in P$ for every $P \in L_i$. Hence, $\mathcal{S}$ satisfies Definition 4. Moreover, since s is winning, we have that every play in $\text{Plays}(\mathsf{s})$ is winning, and therefore $\mathcal{S}$ is a winning sequential strategy.

Now let $\mathcal{S} = P_1, P_2, \ldots, P_{2^n}$ be a winning sequential strategy. We give a high level description of a winning strategy for the SSG. At the start of the strategy we set $L_0 = [1, 2^n]$. In each round i of the game, let $D_i \in \{A_i, B_i\}$ be the decision made by the universal player. We select L_i to be the unique i-block in L_{i-1} such that $D_i \in P_j$ for all $j \in L_i$. We play E_i if $E_i \in P_j$ for all $j \in L_i$, and we play F_i if $F_i \in P_j$ for all $j \in L_i$. It is straightforward to encode this strategy in the form $\mathsf{s} = (s_1, s_2, \ldots, s_n)$. By construction, when we play s, the outcome of the game will be some play P_j from $\mathcal{S}$. Since every play P_j in $\mathcal{S}$ is winning for the existential player, we have that s is a winning strategy. □

The base automaton. We will describe our construction in two steps. Recall, from Figures 1 and 2, that the top counter is used by the play gadget to store the value of the play, and to test whether the play is winning. We begin by constructing a version of the automaton that omits the top counter. That is, if c_k is the top counter, we modify the play gadget by removing all increases to c_k, and the equality test for c_k between w_1 and w_2. We call this the *base* automaton. Later, we will add the constraints for c_k back in, to construct the *full* automaton.

We now give a formal definition of the base automaton. The location and counter names are consistent with, and extensions of, those used in Figures 1 and 2. For each natural number n, we define a counter-stack automaton $\mathcal{A}_n$. The automaton has the following set of locations:

- locations u_i and e_i for each $i \in [1, n]$,
- locations w_1 and w_2,
- reset locations r_i and r'_i for each $i \in [1, n]$, and
- the goal location t.

The automaton uses $k = 2n + 1$ counters. The top counter c_k is reserved for the full automaton, and will not be used in this construction. We introduce shorthands for the counters 1 through $2n$: for each integer i we define $a_i = c_{4(i-1)+1}$, $b_i = c_{4(i-1)+2}$, $e_i = c_{4(i-1)+3}$, and $f_i = c_{4(i-1)+4}$. For example, in Figure 1, we have $a_1 = c_1$ and $a_2 = c_5$, and these are precisely the counters associated with A_1 and A_2, respectively. The same relationship holds between b_1 and B_1, between b_2 and B_2, and so on.

The transitions of the automaton are defined as follows. Whenever we omit a required equality test against a counter c_i, it should be assumed that the transition includes the test $c_i = 0$.

- Each location u_i has two transitions to e_i: a transition that adds 1 to a_i, and a transition that adds 1 to b_i.
- Each location e_i has two outgoing transitions: a transition that adds 1 to e_i, and transition that adds 1 to f_i. For the locations e_i with $i < n$ these transitions go to u_{i+1}, and for the location e_n the transitions go to w_1.
- Location w_1 has a transition to w_2, and w_2 has a transition to r'_n. These transitions do not increase any counter, and do not test any equalities.

- Each location r'_i has two outgoing transitions to r_i. Firstly, there is a transition that tests $e_i = 2^{n-i}$ and $f_i = 0$, and then resets e_i and f_i. Secondly, there is a transition that tests $e_i = 0$ and $f_i = 2^{n-i}$, and then resets both e_i and f_i.

- Each location r_i has two outgoing transitions. Firstly, there is a transition to u_1 that tests $a_i = 2^{n-i}$ and $b_i = 0$. Secondly, there is a transition that tests $a_i = 2^{n-i}$ and $b_i = 2^{n-i}$ and then resets both a_i and b_i. For locations r_i with $i > 1$, this transition goes to to r'_{i-1}. For the location r_1, this transition goes to location t.

Runs in the base automaton. We now describe the set of runs that are possible in the base automaton. We decompose every run of the automaton into segments, such that each segment contains a single pass through the play gadget. More formally, we decompose R into segments $R_1, R_2, \dots,$ where each segment R_i starts at u_1, and ends at the next visit to u_1. We say that a run gets *stuck* if the run does not end at $(t, 0, 0, \dots, 0)$, and if the final state of the run has no outgoing transitions. We say that a run R gets stuck during an i-block L if there exists a $j \in L$ such that R_j gets stuck. Let R be a run in $\mathcal{A}_n$. The following lemma describes the set of reset states that each segment of R must pass through.

Lemma 6. *Let R be a run in $\mathcal{A}_n$. Either:*

- *R_j visits precisely the reset locations $\{r'_i, r_i\}$ for which $j \bmod 2^{n-i} = 0$, or*

- *R_j gets stuck.*

Proof. We will prove this lemma by induction over i. The base case, where $i = n$, is trivial because $j \bmod 2^{n-n}$ is always equal to 0, and it is clear from the construction that every segment R_j must always visit both r'_n and r_n.

For the inductive step, we suppose that the lemma has been shown for $i + 1$, and we will show that the lemma holds for i. We know that, in order to reach r'_i or r_i, a segment must first visit r'_{i+1}. By the inductive hypothesis, we know that only segments R_j with $j \bmod 2^{n-(i+1)}$ visit r_{i+1}. At the start of R, we have $a_i = b_i = 0$. On the first visit to r_{i+1}, we clearly cannot take the transition to r'_i, because we have $a_i + b_i = 2^{n-(i+1)}$, and the transition to r'_i requires $a_i + b_i = 2^{n-i}$. Thus, we either have to take the transition to u_1, or we get stuck. On the second visit to r_{i+1}, we cannot take the transition to u_1, because we have $a_i + b_i = 2^{n-i}$, and the transition to u_1 requires $a_i + b_i = 2^{n-(i+1)}$. Thus, either we get stuck, or we take the transition to r'_i. The transition between r_{i+1} and r'_i resets both a_i and b_i to 0. Thus, we can repeat the argument, and conclude that locations r'_i and r_i are visited by exactly the segments R_j where $j \bmod 2^{n-i} = 0$. □

We now apply Lemma 6 to give the following characterisation of the runs in $\mathcal{A}_n$.

Lemma 7. *A run R in $\mathcal{A}_n$ does not get stuck if and only if, for every i-block L, all of the following hold.*

- *If L is an even i-block, then R_j must increment a_i for every $j \in L$.*
- *If L is an odd i-block, then R_j must increment b_i for every $j \in L$.*
- *Either R_j increments e_i for every $j \in L$, or R_j increments f_i for every $j \in L$.*

Proof. Let R be a run of $\mathcal{A}_n$. For the counters a_i and b_i, we have the following facts:

- At the start of the first i-block, we have $a_i = b_i = 0$.
- Each i-block contains exactly 2^{n-i} segments. Each segment must increment one of a_i or b_i, but not both.
- At the end of each odd i-block, we must take the transition from r_i to u_1 to avoid getting stuck. This transition requires $a_i = 2^{n-i}$ and $b_i = 0$.
- At the end of each even i-block, we must take the transition from r_i to r'_{i-1} to avoid getting stuck. This transition requires $a_i = 2^{n-i}$ and $b_i = 2^{n-i}$, and resets a_i and b_i to 0.

These facts imply that a_i must be incremented during every run in an odd i-block to prevent the automaton getting stuck, and b_i must be incremented during every run in an even i-block to prevent the automaton getting stuck. It can also be verified that, if a_i is incremented during every run in an odd i-block, and b_i is incremented during every run in an even i-block, then the automaton will never get stuck at r_i.

Similarly, for the counters e_i and f_i we have the following facts.

- At the start of the first i-block, we have $e_i = f_i = 0$.
- Each i-block contains exactly 2^{n-i} runs. Each run must increment one of e_i or f_i, but not both.
- At the end of each i-block, we must take one of the two transitions from r'_i to r_i to avoid getting stuck. These transitions require that $e_i = 2^{n-i}$ and $f_i = 0$, or $e_i = 0$ and $f_i = 2^{n-i}$.

These facts imply that either e_i is incremented during every run in an i-block, or f_i is incremented during every run in an i-block, or the automaton will get stuck when moving from r'_i to r_i at the end of the i-block. It can also be verified that, if the automaton increases e_i during every run in an i-block, then the automaton will not get stuck moving from r'_i to r_i, and if the automaton increases f_i during every run in an i-block, then the automaton will not get stuck moving from r'_i to r_i.

Note that, in $\mathcal{A}_n$, it is only possible for R to get stuck at the locations r'_i and r_i. Therefore, we have shown that R does not get stuck if and only if the three conditions of this lemma hold for R. □

We say that a run is *successful* if it eventually reaches $(t, 0, 0, \ldots, 0)$. The next lemma shows that every run is either successful or eventually gets stuck, by showing that there are no infinite runs in the base automaton.

Lemma 8. *Every run is either successful or gets stuck.*

Proof. We show that a run is successful if and only if it does not get stuck. By definition, if a run gets stuck, then it never reaches location t, and it is therefore not successful. Conversely, let R be a run that does not get stuck. By Lemma 6, we know that the segment R_{2^n} must visit the location r_1. Furthermore, by Lemma 7, we know that when R_{2^n} visits r_1, we must have $a_1 = 2^{n-1}$ and $b_1 = 2^{n-1}$. Thus, R_{2^n} must take the transition from r_1 to t to avoid getting stuck. Therefore, R is successful. □

Note that every successful run must have exactly 2^n segments. If we compare Lemma 7 with Definition 4, then we can see that the set of successful runs in $\mathcal{A}_n$ corresponds exactly to the set of sequential strategies for the existential player in the SSG.

Since we eventually want to implement $\mathcal{A}_n$ as a safe one-counter automaton, it is important to prove that $\mathcal{A}_n$ is b-safe for some $b \in \mathbb{N}$. In the following lemma, we show that $\mathcal{A}_n$ is 2^n-safe.

Lemma 9. *Along every run of $\mathcal{A}_n$ we have that counters a_i and b_i never exceed 2^{n-i+1}, and counters e_i and f_i never exceed 2^{n-i}.*

Proof. This lemma follows from Lemma 6. Let R be a run. Lemma 6 implies that the transition from r_i to r'_{i-1} is taken in every segment R_j such that $j \bmod 2^{n-(i-1)}$. This transition resets both a_i and b_i to 0. Therefore, neither of these counters may exceed $2^{n-(i-1)}$. Similarly, Lemma 6 implies that every segment R_j such that $j \bmod 2^{n-i} = 0$ must move from r'_i to r_i. Both of the transitions between r'_i and r_i reset e_i and f_i, and therefore neither of these counters may exceed 2^{n-i}. □

The full automaton. Let (ψ, T) be an SSG instance, where ψ is:

$$\forall \{A_1, B_1\} \, \exists \{E_1, F_1\} \, \dots \, \forall \{A_n, B_n\} \, \exists \{E_n, F_n\}.$$

We will construct a counter-stack automaton $\mathcal{A}_\psi$ from $\mathcal{A}_n$. Recall that the top counter c_k is unused in $\mathcal{A}_n$. We modify the transitions of $\mathcal{A}_n$ as follows. Let δ be a transition. If δ increments a_i then it also adds A_i to c_k, if δ increments b_i then it also adds B_i to c_k, if δ increments e_i then it also adds E_i to c_k, and if δ increments f_i then it also adds F_i to c_k. We also modify the transition between w_1 and w_2, so that it checks whether $c_k = T$, and resets c_k.

Since we only add extra constraints to $\mathcal{A}_n$, the set of successful runs in $\mathcal{A}_\psi$ is contained in the set of successful runs of $\mathcal{A}_n$. Recall that the set of successful runs in $\mathcal{A}_n$ encodes the set of sequential strategies for the existential player in (ψ, T). In $\mathcal{A}_\psi$, we simply check whether each play in the sequential strategy is winning for the existential player. Thus, we have shown the following lemma.

Lemma 10. *The set of successful runs in $\mathcal{A}_\psi$ corresponds precisely to the set of winning sequential strategies for the existential player in (ψ, T).*

We also have that $\mathcal{A}_\psi$ is b-safe for some $b \in \mathbb{N}$, such that b has polynomially many bits in the size of (ψ, T). Bounds for counters c_1 through c_{k-1} are shown in Lemma 9, and counter c_k may never exceed $\sum\{A_i, B_i, E_i, F_i : 1 \leq i \leq n\}$. This completes the reduction from subset-sum games to reachability in safe counter-stack automata, and gives us our main result.

Theorem 11. *There exists a family of safe counter-stack automata for which the reachability problem is PSPACE-hard.*

Since our construction always produces safe counter-stack automata, we can apply Lemma 3 and Theorem 2 to obtain the following corollaries.

Corollary 12.

- *Reachability in bounded one-counter automata is PSPACE-complete.*
- *Reachability in two-clock timed automata is PSPACE-complete.*

References

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

[2] C. Courcoubetis, M. Yannakakis, Minimum and maximum delay problems in real-time systems, Formal Methods in System Design 1 (4) (1992) 385–415.

[3] F. Laroussinie, N. Markey, P. Schnoebelen, Model checking timed automata with one or two clocks, in: Proc. of CONCUR, Vol. 3170 of Lecture Notes in Computer Science, 2004, pp. 387–401.

[4] G. Naves, Accessibilité dans les automates temporisé à deux horloges, Rapport de Master, MPRI, Paris, France (2006).

[5] C. Haase, J. Ouaknine, J. Worrell, On the relationship between reachability problems in timed and counter automata, in: Proc. of RP, Vol. 7550 of Lecture Notes in Computer Science, 2012, pp. 54–65.

[6] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, J. Srba, Infinite runs in weighted timed automata with energy constraints, in: Proc. of FORMATS, Vol. 5215 of Lecture Notes in Computer Science, 2008, pp. 33–47.

[7] C. Haase, S. Kreutzer, J. Ouaknine, J. Worrell, Reachability in succinct and parametric one-counter automata, in: Proc. of CONCUR, Vol. 5710 of Lecture Notes in Computer Science, 2009, pp. 369–383.

[8] S. Travers, The complexity of membership problems for circuits over sets of integers, Theoretical Computer Science 369 (13) (2006) 211–229.

[9] O. Goldreich, On promise problems: A survey, in: Essays in Memory of Shimon Even, 2006, pp. 254–290.